

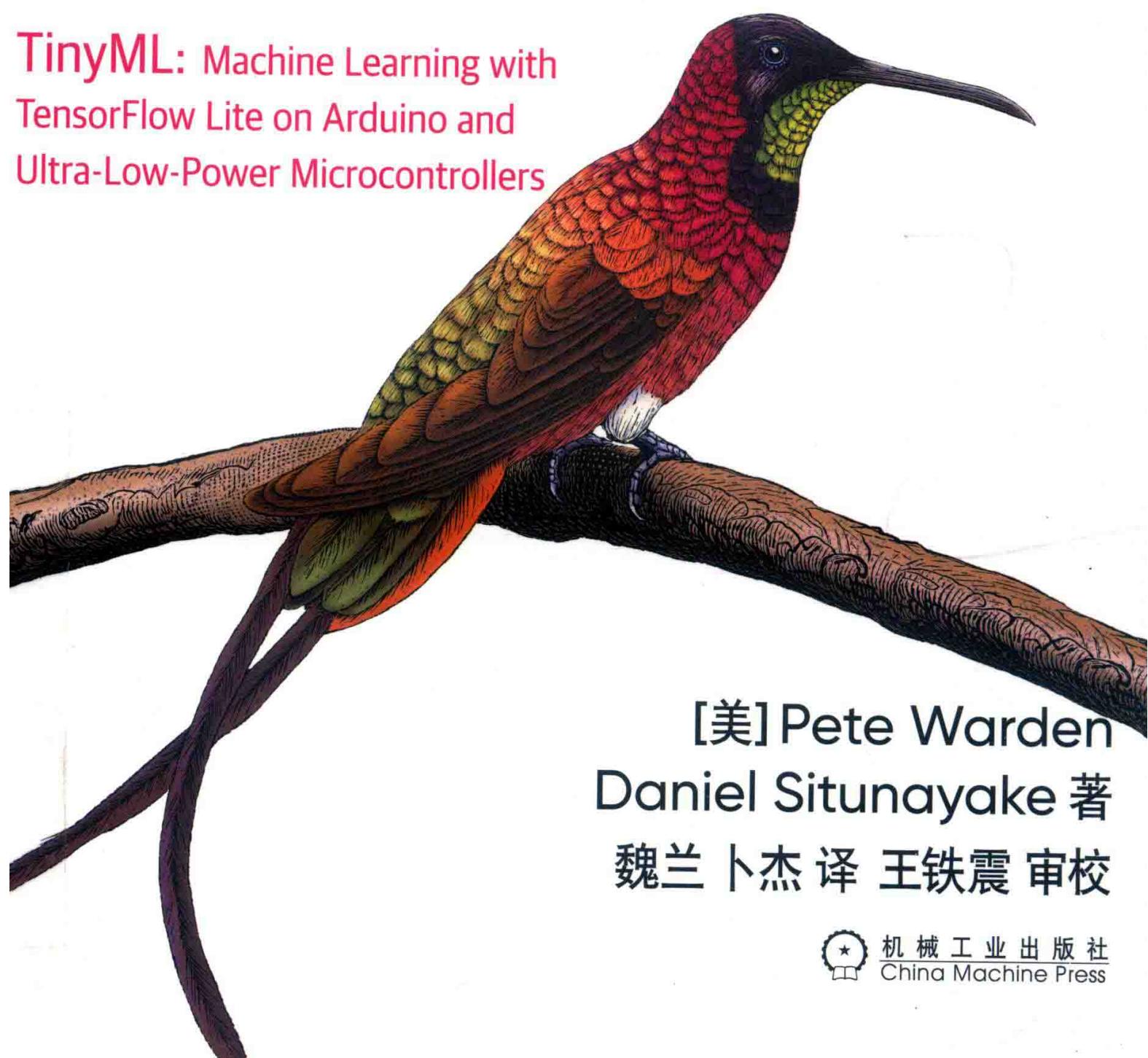
O'REILLY®

HZ BOOKS
华章 IT

TinyML

基于TensorFlow Lite在Arduino和
超低功耗微控制器上部署机器学习

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



[美] Pete Warden
Daniel Situnayake 著
魏兰 卜杰 译 王铁震 审校



机械工业出版社
China Machine Press

TinyML 基于TensorFlow Lite在Arduino和超低功耗微控制器上部署机器学习

深度学习网络正在变得越来越小。Google Assistant团队可以使用大小只有14KB的模型检测单词——模型小到可以在微控制器上运行。在这本实用的书中，你将进入TinyML的世界。TinyML将深度学习和嵌入式系统相结合，使得微型设备可以做出令人惊叹的事情。

本书解释了如何训练足够小的模型以适合任何环境。对于希望在嵌入式系统中搭建机器学习项目的软件及硬件开发人员而言，本书是一个理想的指南，它将一步步地指导你搭建一系列TinyML项目。阅读本书不需要任何机器学习或者微控制器开发经验。

你将深入了解以下内容：

- 如何创建语音识别程序、行人检测程序和响应手势的魔杖程序。
- 如何使用Arduino和超低功耗微控制器。
- 机器学习的基本知识以及如何训练自己的模型。
- 如何训练模型以理解音频、图像和加速度传感器数据。
- 如何使用TensorFlow Lite for Microcontrollers，这是Google用于TinyML的工具包。
- 如何调试程序并提供隐私和安全保障。
- 如何优化延迟、功耗、模型以及二进制文件大小。

Pete Warden是Google公司TensorFlow面向移动和嵌入式设备部分的技术主管，也是TensorFlow团队的创始成员之一。他曾是Jetpac的首席技术官（CTO）和创始人，该公司于2014年被Google收购。

Daniel Situnayake在Google领导TensorFlow Lite的开发宣传工作，并协助运营TinyML meetup小组。他是Tiny Farms的联合创始人，这是美国第一家利用自动化技术以工业规模生产昆虫蛋白的公司。

MACHINE LEARNING / EMBEDDED DEVICES

O'Reilly Media, Inc.授权机械工业出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

投稿热线：(010) 88379604

读者信箱：hxit@hzbook.com

客服电话：(010) 88361066 88379833 68326294

华章网站：www.hzbook.com

网上购书：www.china-pub.com

数字阅读：www.hzmedia.com.cn

“对于那些对在资源受限的设备上实现机器学习感兴趣的人来说，这是一本必读的书。本书是人工智能发展的一个里程碑。”

—Massimo Banzi

Arduino联合创始人

“本书通过清晰有趣的用例讲述了如何在基于Arm的微控制器上部署机器学习。”

—Jem Davies

Arm副总裁、Fellow及机器学习事业部总经理



上架指导：计算机/人工智能

ISBN 978-7-111-66422-2



9 787111 664222 >

定价：149.00元

O'Reilly 精品图书系列

TinyML

基于TensorFlow Lite在Arduino和
超低功耗微控制器上部署机器学习

[美]Pete Warden Daniel Situnayake 著
魏兰 卜杰 译
王铁震 审校

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

O'Reilly Media, Inc. 授权机械工业出版社出版

机械工业出版社

图书在版编目 (CIP) 数据

TinyML：基于 TensorFlow Lite 在 Arduino 和超低功耗微控制器上部署机器学习 /
(美) 皮特·沃登 (Pete Warden), (美) 丹尼尔·西图纳亚克 (Daniel Situnayake) 著；
魏兰, 卜杰译. —北京: 机械工业出版社, 2020.8
(O'Reilly 精品图书系列)

书名原文: TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers

ISBN 978-7-111-66422-2

I. T… II. ①皮… ②丹… ③魏… ④卜… III. 机器学习 IV. TP181

中国版本图书馆 CIP 数据核字 (2020) 第 163722 号

北京市版权局著作权合同登记

图字: 01-2020-2392 号

Copyright © 2020 Pete Warden and Daniel Situnayake.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2020.
Authorized translation of the English edition, 2019 O'Reilly Media, Inc., the owner of all rights to publish
and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2019。

简体中文版由机械工业出版社出版 2020。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

封底无防伪标均为盗版

本法律顾问

北京大成律师事务所 韩光 / 邹晓东

书 名 / TinyML：基于 TensorFlow Lite 在 Arduino 和超低功耗微控制器上部署机器学习

书 号 / ISBN 978-7-111-66422-2

责任编辑 / 孙榕舒

封面设计 / David Futato, 张健

出版发行 / 机械工业出版社

地 址 / 北京市西城区百万庄大街 22 号 (邮政编码 100037)

印 刷 / 北京文昌阁彩色印刷有限责任公司

开 本 / 178 毫米 × 233 毫米 16 开本 27 印张

版 次 / 2020 年 9 月第 1 版 2020 年 9 月第 1 次印刷

定 价 / 149.00 元 (册)

客服电话: (010) 88361066 88379833 68326294

华章网站: www.hzbook.com

投稿热线: (010) 88379604

读者信箱: hzit@hzbook.com

O'Reilly Media, Inc.介绍

O'Reilly以“分享创新知识、改变世界”为己任。40多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来O'Reilly图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能帮助他们一臂之力。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

译者序一

刘慈欣在《三体》中有这样一段对四维世界的描述：“任何东西都不可能挡住它后面的东西，任何封闭体的内部也都是能看到的。这只是一个简单的规则，但如果世界真按这个规则呈现，在视觉上是极其震撼的。当所有的遮挡和封闭都不存在，一切都暴露在外时，目击者首先面对的是相当于三维世界中亿万倍的信息量，对于涌进视觉的海量信息，大脑一时无法把握。”在翻译本书第 11 章时，我看到作者用两张加速度传感器数据的图来说明机器学习可以理解对人类来说看起来毫无道理的数据，这使我很偶然地想起这段对四维世界的描述。这个有些跳跃的联想源于我们常常这样类比低维世界：想象生活在三维空间中的一个二维平面人，不管什么物体穿过二维世界，在他眼中都只是一些长短不一的线段而已，只有从平面中跳出来，他才能看到全貌。虽然低维空间只是高维空间的一个切面，但并不代表其信息量也只有高维空间一个切面的信息量，就像那些变化多端的线段，“变化”本身就是另一个维度的信息，只是对于生来就在感觉和思考二维空间的二维平面人来说难以理解和把握。

在某种意义上，机器学习（Machine Learning, ML）就没有这种维度和先验的负累，其学习和泛化的能力无可限量。机器学习的输入“张量”（tensor）是一种高维结构，其生来就被设计用于变换和投影复杂的高维数据。这个理解也许谬以千里，但是我发现它对于宽慰我面对机器学习的“失控感”很有帮助。作为一个喜欢逐行读代码的老派程序员，我刚刚接触机器学习时总有诸多不适——纵然能看懂每行代码，还是不知道精度为什么不够，模型为什么不能泛化——算法的精髓几乎不在代码中，而更多地隐藏于数以亿计对程序员而言几乎毫无意义的权重中。纵使真的可以通过数学公式推导出哪个权重导致了结果的不精确，其承载的海量信息细节也已然超出我们的处理能力，最终只能通过“与其说是科学，不如说是艺术”^{译注1}的方法不断调参，反复训练并优化模型。以前总以为代码世界是对抗不确定性的最后一片净土，但这也许不过是二进制文明尚太过原始所带来的一种错觉。当然，无视“高维数据”的“歪理”，拥抱这种失控感也不是一件困难的事，因为我们能用代码创造更富生命力的应用，能用代码看到更广阔的世界，这

译注 1：出自 François Chollet 的 *Deep Learning with Python*。

种成就感和喜悦感对每个程序员而言都难以抗拒。

不过本书的重点不完全是机器学习，而是面向微控制器的机器学习，也就是 Pete 所说的 TinyML。我第一次接触微控制器是很多年前在学校和小伙伴一起参加 Intel“创新杯”赛，用 51 单片机、角度传感器和蓝牙模块制作了一款智能魔方，可以在计算机中实时展示魔方的旋转情况。当时的开发环境极其简陋，只能参照芯片的官方开发文档编写非常简单的逻辑代码，每次要处理任何一点意外的抖动，都要加大量的 `if/else` 语句重新烧录。我们的程序最终堆砌了大量让人毫无头绪的“魔数”（magic number），却仍旧没有办法很好地适应真实情况，充其量只能“小心翼翼”地演示效果，完全无法作为真正的产品落地。这之后很多年，我对微控制器开发的认知始终都停留在这种近汇编式“控制流代码”的原始印象中。我在接触本书时才第一次了解到 Arduino 和 TensorFlow Lite for Microcontrollers，我不仅惊讶于微控制器开发已经变得这么友好，更惊讶于机器学习这种我们认为的 CPU 密集操作，依靠如此有限的算力，竟然能做这么多的事情！

TinyML 无疑大大拓展了机器学习和嵌入式应用的疆界。自此，机器学习不再囿于云端超级计算机，而是可以被隐藏于众多小到可以被忽视的电子零件中；嵌入式应用也不再局限于简单的信号处理，而是可以“看懂”“听懂”“感觉到”周围的世界。这意味着“智能”可以适应更恶劣的环境，提供更持久和稳定的服务，同时拥有更强的安全和隐私保障。我们甚至已经可以想象一个“所有的遮挡和封闭都不存在，一切都暴露在外，相当于三维世界中亿万倍的信息量”的新世界。

市面上有关机器学习的优秀资源有很多，但有关微控制器上的机器学习的书籍却寥寥可数。本书不仅填补了这方面的空白，而且无论在机器学习、微控制器应用开发还是工程设计等方面，都是非常优秀的书籍。我在翻译过程中学到了很多知识，不仅仅是“如何在微控制器上部署机器学习”，还了解到很多 TensorFlow Lite for Microcontrollers 框架的哲学，以及有关工程设计的考量。我已经迫不及待要把这本书推荐给你，相信你会和我一样享受阅读本书的过程，并且受益良多！

感谢 Pete Warden 和 Daniel Situnayake 撰写本书并在我们翻译的过程中提供了诸多帮助，感谢 Google 北京 TensorFlow 团队的王铁震、vivo AI 研究院的况辉帮助做技术审校，感谢机械工业出版社华章公司的刘锋编辑筹备和组织本书的出版工作。

我和卜杰都非常荣幸能参与本书中文版的翻译工作，由于能力和时间有限，翻译难免存在纰漏，我们为可能存在的翻译错误和词不达意提前致歉。感谢你阅读本书，也感谢你的谅解与包容。

祝你学有所成，学有所用！

魏兰

2020 年 5 月于北京

译者序二

我从小就对硬件智能充满了幻想。上小学的时候，有一天老师布置了一篇作文，让我们畅想未来世界的生活。我花了整个晚上思考未来我的家会是什么样子：会根据主人的需要自动调节水温的饮水机、自动打扫卫生的机器人、自动调节温度的空调等，以及一台安装在天花板上的中控计算机负责控制这些设备。在那个互联网甚至个人计算机都还没有普及的年代，这些幻想对普通人而言是那么遥远。然而二十多年后的今天再回首，我小时候的这些幻想早已变成普通人日常生活的一部分。每天晚上回到家，呼唤一声家里的语音助手帮自己开灯、调节一下空调的温度，抑或让扫地机器人打扫某个特定房间，一切都是那么自然。不过我当时没有想到的一点是，负责控制这些设备的“中控计算机”不在天花板上，而是在某个数据中心里，抑或是在云端。

如果要评选出过去十年间对计算机领域影响最大的十大技术革命，云计算和机器学习肯定都榜上有名。云计算彻底改变了人们分享与处理数据的方式，让信息的收集与流通更加便利。而机器学习则需要依赖大量的数据进行训练，以找出数据中共有的模式，并对新数据进行预测。云计算与机器学习的有机结合、优势互补，给很多行业带来了革命性的变化，比如电商平台根据用户的搜索和购买记录为用户更精准地推荐产品，很多内容平台（比如短视频应用）会根据用户的浏览记录推送用户更想看到的内容。同样，云计算+机器学习+物联网的组合，凭借云端强大的运算性能以及网络，通过对传感器收集到的各种数据进行分析并将分析结果分发到各个设备，让我小时候幻想的“未来生活”成为现实。这些变革极大地优化了信息流通的效率，使人们的生活更加便利，但也有很多潜在的风险。

首当其冲的就是用户隐私和信息安全。传统的机器学习推断需要用户将行为信息上传到云端统一处理，导致用户隐私在信息传输过程中被泄露的风险上升。并且，不恰当的服务端实现也会造成滥用用户隐私信息的情况，甚至会导致大规模的信息泄露事故。除此之外，还有一些弊端也是无法避免的，比如收集数据、上传数据、处理数据、返回结果的过程对网络条件有着严格的要求，可能会造成较大的延迟，让用户体验变得很糟糕。并且对于某些智能设备而言，还存在一旦断网就“变砖”的风险。

为了应对这些弊端和风险，业界有人提出了边缘计算（edge computing）的概念，这是一种分布式计算架构，将应用程序、数据资料与服务的运算由网络中心节点移至网络逻辑上的边缘节点来处理。这样做可以减少不必要的信息传输，从而降低信息在传递过程中被泄露的风险，并且减少对网络的依赖，降低延迟。而本书中提到的给嵌入式设备使用的微型机器学习（TinyML）技术则更进一步，将部分机器学习运算直接转移到节点设备上进行。一个常见的应用就是家中语音助手的唤醒词检测。该应用的原理是不断地监听环境中的声音，然后在本地持续地运行推断，这个过程是不会上传音频数据的。一旦检测到了所需的唤醒词，比如“小爱同学”“小度小度”，该应用便会唤醒后续的处理程序，将用户后续的音频上传到云端，以借用云端强大的计算能力进行更加精确的处理。想象一下，如果没有第一步在本地检测唤醒词，会有什么后果呢？用户的音频数据会被时刻上传到服务端进行分析，这会产生极其严重的隐私风险以及大量的网络流量消耗，从而使其实用价值大打折扣。

除了解决以上问题外，TinyML 还有更多的应用价值。由于经过了特定优化，运行 TinyML 模型的硬件可能只有几毫瓦（mW）的功率，这使得我们甚至可以在用纽扣电池驱动的设备上运行机器学习模型。头脑风暴一下，比如在心脏起搏器上运行机器学习模型，根据用户的心跳数据预测即将发生的危险并报警，抑或在需要长期安装在野外的野生动物监测设备上运行图像识别模型，用以监测某种野生动物的数量等。也许，TinyML 会出现一款“杀手级”应用，以意想不到的方式彻底颠覆我们的生活习惯。或者，更有可能的是，TinyML 会逐步渗透到我们生活的方方面面，以一种无声的方式潜移默化地提升我们的生活品质。而这个过程需要广大工程师的共同努力，希望我们翻译的这本书能够给读者提供一个解决问题的新视角。希望二十年后再回首，我们现在的一些“幻想”能够成为未来的生活日常。

感谢作者 Pete 和 Daniel 为我们提供了知识盛宴并在我们翻译的过程中提供支持，感谢魏兰组织这次翻译并让我参与其中，感谢家人和朋友对我的支持与鼓励。

由于翻译水平与时间有限，翻译过程中难免存在错误和词不达意的情况，在这里先向广大读者致歉。如有任何问题，可以通过本书最后译者介绍中提到的联系方式与我们联系。感谢理解！

祝大家身体健康，学有所成！

卜杰

2020 年 5 月于北京

目录

前言	1
第 1 章 简介	5
1.1 嵌入式设备	6
1.2 技术变迁	7
第 2 章 入门	8
2.1 本书目标读者	8
2.2 需要的硬件	8
2.3 需要的软件	10
2.4 我们希望你学到的东西	10
第 3 章 快速了解机器学习	12
3.1 什么是机器学习	13
3.2 深度学习的工作流程	14
3.3 小结	27
第 4 章 TinyML 之 “Hello World”：创建和训练模型	28
4.1 我们要创建什么	29
4.2 我们的机器学习工具链	30
4.3 创建我们的模型	32
4.4 训练我们的模型	43
4.5 为 TensorFlow Lite 转换模型	56
4.6 小结	61

第 5 章 TinyML 之 “Hello World”：创建应用程序	62
5.1 详解测试	63
5.2 项目文件结构	79
5.3 详解源文件	80
5.4 小结	87
第 6 章 TinyML 之 “Hello World”：部署到微控制器	88
6.1 什么是微控制器	88
6.2 Arduino	89
6.3 SparkFun Edge	98
6.4 ST Microelectronics STM32F746G Discovery 套件	109
6.5 小结	115
第 7 章 唤醒词检测：创建应用程序	116
7.1 我们要创建什么	117
7.2 应用架构	118
7.3 详解测试	121
7.4 监听唤醒词	139
7.5 部署到微控制器	143
7.6 小结	164
第 8 章 唤醒词检测：训练模型	165
8.1 训练我们的新模型	166
8.2 在我们的项目中使用模型	179
8.3 模型的工作方式	184
8.4 使用你自己的数据训练	194
8.5 小结	198
第 9 章 行人检测：创建应用程序	199
9.1 我们在创建什么	200
9.2 应用程序架构	201
9.3 详解测试	204
9.4 行人检测	210
9.5 部署到微处理器	213
9.6 小结	232

第 10 章 行人检测：训练模型.....	233
10.1 选择机器	233
10.2 配置 Google Cloud Platform 实例	233
10.3 训练框架选择	240
10.4 构建数据集	241
10.5 训练模型	241
10.6 TensorBoard	243
10.7 评估模型	245
10.8 将模型导出到 TensorFlow Lite	245
10.9 训练其他类别	247
10.10 理解架构	248
10.11 小结	248
第 11 章 魔杖：创建应用程序.....	250
11.1 我们要创建什么	252
11.2 应用程序架构	254
11.3 详解测试	255
11.4 检测手势	264
11.5 部署到微处理器	268
11.6 小结	293
第 12 章 魔杖：训练模型.....	294
12.1 训练模型	295
12.2 模型是如何工作的	303
12.3 训练你自己的数据	311
12.4 小结	315
第 13 章 TensorFlow Lite for Microcontrollers.....	317
13.1 什么是 TensorFlow Lite for Microcontrollers	317
13.2 编译系统	323
13.3 支持一个新的硬件平台	331
13.4 支持一个新的 IDE 或新的编译系统	336
13.5 在项目和代码库之间整合代码更改	337
13.6 回馈开源	338
13.7 支持新的硬件加速器	339

13.8 理解文件格式	340
13.9 将 TensorFlow Lite 移动平台算子移植到 Micro	347
13.10 小结	350
第 14 章 设计你自己的 TinyML 应用程序	351
14.1 设计过程	351
14.2 你需要微控制器还是更大的设备	351
14.3 了解可行性	352
14.4 站在巨人的肩膀上	353
14.5 找一些相似的模型训练	353
14.6 查看数据	354
14.7 绿野仙踪	355
14.8 先可以在桌面系统中运行	356
第 15 章 优化延迟	357
15.1 首先确保你要优化的部分很重要	357
15.2 更换硬件	358
15.3 改进模型	358
15.4 量化	360
15.5 产品设计	361
15.6 优化代码	362
15.7 优化算子	363
15.8 回馈开源	368
15.9 小结	368
第 16 章 优化功耗	369
16.1 开发直觉	369
16.2 测量实际功耗	372
16.3 估算模型的功耗	373
16.4 降低功耗	373
16.5 小结	375
第 17 章 优化模型和二进制文件大小	376
17.1 了解系统限制	376
17.2 估算内存使用率	376
17.3 关于不同问题的模型准确率和规模的大致数字	379

17.4 模型选择	380
17.5 减小可执行文件的大小	380
17.6 真正的微型模型	386
17.7 小结	386
第 18 章 调试	387
18.1 训练与部署之间准确率的损失	387
18.2 数值差异	389
18.3 神秘的崩溃与挂起	391
18.4 小结	394
第 19 章 将模型从 TensorFlow 移植到 TensorFlow Lite ...	395
19.1 了解需要什么算子	395
19.2 查看 Tensorflow Lite 中支持的算子	396
19.3 将预处理和后处理移至应用程序代码	396
19.4 按需自己实现算子	397
19.5 优化算子	397
19.6 小结	398
第 20 章 隐私、安全和部署	399
20.1 隐私	399
20.2 安全	401
20.3 部署	403
20.4 小结	404
第 21 章 了解更多	405
21.1 TinyML 基金会	405
21.2 SIG Micro	405
21.3 TensorFlow 网站	406
21.4 其他框架	406
21.5 Twitter	406
21.6 TinyML 的朋友们	406
21.7 小结	407
附录 A 使用和生成 Arduino 库 ZIP 文件	409
附录 B 在 Arduino 上捕获音频	411

前言

自我记事时起，各种电子产品就一直深深吸引着我，赋予我丰富的想象力。人类学会了从地上挖掘矿石，用神秘的方式提炼这些矿石，然后生产出一系列令人眼花缭乱的微型部件。通过各种神奇的规则，人类将这些微小的部件组合在一起，赋予它们无限的生命力。

在我八岁的时候，电池、开关和白炽灯泡对于还是孩子的我来说已经相当迷人，更不用说家用计算机中的处理器了。随着时间的流逝，我对电子和软件的工作原理有了一定的理解。将简单的元素组合成一个系统，就能创造出更加微妙而复杂的事物，这一点始终令我印象深刻，而深度学习无疑将这一点推向了新的高度。

本书包含的示例之一是深度学习网络，从某种意义上说，它是一种学习和理解如何去“看”的网络。深度学习网络由成千上万个虚拟的“神经元”组成，每个神经元都遵循一些简单的规则并输出一个数字。任何一个神经元本身的能力都非常有限，但是将成千上万个神经元组合在一起，利用人类的知识进行启发式训练，它们最终也可以理解我们这个复杂的世界。

这个想法有些神奇——在由沙子、金属和塑料制成的微型计算机上运行的简单算法可以体现人类的认知理解。这就是 TinyML 的本质。TinyML 是 Pete 创造的一个术语，我们将在第 1 章中介绍它。在本书中，你会找到创建这些东西所需的工具。

衷心感谢你成为我们的读者。TinyML 是一个复杂纷繁的主题，但是我们会努力解释所有你需要了解的概念，并尽可能地使它们通俗易懂。希望你会喜欢我们的文字，也非常期待看到你能有所创造！

—— Daniel Situnayake

本书排版约定

本书采用以下排版约定：

楷体

表示新术语。

斜体 (*Italic*)

表示 URL、电子邮件地址、文件名和文件扩展名。

等宽字体 (**Constant width**)

用于代码清单，以及在段落中引用的代码内容，例如变量或函数名、数据库、数据类型、环境变量、声明和关键字等。

等宽粗体 (**Constant width bold**)

表示用户应直接输入的命令行或其他文本。

等宽斜体 (*Constant width italic*)

表示应该被替换为用户提供的值或由上下文确定的值。



这个图案表示提示或建议。



这个图案表示注释。



这个图案表示警告或注意事项。

示例代码

可以从 <https://tinymlbook.com/supplemental> 下载补充材料（示例代码、练习等）。

如果你在使用示例代码时遇到任何问题或有疑问，请发送电子邮件至 bookquestions@oreilly.com。

本书旨在帮助你完成工作。通常来说，如果本书提供了示例代码，你就可以在程序和文档中使用它。除非你要大量复制代码的重要部分，否则无须与我们联系以获取许

可。例如，使用本书中提供的若干代码片段来编写程序不需要获得许可，而销售或发布 O'Reilly 书中的示例则需要获得许可。引用本书以及引用示例代码来回答问题不需要获得许可，将本书中的大量示例代码合并到产品文档中则需要获得许可。

我们鼓励（但一般不要求）注明出处。出处通常包含书名、作者、出版社和 ISBN，例如：*TinyML*，作者 Peny Warden 和 Daniel Situnayake，由 O'Reilly 出版，Pete Warden 和 Daniel Situnayake 版权所有，书号 978-1-492-05204-3。

如果你认为你对代码示例的使用超出了合理使用范围或上述给出的许可范围，请随时通过 permissions@oreilly.com 与我们联系。

O'Reilly 在线学习平台 (O'Reilly Online Learning)

O'REILLY[®] 40 多年来，O'Reilly Media 一直致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的由专家和革新者组成的庞大网络，他们通过书籍、文章、会议和我们的在线学习平台分享他们的知识和经验。O'Reilly 的在线学习平台可以让你按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O'Reilly 和 200 多家其他出版社提供的大量文本和视频资源。更多相关信息请访问 <http://oreilly.com>。

联系我们

对于本书，如果有任何意见或疑问，请按照以下地址联系本书出版商。

美国：

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

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室 (100035)
奥莱利技术咨询（北京）有限公司

本书配套网站 <https://oreil.ly/tiny> 上列出了勘误表、示例以及其他信息。

要询问技术问题或对本书提出建议，请发送电子邮件至 tinyml-book@googlegroups.com。

关于书籍、课程、会议和新闻的更多信息，请访问我们的网站：

<http://www.oreilly.com>
<http://www.oreilly.com.cn>

我们在 Facebook 上的地址：<http://facebook.com/oreilly>

我们在 Twitter 上的地址：<http://twitter.com/oreillymedia>

我们在 YouTube 上的地址：<http://www.youtube.com/oreillymedia>

致谢

特别感谢 Nicole Tache 出色的编辑、Jennifer Wang 令人激动的魔杖示例，以及 Neil Tan 在 uTensor 库中所做的开创性的嵌入式机器学习工作。没有 Rajat Monga 和 Sarah Sirajuddin 在专业方面的支持，我们不可能完成本书。还要感谢我们的伙伴 Joanne Ladolcetta 和 Lauren Ward 极具耐心的合作。

本书是硬件、软件和研究领域（尤其是 TensorFlow 团队）数百名工程师工作的结晶。虽然此处无法提到所有人（向每一个没有提及的人致歉），但我们还是想尽可能多地提名感谢：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、Tiezen Wang、Paul Whatmough、Tom White、Edd Wilder-James 和 Wei Xiao。

简介

本书旨在向具有命令行终端和代码编辑器基本使用经验的开发人员展示，如何在嵌入式设备上搭建和运行自己的机器学习（Machine Learning, ML）项目。

2014年，在我刚加入Google时，我惊奇地发现Google有许多我之前不知道的内部项目，其中最令人兴奋的是OK Google团队正在做的项目：他们运行的神经网络的大小只有14KB！这些网络运行在大多数Android手机的数字信号处理器（Digital Signal Processor, DSP）中，持续地监听“OK Google”唤醒词。因为这些DSP只有几十KB的RAM和闪存，所以神经网络必须非常小。OK Google团队必须使用DSP来完成这项工作，因为主CPU可能会因为手机进入节能模式而被关闭。然而这些专用芯片只会消耗几毫瓦（mW）的功率，可以一直保持运行。

在深度学习图像领域，我从未见过如此小的网络，也没有想到过可以使用功耗如此之低的芯片来运行神经网络模型。当我致力于让TensorFlow和后来的TensorFlow Lite在Android和iOS设备上运行时，我依旧着迷于使用更简单的芯片运行机器学习模型的可能性。我逐渐了解到更多其他的开创性项目，比如与音频相关的项目（例如Pixel上的Music IQ）、用于预测性维修的项目（例如PsiKick）甚至视觉领域的项目（例如高通的Glance摄像头模块）。

我清楚地意识到，一种全新的产品类别出现了，其主要特点是使用机器学习来理解嘈杂的传感器数据，可以使用电池或能源收集装置运行几年，而且成本只有一两美元。我反复听到的一个术语是“即剥即贴传感器”（peel-and-stick sensor），它指的是不需要更换电池并且可以在任何环境、任何地方使用的可以被遗忘的设备。要使这些产品成为现实，就必须将传感器设备本身的数据在传感器上本地转换为可操作的信息，因为事实证明，任何地方的流传输的功耗成本天生就太高而无法实用。

这就是TinyML^{译注1}这个概念的由来。通过同工业界和学术界同事的长时间讨论，我们

译注1：TinyML，即微型机器学习（Tiny Machine Learning），本书统一不翻译。

达成了一个粗略的共识：如果能以低于 1 毫瓦的功耗成本运行神经网络模型，那么将使得许多全新的应用成为可能。“1 毫瓦”似乎是一个有点随意的数字，但如果把它转换为具体的术语，你会发现它意味着一个使用纽扣电池的设备可以具有一年的使用寿命。这样一来就产生了一种产品——它足够小，可以适应任何环境，并且能够在无须人工干预的情况下运行相当长的时间。



我将直接使用一些技术术语来讨论本书将要介绍的内容，如果某些术语对你来说比较陌生，请不要担心。我们会在第一次使用时解释它们的含义。

在这一点上，你可能想到了树莓派（Raspberry Pi）或 NVIDIA 的 Jetson 等平台。这些都是很棒的设备和平台，我自己也经常使用它们，但是即使是最小的 Pi 也类似于手机的主 CPU，因此会消耗数百毫瓦的电量。要让一个 Pi 连续运行几天，就需要使用类似智能手机电池的电源，因此也就很难建立真正不受限制的体验。NVIDIA 的 Jetson 基于强大的 GPU，我们已经看到它在全速运行时会消耗高达 12 瓦的功率，因此，如果没有大型外部电源，它使用起来会非常困难。在汽车或机器人技术应用中，外部电源通常不是问题，因为其机械部分本身就需要大功率的电源，但我感兴趣的产品通常需要在没有可连接电源的情况下运行，这确实使得 Jetson 等平台很难应用。当然，令人高兴的是，在使用这些平台时，没有资源约束意味着我们可以使用 TensorFlow、TensorFlow Lite 和 NVIDIA TensorRT 之类的框架，因为这些平台通常基于支持 Linux 的 Arm Cortex-A CPU，而且有数百兆的内存。本书不会重点介绍如何在这些平台上运行我们的项目，但是，如果你有兴趣，有很多资源和文档可供参考：例如 TensorFlow Lite 的官方文档。

我关心的另一个产品特性是成本。对于创客来说，最便宜的树莓派 Zero 价格为 5 美元，但要以这样的价格大量购买这类芯片是非常困难的。零售购买通常会受到数量的限制，虽然工业购买的价格并不透明，但很明显，5 美元绝对不是寻常价格。相比之下，最便宜的 32 位微控制器（microcontroller）的价格远低于 1 美元。如此低廉的价格可以使制造商用软件定义的替代产品来代替传统的模拟或机电控制电路，从玩具到洗衣机，无所不包。我希望我们可以在无须对现有设计进行大量改动的前提下，向这些设备中无处不在的微控制器引入人工智能作为软件更新。另外，我们应该有可能在一定的成本和可用资金范围内，在建筑物或野生动植物保护区之类的环境中大量部署智能传感器。

1.1 嵌入式设备

TinyML 的定义是功耗成本低于 1 毫瓦，这意味着我们需要为我们的硬件平台寻找合适的嵌入式设备。几年之前，我本人对它们也都不熟悉，对我来说，嵌入式设备被笼罩在一层神秘的面纱之下。传统意义上，嵌入式设备都只有 8 位 CPU，而且使用晦涩

难懂的专业工具链，因此使用任何一个嵌入式设备似乎都非常吓人。Arduino 在开发方面向前迈出了一大步，它引入了用户友好的集成开发环境（Integrated Development Environment, IDE）和标准化硬件。从那以后，32 位 CPU 已成为标准配置，这在很大程度上归功于 Arm 的 Cortex-M 系列芯片。直到几年前，当我开始进行一些机器学习试验的原型设计时，我才惊讶于嵌入式领域的开发过程竟已变得如此简单。

但是，嵌入式设备仍然存在一些严格的资源限制。它们通常只有几百 KB 的 RAM，有时甚至还要小得多，并且具有类似数量级的闪存，用于持久编程和数据存储。仅仅几十兆赫兹的时钟速度也非常常见。它们肯定不会拥有完整的 Linux 系统（因为这需要一个内存控制器和至少一兆字节的 RAM），并且即使有操作系统，它也很可能无法提供全部或任何你所期望的 POSIX 或者 C 标准库函数。许多嵌入式系统避免使用诸如 `new` 或 `malloc()` 之类的动态内存分配函数，因为这些系统被设计为可靠且可长时间运行，如果有一个可以被分割的堆，这一点就很难确保。你可能还会发现使用调试器或其他任何你所熟悉的桌面开发工具都是非常困难的，因为用来访问芯片的接口通常都不是通用的接口。

不过，当我开始学习嵌入式开发后，还有一些小惊喜。当一个系统不会有来自其他程序的中断程序时，建立心智模型（mental model）就会变得非常简单，而且没有分支预测或指令流水线的处理器，其直截了当的特性也使得在其上进行手动汇编优化要比在复杂的 CPU 上容易得多。当我看到通过手机控制在微控制器上点亮的 LED 时，我也感到一种单纯的乐趣，因为我知道它每秒在运行着数百万条指令以了解周围的世界。

1.2 技术变迁

直到最近，我们才能够在微控制器上完整地运行机器学习算法，这个领域还很年轻，这也意味着硬件、软件和学术研究都在快速地变化着。本书是对 2019 年世界的快照，对于这个飞速发展的领域来说，这也意味着有些内容在我们写完最后一章前就已经过时了。我们试图确保我们所依赖的硬件平台是长期可用的，但是不可避免的是，设备会持续不断地改进和发展。我们使用的 TensorFlow Lite 软件框架具有稳定的 API，将持续地支持本书中提供的示例。但同时，我们也提供了所有示例代码和文档的最新版本的 Web 链接。例如，你可以期望参考应用程序涵盖的用例比本书中添加到 TensorFlow 代码仓库中的用例还要多。我们还致力于讲解诸如调试、模型创建以及对深度学习原理的理解等，即使你使用的基础架构发生了变化，这些知识也将非常有用。

我们希望本书能为你提供开发嵌入式机器学习产品所需的基础，以帮助你解决任何你所关心的现实世界中的问题。希望我们能够帮助你从零开始搭建一些新的、令人兴奋的应用程序，我敢肯定，未来几年内这个领域一定会出现很多激动人心的应用。

第2章

入门

本章我们将介绍开始在低功耗设备上创建和修改机器学习应用程序之前需了解的一些知识。本书使用的所有软件都是免费的，而硬件开发套件的价格不到 30 美元，因此最大的挑战很可能是对开发环境的不熟悉。为了解决这个问题，在本章中，我们推荐了一套我们发现的协作良好的工具组合。

2.1 本书目标读者

要创建 TinyML 项目，你需要了解一定的机器学习和嵌入式软件开发方面的知识。这两种技能都不是普通的技能，很少有人同时是这两方面的专家，因此本书将假设你没有任何这两方面的背景知识。本书的唯一要求是你对在终端（或 Windows 上的命令提示符）中运行命令有一定的了解，并且能够将程序源文件加载到编辑器中，进行修改并保存。虽然这听起来没什么难度，我们仍将一步一步地介绍我们讨论的所有内容，就像一本精心编写的菜谱一样，我们会提供各种情况下的截屏（包括在线截屏），因此我们希望本书能面向尽可能广泛的读者。

我们将向你展示一些嵌入式设备上机器学习的实际应用，例如简单的语音识别、使用运动传感器检测手势以及使用摄像头传感器检测行人。我们希望能让你自己创建这些程序，并扩展它们以解决你关心的实际问题。例如，你可以修改语音识别程序以检测狗吠声而不是人声，或者修改摄像头的项目以检测狗而不是检测行人，我们也在书中提供了有关如何自己处理这些改动的想法。我们的目标是为你提供所需的所有工具，帮助你创建你所关心的令人兴奋的应用程序。

2.2 需要的硬件

你需要一台带有 USB 接口的笔记本电脑或台式机。这将是你的主要编程环境，你可以在计算机上编辑并编译要在嵌入式设备上运行的程序。你将使用 USB 接口和专用的适

配器将计算机与嵌入式设备连接，具体的接口和适配器取决于你用于开发的硬件。主计算机可以运行 Windows、Linux 或 macOS。对于大多数示例，我们使用 Google Colab 在云端训练我们的机器学习模型，因此你不必担心是否还需要拥有一台特殊配置的计算机。

你还将需要一个嵌入式开发板来测试你的程序。为了做一些更有趣的事情，你还会需要一个麦克风、一些加速度计或者一个摄像头，还需要一个电池，并且你可能会希望这些东西足够小，以构建一个现实场景中的原型项目。当我们刚开始编写本书时，这些东西很难找到，因此我们与芯片制造商 Ambiq 和创客零售商 SparkFun 合作，生产了价格为 15 美元的 SparkFun Edge 开发板。本书中的所有示例均适用于这款开发板。

我们还提供了有关如何使用 Arduino 和 Mbed 开发环境运行项目的说明。我们推荐使用 Arduino Nano 33 BLE Sense 开发板和基于 Mbed 的 STM32F746G Discovery 套件开发板，不过这只是推荐而已，如果你能按所需格式捕获传感器数据，则所有项目都可适用于其他设备。表 2-1 展示了每个项目所在的章以及使用的设备。

表 2-1：每个项目所使用的硬件

项目名称	章	SparkFun Edge	Arduino Nano 33 BLE Sense	STM32F746G Discovery 套件
Hello World	第 5 章	使用	使用	使用
唤醒词检测	第 7 章	使用	使用	使用
行人检测	第 9 章	使用	使用	未使用
魔杖	第 11 章	使用	使用	未使用

如果我想要使用的开发板未在此处列出怎么办

本书中所有项目的源代码都托管在 GitHub 上，我们会不断对其进行更新以支持其他设备。本书的每一章都提供了项目 *README.md* 的链接，其中列出了所有支持的设备，并提供了有关如何部署到这些设备上的说明，因此你可以查看以了解是否已经支持了你想要使用的设备。

如果你有一些嵌入式开发经验，那么即使你的设备没有被列出，你应该也可以轻松地将示例移植到你的设备上。

除了行人检测项目需要摄像头模块以外，这些项目都不需要任何其他电子元件。如果你使用的是 Arduino，你将需要 Arducam Mini 2MP Plus。如果你使用的是 SparkFun Edge，则需要 SparkFun 的 Himax HM01B0 扩展板。

2.3 需要的软件

本书中的所有项目均基于 TensorFlow Lite for Microcontrollers 框架。它是 TensorFlow Lite 框架的一个变体，设计用于在只有几十 KB 可用内存的嵌入式设备上运行。本书中所有的项目均作为示例包含在 TensorFlow Lite for Microcontrollers 库中。TensorFlow Lite 是开源的，因此你可以在 GitHub 上找到它。



由于本书中的代码示例是一个活跃的开源项目的一部分，因此随着我们不断地优化、修复错误以及支持新的设备，代码也会不断地变化和发展。你可能会发现本书中印刷的代码与 TensorFlow 代码仓库中的最新代码之间存在一些差异。也就是说，尽管代码可能会随着时间的推移而有所变化，但是你将在本书中学到的基本原理是不变的。

你需要某种编辑器来检查和修改代码。如果你不确定应该使用哪个软件，那么免费的微软 VS Code 应用程序会是一个很好的起点。它可以在 macOS、Linux 和 Windows 上运行，并且有许多非常便利的特性，诸如语法高亮和自动补全等。当然，如果你已经有了自己喜欢的编辑器，请随意。对于本书的任何项目，所需的代码修改其实并不多。

你还需要在某个地方输入命令。在 macOS 和 Linux 上，这就是所谓的终端（terminal），你可以在“应用”（Applications）文件夹中找到它。在 Windows 上，它被称为“命令提示符”（Command Prompt），你可以在“开始”（Start）菜单中找到它。

你还需要一些额外的软件与嵌入式开发板进行通信，但是具体用什么软件取决于你所使用的设备。如果你使用的是 SparkFun Edge 开发板或 Mbed 设备，则需要安装 Python 以编译并运行某些脚本，然后可以在 Linux 或 macOS 上使用 GNU Screen，或者在 Windows 上使用 Tera Term 来访问调试日志控制台，显示从嵌入式设备输出的文本。如果你使用的是 Arduino 开发板，则所需的所有内容都将作为 IDE 的一部分安装，因此你只需要下载 Arduino 主软件包即可。

2.4 我们希望你学到的东西

本书的主要目的是帮助在这个新兴领域创造更多的应用程序。TinyML 目前还没有一款“杀手级应用程序”，也许永远都不会有。但是从历史经验中我们知道，世界上的很多问题都可以使用 TinyML 提供的工具箱来解决。我们希望能帮助你熟悉可能的解决方案。我们更希望吸收来自农业、太空探索、医药、消费品以及任何其他领域的专家，让他们了解如何使用这些技术解决问题（或者至少可以交流使用这些技术可以解决的专业领域问题）。

考虑到这一点，我们希望当你读完本书的时候，能够对当前在嵌入式系统上使用机器学习的可能性有一个很好的了解，并且对未来几年内的可行性有一些自己的想法。我们希望你能够使用时间序列数据（例如音频或加速度传感器的输入）以及来自低能耗硬件的视觉数据，来构建和修改一些实际的项目。我们希望你能对整个系统有足够的了解，至少能够有意义地参与或者与领域专家进行有关新产品的设计的讨论，并希望你能够自己制作早期版本的原型。

由于我们希望看到完整产品的出现，因此我们会从整个系统的角度来思考我们正在讨论的所有内容。通常，硬件供应商会把重点放在要出售的特定组件的功耗上，而不会考虑其他必要组件会如何增加所需的功率。例如，如果你有一个仅消耗 1 毫瓦的微控制器，但唯一与之配合使用的摄像头传感器需要 10 毫瓦的功耗，那么任何基于这款摄像头的视觉产品都将无法有效发挥处理器的低功耗带来的优势。这意味着我们不会深入研究不同领域的基础工作，相反，我们关注的是使用和修改所有相关组件所需的知识。

例如，当你在 TensorFlow 中训练模型时，我们不会特意关注模型背后的细节，比如梯度和反向传播的工作原理；取而代之的是，我们将向你展示如何从零开始训练和编译模型、你可能会遇到哪些常见的错误、你应该如何处理它们，以及如何自定义创建模型的所有过程，来使用新数据集解决你自己的问题。

第3章

快速了解机器学习

在技术方面，很少有领域能像机器学习和人工智能（Artificial Intelligence, AI）一样充满神秘感。即使你是一位在其他领域经验丰富的工程师，机器学习对你来说也可能是一门需要有大量假定知识背景的繁重课题。在开始了解机器学习相关知识时，很多开发者都会为理解学术论文、晦涩的 Python 库以及高等数学知识而感到灰心。即使知道从哪里开始学习，机器学习仍让人心生畏惧。

实际上，机器学习很容易理解，任何使用文本编辑器的人都可以使用机器学习。在学习了一些核心思想后，就可以轻松地在自己的项目中使用它。隐藏在所有神秘感背后的其实是一套方便的工具，能够用于解决各种类型的问题。机器学习有时候看起来像魔法一样，但其实它只是代码而已，你并不需要获得博士学位就可以使用它。

本书主要介绍在微型设备上使用机器学习。在本章的其余部分，你将学习入门所需的所有机器学习相关知识。我们将介绍一些基本概念，探索一些工具的用法，并训练一个简单的机器学习模型。由于我们的重点是微型硬件，因此我们不会花太多时间在深度学习背后的理论上，也不会花很多时间在使之起作用的数学原理上。在后面的章节中，我们将更深入地研究工具，以及如何为嵌入式设备优化模型。在本章结束时，你将熟悉关键的术语，了解一般的工作流程，并知道可以去哪里了解更多信息。

在本章中，我们将介绍以下几点：

- 什么是机器学习。
- 它能解决的问题类型。
- 关键术语和思想。
- 使用深度学习——一种十分流行的机器学习方法——来解决问题的流程。



由于目前已经有很多书籍和课程介绍了深度学习背后的科学原理，因此在这里将不会多作解释。当然，这是一个十分有趣的话题，我们鼓励你自己去探索！我们在 12.4 节列出了一些我们十分喜欢的资源。但是请记住，你并不需要了解所有的理论就可以开始搭建有用的程序。

3.1 什么是机器学习

想象一下你拥有一台生产小部件的机器，它时不时会出现故障而且维修费用十分昂贵。也许，如果你能在操作过程中收集有关机器的数据，那么有可能可以通过这些数据预测机器会在何时发生故障，并在发生损坏之前就停止操作。举个例子，你可以记录机器的生产率、温度以及振动水平，这些因素的某种组合可能预示着一个故障即将发生。但是你如何才能找到这种组合呢？

这是可以用机器学习来解决的一类问题的一个例子。从根本上说，机器学习是一种使用计算机基于过去的观察结果来预测事件的技术。我们可以收集有关工厂机器性能的数据，然后创建一个计算机程序来分析这些数据，并用它预测未来的状态。

创建机器学习程序的过程与编写普通代码的过程不太一样。在传统的软件编写中，程序员需要设计一个算法，接收输入，应用各种规则最终返回一个输出。这个算法的内部操作是由程序员规划并通过编写一行行明确的代码实现的。为了预测工厂机器的故障，程序员需要了解数据中的哪些测量值会指示问题，并编写能够检查这种指示的代码。

这种方法适用于许多问题。举个例子，我们知道水在海平面的沸点是 100°C，所以编写一个基于当前温度和海拔来预测水是否沸腾的程序非常容易。但是在许多情况下，我们可能很难知道预测给定状态所需因素的精确组合。继续工厂机器的例子，生产率、温度和振动水平可能会存在各种不同的组合，其中某些组合可能可以指示机器将出现故障，但是我们却难以从数据中直接找出这种组合。

要创建一个机器学习程序，程序员需要将数据输入一种特殊的算法，并让算法来发现其中的规则。这意味着作为程序员，我们可以创建基于复杂数据进行预测的程序，而不必自己去理解数据背后所有的复杂性。机器学习算法能够根据我们提供的数据，通过称为训练 (training) 的过程来建立系统的模型 (model)。该模型是一种计算机程序，我们通过模型来处理数据并进行预测，这个过程称为推断 (inference)。

机器学习有许多不同的方法，其中最流行的是深度学习 (deep learning)，这是一种基于人类大脑工作机制的简化式概念的算法。在深度学习中，一个由模拟神经元组成的网络 (network)（用数字数组表示）会被训练来模拟各种输入和输出之间的关系。不同的架构

(architecture) 或模拟神经元的排列适用于不同类型的任务。例如，某些架构擅长从图像数据中提取含义，而其他一些架构更适合预测序列中的下一个值。

本书中的例子将侧重于深度学习，因为这是一种灵活且强大的工具，非常适合用于解决微控制器的相关问题。可能令人惊讶的是，即使在内存和处理能力有限的设备上，深度学习也能发挥作用。在阅读本书的过程中，你将学习如何创建深度学习模型，即使受到微型设备的限制，这些模型仍能做到一些令人十分惊奇的事情。

下一节我们将介绍创建和使用深度学习模型的基本工作流程。

3.2 深度学习的工作流程

在上一节中，我们概述了一个使用深度学习来预测工厂机器何时可能发生故障的场景。在本节，我们将介绍实现这种功能所需要的工作。

这个流程将涉及以下任务：

1. 选择一个目标。
2. 收集数据集。
3. 设计一个模型架构。
4. 训练模型。
5. 转换模型。
6. 运行推断。
7. 评估和故障排除。

接下来我们将逐一介绍。

3.2.1 选择一个目标

在设计各种类型的算法时，首先确定你希望它执行什么操作至关重要，对于机器学习来说也是如此。你需要先决定想要预测什么，这样才能决定要收集什么数据，以及使用什么模型架构。

在我们的示例中，我们想要预测工厂机器是否即将出现故障。我们可以将其表示为一种分类 (classification) 问题。分类是一种机器学习任务，它接收一组输入数据，并返回该数据属于一组已知类别 (class) 中每个类别对应的概率。在我们的示例中，可以有两种类别：“正常” (normal) 意味着机器正在正常运行；“异常” (abnormal) 意味着机器表现出一些即将出现故障的征兆。

这意味着我们的目标是创建一个将输入数据分类为“正常”或“异常”的模型。

3.2.2 收集数据集

我们的工厂中可能有很多可用数据，从机器的工作温度到餐厅某一天提供的食物类型。有了刚刚确立的目标后，我们就可以开始着手确定需要什么样的数据了。

选择数据

深度学习模型可以学习忽略嘈杂和不相关的数据。也就是说，我们最好仅使用与解决问题有关的信息来训练模型。由于餐厅今天提供的食物不太可能会影响机器的运行，因此我们可以将其从数据集中排除。否则，我们的模型将需要学习否定这种不相关的输入，并且很容易受到学习虚假关联的影响——也许我们的机器总是碰巧在餐厅供应比萨的日子出现故障。

在决定是否包含数据时，你应该始终尝试将你的领域专业知识与试验结合起来。你还可以使用统计技术来尝试确定哪些数据是重要的。如果你仍然不确定是否需要包含某种数据源，那么你可以同时训练两个模型，然后看看哪个模型的效果更好！

假设我们已经确定了最有希望的数据是生产率（rate of production）、温度（temperature）和振动（vibration）。下一个步骤就是收集一些数据以便训练一个模型。



非常重要的一点是，当你进行预测时，被用来训练模型的数据要持续可用。举个例子，由于我们已经选择了使用温度读数来训练模型，所以当运行推断时，我们需要提供从相同的物理位置获取的温度读数，这是因为模型学会理解的是如何使用这种温度数据作为输入来预测它的输出。如果我们最初是根据机器内部的温度数据训练模型，那么使用当前室温去运行模型将不可能得出我们想要的结果。

收集数据

我们很难确切地知道训练一个有效的模型到底需要多少数据，因为这取决于许多因素，例如变量之间关系的复杂性、噪声量以及区分类别的难易程度等。然而，有一条经验法则几乎总是正确的：数据越多越好！

你需要收集的数据应该代表了系统中所有可能发生的条件和事件。如果我们的机器可能以几种不同的方式出现故障，那我们应该确保捕获与每种类型的故障相关的数据。如果一个变量会随着时间自然变化，那么收集其整个范围内的数据非常重要。例如，如果在温暖的日子里机器的温度会升高，那么你所收集的数据应该确保包括机器分别在冬季和夏季产生的温度数据。这种多样性将会帮助你的模型学习所有可能的场景，而不仅仅是

少数的几种情况。

我们收集的关于工厂的数据很可能被记录为一个时间序列（time series）的集合，代表着一系列定期收集的读数。例如，我们可能会记录机器每分钟的温度、每小时的生产率以及每秒的振动水平。当收集这些数据后，我们需要将这些时间序列转换为适合我们模型的形式。

标记数据

除了收集数据之外，我们还需要确定哪些数据代表“正常”的操作，哪些数据代表“异常”的操作。我们将需要在训练过程中提供这些信息，以便于模型能够学习如何对输入进行分类。这种将数据与类别相关联的过程称为标记（labeling）。而“正常”和“异常”这两种类别就是我们的标签（label）。



在训练过程中为算法提供标明含义的数据，我们将这种类型的训练称为监督学习（supervised learning）。由此产生的分类模型将能够处理输入的数据并预测它可能属于哪个类别。

为了标记收集的时间序列数据，我们需要记录机器工作的时间段和故障的时间段。我们可以假设，机器故障前的一段时间通常代表异常操作。但是，由于不一定能从表面上观察数据来发现异常操作，因此我们可能需要进行一些试验来获得正确的结果。

在决定了如何标记数据之后，我们可以生成一个包含标签的时间序列并将其添加进我们的数据集。

我们的最终数据集

表 3-1 列出了我们据此在工作流程中收集的数据源。

表 3-1：数据源

数据源	记录间隔	样本读数
生产率	每 2 分钟一次	100 单位
温度	每分钟一次	30°C
振动（相对于正常情况的百分比）	每 10 秒一次	23%
标签（“正常”或者“异常”）	每 10 秒一次	正常

表 3-1 展示了每个数据源的记录间隔，例如温度是每分钟记录一次。我们还生成了一个包含数据标签的时间序列。标签记录间隔是每 10 秒一次，这与其他所有时间序列中的最小间隔相同。这意味着我们可以轻松地确定数据中每个数据点的标签。

现在我们已经收集了需要的数据，是时候用它来设计和训练一个模型了。

3.2.3 设计一个模型架构

深度学习模型的架构类型有很多，它们被用于解决各种各样的问题。当训练一个模型时，你可以选择设计一个你自己的架构，也可以选择基于研究人员已开发的现有架构。对于许多常见的问题，你可以在互联网找到免费的已经训练过的模型。

在本书中，我们将向你介绍几种不同的模型架构，但是请了解，除了此处介绍的内容之外，模型架构设计还存在着丰富的可能性。设计一个模型既是一门艺术又是一门科学，而模型架构是机器学习研究的主要领域，每天都有新的架构被发明出来。

在决定架构时，你需要考虑要解决的问题类型、可以访问的数据类型以及在将数据输入模型之前转换数据的方式，我们稍后将讨论如何转换数据。事实上，由于最有效的架构会因你使用的数据类型而异，所以数据和模型的架构是紧密联系在一起的。尽管我们将在不同的章节介绍它们，但是它们始终需要被放在一起考虑。

你还需要考虑即将运行模型的设备约束，因为微控制器通常有着有限的内存和较慢的处理器，而越大的模型需要越多的内存和时间去运行。模型的大小取决于它包含的神经元数量以及这些神经元之间的连接方式。此外，由于某些设备配备了硬件加速功能，可以加速某些类型的模型架构的执行，因此你可能需要根据所考虑设备的优势来定制模型。

在我们的案例中，我们可能会先训练一个带有几层神经元的简单模型，然后以迭代的方式完善架构，直到获得有用的结果。你将在本书后面的内容中看到如何执行此操作。

深度学习模型接受输入并以张量 (tensor) 的形式生成输出。就本书而言^{注1}，张量本质上是一个可以包含数字或其他张量的列表，你可以认为它类似于数组。我们假设的简单模型将以张量作为输入。接下来将介绍如何将数据转换为张量。

维度

张量的结构被称为张量的形状 (shape)，它可以有多个维度 (dimension)。我们将在整本书中都用到张量，因此这里有一些有用的术语：

向量 (vector)

向量是一个数字列表，类似于数组。这是我们给只有一个维度的张量（一维张量）的命名。以下是形状为 (5,) 的向量，因为它在一个维度中包含 5 个数字：

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

注 1：张量这个词在这里的定义不同于其在数学或物理中的定义，但是它已经成为数据科学中的规范。

矩阵 (*matrix*)

矩阵是一个二维张量，类似于二维数组。以下是一个形状为 $(3, 3)$ 的矩阵，因为其包含了 3 个有着 3 个数字的向量：

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

高维张量 (*higher-dimensional tensor*)

任何具有两个以上维度的形状都称为张量。以下是一个形状为 $(2, 3, 3)$ 的三维张量，因为其包含了两个形状为 $(3, 3)$ 的矩阵：

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

标量 (*scalar*)

从技术上讲，单个数字被称为标量，它是零维张量。例如，数字 42 就是一个标量。

从数据生成特征

我们已经确定我们的模型将接受某种类型的张量作为输入。但是正如之前讨论的，我们的数据是以时间序列的形式出现的。如何才能将时间序列数据转换为可传递给模型的张量呢？

我们现在的任务是决定如何从数据中生成特征。在机器学习中，特征 (*feature*) 一词是指训练模型所依据的特定类型的信息，不同的模型需要使用不同的特征进行训练。例如，一个模型可能接受单个标量值作为其唯一的输入特征。

但实际上输入可能比这复杂得多：被设计用于处理图像的模型可能要接受图像数据的多维张量作为输入，而被设计用于基于多个特征进行预测的模型可能要接受包含多个标量值的向量数据，每个标量对应一个特征。

回想一下我们决定模型应该使用生产率、温度和振动来进行预测的情况。在数据的原始格式中，由于时间序列有着不同的间隔，这些格式将不适合被传递到模型中。下面将说明原因。

窗口化 (windowing)。在下面的图表中，时间序列中的每个数据片段都由一个星号表

示。当前的标签也被包含在了数据中，因为需要用这些标签来进行训练。我们的目标是训练一个模型，可以使用该模型根据当前条件预测机器在任何给定时刻是否正常工作。

生产率：	*	*	(每 2 分钟)
温度：	*	*	(每分钟)
振动：	* * * * *	* * * * *	(每 10 秒)
标签：	* * * * *	* * * * *	(每 10 秒)

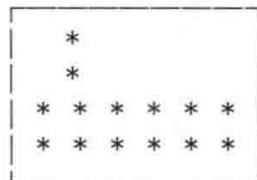
但是，由于我们的时间序列有着不同的间隔（比如每分钟一次，或者每 10 秒一次），如果我们只传递给定时刻的可用数据，那么它可能不包括所有可用的数据类型。例如，在下图标记的时刻中仅有振动数据可用。这将意味着，当我们的模型试图进行预测时，仅有振动信息可用。

生产率：	*	*	
温度：	*	*	
振动：	* * * * *	* * * * *	
标签：	* * * * *	* * * * *	



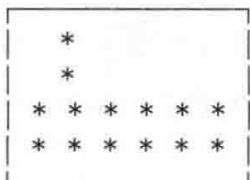
解决此问题的一种方法可能是在时间中选择一个窗口，并将该窗口中的所有数据合并为一组值。例如，我们可以选择一个 1 分钟的窗口并且查看其中包含的值：

生产率：	*	*	
温度：	*	*	
振动：	* * * * *	* * * * *	
标签：	* * * * *	* * * * *	



如果我们将每个时间序列窗口中的所有值求平均值，并对当前窗口中缺少数据点的所有值取其最新值，那么最终将得到一组单个值。我们可以根据窗口中是否存在任何“异常”标签来决定如何标记此快照。如果窗口中存在任何“异常”标签，那么该窗口应该被标记成“异常”，否则应该被标记成“正常”。

生产率：	*	*	
温度：	*	*	
振动：	* * * * *	* * * * *	
标签：	* * * * *	* * * * *	



平均：102
平均：34°C
平均：18%
标签：“正常”

这三个非标签值就是我们的特征！我们可以将其作为一个向量输入我们的模型，每个时间序列都有一个元素：

[102 34 .18]

在训练过程中，我们可以用每 10 秒的数据计算一个新窗口，并将其输入我们的模型，使用

标签将期望的输出告知训练算法。在推断过程中，每当我们想使用模型来预测异常行为时，我们可以查看数据，计算最新的窗口，然后在模型中运行它，最后接收一个预测结果。

这是一种简化过的方法，在实践中可能并不总是有效，但这已经是一个足够好的起点。你很快就会发现，机器学习完全就是试错法！

在继续训练之前，让我们回顾一下关于输入值的最后一件事。

标准化 (normalization)。通常，输入神经网络的数据将采用张量的形式，这个张量被浮点值 (floating-point) (又称浮点数) 所填充。浮点数是一种数据类型，用于表示有小数点的数字。为了使训练算法有效地运行，这些浮点值的大小必须彼此接近。实际上，最好将所有值都表示为介于 0 和 1 之间的数字。

让我们再看一下之前的输入张量：

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

这些数字的取值范围各不相同：温度值大于 100，而振动为小于 1 的小数。为了将这些值传递到我们的网络中，我们需要对其进行标准化，以使它们都在相似的范围内。

一种方法是计算数据集中每个特征的平均值，然后从值中减去平均值，这具有将数字压缩至更接近零的值的效果。下面是一个例子：

温度序列：

```
[108 104 102 103 102]
```

平均值：

```
103.8
```

标准化值，通过从每个温度中减去 103.8 得到：

```
[ 4.2 0.2 -1.8 -0.8 -1.8 ]
```

当你将图像输入神经网络时，经常会遇到以不同方式实现标准化的情况。计算机通常将图像存储为 8 比特整数的矩阵，其值的范围为 0~255。为了标准化这些值，使它们都在 0 和 1 之间，我们需要将每个 8 比特整数值都乘以 1/255。这是一个 3 像素 ×3 像素灰度图像的示例，其中每个像素的值代表其亮度：

原始的 8 比特整数值：

```
[[255 175 30]
 [0 45 24]
 [130 192 87]]
```

标准化后的值：

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

用机器学习思考

到目前为止，我们已经学会了如何开始思考用机器学习解决实际问题。在我们的工厂场景中，我们已经确定了一个合适的目标，收集并标记了合适的数据，设计了要传递到模型中的特征，并选择了一个模型架构。不管我们想解决什么样的问题，我们都会用同样的方法。需要注意的是，这是一个迭代的过程，我们经常需要在机器学习工作流的各个阶段中来回切换，直到得到一个可以工作的模型，抑或发现这个任务过于困难。

举个例子，想象一下我们正在搭建一个模型来预测天气。我们需要确定一个目标（例如预测明天是否会下雨），收集并标记数据集（例如过去几年的天气报告），设计将提供给模型的特征（可能是过去两天的平均条件），并选择适合此类数据的模型架构以及运行这个模型的设备。我们将提出一些初步的想法，并对其进行测试，然后调整方法，直到获得不错的结果。

工作流程中的下一步是训练，我们将在下一节进行探讨。

3.2.4 训练模型

训练是一个模型学习为给定的一组输入产生正确输出的过程。它涉及为模型提供训练数据，并对其进行微调，直到做出尽可能准确的预测为止。

如前所述，模型是由层次排列的数组表示的模拟神经元网络。这些数字称为权重（weight）和偏差（bias），或者统称为网络参数（parameter）。

当数据被输入网络时，网络将通过连续的数学运算对数据进行转换，这些运算涉及每层中的权重和偏差，模型的输出是对输入执行这些数学运算的结果。图 3-1 展示了一个简单的两层网络。

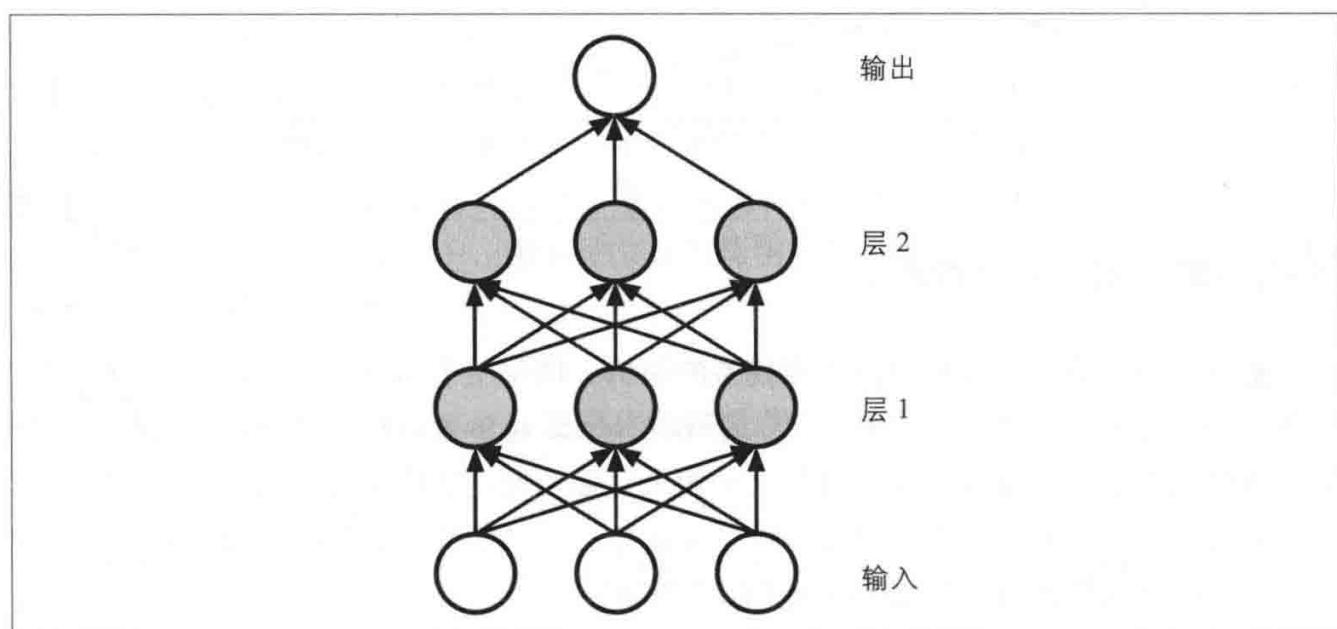


图 3-1：一个简单的两层深度学习网络

模型的权重始于随机值，偏差通常始于 0。在训练过程中，批次（batch）的数据被输入模型中，并将模型的输出与期望的输出（在我们的例子中是数据对应的标签，即“正常”或者“异常”）进行比较。一种称为反向传播（backpropagation）的算法会逐步调整权重和偏差，以使得模型的输出可以随着时间的推移越来越匹配期望的值。训练中我们以轮次（epoch）（即迭代的次数）为单位来决定是否要停止训练。

当模型的表现停止改善时，我们通常会停止训练。当模型能够做出准确预测时，就认为其已经收敛（converge）了。为了确定模型是否收敛，我们可以分析训练过程中模型表现的图表。两种常见的模型表现指标是损失（loss）和准确率（accuracy）。损失指标为我们提供了一个数字估算值，用于估算模型距离产生预期答案还有多远，而准确率指标告诉我们模型选择了正确预测的次数百分比。一个完美的模型将有着 0.0 的损失以及 100% 的准确率，然而实际上，很少有模型是完美的。

图 3-2 展示了深度学习网络训练期间的损失和准确率。你可以看到随着训练的进行，准确率在不断提升，损失逐渐减少，直至模型不再改进。

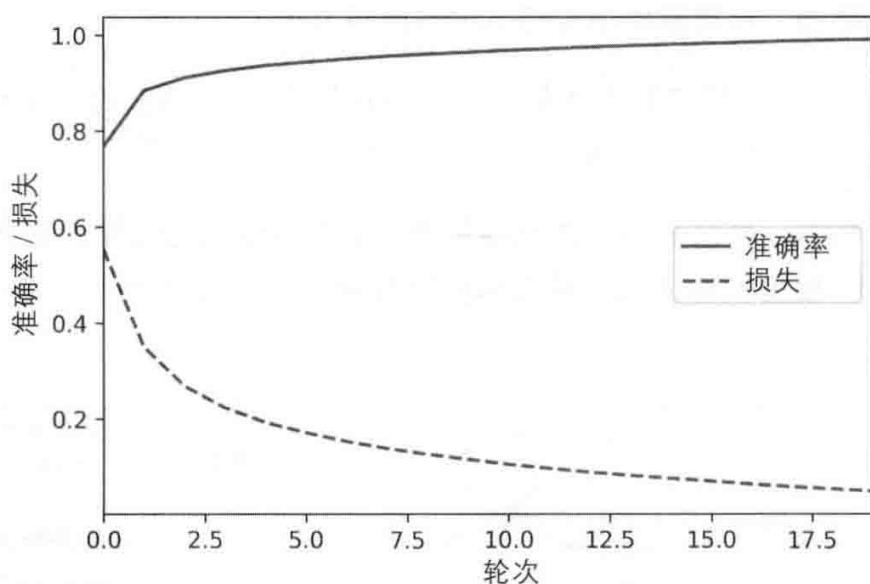


图 3-2：模型在训练过程中的收敛

为了提高模型的表现，我们可以改变模型的架构，调整用于建立模型的各种值，并调整训练过程。这些值统称为超参数（hyperparameter），超参数包括一些变量，比如要运行的训练轮次的数量以及每一层中神经元的数量。每当对这些值进行更改时，我们都可以重新训练模型、查看指标并决定是否进一步优化。希望坚持不懈的时间投入和不遗余力的迭代改进可以使我们得到一个可接受的准确率！



重要的是请记住，对于你试图解决的问题，并不能保证总能获得足够好的准确率。数据集中并不会总是有足够的信息来进行准确的预测，即使采用最先进的深度学习，某些问题也无法被解决。也就是说，即使模型的准确率不是 100%，它也可能是有用的。以我们的工厂示例为例，即使我们只能在部分时间内预测到异常操作，这也会有很大的帮助。

欠拟合和过拟合

模型无法收敛的两个最常见的原因是欠拟合 (underfit) 和过拟合 (overfit)。

神经网络会学习如何将其行为与从数据中识别出的模式进行拟合 (fit)。如果模型拟合正确，它将为给定的一组输入产生正确的输出。当模型欠拟合时，它将无法学习到足以表示这些模式的方法，从而无法做出良好的预测。发生这种情况的原因有很多，最常见的原因是架构太小而难以适应被建模系统的复杂性，或者没有使用足够多的数据对模型进行训练。

而如果模型过拟合，则表明它过于好地理解了用来对其进行训练的数据。该模型能够准确预测训练数据的细节，却无法将学习结果泛化到以前未见过的数据上。通常，发生这种情况是因为模型设法完全记住了训练数据，或者模型已经学会了依靠只存在于训练数据（而非现实世界中）的捷径来进行预测。

例如，假设你正在训练一个模型来对一组猫和狗的照片进行分类。如果你的训练数据中所有狗的照片都是在户外拍摄的，而所有猫的照片都是在室内拍摄的，你的模型可能会学会作弊：利用照片中是否存在天空来预测是哪种动物。这意味着，如果未来狗的照片是在室内拍摄的，那么这张照片很可能会被模型错误分类。

有很多方法可以解决过拟合的问题。其中一种可能的方法是减小模型的大小，使其没有足够的能力学习其训练集的精确表示。另一类称为正则化 (regularization) 的技术也可以用来减少训练过程中过拟合的程度。为了充分利用有限的数据，还可以使用一种称为数据增强 (data augmentation) 的技术，通过对现有数据进行切片和切块生成新的人工数据点。但是，在可能的情况下，避免过拟合的最佳方法是使用一个更大、更多样化的数据集。更多的数据总是有用的！

正则化和数据增强

正则化技术被用来减少深度学习模型对其训练数据的过拟合。通常，正则化技术涉及以某种方式对模型进行约束，以防止模型完整地记住训练过程中输入的数据。

正则化有几种不同的方法。有些算法，例如 L1 正则化 (L1 regularization) 和 L2 正

则化 (L2 regularization)，涉及调整训练过程中使用的算法，以惩罚容易过拟合的复杂模型。另一种名为 dropout 的方法是在训练过程中随机切断神经元之间的连接。我们将在本书后面的内容中讨论正则化在实践中的应用。

我们还将探讨数据增强技术，这是一种人为地扩展训练数据集的方法。数据增强是通过为每个训练输入创建多个附加版本完成的，每个版本的转换都保留了其原本的含义，但改变了数据的确切构成。在一个示例中，我们训练了一个模型来识别音频样本中的语音。我们通过添加人工噪声并及时移动样本来增强原始训练数据。

训练、验证和测试

为了评估一个模型的表现，我们可以观察它在训练数据上的表现。然而，这只能告诉我们故事的一部分。在训练过程中，模型会学习尽可能地拟合训练数据。正如在前面所看到的，在某些情况下，模型将开始过拟合训练数据，这意味着模型可以在训练数据上很好地工作，在现实生活中却不行。

要了解这种情况何时发生，我们需要使用训练中未被使用的新数据来验证 (validate) 模型。我们通常将数据集分为三个部分——训练 (training)、验证 (validation) 和测试 (test)。典型的划分是 60% 的训练数据、20% 的验证数据和 20% 的测试数据。这种拆分必须使每个部分包含相同的信息分布，并保留数据结构。例如，由于我们的数据是一个时间序列，因此可以将其分为三个连续的时间块。如果数据不是时间序列，就可以随机采样数据点。

在训练过程中，训练数据集被用来训练模型，而来自验证数据集的数据将被周期性地输入模型来计算损失。由于模型以前没有看到过这些数据，因此其损失评分是衡量模型运行情况的更可靠的指标。通过比较一段时间内的训练数据和验证数据的损失（以及准确率或任何其他可用的指标），你可以查看模型是否出现过拟合的问题。

图 3-3 展示了一个过拟合的模型。你可以看到训练损失在不断减少，而验证损失在增加。这意味着模型在预测训练数据方面变得越来越好，但是失去了将预测泛化到新数据的能力。

当我们调整模型和训练过程以改善模型表现并避免过拟合时，我们将有望开始看到验证指标得到改善。

然而不幸的是，调整的过程也会有副作用。通过不断地优化模型来改善验证指标，我们可能会使得模型同时对训练数据和验证数据过拟合！我们所做的每次调整都会使模型更好地拟合验证数据，最后，我们可能会遇到与以前相似的过拟合问题。

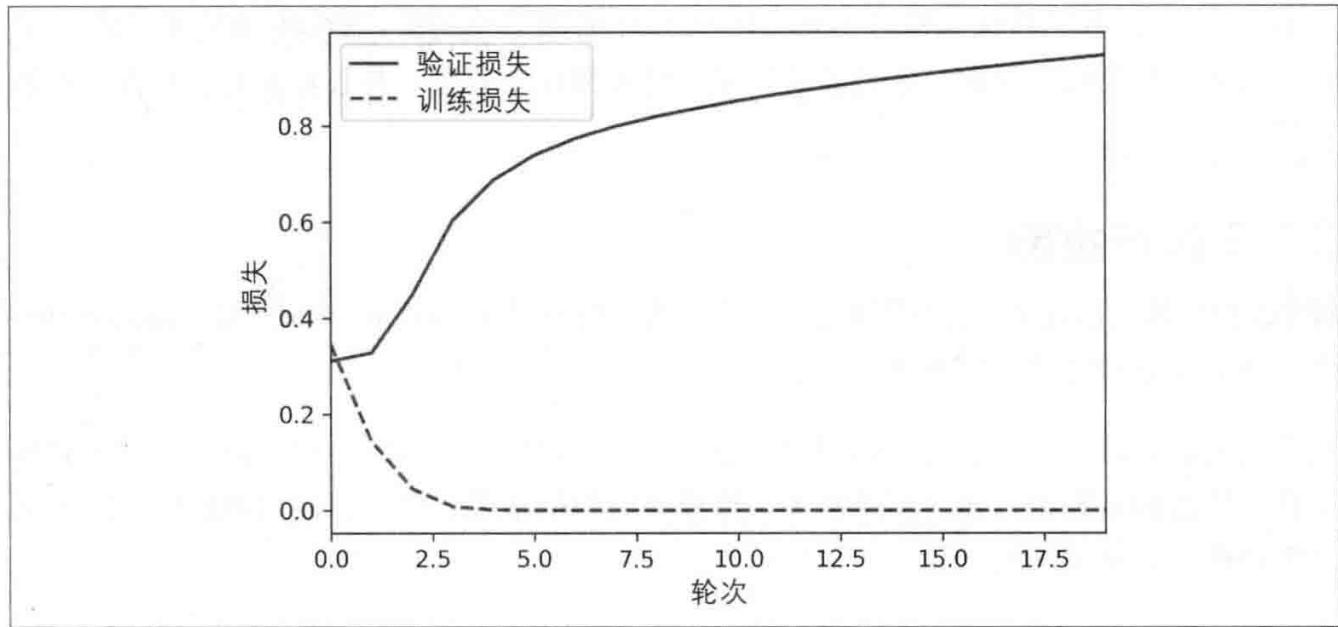


图 3-3：模型在训练过程中的过拟合

为了验证这种情况是否发生，我们训练模型的最后一步是在测试数据集上运行模型，并确认其表现是否和验证期间的表现一样。如果不一样，则说明我们已经将模型“优化”到同时对训练数据和验证数据过拟合。在这种情况下，我们可能需要提出一个新的模型架构，因为如果我们继续调整模型以提高其在测试数据上的表现，该模型可能也将对测试数据过拟合。

在有了一个能够很好地处理训练数据、验证数据和测试数据的模型之后，工作流程的训练部分就结束了。接下来，我们将准备好在设备上运行我们的模型。

3.2.5 转换模型

在本书中，我们使用 TensorFlow 来构建和训练模型。TensorFlow 模型本质上是一组指令，这些指令告诉解释器（interpreter）如何转换数据以产生输出。当我们想使用我们的模型时，我们只是将其加载到内存中，然后使用 TensorFlow 解释器执行。

然而，TensorFlow 的解释器被设计为在性能强大的台式计算机和服务器上运行模型。由于我们将在微控制器上运行模型，因此需要为我们的用例设计一个不同的解释器。幸运的是，TensorFlow 提供了一个解释器和一些附带的工具，可以在小型低能耗设备上运行模型。这组工具称为 TensorFlow Lite。

在 TensorFlow Lite 运行模型之前，我们需要先将模型转换为 TensorFlow Lite 的格式并将其作为文件保存在磁盘上。我们使用名为 TensorFlow Lite 转换器（TensorFlow Lite Converter）的工具执行此操作。这个工具还可以应用一些特殊的优化，在不牺牲性能的前提下，通过减小模型的尺寸帮助模型更快地运行。

在第 13 章中，我们将深入探讨 TensorFlow Lite 的细节，以及它是如何帮助我们在微型设备上运行模型的。目前，你只需要知道你将转换你的模型，并且转换的过程很迅速也很简单。

3.2.6 运行推断

模型在被转换之后就可以进行部署了。现在，我们将使用 TensorFlow Lite for Microcontrollers C++ 库来加载模型并进行预测。

由于这是我们的模型与应用代码相交的地方，因此我们需要编写一些代码，以从传感器中获取原始输入数据，并将其转换为训练模型所用的格式。然后，我们将转换后的数据传递到模型中并运行推断。

这将产生包含预测的输出数据。在我们的分类器模型中，输出将是每个类别（“正常”和“异常”）的得分。

对于对数据进行分类的模型，通常所有类别的得分总和为 1，而得分最高的类别将作为预测结果。得分之间的差异越大，预测的置信度就越高。表 3-2 列出了一些样例输出。

表 3-2：样例输出

“正常”得分	“异常”得分	解释
0.1	0.9	“异常”状态具有高置信度
0.9	0.1	“正常”状态具有高置信度
0.7	0.3	“正常”状态具有稍高置信度
0.49	0.51	结果尚无定论，因为两种状态都没有明显的优势

在我们的工厂机器示例中，每个单独的推断都只考虑了一份数据快照——根据各种传感器读数，告诉我们最近 10 秒内出现异常状态的可能性。由于现实世界中的数据通常比较混乱，并且机器学习模型的准确性也常常不尽如人意，因此临时故障可能会导致分类错误。例如，由于传感器临时故障，我们可能会观测到温度突然升高。这种短暂、不可靠的输入可能会导致输出的分类结果暂时无法反映真实的情况。

为了防止这些瞬时故障引起的问题，我们可以对一段时间内模型所有的输出取平均值。例如，我们可以在当前数据窗口上每 10 秒运行一次模型，并取最后 6 个输出的平均值，以得出每个类别平滑后的分数。这意味着短暂故障将被忽略，我们仅对一致的行为采取行动。在第 7 章中，我们将使用这一技术来帮助检测唤醒词。

在得到每个类别的得分之后，我们的应用程序代码将决定接下来要做什么。比如，如果在一分钟内持续检测到异常状态，我们的代码将发送信号以关闭机器并警告维护团队。

3.2.7 评估和故障排除

在我们部署了模型并使其在设备上运行之后，我们将开始观察它的实际表现是否达到了我们的期望。尽管我们已经证明了该模型可以对测试数据做出准确的预测，但它在解决实际问题时的表现可能会有所不同。

发生这种情况的原因有很多。例如，训练中使用的数据可能无法完全代表实际操作中的可用数据。也许由于当地气候，机器的温度比数据集中所收集的温度要低。这可能会影响我们的模型做出的预测，从而使预测不再像预期的那样准确。

另一种可能性是，模型可能在我们没有意识到的情况下过拟合了数据集。在 3.2.4 节中，我们了解到，当数据集碰巧包含模型可以学习识别的额外信号而非我们期望的信号时，也会出现过拟合。

如果我们的模型不能在实际生产中正常工作，那么我们需要先排除一些故障。首先，我们要排除所有可能影响传入模型的数据的硬件问题（例如传感器故障或意外的噪声）。其次，我们可以从部署模型的设备中截获一些数据，并将其与原始数据集进行比较，以确保数据保持在相同的范围内。如果不一致，那么也许是因为我们没有预料到的环境条件或者不同的传感器特性。如果数据证实这些都没有问题，那么问题可能在于模型出现过拟合。

在排除了硬件问题之后，解决过拟合问题的最佳方法通常是使用更多的数据进行训练。我们可以从部署的硬件中捕获其他数据，将其与原始数据集结合起来，然后重新训练我们的模型。在此过程中，我们可以应用正则化和数据增强技术来帮助我们充分利用现有的数据。

为了获得良好的实际表现，有时需要对模型、硬件和附带的软件进行一些迭代。如果遇到问题，就要像对待其他技术问题一样：采用科学的方法进行故障排除，消除可能的因素，并分析数据以找出潜在的问题。

3.3 小结

现在你已经熟悉了机器学习从业者的基本工作流程，那么我们可以开始 TinyML 的下一步冒险了。

在第 4 章中，我们将创建我们的第一个模型，并将它部署到一些微型硬件上！

第4章

TinyML 之 “Hello World”： 创建和训练模型

在第3章中，我们学习了机器学习的基本概念以及机器学习项目遵循的一般工作流程。在本章和下一章中，我们开始将学到的知识付诸实践。我们将从零开始创建和训练模型，然后将训练好的模型集成到一个简单的微控制器程序中。

在这个过程中，你将会接触到一些功能强大的开发工具，这些工具每天都被前沿的机器学习开发人员所使用。你还将学习如何把机器学习模型集成到C++程序中，并将其部署到微控制器中以控制电路中的电流。这可能是你第一次尝试混合硬件和机器学习，应该会很有趣！

你可以在你的Mac、Linux或者Windows机器上测试我们在这些章节中编写的代码，但是要获得完整的体验，你将需要2.2节提到的硬件：

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

为了创建机器学习模型，我们将使用Python、TensorFlow和Google的Colaboratory，这是一个基于云的交互式笔记本，用于试验Python代码。对于现实世界中的机器学习工程师来说，这些都是非常重要的工具，而且它们都是免费的。



你是否在想这一章的标题是什么意思？通过示例代码介绍新技术是编程教学的传统，这些示例代码通常会演示如何做一些非常简单的事情。常见的简单任务是让程序输出“Hello, world”。在机器学习中没有明确的等价单词（程序），因此我们使用术语“Hello World”来指代一个简单易读的机器学习应用程序示例。

在本章中，我们将执行以下操作：

1. 获取一个简单的数据集。
2. 训练深度学习模型。
3. 评估模型的表现。
4. 转换模型格式以使其可以在设备上运行。
5. 编写代码以执行设备上的推断。
6. 将代码编译为二进制文件。
7. 将二进制文件部署到微控制器上。

我们使用的所有代码都可以在 TensorFlow 的 GitHub 代码仓库中找到。

我们建议你阅读本章的每个部分，并尝试运行代码。这一路上都有详尽的说明。但是在开始之前，让我们讨论一下到底要创建什么。

4.1 我们要创建什么

在第 3 章中，我们讨论了深度学习网络如何从训练数据中学习模式，以对新的数据进行预测。现在，我们要训练一个网络来模拟一些非常简单的数据。你可能听说过正弦函数。它在三角学中被用来描述直角三角形的属性。我们要训练的数据是一个正弦波，这是通过绘制正弦函数随时间变化的结果而得到的图形（见图 4-1）。

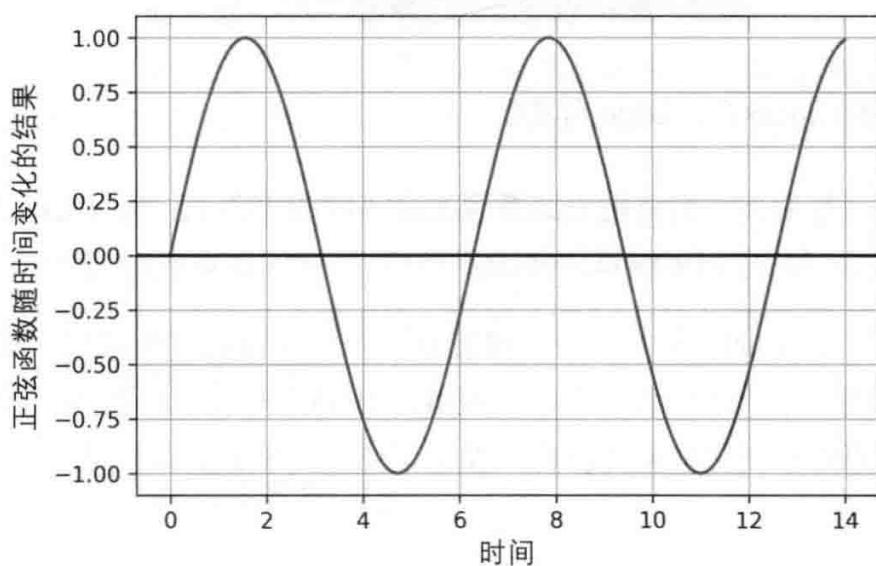


图 4-1：正弦波

我们的目标是训练一个可以对给定输入值 x 预测其正弦值 y 的模型。在实际应用中，如

果你需要 x 的正弦值，可以直接计算得到。但是，这里我们不直接计算而是通过训练模型来近似结果，以演示机器学习的基本原理。

我们项目的第二部分是在硬件设备上运行这个模型。从视觉上看，正弦波是一条平滑的曲线，从 -1 到 1 平滑地循环。这使得它非常适合控制灯光进行令人愉悦的视觉表演！我们将使用模型的输出来控制一些闪烁的 LED 或者图形动画，具体效果取决于设备的能力。

在网上，你可以找到编写代码使得 SparkFun Edge 开发板的 LED 闪烁的 GIF 动画。图 4-2 是这个动画中一帧静止的画面，它显示了设备的几个 LED 亮起。对于机器学习项目来说，这可能不是一个特别有用的应用，但是本着“Hello World”示例的精神，它简单、有趣，并且有助于演示你需要知道的基本原则。

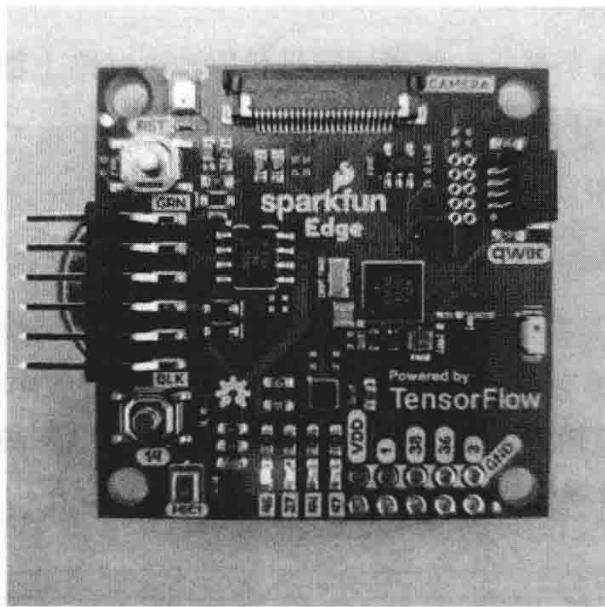


图 4-2：运行代码的 SparkFun Edge 开发板

在基本代码正常运行后，我们将把它部署到三种不同的设备上：SparkFun Edge、Arduino Nano 33 BLE Sense 和 ST Microelectronics STM32F746G Discovery 套件。



由于 TensorFlow 是一个不断发展、积极开发的开源项目，因此你可能会注意到此处打印的代码与在线托管的代码之间存在一些细微的差别。不过不用担心——即使有几行代码发生了变化，它们的基本原理仍然是一样的。

4.2 我们的机器学习工具链

为了搭建项目中的机器学习部分，我们使用了一些工具，现实世界中的机器学习从业者

员都在使用这些工具。在本节中，我们将向你介绍这些工具。

4.2.1 Python 和 Jupyter Notebook

Python 是机器学习科学家和工程师最喜欢的编程语言。它易于学习，适用于许多不同的应用程序，并且拥有大量的数据处理库和数学库。绝大多数深度学习的研究都是使用 Python 完成的，研究人员也经常公开他们创建模型的 Python 源代码。

当与 Jupyter Notebook 结合使用时，Python 尤其出色。Jupyter 是一种特殊的文档格式，它允许你混合文字、图形和代码，并且通过单击按钮就可以运行代码。Jupyter Notebook 被广泛用于描述、解释和探索机器学习代码及问题。

我们将在 Jupyter Notebook 中创建模型，这将使我们能够做一些很棒的事情，以在开发过程中可视化数据。比如以图的方式展示模型的准确率和收敛性。

Python 很容易阅读和学习，如果你有一定的编程经验，那么你应该能够毫无困难地学习本书。

4.2.2 Google Colaboratory

为了运行我们的笔记本，我们将使用一个叫作 Colaboratory 的工具，它简称 Colab。Colab 是由 Google 开发的，它为运行 Jupyter Notebook 提供了一个在线环境。作为促进机器学习研究和发展的工具，它是可以免费使用的。

传统意义上，你需要在计算机上创建一个笔记本（notebook）。这需要安装很多依赖项，比如 Python 库，这个过程可能很令人头疼。与其他人生共享笔记本也很困难，因为别人可能正在使用不同版本的依赖项，这就意味着笔记本可能无法按照预期的方式运行。此外，机器学习可能需要大量密集的计算，因此在开发计算机上训练模型可能会很慢。

Colab 允许你在 Google 的强大硬件上运行笔记本，而且没有任何成本。你可以通过任何 Web 浏览器编辑和查看笔记本，还可以与其他人共享你的笔记本，共享者在运行笔记本时会得到相同的结果。你甚至可以将 Colab 配置为在特定的加速硬件上运行，该硬件可以比普通计算机更快地进行模型训练。

4.2.3 TensorFlow 和 Keras

TensorFlow 是一套用于构建、训练、评估和部署机器学习模型的工具。TensorFlow 最初是由 Google 开发的，现在是一个开源项目，由全世界成千上万的贡献者构建和维护。它是当前最流行、应用最广泛的机器学习框架。大多数开发人员通过 Python 库与 TensorFlow 交互。

TensorFlow 可以做很多不同的事情。在本章中，我们将使用 TensorFlow 的高级 API——Keras，它使得创建和训练深度学习网络变得非常容易。我们还将使用 TensorFlow Lite——一组用于将 TensorFlow 模型部署到移动端和嵌入式设备的工具，用于在设备上运行我们的模型。

我们将在第 13 章更详细地介绍 TensorFlow。现在，你只需要知道它是一个非常强大的行业标准工具，无论你是深度学习初学者还是专家，它都能很好地满足你的需求。

4.3 创建我们的模型

现在，我们将逐步完成模型的创建、训练和转换。我们将在本章中包含所有代码，你也可以在 Colab 中进行后续操作，并随时运行代码。

首先，加载笔记本。页面加载后，在顶部单击“Run in Google Colab”（在 Google Colab 中运行）按钮，如图 4-3 所示。这会将笔记本从 GitHub 复制到 Colab，从而允许你运行并编辑它。

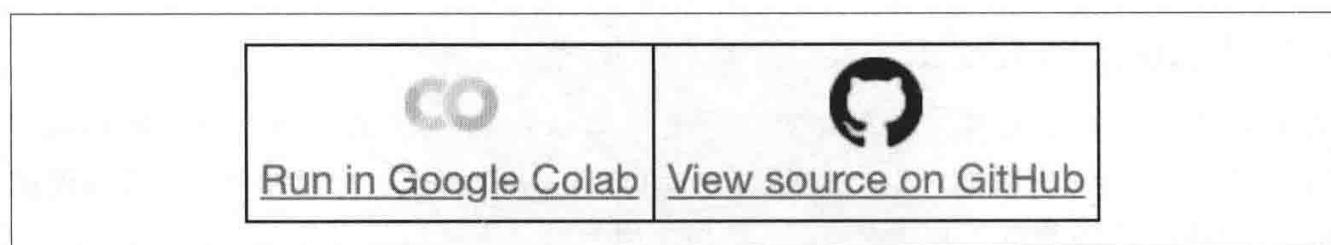


图 4-3：“Run in Google Colab”按钮

加载笔记本时会出现的问题

在撰写本书时，GitHub 存在一个已知问题：在显示 Jupyter Notebook 时会间歇性地出现错误消息。如果你在尝试访问笔记本时看到信息“Sorry, something went wrong. Reload?”（对不起，出了点问题。重新加载？），可以通过以下过程在 Colab 中直接打开它。复制笔记本的 GitHub URL 中出现在 <https://github.com> 的部分：

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

并以 <https://colab.research.google.com/github> 作为前缀。这将生成完整的 URL：

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

在浏览器中导航到该 URL 可直接在 Colab 中打开笔记本。

默认情况下，除了代码之外，笔记本还包含了运行代码时你应该看到的输出示例。因为我们将在本章中遍历所有代码，所以让我们先清除这个输出，使笔记本处于原始状态。为此，在 Colab 的菜单中，单击“Edit”（编辑），然后选择“Clear all outputs”（清除所有输出），如图 4-4 所示。

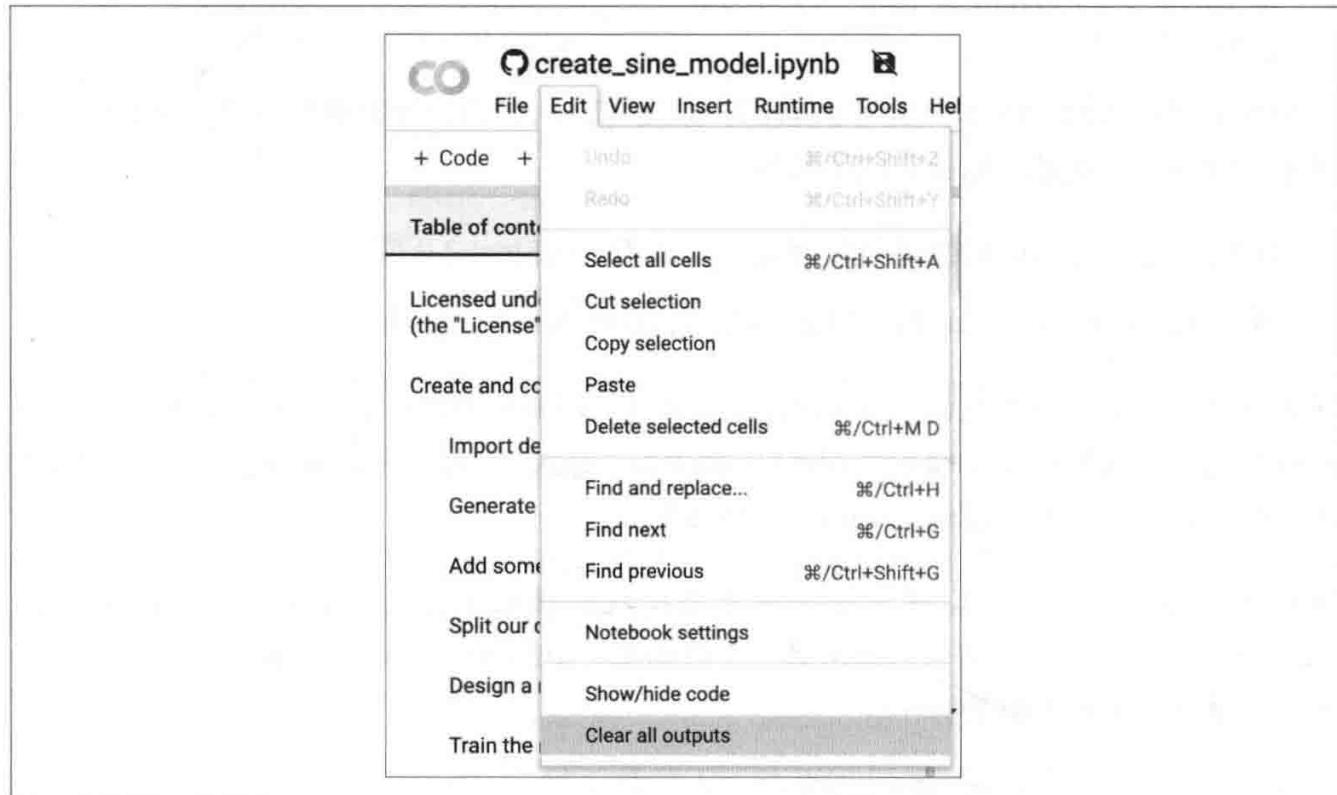


图 4-4：“Clear all outputs”选项

很好，我们的笔记本已经准备好了！



如果你已经熟悉了机器学习、TensorFlow 和 Keras，那么你可能想要直接跳转到将模型转换为与 TensorFlow Lite 一起使用的部分。你可以跳转至 4.5 节。

4.3.1 导入依赖项

第一个任务是导入我们所需要的依赖项。在 Jupyter Notebook 中，代码和文本按照单元格（cell）排列。单元格分为代码（code）单元格（包含可执行的 Python 代码）和文本（text）单元格（包含格式化的文本）。

我们的第一个代码单元格位于“Import dependencies”（导入依赖项）下面。它配置了我们训练和转换模型所需的所有库。代码如下：

```
# 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
```

在 Python 中，`import` 语句用于加载库，以便可以从我们的代码中使用它。从代码和注释中可以看到，此单元格执行以下操作：

- 使用 `pip`（Python 的软件包管理器）安装 TensorFlow 2.0 库。
- 导入 TensorFlow、NumPy、Matplotlib 库和 Python 的 `math` 库。

导入库时，我们可以给它起一个别名，这样可以使得在后续很容易引用库。例如，在前面的代码中，我们使用 `import numpy as np` 来导入 NumPy 库，并将它命名为 `np`。因此我们可以在代码中通过名字 `np` 使用 NumPy。

选中单元格时，我们可以通过单击左上角出现的按钮来运行代码单元中的代码。在“Import dependencies”部分，单击第一个代码单元中的任何位置即可选中单元格。图 4-5 展示了单元格被选中时的样子。

▼ Import dependencies

Our first task is to import the dependencies we need. Run the following cell to do so:

```
# TensorFlow is an open source machine learning library
# Note: The following line is temporary to use v2
!pip install tensorflow==2.0.0-beta0
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
```

图 4-5：被选中的“Import dependencies”单元格

要运行代码，需要单击左上角出现的按钮。在运行代码时，该按钮将展示如图 4-6 所示的圆圈动画。

开始安装依赖项时，你会看到一些输出。最终你应该看到以下输出信息，这意味着该库已成功安装：

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

```
# TensorFlow is an open source machine learning library
# Note: The following line is temporary to use v2
!pip install tensorflow==2.0.0-beta0
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

... Collecting tensorflow==2.0.0-beta0
```

图 4-6：运行中的“Import dependencies”单元格

在 Colab 中运行单元格后，当你不再选中它时，你会看到左上角显示 [1]，如图 4-7 所示。这是一个计数器，它的值在每次单元格运行时都会递增。

```
[1] # TensorFlow is an open source machine learning library
# Note: The following line is temporary to use v2
!pip install tensorflow==2.0.0-beta0
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
```

图 4-7：左上角的单元格运行计数器

你可以通过这个计数器来了解哪些单元格已被运行，以及运行了多少次。

4.3.2 生成数据

深度学习网络通过学习数据内在的模式来构建模型。就像之前提到的，我们将训练一个网络来对正弦函数生成的数据建模。这将产生一个模型，该模型可以获取一个值 x ，并预测其正弦值 y 。

在进一步研究之前，我们需要一些数据。在现实世界中，我们的数据可能是从传感器或

产品日志中收集而来的。然而，对于当前的例子，我们将使用一些简单的代码来生成一个数据集。

下一个单元格是用来生成数据的。我们的计划是生成 1000 个代表正弦波上随机点的值。让我们先看看图 4-8，提醒自己真正的正弦波是什么样子的。

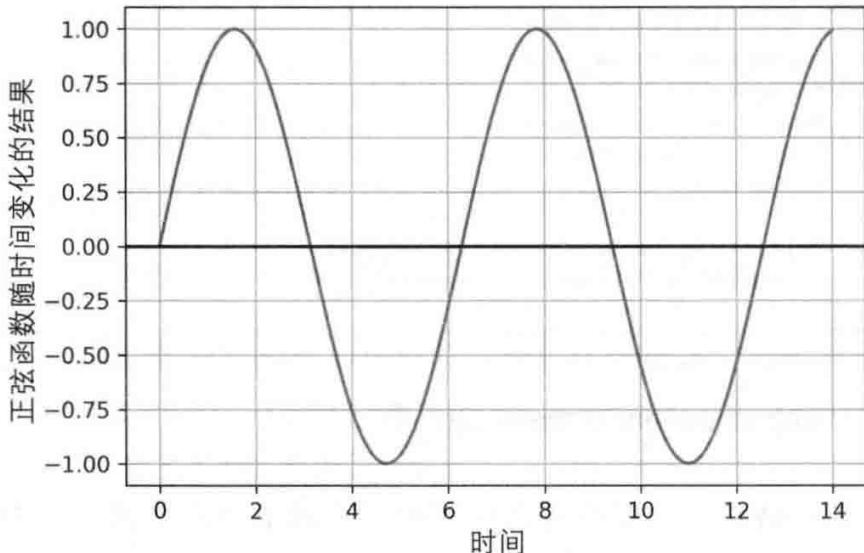


图 4-8：正弦波

波的每一个完整循环称为周期（period）。从图 4-8 中我们可以看到， x 轴上大约每 6 个单元形成一个完整的周期。实际上，正弦波的周期是 2π 。

因此，为了获得一个完整的正弦波的训练数据，我们的代码将随机产生 $0 \sim 2\pi$ 的 x 值，然后计算这些 x 值的正弦值。

下面是这个单元的完整代码，它使用 NumPy（我们之前导入的 `np`）来生成随机数并计算它们的正弦值：

```
# 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  $2\pi$ , 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()
```

除了我们前面讨论过的方法之外，这段代码还有一些地方值得强调。首先，你会看到我们使用 `np.random.uniform()` 生成 `x` 值。这个方法会返回指定范围内的随机数数组。NumPy 包含了很多可以对整个数组的值进行操作的有用方法，这在处理数据时非常方便。

其次，在生成数据之后，我们将对数据进行随机排列（shuffle）。这一点非常重要，因为在深度学习中使用的训练过程依赖于以真正的随机顺序向其输入数据。如果数据是有序的，那么得到的模型可能不会那么准确。

接下来，请注意我们使用 NumPy 的 `sin()` 方法来计算正弦值。NumPy 可以一次性地对所有的 `x` 值执行此操作，并返回一个数组。NumPy 真的太棒了！

最后，你会看到一些使用 `plt`（Matplotlib 的别名）进行调试的神秘代码：

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

这个代码在做什么？它绘制了我们的数据图表。Jupyter Notebook 最好的功能之一就是能够图形化展示你所运行代码的输出。Matplotlib 是一个非常优秀的通过数据创建图形的工具。可视化数据是机器学习工作流程中一个非常关键的部分，这将对我们训练模型非常有帮助。

要生成数据并将其呈现为图形，需要在单元格中运行代码。代码单元运行完成后，你应该会看到下面出现了一个漂亮的图，如图 4-9 所示。

这就是我们生成的数据！它是在一条平滑的正弦曲线上随机选择的点。我们可以用这些数据来训练我们的模型。不过，这些数据有点过于简单了。深度学习网络令人兴奋的一点是它能够从噪声中过滤出模式。这使得深度学习即使是在混乱的、真实的数据上训练，也能有效地学习并做出很好的预测。为了展示这一点，让我们向数据中添加一些随机噪声，并绘制另一个图：

```
# 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()
```

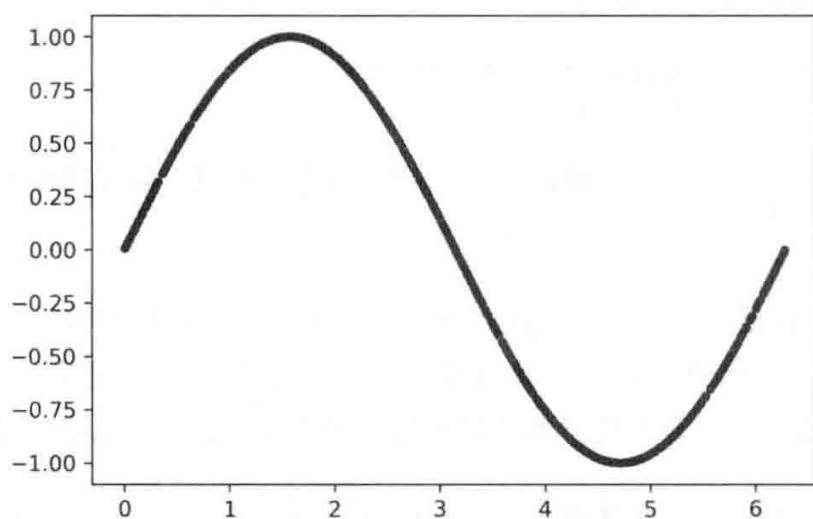


图 4-9：我们生成的数据的图形

运行这个单元格并查看结果，如图 4-10 所示。

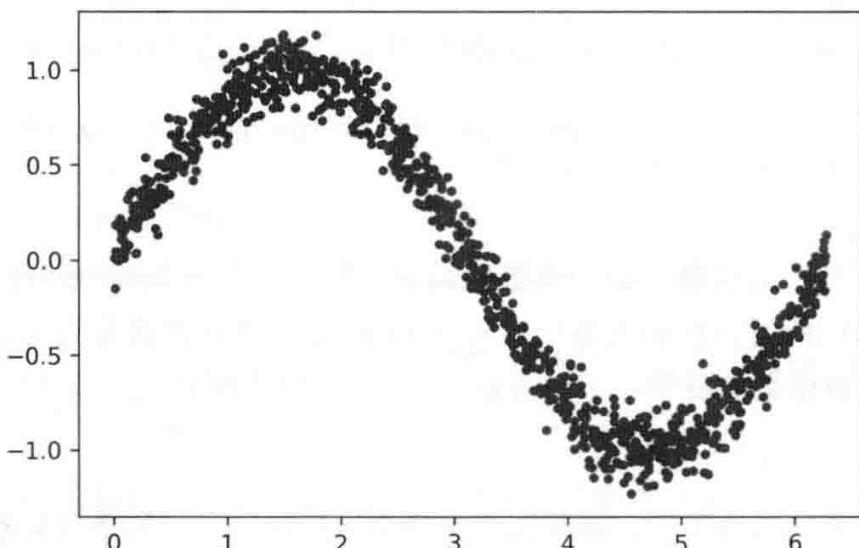


图 4-10：加入噪声之后生成的数据图形

好多了！由于我们的点是随机的，因此它们代表了正弦波的分布，而不是平滑的完美正弦曲线。这更能反映现实世界的情况——在现实世界中，数据通常是非常杂乱的。

4.3.3 拆分数据

你可能还记得，我们在上一章中提到数据集通常会分为三个部分：训练集、验证集和测试集。为了评估训练的模型的准确率，我们需要将模型预测的数据与实际的数据进行比较，以检查它们的匹配程度。

评估会发生在训练期间（称为验证）和训练之后（称为测试）。在每种情况下，非常重要的一点是，我们都必须使用没有被用于模型训练的新数据。

为了确保有数据可以被用于评估，我们将在开始训练之前预留出一些数据。让我们保留 20% 的数据用于验证，另外 20% 的数据用于测试。然后，我们将用余下 60% 的数据来训练模型。这是训练模型时经常使用的经典拆分比例。

以下代码将拆分数据，然后用不同的颜色绘制数据：

```
# 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()
```

为了拆分数据，我们将使用 NumPy 中另外一个非常简洁的方法：`split()`。该方法输入一个数据数组和一个索引数组，然后会根据提供的索引将数据分成若干部分。

运行该单元格以查看我们的拆分结果。每种类型的数据都将以不同的颜色被绘制出来，如图 4-11 所示。

4.3.4 定义基本模型

现在我们有了自己的数据，是时候创建一个合适的模型来训练它了。

我们将创建一个模型，该模型将接收一个输入值（在本例中为 x ），并用它来预测一个数字输出值（即 x 的正弦）。这种类型的问题称为回归（regression）。我们使用回归模型训练任何需要输出数字的任务。例如，回归模型可以尝试通过加速度传感器的数据来预测一个人的跑步速度（以英里^{编注 1}/时为单位）。

编注 1：1 英里=1609.344 米。

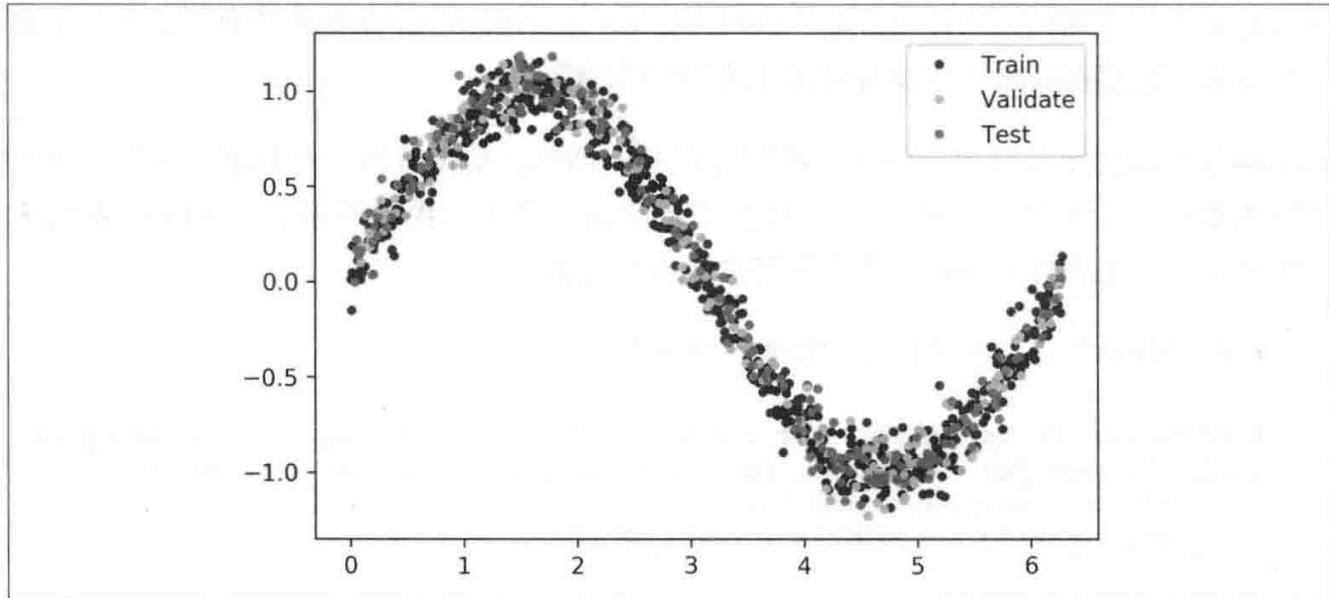


图 4-11：数据被分为训练集、验证集和测试集（以不同灰度显示）

为了创建模型，我们将设计一个简单的神经网络。它使用神经元层来尝试学习隐藏在训练数据内的任何模式，以便对新的数据进行预测。

事实上，完成此项任务的代码非常简单。它使用 TensorFlow 的高级 API——Keras 来创建深度学习网络：

```
# 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()
```

首先，我们使用 Keras 创建一个 `Sequential`（序列）模型，它表示模型中的每一层神经元都堆叠在下一层之上，如图 3-1 所示。然后，我们定义两个层。下面是第一层的定义：

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

第一层只有一个输入（ x 值）和 16 个神经元。这是一个 `Dense`（密集）层，也称为全连接层（fully connected layer），这意味着当我们进行预测时，在推断过程中的输入将被输

入这一层的每个神经元中。然后，每个神经元将在一定程度上被激活（activate）。每个神经元的激活量取决于它在训练过程中获得的权重（weight）和偏差（bias），以及它的激活函数（activation function）。神经元的激活程度将以数字的形式输出。

如 Python 代码所示，激活程度是通过一个简单的公式计算得来的。由于它是由 Keras 和 TensorFlow 处理的，因此我们不需要自己编写代码。但是了解它会有助于我们进一步深入学习：

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

为了计算神经元的激活程度，需要先将输入乘以权重，再加上偏差。计算出的值将被传递到激活函数中，得到的结果就是神经元的激活程度。

激活函数是一个数学函数，用于模拟神经元。在我们的网络中，我们使用的激活函数是线性整流单元（Rectified Linear Unit，ReLU）。这在 Keras 中通过参数 `activation = relu` 指定。

ReLU 是一个简单的函数，其 Python 代码如下所示：

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

ReLU 返回输入和零两者中较大的值。如果输入值为负，ReLU 将返回零；如果输入值大于零，ReLU 将返回输入值。

图 4-12 显示了一系列输入值对应的 ReLU 输出。

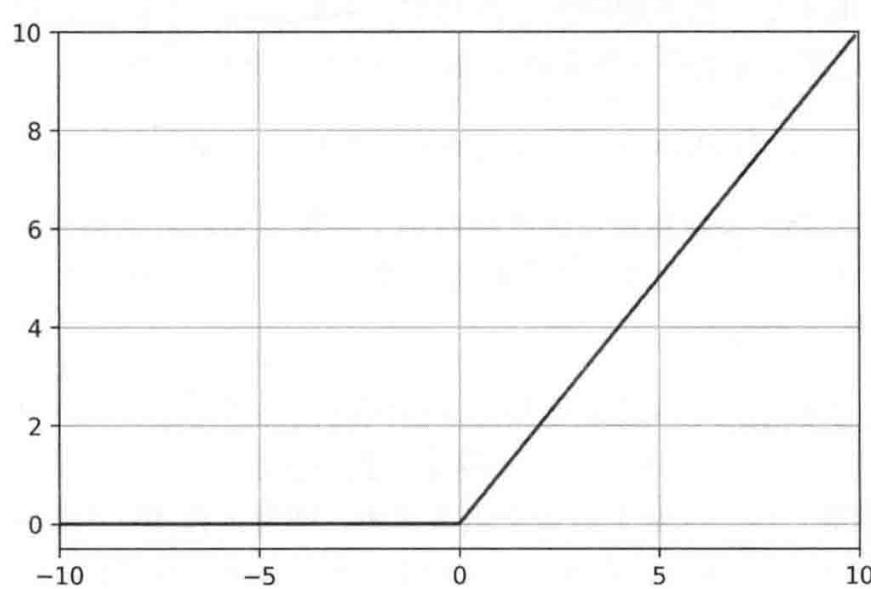


图 4-12：-10~10 对应的 ReLU 输出

如果没有激活函数，神经元的输出将始终是其输入的线性函数。这意味着网络只能对线性关系建模，在线性关系中， x 和 y 的比值在一定范围内相同。因为正弦波是非线性的，所以这会使得网络无法对正弦波建模。

由于 ReLU 是非线性的，所以它允许多层神经元相互作用，以建模复杂的非线性模型，其中 y 值不会随着 x 的增加而增加相同的量。



当然还有很多其他激活函数，不过 ReLU 是最常用的。你可以在有关激活函数的 Wikipedia 文章中找到一些其他的选项。每个激活函数都有不同的权衡，机器学习工程师会通过试验来找出最适合给定问题的激活函数。

来自第一层的激活数据将作为输入提供给第二层，定义如下：

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

由于第二层由单个神经元组成，它将接收 16 个输入，即上一层中的每个神经元的输出。其目的是将上一层的所有激活程度值合并为一个输出值。因为这是我们的输出层，所以不需要指定激活函数——只需要原始的结果。

由于这个神经元有多个输入，因此每个输入都有一个对应的权重值。神经元的输出通过以下公式计算得到，如 Python 代码所示：

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

将每个输入与其相应的权重相乘、求和，然后加上神经元的偏差，可以得到输出值。

该网络的权重和偏差是在训练期间学习得到的。本章前面显示的代码中的 `compile()` 步骤配置了训练过程中使用的一些重要参数，并准备了要训练的模型：

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

`optimizer`（优化器）参数指定在训练期间用于调整网络以对其输入进行建模的算法。优化器参数有很多选择，找到最好的选择通常要经过一系列试验。你可以在 Keras 文档中阅读有关优化器选择的信息。

`loss`（损失值）参数指定了训练期间用来计算网络预测值与真实值之间的距离的方法。此方法称为损失函数（loss function）。在这里，我们使用的是 `mse`，即均方误差（mean squared error）函数。回归问题中经常使用该函数，即尝试预测一个数字的情况。Keras 中提供了各种损失函数。你可以在 Keras 文档中看到一些可选的函数。

`metrics`（度量指标）参数允许我们指定一些用于评测模型表现的函数。我们指定 `mae`，即平均绝对误差（mean absolute error）函数，这是一个对于测量回归模型的表现

非常有用的函数。模型在训练期间会计算该指标，并在训练结束后将结果返回给我们。

模型编译完成后，我们可以使用以下代码来打印一些有关模型架构的摘要信息：

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

在 Colab 中运行单元格以定义模型。你将会看到以下输出：

```
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		

该表显示了网络的各个层、它们的输出形状以及它们的参数（parameter）数目。网络的大小——占用多少内存——主要取决于它的参数数量，即其权重和偏差的总数。在讨论模型的大小和复杂性时，这可能是一个很有用的度量标准。

对于这样的简单模型（且指定每个连接都有权重），可以通过计算模型中神经元之间的连接数来得到权重的数目。

我们刚刚设计的网络由两层组成。第一层有 16 个连接——每个输入和每个神经元之间都有一个连接。第二层有一个神经元，它也有 16 个连接——这个神经元到第一层的每个神经元之间都有一个连接。于是总连接数为 32。

由于每个神经元都有一个偏差，因此网络具有 17 个偏差，这意味着它总共具有 $32+17=49$ 个参数。

现在，我们已经介绍了创建模型的代码。接下来，我们将开始模型训练的过程。

4.4 训练我们的模型

在定义完我们的模型之后，是时候开始训练它，然后评估它的表现，看看效果如何了。当看到度量指标（metric）时，我们可以确定它是否已经足够好，或者是否还应该对我们的设计进行改进并再次训练。

要在 Keras 中训练模型，我们只需调用模型的 `fit()` 方法，并传入所有数据和其他一些重要参数即可。下一个单元格中的代码显示了如何进行训练：

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

运行单元格中的代码以开始训练。你会开始看到出现如下一些日志：

```
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
```

我们的模型正在训练中。这将需要一些时间，因此，在等待的同时，让我们来看看调用 `fit()` 函数的细节：

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

首先，你会注意到我们将 `fit()` 调用的返回值赋给了一个名为 `history_1` 的变量。这个变量包含大量有关我们的训练运行情况的信息，稍后我们将使用它来查看训练的情况。

接下来，让我们看一下 `fit()` 函数的参数：

`x_train` 和 `y_train`

`fit()` 的前两个参数是训练数据的 `x` 值和 `y` 值。请记住，我们将会保留部分数据用于验证和测试，因此只使用训练集来训练网络。

`epochs`

该参数指定在整个训练期间训练集通过网络运行的轮次总数。`epochs` 越大，训练次数也就越多。你可能会认为训练次数越多，网络就会越好。然而，一些网络会在特定的训练次数之后开始过拟合它们的训练数据，所以我们可能想要限制进行的训练次数。

此外，即使没有过拟合，在经过一定的训练次数后，网络也会不再改善。由于训练需要时间和计算资源，所以如果网络没有变得更好，最好不要再继续训练它！

我们从 1000 个训练轮次开始。当训练完成时，我们可以深入度量指标来看这是否是合适的训练次数。

`batch_size`

`batch_size` 参数指定在测量网络的准确率并更新其权重和偏差之前，需要向网络输入多少训练数据。如果需要，我们可以将 `batch_size` 的值设定为 1，这意味着我们将在单个数据点上运行推断，测量网络预测的损失，更新权重和偏差以使下

一次预测更准确，然后对其余数据继续此循环。

因为我们有 600 个数据点，所以每个轮次都会导致网络更新 600 次。这需要大量的计算，因此我们的训练需要很长时间！另一种方案是选择在多个数据点上进行推断，测量总体损失，然后根据损失相应地更新网络。

如果我们将 `batch_size` 设置为 600，则每个批次将包含我们的所有训练数据。现在，我们每个轮次只需要对网络进行一次更新即可，这样速度要快得多。问题是，这会导致模型的准确率降低。研究表明，使用大批量进行训练的模型推广到新数据的能力较低，也就是说它们更容易出现过拟合的问题。

折中方案是使用居中的批次大小。在我们的训练代码中，批次大小为 16。这意味着我们每次将随机选择 16 个数据点，对其进行推断、计算总损失，而且每个批次会更新一次网络。如果我们有 600 个训练数据点，则该网络将在每个轮次更新约 38 次，这比更新 600 次要好得多。

选择批次大小时，我们会在训练效率和模型准确率之间做出折中。理想的批次大小会因模型而异。但最好以 16 或 32 的批次大小开始，然后不断试验以找出最有效的批次大小。

`validation_data`

我们可以使用这个参数来指定验证数据集。来自验证数据集的数据将在整个训练过程中通过网络，并将网络的预测与预期值进行比较。我们将在日志中 `history_1` 对象的部分看到验证结果。

4.4.1 训练度量指标

希望在你读到这里的时候，训练已经结束了。如果还没有，请稍等片刻，直至训练完成。

现在，我们将检查各种指标，看看网络学得有多好。首先，让我们看一下训练期间输出的日志。这将显示网络在训练过程中是如何从随机初始状态开始一步步改进的。

以下是我们第一个和最后一个轮次的日志：

```
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
```

`loss`、`mae`、`val_loss` 和 `val_mae` 会告诉我们各种各样的信息：

`loss`

这是损失函数的输出。我们使用的是均方误差，它的值都是正数。一般来说，损失值越小越好，所以在评估网络时，输出较小的损失是一件值得关注的好事情。

比较第一个和最后一个轮次，网络在训练期间有了明显的改进，损失从大约 0.7 减少到大约 0.15。让我们看看其他的数字，看这个改进是否足够！

`mae`

这是训练数据的平均绝对误差。它显示了网络预测值与训练数据中 y 值之间的平均差。考虑到我们的初始错误是基于未经训练的网络，我们可以预期最初的误差值会很令人沮丧。的确是这样：刚开始时网络预测的平均误差大约为 0.78，考虑到 y 值是 $-1 \sim 1$ ，0.78 是一个很大的数字！

然而，即使经过训练之后，我们的平均绝对误差仍为 0.30 左右。这意味着我们的预测的平均误差大约为 0.30，这个结果仍然非常糟糕。

`val_loss`

这是损失函数在验证数据上的输出。在最后一个轮次中，训练损失值（大约 0.15）略低于验证损失值（大约 0.17）。这表明我们的网络可能出现了过拟合，因为它在处理从未见过的新数据时表现比较差。

`val_mae`

这是验证数据的平均绝对误差。它的值大约为 0.32，比训练集的平均绝对误差还差，这是网络可能过拟合的另一个迹象。

4.4.2 绘制历史数据

到目前为止，很明显，我们的模型在做出准确预测方面做得还不是很好。现在，我们的任务是弄清楚为什么模型的结果不好。为此，让我们开始利用在 `history_1` 对象中收集的数据。

下一个单元格从 `history_1` 对象中提取训练和验证损失数据，并将它们绘制在图表上：

```
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()
```

`history_1` 对象包含一个名为 `history_1.history` 的属性，它是一个字典，用于记录训练期间和验证期间的度量指标。我们用它来收集要绘制的数据。我们使用轮次作为 x 轴，然后绘制每个轮次的损失值。运行单元格，你将看到如图 4-13 所示的图。

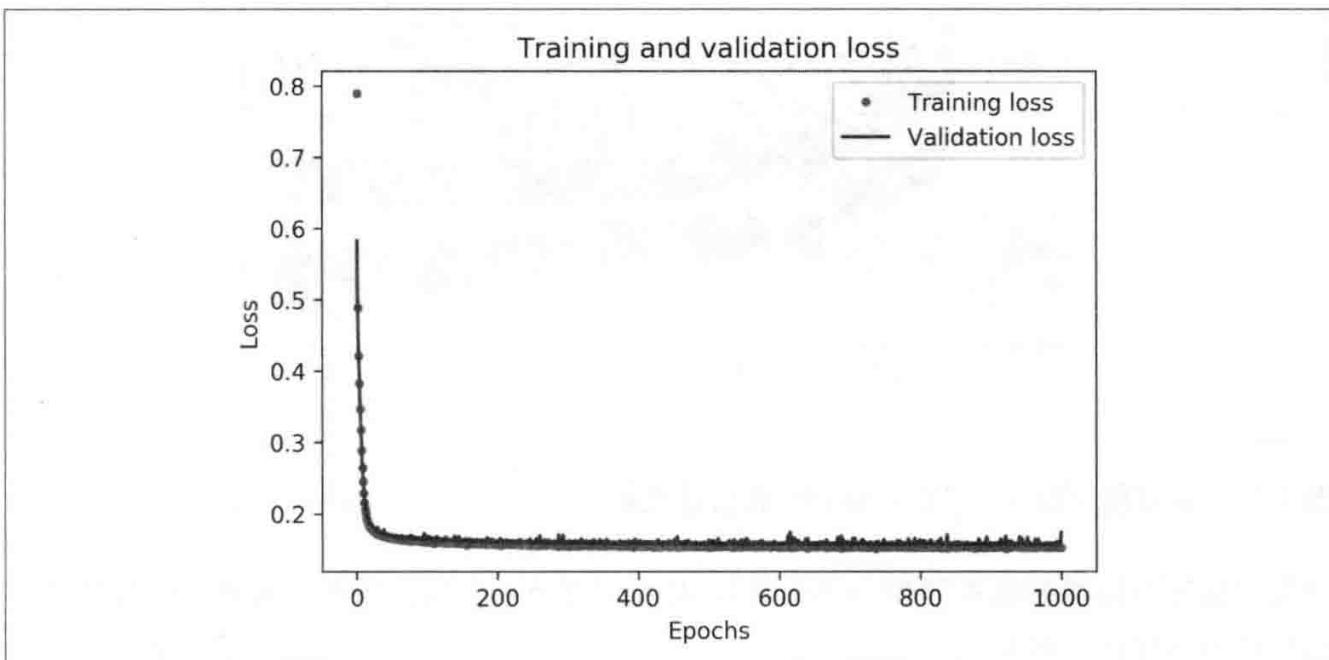


图 4-13：训练损失和验证损失

如图所示，在前 50 个轮次，损失迅速减少，然后逐渐趋于平缓。这意味着该模型正在改进并产生更准确的预测。

我们的目标是，当模型不再改进或者训练损失小于验证损失时停止训练，因为这意味着模型已经学会了很好地预测训练数据，而且不能再泛化推广到新的数据。

损失在前几个轮次急剧下降，这使得图的其余部分很难读懂。让我们通过运行下一个单元格跳过前 100 个轮次：

```
# 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()
```

图 4-14 展示了此单元格生成的图。

现在我们已经放大了后面的轮次，你可以看到损失仍在持续减小，直到大约 600 个轮次为止，从此之后，它基本上很稳定。这意味着网络可能不需要这么多的训练轮次。

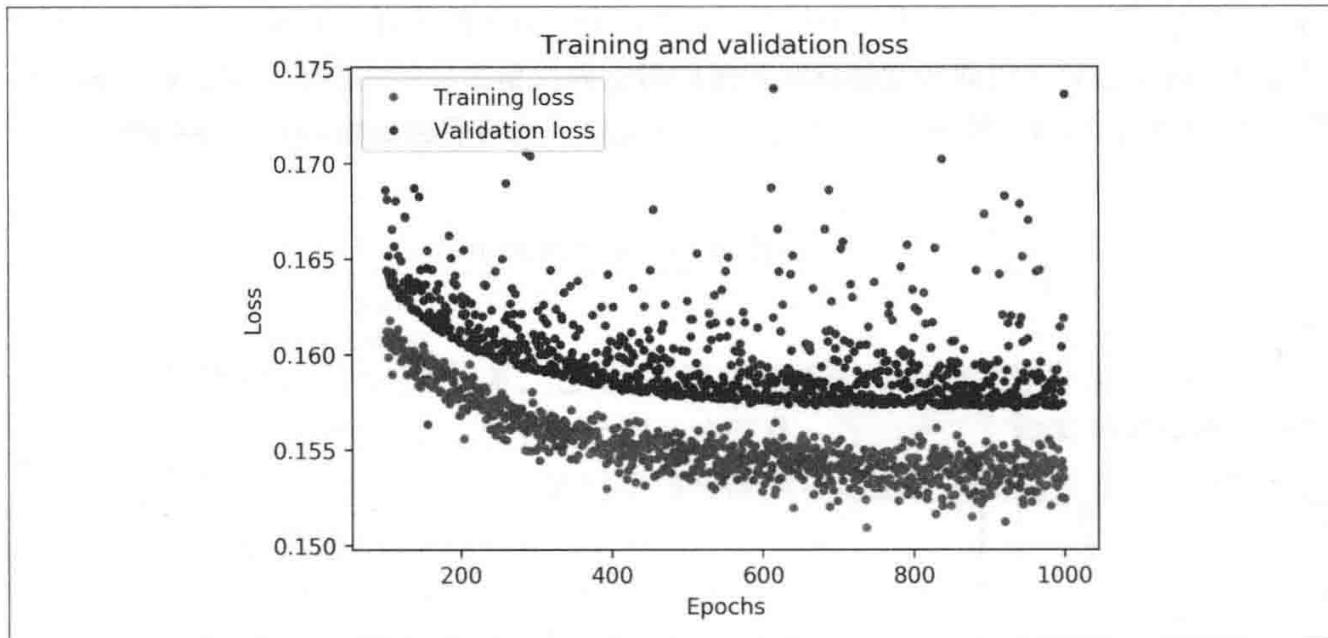


图 4-14：跳过前 100 个轮次的训练损失和验证损失

但是，你还可以看到最小的损失仍为 0.15 左右。这个损失值似乎相对较高。另外，验证损失值始终比训练损失高。

为了更深入地了解模型的表现，我们可以绘制更多的数据。这次，让我们绘制平均绝对误差。运行下一个单元格：

```
# 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()
```

图 4-15 展示了此单元格的结果。

这个平均绝对误差图为我们提供了更多进一步的线索。我们可以看到，平均而言，训练数据的误差低于验证数据的误差，这意味着网络可能存在过拟合，或者对训练数据学习过度，从而无法对新数据做出有效的预测。

此外，平均绝对误差非常高，大约为 0.31，这意味着我们模型的某些预测误差至少为 0.31。由于我们期望的输出值的范围为 $-1 \sim +1$ ，因此 0.31 的误差意味着我们离精确建模正弦波还差很远。

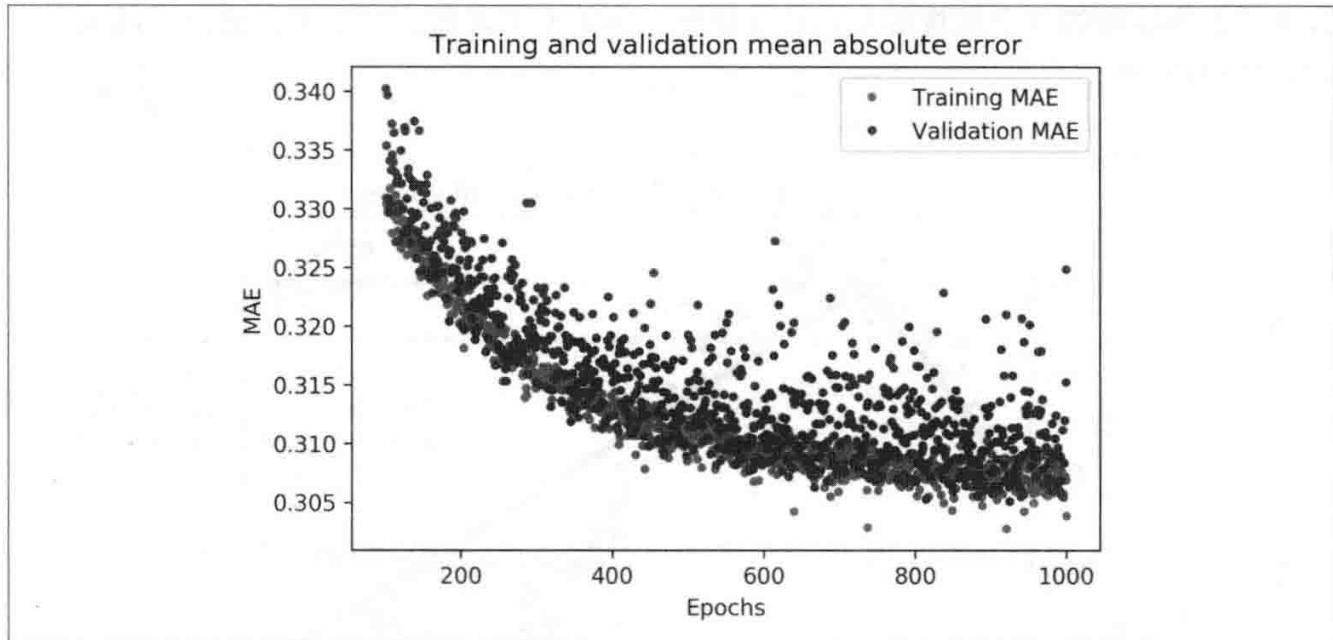


图 4-15：训练期间和验证期间的平均绝对误差

为了更深入地了解正在发生的事情，我们可以绘制网络对训练数据的预测值与期望值。运行下面这个单元格：

```
# 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()
```

通过调用 `model_1.predict(x_train)`，我们可以对训练数据中的所有 `x` 值进行推断。该方法返回一个预测数组。让我们将数组中的值与训练集中实际的 `y` 值一起绘制在图上。运行单元格以得到图 4-16。

这张图清楚地表明，我们的网络已经学会了以非常有限的方式近似正弦函数。这些预测是高度线性的，并且只能非常粗略地拟合数据。

这种僵化的拟合表明，我们的模型没有足够的能力来学习正弦波函数的全部复杂性，因此它只能以一种非常简单的方式对其进行近似。通过扩大模型，我们应该能够改善它的表现。

4.4.3 改进我们的模型

在了解到原始模型太小而无法学习数据的复杂性之后，我们可以尝试改进它。这也是机

器学习工作流程的一个常规部分：设计模型，评估它的表现，然后进行更改，以期望得到更好的模型。

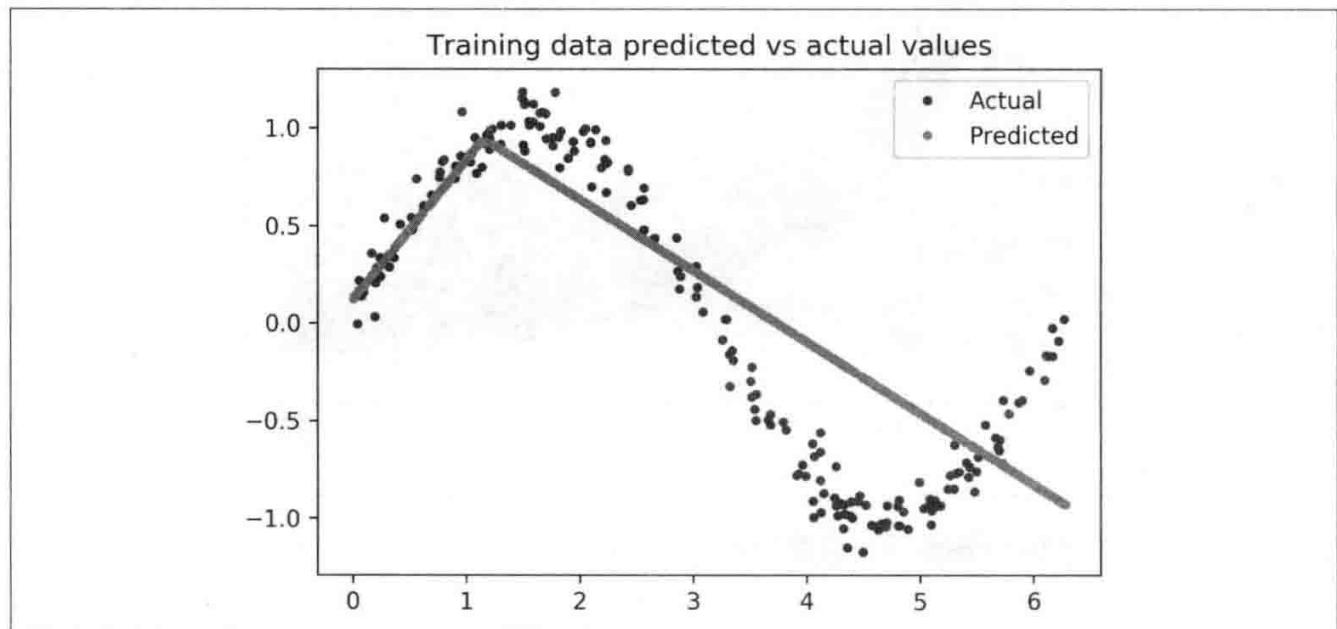


图 4-16：训练数据的预测值与实际值之间的关系图

使网络更大的一个简单方法是添加另一层神经元。神经元的每一层都代表对输入的一次转换，并希望可以使转换后的结果更接近预期的输出。网络具有的神经元层数越多，这些转换也就越复杂。

运行下面的单元格来重新定义模型，它与前面的方式相同，但这次在中间增加了一层包含 16 个神经元的层：

```
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()
```

正如你所看到的，该代码与我们的第一个模型的代码基本相同，只是增加了一个额外的 Dense 层。让我们运行单元格以查看 `summary()` 结果：

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	

由于我们的新模型有两层包含 16 个神经元的层，所以它比旧模型要大得多。它具有 $(1 \times 16) + (16 \times 16) + (16 \times 1) = 288$ 个权重，再加上 $16 + 16 + 1 = 33$ 个偏差，总共有 $288 + 33 = 321$ 个参数。我们的原始模型总共只有 49 个参数，因此模型的大小增加了 555%。希望这些额外的参数能有助于学习和表示数据的复杂性。

以下单元格将训练我们的新模型。由于我们的第一个模型很快就停止了改进，因此这次我们选择较小的 epochs——只有 600。运行此单元格开始训练：

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

训练完成后，我们可以查看最终日志，以快速了解情况是否有所改善：

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

你可以看到我们已经取得了巨大的改进——验证损失从 0.17 下降到 0.01，验证平均绝对误差从 0.32 下降到 0.08。这看起来很有希望。

为了查看进展情况，让我们运行下一个单元格。它被设置为生成与上次相同的图。首先，我们绘制损失图：

```
# 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()
```

结果如图 4-17 所示。

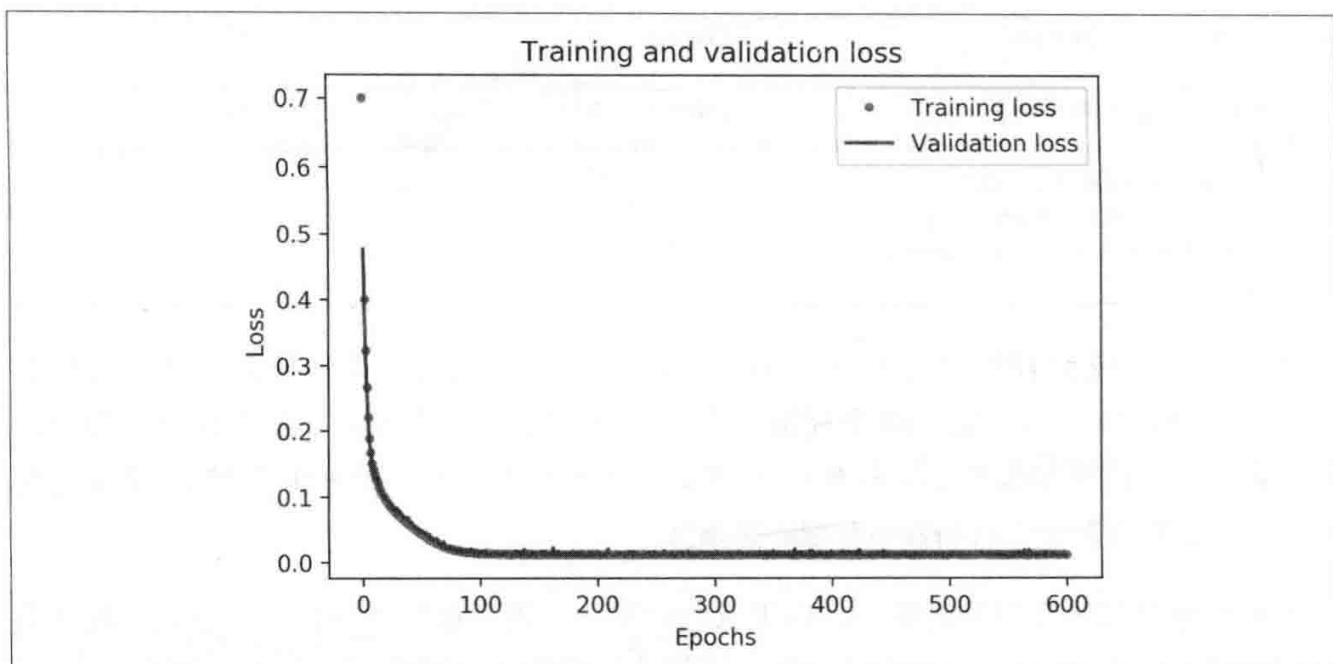


图 4-17：训练损失和验证损失

接下来，我们绘制同样的损失图，但跳过前 100 个轮次，以便更好地查看细节：

```
# 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()
```

图 4-18 给出了输出。

最后，我们对相同的轮次绘制平均绝对误差：

```
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()
```

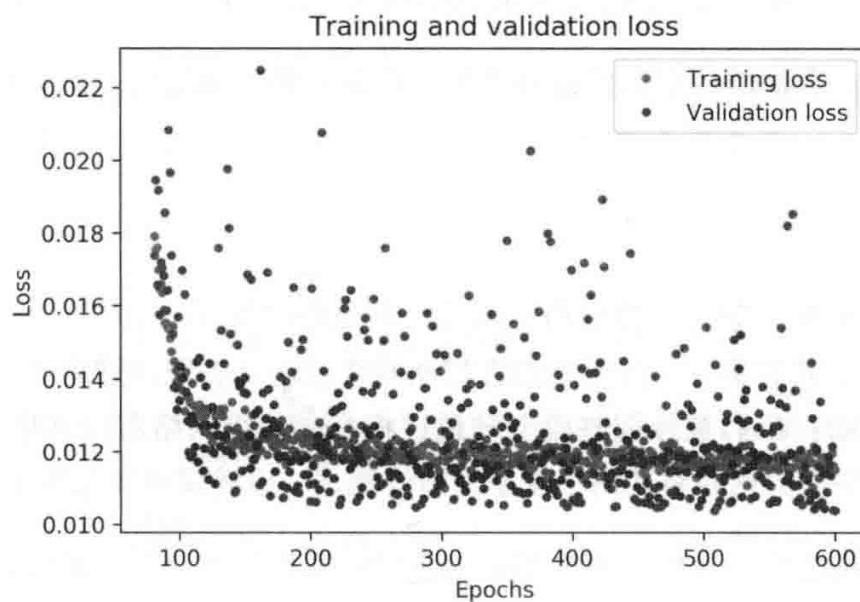


图 4-18：跳过前 100 个轮次的训练损失和验证损失

图 4-19 为单元格运行结果。

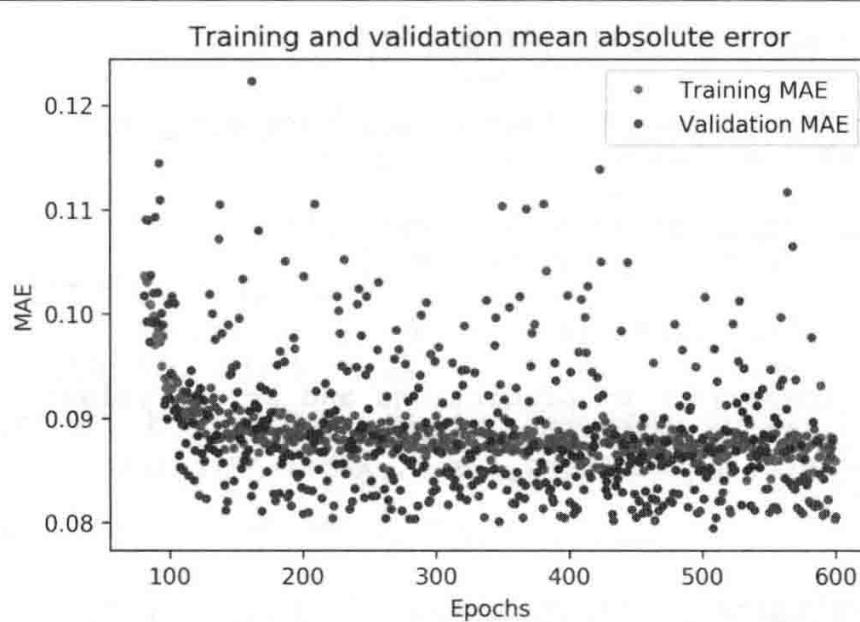


图 4-19：训练期间和验证期间的平均绝对误差

我们得到了很好的结果！从这些图中，我们可以看到两个令人兴奋的事情：

- 总体来说，验证集上的度量数据要好于训练集，这意味着网络没有出现过拟合。
- 整体损失和平均绝对误差比我们之前的网络要好得多。

你可能想知道为什么用于验证的指标要比用于训练的指标更好，而不仅仅是相等。原因是验证集的度量数据是在每个轮次结束时计算的，而训练数据的度量数据是在训练期间计算的。这意味着验证是在训练了较长时间的模型上进行的。

据验证数据来看，我们的模型似乎运行良好。但是，为了确定这一点，我们还需要运行一个最终测试。

4.4.4 测试

在此之前，我们预留了 20% 的数据用于测试。正如我们前面讨论的，拥有独立的验证集和测试集数据是非常重要的。由于我们会根据网络的验证表现对网络进行微调，因此可能存在这样的风险：我们意外调整模型导致过拟合，使得网络难以被推广到新的数据。我们可以保留一些新数据并将其用于模型的最终测试，以确保这种情况不会发生。

在使用了测试数据之后，我们需要抑制进一步调整模型的冲动。如果我们确实以提高测试表现为目标进行了更改，就可能会导致网络针对测试集过拟合。如果这样做，我们将无法知道这一点，因为已经没有可供测试的新数据了。

这意味着，如果我们的模型在测试数据上的表现不佳，那么是时候重新开始了。我们需要停止优化当前模型，并提出一种全新的架构。

考虑到这一点，以下单元格将根据测试数据评估我们的模型：

```
# 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()
```

首先，我们使用测试数据调用模型的 `evaluate()` 方法。这个方法会计算并打印损失值和平均绝对误差，告诉我们模型的预测值与实际值的偏离程度。接下来，我们进行一系列预测，并将预测值与实际值一起绘制在图上。

现在，我们可以运行该单元来了解模型的表现了！首先，让我们看一下 `evaluate()` 的结果：

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

这段输出表明我们已经评估了 200 个数据点，也就是我们的整个测试集。该模型花费了 71 微秒来对每个数据点进行预测。损失值为 0.0103，非常好，这非常接近验证损失 0.0104。我们的平均绝对误差是 0.0718，也非常小，非常接近验证集中的平均绝对误差 0.0806。

这意味着我们的模型运行良好，并且没有过拟合！如果模型过拟合了验证数据，我们可以预期测试集上的度量数据将比验证集上所产生的度量数据差很多。

我们的预测值与实际值的关系图（如图 4-20 所示）清楚地展示了模型的表现。

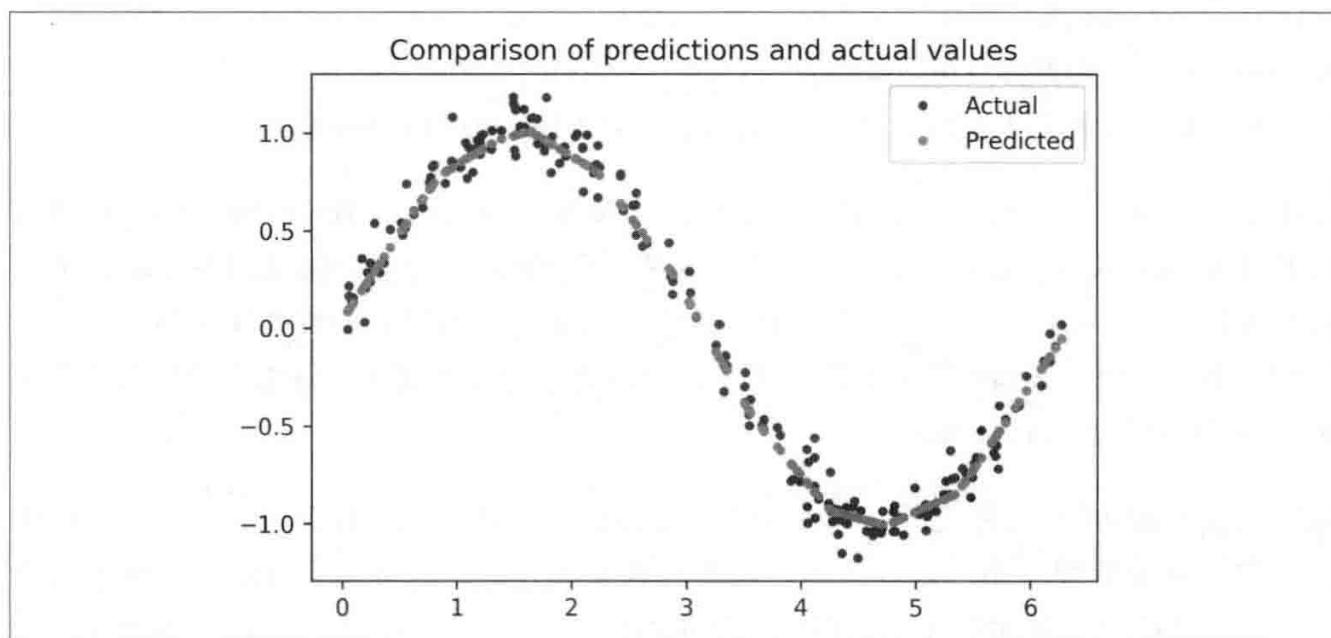


图 4-20：测试数据的预测值与实际值的关系图

你可以看到，在大多数情况下，代表预测值的点沿实际值分布的中心形成一条平滑曲线。虽然数据集含有噪声，但我们的网络还是学会了近似正弦曲线！

但是，如果仔细观察，你会发现其中还是存在一些缺陷。我们预测的正弦波的波峰和波谷并不像真实的正弦波那样完全平滑。由于训练数据是随机采样的，所以我们的模型已经学习到了数据随机分布的变化。这是一个轻微过拟合的情况：我们的模型学习的不是平滑的正弦函数，而是复制了数据的精确形状。

就我们的目的而言，这种过拟合还是一个大问题。我们的目标是使模型控制 LED 变亮或变暗，这其实并不需要完美的平滑效果。如果我们认为过拟合的程度有问题，可以尝试使用正则化技术或者获取更多的训练数据来解决。

现在，我们的模型已经足够令人满意了，让我们准备好在设备上部署它吧！

4.5 为 TensorFlow Lite 转换模型

在本章开始时，我们简要介绍了 TensorFlow Lite，这是一套用于在“边缘设备”（edge device）——从手机到微控制器——上运行 TensorFlow 模型的工具。

第 13 章会详细介绍 TensorFlow Lite for Microcontrollers。目前，我们可以先粗略地认为它主要由两部分组成：

TensorFlow Lite 转换器 (converter)

它会将 TensorFlow 模型转换为一种特殊的、更节省空间的格式，以便在内存受限制的设备上使用。它还可以通过进一步优化来减小模型大小，使其在小型设备上运行得更快。

TensorFlow Lite 解释器 (interpreter)

它以最有效的方式在特定设备上运行转换后的 TensorFlow Lite 模型。

在使用 TensorFlow Lite 之前，我们需要先转换模型。我们使用 TensorFlow Lite 转换器的 Python API 来执行此操作。它接收 Keras 模型作为输入，然后将其以 FlatBuffer 的格式写入磁盘，FlatBuffer 是一种特殊的、以节省空间为目的设计的特殊文件格式。由于我们将要把模型部署到内存有限的设备上，所以更小的模型文件会非常有用！我们将在第 13 章详细介绍 FlatBuffers。

除了创建 FlatBuffer 文件之外，TensorFlow Lite 转换器还可以对模型进行优化。这些优化通常会减少模型大小、运行时间，或者两者兼而有之。这可能会以降低准确率为代价，但是降低幅度通常很小，所以优化是值得的。你可以在第 13 章中阅读到更多有关优化的信息。

最有用的优化之一是量化（quantization）。默认情况下，模型中的权重和偏差被存储为 32 位浮点数，以便在训练过程中可以进行高精度计算。量化允许你降低这些权重和偏差的精度，以使其可以存储为 8 位整数——大小会减小为原来的 1/4。更好的是，由于 CPU 使用整数执行数学运算要比使用浮点数更容易，所以量化后模型的运行速度会更快。

关于量化的最酷的地方在于，它的精度损失程度通常非常小。这意味着在部署到低内存的设备上时，量化优化几乎总是值得的。

在下面的单元格中，我们使用转换器创建和保存模型的两个新版本。第一种转换为 TensorFlow Lite FlatBuffer 格式，但没有进行任何优化。第二个版本应用量化优化。

运行该单元以将模型转换为两个版本：

```

# 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)

```

为了创建一个尽可能高效运行的量化模型，我们需要提供一个代表性数据集（representative dataset），即一组数字，这些数字表示模型训练数据集所有输入值的范围。

在前面的单元格中，我们可以将测试数据集的 x 值用于代表性数据集。我们定义一个函数 `representative_dataset_generator()`，该函数使用 `yield` 运算符将它们逐个返回。

为了证明这些模型在转换和量化之后仍然是准确的，我们使用转换后的模型进行预测，并将结果与测试结果进行比较。考虑到这些都是 TensorFlow Lite 模型，我们需要使用 TensorFlow Lite 解释器来实现这一点。

由于 TensorFlow Lite 解释器主要是以提高效率为目的设计的，因此它使用起来比 Keras API 稍微复杂一些。要使用 Keras 模型进行预测，我们只需要调用 `predict()` 方法，并传入输入数组即可。而使用 TensorFlow Lite，我们需要执行以下操作：

1. 实例化一个 `Interpreter` 对象。
2. 调用一些为模型分配内存的方法。
3. 将输入写入输入张量。
4. 调用模型。
5. 读取输出张量的输出。

这听起来有很多工作要做，但现在先不必担心。我们将在第 5 章中详细介绍它。现在，运行以下单元格对两个模型进行预测，并将预测结果与原始未转换模型的结果一起绘制在图上：

```
# 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()
```

运行此单元格将产生图 4-21 所示的图。

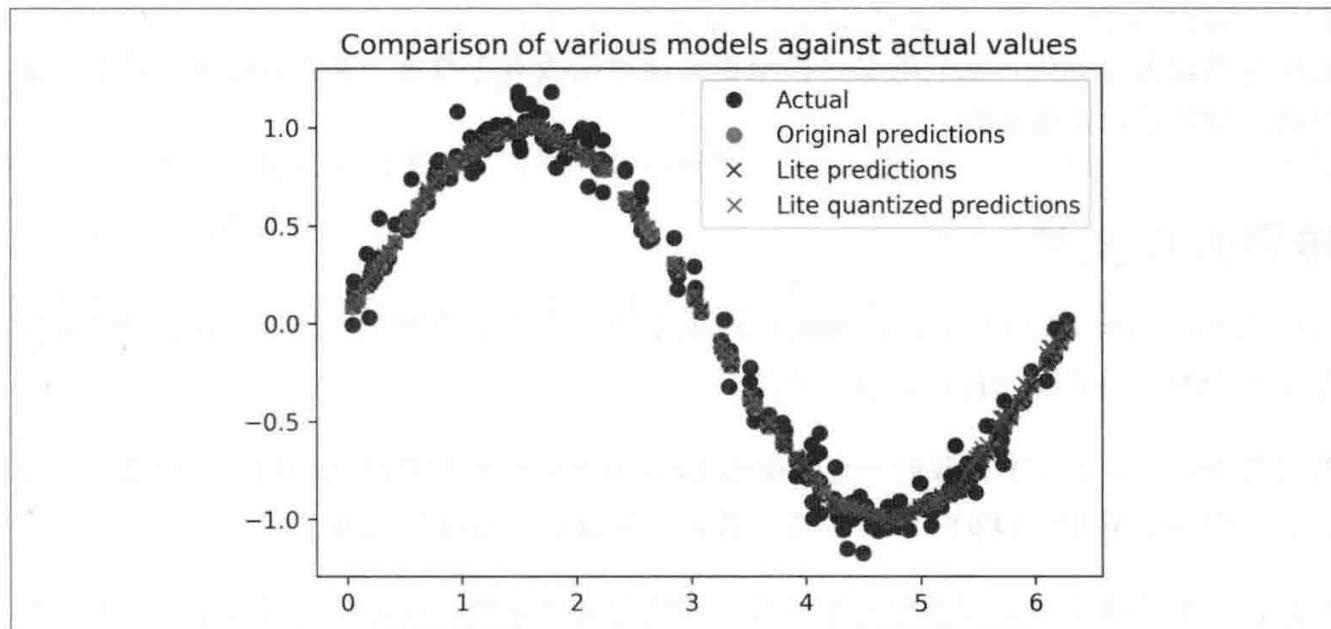


图 4-21：比较模型预测值与实际值的图

从图 4-21 可以看出，无论是原始模型、转换模型还是量化模型，预测的值都非常接近，几乎无法区分。情况看起来很不错！

由于量化会使模型变小，因此，让我们来比较两个转换后的模型，看看大小的差异。运行以下单元格以计算它们的大小并进行比较：

```
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)
```

你应该看到以下输出：

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

我们的量化模型比原始版本小 224 字节，这虽然很棒，但是实际上它的大小减少得很少。该模型的大小约为 2.4KB，这已经很小了，权重和偏差仅占整体大小的一小部分。除了权重以外，我们的模型还包含构成深度学习网络架构的所有逻辑，也就是计算图 (computation graph)。对于本身已经很小的模型而言，计算图的大小可能超过模型权重的大小，这也意味着量化的影响很小。

更加复杂的模型会有更多的权重，这也意味着量化将会节省更多的空间。对于大多数复杂的模型来说，量化预计会使大小减小为原来的约 1/4。

无论其确切大小如何，我们的量化模型都会比原始版本花费更少的执行时间，这对于微型微控制器是非常重要的。

转换为 C 文件

使用 TensorFlow Lite for Microcontrollers 的最后一个准备步骤是，将模型转换为可以包含在我们的应用程序中的 C 语言源文件。

到目前为止，在本章中，我们一直在使用 TensorFlow Lite 的 Python API。也就是说，我们已经能够使用 `Interpreter` 构造函数来从磁盘中加载模型文件。

然而，大多数微控制器都没有文件系统，即使有文件系统，从磁盘加载模型所需的额外代码也会浪费我们有限的空间。相反，作为一种更优雅的解决方案，我们直接在 C 语言源文件中提供模型，我们可以在二进制文件中直接将 C 语言文件加载到内存中。

在文件中，模型定义为字节数组。幸运的是，有一个很方便的名为 `xxd` 的 UNIX 工具，它能够很方便地将给定的文件转换成我们所需的格式。

以下单元格在我们的量化模型上运行 `xxd`，将输出写入名为 `sine_model_quantized.cc` 的文件中，并将其打印到屏幕上：

```
# 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
```

其输出非常长，因此我们不会在这里复制所有的内容，下面的代码片段只包含了开头和结尾：

```
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, 0xa, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x09,
    0x04, 0x00, 0x00, 0x00
};
unsigned int sine_model_quantized_tflite_len = 2512;
```

要在项目中使用此模型，你可以复制并粘贴源代码，或者从笔记本中下载文件。

4.6 小结

至此，我们完成了模型的构建。我们已经训练、评估和转换了 TensorFlow 深度学习网络，我们的网络可以接收 $0 \sim 2\pi$ 的数字，并且输出一个很好地逼近其正弦值的结果。

这是我们第一次尝试使用 Keras 训练一个小模型。在未来的项目中，我们将训练依旧很小但是复杂度高得多的模型。

现在，让我们进入第 5 章，在那里我们将编写代码，并在微控制器上运行我们在本章得到的模型。

TinyML 之 “Hello World”： 创建应用程序

模型只是机器学习应用程序的一部分。如果只有模型，它就只是一小段信息，什么也做不了。要使用模型，我们需要将它封装在代码中，然后为代码设置必要的运行环境，为它提供输入，并使它的输出产生行为。图 5-1 展示了模型（右侧）在一个基本的 TinyML 应用程序中扮演的角色。

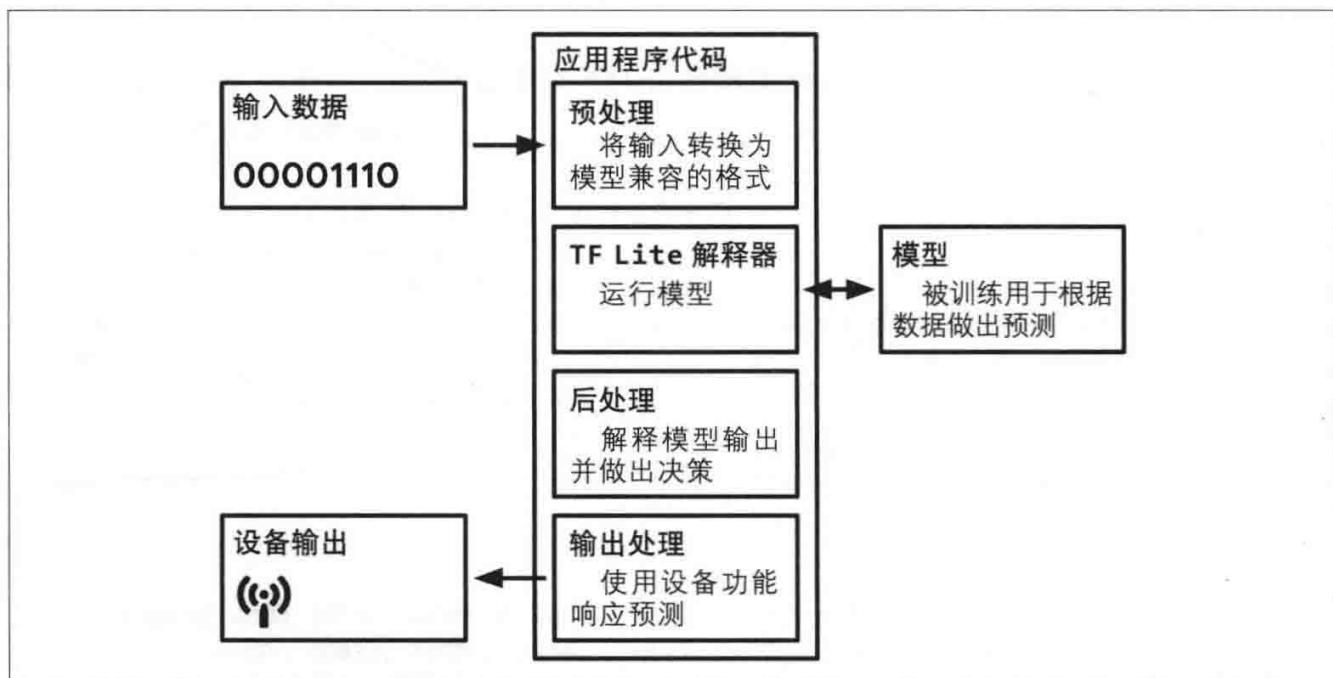


图 5-1：一个基本的 TinyML 程序架构

在本章中，我们将创建一个嵌入式应用程序，使用正弦波模型生成一个小型的灯光秀。我们将建立一个连续的循环，将 x 值输入模型中，通过模型推断的结果打开和关闭 LED，如果我们的设备具有 LCD 显示屏，我们就控制显示屏生成动画。

我们的应用程序已经写好了。这是一个 C++ 11 程序，它的代码旨在完整地展示 TinyML 应用程序的最小可能实现，所以几乎避免了任何复杂的逻辑。这种简单性使它成为我们学习如何使用 TensorFlow Lite for Microcontrollers 的有用工具，因为你可以准确地看到什么代码是必需的，而什么代码是不需要的。代码的简单性也使得它成为一个有用的模板。阅读本章后，你将了解 TensorFlow Lite for Microcontrollers 的一般结构，并且可以在自己的项目中重用该结构。

本章将逐步介绍应用程序代码，并解释它的工作原理。下一章将提供有关创建和部署应用程序到多个设备上的详细说明。如果你不熟悉 C++，请不要着急。我们的代码相对简单，而且我们将详细解释所有内容。阅读本章后，你应该会熟悉运行模型所需的所有代码，甚至可能还会在此过程中学习一些 C++ 的知识。



请记住，由于 TensorFlow 是一个正在积极开发中的开源项目，所以本书中的代码和线上托管的代码之间可能会存在一些细微的差异。即使有几行代码发生了变化，也请不要担心，代码的基本原理仍然是一样的。

5.1 详解测试

通常在弄清楚应用程序代码之前，最好编写一些测试。测试是一些简短的代码片段，用于演示特定的逻辑。由于它们由有效代码组成，因此我们可以运行它们来证明某段代码能够实现预期的功能。编写完代码后，代码通常会持续地自动运行测试，用以不断地验证——即使未来我们可能对代码进行一些更改，但它们仍在做我们期望它做的事情。另一方面，测试也可以用作某段代码的工作示例。

我们的 `hello_world` 示例也有测试，定义在 `hello_world_test.cc` 文件中。这段测试加载我们的模型并使用它运行推断，来检查推断的预测值是否符合我们的期望。它包含执行此操作所需的精确代码，除此之外没有别的内容，因此它将是学习如何使用 TensorFlow Lite for Microcontrollers 的好的开始。在本节中，我们将逐步讲解测试的各个部分。阅读完代码后，我们可以运行测试来证明它是正确的。

现在，让我们开始逐步讲解它。如果你正在计算机前，那么请打开 `hello_world_test.cc` 并且跟随本章的讲解浏览它，这可能有助于你理解代码。

5.1.1 导入依赖项

第一部分位于许可证（它指定任何人都可以在 Apache 2.0 开源许可下使用或共享此代码）下面，如下所示：

```
#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"
```

#include 指令是 C++ 代码中用于指定依赖项的一种方式。当使用 **#include** 引用代码文件时，文件中定义的任何逻辑或者变量都可以供我们使用。在本节中，我们使用 **#include** 导入以下依赖项：

tensorflow/lite/micro/examples/hello_world/sine_model_data.h

我们训练并转换得到的正弦模型，通过 `xxd` 转换为 C++ 代码。

tensorflow/lite/micro/kernels/all_ops_resolver.h

一个允许解释器加载我们的模型所需要使用的操作的类。

tensorflow/lite/micro/micro_error_reporter.h

一个可以记录并输出错误以帮助调试的类。

tensorflow/lite/micro/micro_interpreter.h

TensorFlow Lite for Microcontrollers 解释器，它会运行我们的模型。

tensorflow/lite/micro/testing/micro_test.h

用于编写测试的轻量级框架，可以帮助我们运行测试。

tensorflow/lite/schema/schema_generated.h

定义 TensorFlow Lite FlatBuffer 数据结构的 schema，用于理解 *sine_model_data.h* 中的模型数据。

tensorflow/lite/version.h

schema 当前的版本号，根据它我们可以检查模型是否使用了兼容版本的定义。

在深入研究代码时，我们将更多地讨论这些依赖项。



按照惯例，为了与 **#include** 指令一起使用，C++ 代码会被编写为两个文件：*.cc* 文件，即源文件（source file）；*.h* 文件，即头文件（header file）。头文件定义允许代码连接到程序其他部分的接口。它们包含变量和类声明之类的内容，但是逻辑很少。源文件则实现了执行计算并使代码真正工作的逻辑。当我们使用 **#include** 导入依赖项时，我们指定它的头文件。例如，我们正在进行的测试涉及 *micro_interpreter.h*。如果我们查看该文件，就会发现它定义了一个类，但没有包含太多逻辑。相反，*micro_interpreter.cc* 源文件中包含了它所有的逻辑。

5.1.2 配置测试

代码的下一部分由 TensorFlow Lite for Microcontrollers 测试框架使用。它看起来像这样：

```
TF_LITE_MICRO_TESTS_BEGIN  
TF_LITE_MICRO_TEST(LoadModelAndPerformInference) {
```

在 C++ 中，你可以定义具有特殊名字的代码块，以便在其他地方可以通过包含它们的名字来重用这些代码。这些代码块称为宏（macro）。此处的两个语句 `TF_LITE_MICRO_TESTS_BEGIN` 和 `TF_LITE_MICRO_TEST` 就是宏的名称。它们在 `micro_test.h` 文件中定义。

这些宏将其余的代码封装在必要的设备中，以便由 TensorFlow Lite for Microcontrollers 测试框架执行。我们不需要关心它的工作原理，只需要知道我们可以用这些宏快捷地配置测试即可。

第二个名为 `TF_LITE_MICRO_TEST` 的宏接收一个参数。在这种情况下，传入的参数是 `LoadModelAndPerformInference`。这个参数是测试的名称，在运行测试时，它将与测试结果一起输出，以便我们查看测试是通过还是失败。

5.1.3 准备记录数据

文件中剩余的代码是我们测试的实际逻辑。让我们来看看第一部分：

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

在第一行中，我们定义了一个 `MicroErrorReporter` 实例。`MicroErrorReporter` 类定义在 `micro_error_reporter.h` 文件中。它提供了一种在推断过程中记录调试信息的机制。我们将调用它来打印调试信息，TensorFlow Lite for Microcontrollers 的解释器将使用它来打印遇到的任何错误。



你可能已经注意到每个类型名称前面都有 `tfLite::` 前缀，比如 `tfLite::MicroErrorReporter`。`tfLite` 是一个命名空间（namespace），它只是一种用于帮助组织 C++ 代码的机制。TensorFlow Lite 在 `tfLite` 命名空间下定义了它所有有用的东西，这意味着，即使另一个库碰巧实现了同名的类，它们也不会与 TensorFlow Lite 提供的类冲突。

第一个声明看起来很简单，但是第二行带有 * 和 & 字符，看起来有些奇怪？我们已经有了 `MicroErrorReporter`，为什么还要声明一个 `ErrorReporter`？

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

为了解释这里在做什么，我们需要了解一些背景知识。

`MicroErrorReporter` 是 `ErrorReporter` 类的子类，它提供了一个模板，指定在 TensorFlow Lite 中这些调试日志记录应该如何工作。`MicroErrorReporter` 会覆盖 `ErrorReporter` 的一种方法，将其替换为专门为微控制器编写的逻辑。

在前面的代码行中，我们创建了一个名为 `error_reporter` 的变量，它的类型是 `ErrorReporter`。声明中的 * 表示它是一个指针 (pointer)。

指针是一种特殊类型的变量，它不保存值，而是保存内存中某个地址的引用，在这个地址可以找到对应的值。在 C++ 中，某个类（如 `ErrorReporter`）的指针可以指向它的一个子类（如 `MicroErrorReporter`）。

正如我们在之前提到的，`MicroErrorReporter` 会覆盖 `ErrorReporter` 的一个方法。在不深入讨论太多细节的情况下，覆盖某个方法的过程会产生一些副作用，比如隐藏某些其他方法。

为了仍然能访问 `ErrorReporter` 中没有被重写的方法，我们需要将 `MicroErrorReporter` 实例视为一个 `ErrorReporter`。我们通过创建 `ErrorReporter` 指针并将它指向 `micro_error_reporter` 变量实现这一点。在赋值过程中，`micro_error_reporter` 前面的 & 符号表示我们赋值的是指针，而不是值。

这听起来很复杂。如果你觉得很难跟上我的讲解，请不要惊慌；C++ 可能不那么灵巧，但是对于我们想要实现的目的而言，我们只需要知道，我们应该使用 `error_reporter` 来打印调试信息并且它是一个指针。

5.1.4 映射我们的模型

随后我们设置打印调试信息的机制，因为这样我们就可以记录在代码其余部分发生的任何问题。我们在下一段代码使用它：

```
// 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;
}
```

在第一行中，我们使用我们的模型数据数组（在文件 `sine_model_data.h` 中定义），并将

其传递给名为 `GetModel()` 的方法。这个方法返回一个 `Model` 指针，该指针被分配给一个名为 `model` 的变量。如你所料，这个变量就代表我们的模型。

`Model` 类型是一个结构体（struct），在 C++ 中结构体与类非常相似。它是在 `schema_genic.h` 中定义的。它保存了我们模型的数据，并允许我们查询有关模型的信息。

数据对齐

如果在 `sine_model_data.cc` 中查看模型的源文件，你就会看到 `g_sine_model_data` 的定义引用了 `DATA_ALIGN_ATTRIBUTE` 宏：

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

当数据在内存中被对齐（align）后，处理器就可以用最有效的方式读取数据，这意味着数据结构被这样存储：它们不会超过处理器单个操作能读取的内容界限。通过指定这个宏，我们可以确保在可能的情况下，模型数据被正确地对齐，以获得最佳的读取性能。如果你对这里的内容有兴趣，可以在 Wikipedia 文章中阅读更多有关对齐的信息。

当 `model` 准备好后，我们会调用下面的方法来检索模型的版本号：

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

然后，我们将模型的版本号与 `TFLITE_SCHEMA_VERSION` 进行比较，`TFLITE_SCHEMA_VERSION` 表示我们当前正在使用的 TensorFlow Lite 库的版本。如果版本匹配，我们的模型将使用兼容版本的 TensorFlow Lite 转换器进行转换。检查模型版本是非常推荐的做法，因为版本不匹配可能会导致难以调试的奇怪行为。



在前面的代码行中，`version()` 是属于 `model` 的方法。注意从 `model` 指向 `version()` 的箭头（`->`）。这是 C++ 的箭头运算符（arrow operator），我们使用它访问一个指针对象的成员。如果有对象本身（而不仅仅是指针），我们就应该使用点（`.`）来访问它的成员。

如果版本号不匹配，我们仍然会继续进行，但是我们会使用 `error_reporter` 来打印警告：

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

我们调用 `error_reporter` 的 `Report()` 方法来记录这条警告。由于 `error_reporter`

也是一个指针，因此我们还是使用 `->` 运算符来访问 `Report()`。

`Report()` 方法的设计与常用的 C++ 方法 `printf()` 颇为相似，后者用于打印文本。作为它第一个参数，我们传入一个要记录的字符串。该字符串包含两个 `%d` 格式说明符，它们充当占位符，在打印消息时将会被插入的变量替换。接下来传入的两个参数是模型版本号和 TensorFlow Lite schema 版本号。这些将被插入字符串中以替换 `%d` 占位符。



`Report()` 方法支持用作不同类型变量的格式说明符。`%d` 是整数的占位符，`%f` 是浮点数的占位符，`%s` 是字符串的占位符。

5.1.5 创建一个 AllOpsResolver

到目前为止，一切都还很顺利！我们的代码可以记录错误，我们已经将模型加载到一个方便的结构中，并且检查了它的版本是否兼容。考虑到我们一直在回顾一些 C++ 的概念，我们的进度有些缓慢，但是不要急，事情马上就会变得有意义了。接下来，我们将创建 `AllOpsResolver` 类的一个实例：

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

`AllOpsResolver` 类定义在 `all_ops_resolver.h` 中，它允许 TensorFlow Lite for Microcontrollers 解释器访问各种可用的算子（operation）。

在第 3 章中，你了解了机器学习模型由各种数学运算组成，这些数学运算依次运行，以将输入转换为输出。`AllOpsResolver` 类包含所有 TensorFlow Lite for Microcontrollers 可用算子，并将它们提供给解释器。

5.1.6 定义一个 Tensor Arena

我们几乎准备好了创建一个解释器所需的所有内容。最后一件需要做的事情就是分配一块工作内存来运行我们的模型：

```
// 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];
```

就像注释中所说的那样，这个内存区域将用于存储模型的输入、输出和中间张量。我们称它为 tensor arena。在我们的例子中，我们分配了一个大小为 2048 字节的数组。我们通过表达式 2×1024 来指定大小。

那么，我们的 tensor arena 应该有多大？这是个好问题。不幸的是，它并没有一个简单

的答案。对于不同架构的模型，它们的输入、输出和中间张量的大小和数量都不相同，因此很难知道我们到底需要多少内存。不过这里并不需要一个完全精确的数字——我们可以保留比实际需求多的内存，但是由于微控制器的 RAM 有限，我们还是应该尽可能地使它小一些，以便为程序的其余部分留出空间。

我们可以通过反复试验来找到一个相对合适的大小。这就是为什么我们将数组大小表示为 $n \times 1024$ ：以便轻松地（通过更改 n ）按比例放大或者缩小内存空间，同时保证它是 8 的倍数。要找到正确的数组大小，请从较大的数字开始，以确保它可以正常工作。本书示例中使用的最大数字是 70×1024 。然后，逐渐减少它直到模型不能运行。最后一个可以有效工作的数字就是我们想要的！

5.1.7 创建一个解释器

现在我们已经声明了 `tensor_arena`，然后就可以开始设置解释器了。它看起来像这样：

```
// 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();
```

首先，我们声明一个名为 `interpreter` 的 `MicroInterpreter`。这个类是 TensorFlow Lite for Microcontrollers 的核心：一段神奇的、将在我们提供的数据上执行模型的代码。我们把到目前为止创建的大多数对象都传递给它的构造函数，然后调用 `AllocateTensors()`。

在上一节中，我们通过定义一个名为 `tensor_arena` 的数组来预留内存区域。`AllocateTensors()` 方法会遍历模型定义的所有张量，并将 `tensor_arena` 中的内存分配给每个张量。在尝试进行推断之前，先调用 `AllocateTensors()` 非常重要，因为不调用这个函数会导致推断失败。

5.1.8 检查输入张量

在创建解释器之后，我们需要为模型提供一些输入。为此，我们把输入数据写到模型的输入张量中：

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

为了获取指向输入张量的指针，我们调用了 `interpreter` 的 `input()` 方法。由于一

个模型可以有多个输入张量，因此我们需要向 `input()` 方法传递一个索引，用以指定所需的张量。对于我们的情况，模型只有一个输入张量，因此它的索引是 0。

在 TensorFlow Lite 中，张量由 `TfLiteTensor` 结构体表示，这个体构体定义在 `c_api_internal.h` 中。它提供了用于与张量交互和了解张量的 API。在下一部分代码中，我们使用它来验证我们的张量是否正确。由于我们会经常用到张量，因此让我们浏览一遍下面的代码，以熟悉 `TfLiteTensor` 结构体是如何工作的：

```
// 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);
```

你首先会注意两个宏：`TF_LITE_MICRO_EXPECT_NE` 和 `TF_LITE_MICRO_EXPECT_EQ`。这些宏是 TensorFlow Lite for Microcontrollers 测试框架的一部分，让我们能够断言变量的值，证明它们等于我们期望的值。

例如，宏 `TF_LITE_MICRO_EXPECT_NE` 设计为断言调用它的两个变量不相等。因此，它的名称中 `_NE` 部分代表不相等（Not Equal）。如果变量不相等，代码将继续执行。如果它们相等，将记录错误，并且标记测试失败。

更多的断言

`micro_test.h` 中定义了各种断言的宏，你可以读取文件来查看它们是如何工作的。以下是可用的断言：

`TF_LITE_MICRO_EXPECT(x)`

断言 `x` 的计算结果为 `true`。

`TF_LITE_MICRO_EXPECT_EQ(x, y)`

断言 `x` 等于 `y`。

`TF_LITE_MICRO_EXPECT_NE(x, y)`

断言 `x` 不等于 `y`。

```
TF_LITE_MICRO_EXPECT_NEAR(x, y, epsilon)
```

对于数值输入，断言 x 和 y 之间的差值小于或等于 ϵ 。例如，`TF_LITE_MICRO_EXPECT_NEAR(5, 7, 3)` 会通过测试，因为 5 和 7 的差是 2。

```
TF_LITE_MICRO_EXPECT_GT(x, y)
```

对于数值输入，断言 x 大于 y 。

```
TF_LITE_MICRO_EXPECT_LT(x, y)
```

对于数值输入，断言 x 小于 y 。

```
TF_LITE_MICRO_EXPECT_GE(x, y)
```

对于数值输入，断言 x 大于或等于 y 。

```
TF_LITE_MICRO_EXPECT_LE(x, y)
```

对于数值输入，断言 x 小于或等于 y 。

我们要检查的第一件事是输入张量是否真的存在。为此，我们断言它不等于 `nullptr`，这是一个特殊的 C++ 值，表示一个指针实际上并未指向任何数据：

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

接下来我们要检查的是输入张量的形状。如第 3 章中所讨论的，所有张量都有一个形状，用来描述张量的维数。我们模型的输入是一个标量（表示单个数字）。但是，限于 Keras 层可以接受的输入形式，我们的输入必须包含在一个二维张量中提供给 Keras。比如，输入为 0 应该表示成这样：

```
[[0]]
```

请注意输入标量 0 如何被包裹在两个向量中，从而使其成为二维张量。

`TfLiteTensor` 结构体包含一个 `dims` 成员，用于描述张量的位数。`dims` 是一个 `TfLiteIntArray` 类型的结构体，`TfLiteIntArray` 也定义在 `c_api_internal.h` 中。它的大小代表张量的维数。由于输入张量应为二维，因此我们可以断言 `size` 的值等于 2：

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

我们可以进一步检查 `dims` 结构体，以确保张量的结构符合我们的预期。它的 `data` 变量是一个数组，每个维度都有一个元素。每个元素都是一个整数，代表这个维度的大小。因为我们期望的是一个二维张量，且在每个维度上都包含一个元素，所以我们可以断言两个维度都包含一个元素：

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

现在我们可以确信我们的输入张量具有正确的形状。最后，由于张量可以包含各种不同类型的数据（比如整数、浮点数和布尔值），因此我们应确保输入张量的类型是正确的。

张量结构体的 `type` 变量会告诉我们张量的数据类型。我们将提供一个 32 位浮点数类型，由常量 `kTfLiteFloat32` 表示，我们可以很容易地断言该类型正确：

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

完美！现在，我们确保了输入张量具有正确的大小和形状——它是单个浮点值。我们已经准备好进行推断了！

5.1.9 对输入运行推断

为了运行推断，我们需要向输入张量中添加一个值，然后指示解释器来调用模型。随后，我们将检查模型是否成功运行。代码如下：

```
// 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 的 `TfLiteTensor` 结构体有一个变量 `data`，我们可以用它来设置输入张量的内容。其用法如下：

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

`data` 变量是一个 `TfLitePtrUnion`，它是一个共用体（union），这是一种特殊的 C++ 数据类型，它允许在内存中的同一位置存储不同类型的数据。由于给定的张量可以包含多种不同类型的数据（如浮点数、整数或布尔值），所以共用体是帮助我们存储它的最理想的类型。

`TfLitePtrUnion` 声明在 `c_api_internal.h` 中。它看起来是这样的：

```
// 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;
```

你会看到有一群成员，每个成员代表一个特定的类型。每个成员都是一个指针，可以指向内存中应存储这种数据的位置。当我们像以前一样调用 `interpreter.AllocateTensors()` 时，指针会被赋值为指向为张量分配的内存块。因为每个张量都有特定的数据类型，所以只有对应类型的指针会被设置。

这意味着要存储数据，我们可以使用 `TfLitePtrUnion` 中适当的指针。例如，如果张量的类型是 `kTfLiteFloat32`，我们将使用 `data.f`。

由于指针指向一个内存块，因此我们可以在指针名称后使用方括号（`[]`）来指示程序数据存储在何处。在我们的示例中，我们执行以下操作：

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

我们写入的值为 `0.`，也就是 `0.0` 的简写。通过指定小数点，我们明确地告诉 C++ 编译器这个值为浮点数，而不是整数。

你可以看到我们将此值分配给了 `data.f[0]`。这意味着我们将它分配给已分配内存块中的第一项。由于我们只有一个值，所以这就是全部要做的。

更复杂的输入

在正在进行的示例中，我们的模型接受的输入是一个标量，因此我们只需要分配一个值 (`input->data.f[0] = 0.`)。如果模型的输入是由多个值组成的向量，那么我们可以将它们添加到后续的存储位置。

这是一个包含数字 1、2 和 3 的向量的示例：

```
[1 2 3]
```

在 `TfLiteTensor` 中，我们可以这样设置这些值：

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

但如果是由多个向量组成的矩阵呢？这里有一个例子：

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

要把它放入 `TfLiteTensor`，我们只需按顺序分配值：从左到右，从上到下。由于我们把结构从二维压缩为一维，所以这种方式也称为展平（flattening）：

```
// 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.;
```

由于 `TfLiteTensor` 结构体记录了张量的维度，所以即使内存内的数据是平面的，它也知道内存中的某个位置对应于多维形状中的哪个元素。在后面的章节中，我们将使用二维输入张量来输入图像和其他二维数据。

当设置好输入张量之后，就可以进行推断了。这只有一行代码：

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

当我们通过解释器调用 `Invoke()` 时，TensorFlow Lite 解释器将运行模型。模型包含一个由数学操作组成的计算图，解释器执行这些计算图以将输入数据转换为输出。模型的输出存储在模型的输出张量中，稍后我们将深入讨论它。

`Invoke()` 方法返回一个 `TfLiteStatus` 对象，这个对象告诉我们推断成功或存在问题。它的值是 `kTfLiteOk` 或 `kTfLiteError`。我们检查它的值，如果存在错误，就将错误打印出来：

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

最后，我们断言状态必须为 `kTfLiteOk` 才能使测试通过：

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

就是这样——推断已经成功运行！接下来，我们将获取输出并确保它看起来不错。

5.1.10 读取输出

就像访问输入一样，我们也可以通过 `TfLiteTensor` 访问模型的输出，获得一个指向它的指针也很简单：

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

就像输入一样，输出也是一个嵌套在二维张量中的浮点数标量。为了进行测试，我们再次检查输出张量是否具有预期的大小、尺寸和类型：

```
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);
```

是的，一切看起来都还不错。现在，我们获取输出值并检查它，以确保它符合我们预期的标准。首先，我们将其分配给一个浮点数变量：

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

每次运行推断时，输出张量都会被新的值覆盖。这意味着，如果想在继续运行推理的同时在程序中保留一个输出值，你就需要像我们刚才做的那样，从输出张量中复制它。

接下来，我们使用 `TF_LITE_MICRO_EXPECT_NEAR` 宏来证明存储的值接近我们期望的值：

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

正如我们在前面看到的，`TF_LITE_MICRO_EXPECT_NEAR` 断言它的第一个参数和第二个参数之间的差小于第三个参数的值。所以上面这个语句中，我们正在测试输出是否在 -0.05 和 0.05 之间，其中点 0 是 0 的正弦值。



我们期望一个数字接近 (near) 我们想要的数字，而不是期望它完全等于一个确切的值，其中有两个原因：第一，我们的模型只是近似 (approximate) 真实的正弦函数，因此我们知道它可能不会输出完全正确的正弦值；第二，计算机上的浮点计算存在一定的误差。这个误差可能因计算机而异：例如，笔记本电脑的 CPU 的结果与 Arduino 的结果可能略有不同。通过设置有一定灵活性的期望值，可以使我们的测试更有可能在任何平台上都能通过。

如果这个测试通过了，那么一切看起来都很好。其余的测试会多次进行推断，以进一步证明我们的模型是有效的。为了再次运行推断，我们要做的是给输入张量赋一个新的值，调用 `interpreter.Invoke()`，然后从输出张量中读取输出即可：

```
// 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);
```

注意我们是如何重用相同的输入和输出张量指针的。由于我们已经有了指针，所以不需要再次调用 `interpreter.input(o)` 或者 `interpreter.output(o)`。

至此，我们已经在测试中证明了 TensorFlow Lite for Microcontrollers 可以成功地加载我们的模型、分配适当的输入和输出张量、运行推断并返回预期结果。最后要做的是使用宏指示测试的结束：

```
}
```

```
TF_LITE_MICRO_TESTS_END
```

这样，我们完成了全部测试。接下来，让我们来运行这些测试！

5.1.11 运行测试

虽然这些代码最终要运行在指定的微控制器上，但我们仍然可以在开发计算机上创建和运行这些测试。这会使编写和调试代码更加容易。与微控制器相比，个人计算机有更方便的工具来记录输出，而且可以逐步执行代码，这也能够帮助我们更方便地查找任何错误。此外，将代码部署到设备上需要一些时间，而在本地运行代码就要快得多。

创建嵌入式应用程序（或者说，任何嵌入式软件）的一个很好的工作流程是，将尽可能多的逻辑编写在可以在普通开发机器上测试的部分中。当然，终究有一些代码需要实际的硬件才能运行，但是你在本地测试得越多，你的工作就会越轻松。

实际上，这也意味着我们应尝试先编写代码，对输入进行预处理，对模型运行推断，然后在一组测试中处理所有输出，最后再尝试在设备上运行。在第 7 章中，我们将介绍一个比本章的示例复杂得多的语音识别应用程序。你将在那里看到我们如何为每个组件编写详细的单元测试。

获取代码

到目前为止，在 Colab 和 GitHub 之间，我们一直在云上进行所有操作。要运行我们的测试，我们需要将代码下载到开发计算机中，并且编译这些代码。

为此，我们需要安装以下软件：

- 终端模拟器，例如 macOS 中的终端。
- bash shell（默认安装在 macOS Catalina 的之前版本和大多数 Linux 发行版中）。
- Git（默认安装在 macOS 和大多数 Linux 发行版中）。
- 3.82 或者更高版本的 Make。

Git 和 Make

Git 和 Make 通常安装在现代操作系统上。要检查系统里是否安装了它们，请打开终端并执行以下操作：

关于 Git

任何版本的 Git 都可以使用。要确认已安装 Git，请在命令行中输入 `git`。你应该看到打印出的使用说明。

关于 Make

要检查已安装的 Make 的版本，请在命令行中输入 `make --version`。你需要 3.82 以上的版本。

如果缺少这两种工具的任何一个，你应该在网上搜索有关如何在特定操作系统中安装它们的说明。

一旦确认已有了所有的工具，你就可打开终端并输入以下命令来下载 TensorFlow 源代码，其中包括我们正在使用的示例代码。它将在你运行的位置创建一个包含源代码的目录：

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

接下来，切换到刚刚创建的 `tensorflow` 目录：

```
cd tensorflow
```

很好，我们现在可以运行一些代码了！

使用 Make 运行测试

正如你在我们的工具列表中所看到的，我们使用一个名为 *Make* 的程序来运行测试。Make 是用于在软件中自动执行创建任务的工具。它自 1976 年以来一直在被使用，从计算历史的角度来说，这几乎等同于“永远”。开发人员在名为 *Makefiles* 的文件中使用一种特殊的语言指示 Make 如何构建和运行代码。TensorFlow Lite for Microcontrollers 在 `micro/tools/make/Makefile` 中定义了一个 Makefile，在第 13 章中有更多关于它的信息。

要使用 Make 运行我们的测试，可以使用以下命令，确保在使用 Git 下载的 *tensorflow* 目录的根目录中运行它。我们首先指定要使用的 Makefile，然后指定目标（target），也就是我们要创建的组件：

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

设置 Makefile 是为了运行测试，我们提供了一个前缀为 `test_` 的目标，随后是我要创建的组件的名称。在我们的例子中，该组件是 `hello_world_test`，因此完整的目标名为 `test_hello_world_test`。

尝试运行此命令。你应该开始看到大量输出掠过！首先会下载一些必要的库和工具。接下来，将创建我们的测试文件及其所有依赖项。我们的 Makefile 会指示 C++ 编译器编译代码并创建二进制文件，然后运行它。

你需要几分钟等待这个过程完成。当停止打印文本时，最后几行应该是这样的：

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

不错！这个输出表明我们的测试如期通过。你可以在源文件的顶部看到测试名：`LoadModelAndPerformInference`。虽然还没有部署到微控制器上，但至少表明我们的代码可以成功地运行推断。

如果想看看测试失败时会发生什么，我们可以引入一个错误。打开测试文件 `hello_world_test.cc`，相对于根目录，它位于以下路径：

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

为了使测试失败，我们将为模型提供一个不同的输入。这将导致模型的输出发生变化，进而导致检查输出值的断言失败。查找以下行：

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

然后更改赋值，如下所示：

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

现在，保存文件，然后使用以下命令再次运行测试（记住要从 *tensorflow* 目录的根目录执行此操作）：

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

代码将被重新编译，并运行测试。你看到的最终输出应该是这样的：

```
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~~~
```

输出包含一些关于测试失败原因的有用信息，包括失败发生所在的文件和行号 (`hello_world_test.cc:94`)。如果错误是由真正的漏洞引起的，那么这里的输出会帮助你跟踪和查找问题。

5.2 项目文件结构

借助于我们的测试，你已经了解了如何使用 TensorFlow Lite for Microcontrollers 库在 C++ 中运行推断。接下来，我们将逐步解释实际应用程序的源代码。

如我们之前所讲的，我们正在构建的程序由一个连续的循环组成，这个循环将 x 值输入模型中，进行推断，并针对不同的平台将结果转换成某种可视化的输出（如 LED 的闪烁模式）。

由于该应用程序很复杂且分散在多个文件中，因此让我们看一下它的文件结构，以及这些文件是如何组合到一起的。

应用程序的根目录位于 `tensorflow/lite/micro/examples/hello_world`。它包含以下文件：

BUILD

该文件列出了可以通过应用程序的源代码编译的各种内容，包括主应用程序二进制文件和我们之前运行的测试。目前，我们还不需要太关心它的内容。

Makefile.inc

一个 `Makefile`，其中包含有关我们应用程序的编译目标的信息，包括 `hello_world_test`（我们之前运行的测试）和 `hello_world`（主应用程序二进制文件）。它定义了哪些源文件是这些编译目标的一部分。

README.md

包含有关编译和运行应用程序说明的自述文件。

constants.h 和 *constants.cc*

一对包含各种常量（constant）（在程序生命周期中不会改变的变量）的文件，这些常量对于定义程序的行为非常重要。

create_sine_model.ipynb

上一章中使用的 Jupyter 笔记本。

hello_world_test.cc

使用我们的模型进行推断的测试。

main.cc

程序的入口点，在应用程序部署到设备时，它会最先开始运行。

main_functions.h 和 *main_functions.cc*

一对定义了 `setup()` 函数和 `loop()` 函数的文件。`setup()` 函数会执行程序所需的所有初始化操作；`loop()` 函数包含程序的核心逻辑，并设计为在循环中重复调用。程序启动时，*main.cc* 会调用这些函数。

output_handler.h 和 *output_handler.cc*

一对定义输出函数的文件，在每次运行推断时，我们使用这个函数来显示输出。*output_handler.cc* 中的默认实现会将结果打印到屏幕上。我们可以重写它的实现，使它在不同的设备上执行不同的操作。

output_handler_test.cc

用以验证 *output_handler.h* 和 *output_handler.cc* 中的代码可以正常工作的测试文件。

sine_model_data.h 和 *sine_model_data.cc*

一对定义了代表我们模型的数据数组的文件，即本章第一部分中使用 `xxd` 导出的文件。

除了这些文件之外，目录中还包含了以下子目录（可能还有更多）：

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

由于不同的微控制器平台具有不同的功能和 API，因此我们的项目结构允许我们提供针对不同设备的、不同版本的源文件，如果应用程序是针对某个设备编译的，它就会使用特定设备版本的源文件。例如，*arduino* 目录包含 *main.cc*、*constants.cc* 和 *output_handler.cc* 的定制版本，这些源文件会定制应用程序，以保证其可以与 Arduino 一起工作。我们稍后将深入讨论这些定制实现。

5.3 详解源文件

现在，我们知道应用程序源文件的组织结构，接下来让我们深入研究代码。我们将从 *main_functions.cc* 开始——这就是奇迹发生的地方，然后从这个文件再分支到其他文件中。



很多代码看起来都和我们之前在 `hello_world_test.cc` 中看到的有些相似。如果某些内容我们已经讲过了，就不会再深入地探讨它的工作方式。我们宁愿把重点放在你从未见过的代码上面。

5.3.1 从 `main_functions.cc` 开始

这个文件包含我们程序的核心逻辑。它是这样开始的，包含一些熟悉的和一些新出现的 `#include` 语句：

```
#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"
```

很多 `#include` 语句在 `hello_world_test.cc` 中已经出现过。新出现的是 `constants.h` 和 `output_handler.h`，我们在之前的文件列表中大概了解了它们的含义。

文件的下一部分定义了将在 `main_functions.cc` 中使用的全局变量：

```
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
```

你一定注意到这些变量被定义在命名空间中。这意味着可以在 `main_functions.cc` 中的任何位置访问这些变量，却无法在项目中的其他文件中访问它们。如果两个不同的文件碰巧定义了同名的变量，这将有助于防止同名引用的问题发生。

所有这些变量你应该都很熟悉，我们在测试中见过它们。我们设置变量来保存所有熟悉的 TensorFlow 对象以及 `tensor_arena`。唯一的新事物是一个拥有 `inference_count` 的 `int`，我们用它跟踪程序执行了多少次推断。

文件的下一部分声明一个名为 `setup()` 的函数。程序在第一次启动时会调用此函数，

且只调用一次。我们使用它来执行在开始进行推理之前需要执行的所有一次性工作。

`setup()`的第一部分与我们的测试几乎完全相同：设置日志记录、加载模型、设置解释器，并分配内存：

```
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;
    }
}
```

到目前为止都是熟悉的内容。不过在这之后，情况就有所不同了。首先，我们获取同时指向输入和输出张量的指针：

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

你可能会疑惑，为什么我们可以在运行推断之前访问输出？好吧，请记住，`TfLiteTensor`只是一个结构体，它有一个成员 `data` 指向分配给存储输出的内存区域。即使尚未写入任何输出，这个结构体和它的数据成员仍然存在。

最后，为了结束 `setup()` 函数，我们将 `inference_count` 变量设置为 0：

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

至此，我们所有的机器学习基础架构都已经创建好，而且准备就绪。我们已经有了运行推断并获得结果所需的所有工具。接下来要定义的是应用程序逻辑。这个程序到底要做什么？

我们的模型被训练来预测从 0 到 2π （正弦波的整个周期）之间任意数字的正弦。为了演示我们的模型，我们可以输入范围内的数字，预测它的正弦值，然后以某种方式输出这个值。我们可以按顺序执行此操作，以此显示模型在整个范围内都可以工作。这听起来是个好计划！

为此，我们需要编写一些运行在循环中的代码。首先，我们声明一个名为 `loop()` 的函数，接下来我们将遍历整个函数。这个函数中的代码将会一遍又一遍地运行：

```
void loop() {
```

在 `loop()` 函数中，我们首先要确定将要被输入模型的值（我们称其为 `x` 值）。我们使用两个常量来确定这一点：`kXrange` 和 `kInferencesPerCycle`。`kXrange` 将 `x` 值的最大值指定为 2π ，而 `kInferencesPerCycle` 定义从 0 到 2π 之间要执行的推断次数。接下来的几行代码计算 `x` 值：

```
// 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;
```

前两行代码仅将 `inference_count`（我们到目前为止完成的推断次数）除以 `kInferencesPerCycle`，来获得该范围内的当前“位置”。下一行将该值乘以 `kXrange`，它表示范围（ $0 \sim 2\pi$ ）中的最大值。得到的结果 `x_val` 就是我们要输入到模型中的值。



`static_cast<float>()` 用于将均为整数的 `inference_count` 和 `kInferencesPerCycle` 转换为浮点数。我们这样做是为了正确地执行除法。在 C++ 中，如果将两个整数相除，那么得到结果仍为整数，结果中的任何小数部分都将被丢弃。由于我们希望 `x` 值是一个包含小数部分的浮点数，所以我们需要将这些整数转换为浮点数。

我们使用的两个常量 `kInferencesPerCycle` 和 `kXrange` 定义在文件 `constants.h` 和 `constants.cc` 中。按照 C++ 的约定，常量名称以 `k` 开头，这样当在代码中使用它们时，会很容易认出它们是常量。在一个独立的文件中定义常量通常会很有帮助，这样我们就可以在任何包含这些文件的地方使用它们。

代码的下一部分应该看起来很熟悉：我们将 x 值写入模型的输入张量，进行推断，然后从输出张量中获取结果（我们将其称为 y 值）：

```
// 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];
```

现在，我们有了一个正弦值。由于对每个数字进行推断都需要花费少量的时间，而且这段代码是循环运行的，因此随着时间的推移，我们将生成一系列的正弦值。它将完美控制某些闪烁的 LED 或动画。我们的下一个工作是以某种方式输出它。

下面的代码调用在 *output_handler.cc* 中定义的 `HandleOutput()` 函数：

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

我们传入 x 和 y 值以及 `ErrorReporter` 实例，`ErrorReporter` 实例用于记录日志。让我们探索 *output_handler.cc*，看看接下来会发生什么。

5.3.2 通过 *output_handler.cc* 处理输出

output_handler.cc 文件定义了我们的 `HandleOutput()` 函数。它的实现非常简单：

```
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);
}
```

这个函数所做的就是使用 `ErrorReporter` 实例记录 x 值和 y 值。这只是一个最基本的实现，我们可以使用它来测试应用程序的基本功能，例如，通过在开发计算机上运行它进行测试。

但是，我们的目标是将程序部署到不同的微控制器上，使用每个平台的专用硬件来显示输出。对于我们计划部署到的每个平台，比如 Arduino，我们都提供了 *output_handler.cc* 的定制替换版本，这些替换版本使用特定平台的 API 来控制输出——例如，点亮一些 LED。

就像之前讨论过的，这些替换文件位于以每个平台名字命名的子目录下：`arduino/`、`disco_f76ng/` 和 `sparkfun_edge/`。稍后，我们将深入探讨特定于平台的实现。现在，让我们先回到 `main_functions.cc` 文件。

5.3.3 完成 main_functions.cc

我们在 `loop()` 函数中做的最后一件事是递增我们的 `inference_count` 计数器。如果达到了 `kInferencesPerCycle` 中定义的每个周期的最大推理次数，则将其重置为 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;
```

在下一次循环迭代时，它将产生这样的效果：要么将我们的 `x` 值向前移动一步，要么由于到达范围末尾而将它重置为 0。

现在，我们已经到达了 `loop()` 函数的结尾。每次运行时，它都会计算一个新的 `x` 值、运行推断然后由 `HandleOutput()` 输出结果。如果 `loop()` 被连续调用，它将对 $0 \sim 2\pi$ 范围内离散的 `x` 值进行推断，然后重复。

但是，是什么使 `loop()` 函数反复运行呢？答案就在 `main.cc` 文件中。

5.3.4 理解 main.cc

C++ 标准要求每个 C++ 程序都必须包含一个名为 `main()` 的全局函数，该函数将在程序启动时运行。在我们的程序中，`main()` 函数定义在文件 `main.cc` 中。`main()` 函数的存在也是 `main.cc` 代表程序入口点的原因。每当微控制器启动时都将运行 `main()` 中的代码。

`main.cc` 文件非常简短。首先，它包含 `main_functions.h` 的 `#include` 语句，这会引入定义在其中的 `setup()` 和 `loop()` 函数：

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

接下来，它声明 `main()` 函数本身：

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

当 `main()` 函数执行时，它首先调用 `setup()` 函数。这个函数将仅执行一次。之后，

它将进入一个 `while` 循环，循环将不断地重复调用 `loop()` 函数。

这个循环将无限期地运行。如果你有服务器编程或 Web 编程的背景，这听起来可能不是一个好主意。这个循环将阻塞我们的单线程执行其他命令，而且没有办法退出程序。然而，在为微控制器编写软件时，这种无限循环实际上非常常见。由于没有多任务处理，只会运行一个应用程序，所以循环不断进行没有问题。只要微控制器接通电源，我们就会持续地进行推断并输出数据。

现在，我们遍历了整个微控制器应用程序。在下一节中，我们将在开发计算机上试运行应用程序代码。

5.3.5 运行我们的应用程序

为了让我们应用程序代码运行测试，我们需要先编译它。输入以下 Make 命令来为我们的程序创建可执行二进制文件：

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

编译完成后，根据你的操作系统选择相应的命令来运行应用程序二进制文件：

```
# 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
```

如果找不到正确的路径，请列出 `tensorflow/lite/micro/tools/make/gen/` 中的目录。

运行二进制文件后，你应该看到一堆滚动过去的输出，看起来像这样：

```
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
```

非常激动人心！这些就是由 `output_handler.cc` 中的 `HandleOutput()` 函数写出的日志。每次推断都有一个日志，并且 `x_value` 的值从 0 逐渐递增，直到达到 2π ，这时它又回到 0 并重新开始。

兴奋过后，你可以按 Ctrl+C 终止程序。



你可能会注意到，数字以 2 的幂的形式输出，比如 `1.4137159*2^1`。这是一种在微控制器上记录浮点数的有效方法，因为很多微控制器通常不支持浮点运算。

要获取原始值，只需拿出计算器：例如，`1.4137159*2^1` 的计算结果为 `2.8274318`。如果你对输出方式感到好奇。可以在 `debug_log_numbers.cc` 中找到打印这些数字的代码。

5.4 小结

现在，我们已经在开发计算机上确认程序可以正常工作。在下一章中，我们将让程序在一些微控制器上运行！

TinyML 之 “Hello World”： 部署到微控制器

现在是动手的时候了。在本章中，我们将把代码部署到三种不同的设备上：

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

我们将逐一介绍在每个设备上的构建和部署过程。



TensorFlow Lite 会定期添加对新设备的支持，因此，如果你想使用的设备没有列在这里，可以检查示例的 *README.md* 查看是否支持。

如果你在执行以下步骤时遇到问题，也可以在那里查看更新的部署说明。

每个设备都有自己独特的输出功能，从一排 LED 到完整的 LCD 显示屏，因此我们的示例包含针对每个设备的 `HandleOutput()` 的定制实现。我们还将逐一介绍它们，并讨论其背后的逻辑原理。即便你没有所有设备，浏览所有的代码也是很有意思的，因此我们强烈建议你看一看所有的实现。

6.1 什么是微控制器

也许你没有太多硬件的背景，可能还不熟悉微控制器与其他电子组件的交互方式。由于我们即将开始使用硬件，所以在继续之前值得介绍一些背景知识。

在诸如 Arduino、SparkFun Edge 和 STM32F746G Discovery 套件之类的微控制器开发板

上，实际的微控制器只是连接到电路板上的诸多电子组件之一。图 6-1 展示了 SparkFun Edge 上的微控制器。

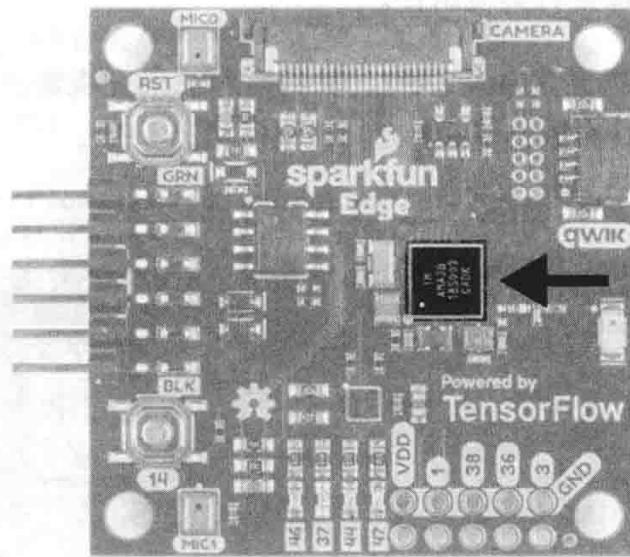


图 6-1：SparkFun Edge 开发板以及其中的微控制器（箭头所指）

微控制器通过引脚（pin）连接到它所处的电路板上。一个典型的微控制器通常有数十个引脚，它们有各种各样的用途。有些引脚为微控制器供电，其他一些则负责连接一些重要的组件。微控制器上运行的程序为数字信号的输入和输出预留了一些引脚，称为 GPIO 引脚，即通用输入 / 输出（general-purpose input/output）引脚。它们可以充当输入，以确定是否有电压输入，也可以充当输出，以提供可为其他组件供电或与之通信的电流。

GPIO 引脚是数字（digital）的。这意味着在输出模式下，它们就像开关一样，只能完全打开或者关闭。在输入模式下，它们可以检测另一组件施加给它们的电压是高于还是低于某个阈值。

除了 GPIO 外，某些微控制器还具有模拟（analog）输入引脚，这些引脚可以测量施加到它们上的确切电压值。

运行在微控制器上的程序可以通过调用特殊的函数，控制给定的引脚是处于输入模式还是输出模式。还可以使用其他的函数打开或者关闭输出引脚，或者读取输入引脚的当前状态。

现在，你对微控制器有了更多的了解，让我们详细介绍一下我们的第一个设备：Arduino。

6.2 Arduino

有各种各样的 Arduino 开发板，它们都有不同的功能。并非所有的 Arduino 开发板都可

以运行 TensorFlow Lite for Microcontrollers。我们为本书推荐的开发板是 Arduino Nano 33 BLE Sense。除了与 TensorFlow Lite 兼容之外，它还自带一个麦克风和一个加速度传感器（我们将在后面的章节中使用）。我们建议购买带接头的电路板版本，这样可以不需要焊接就能轻松地连接各种其他组件。

大多数 Arduino 开发板都带有内置 LED，我们将用它来直观输出正弦值。图 6-2 展示了一个 Arduino Nano 33 BLE Sense 开发板，箭头所指为 LED。

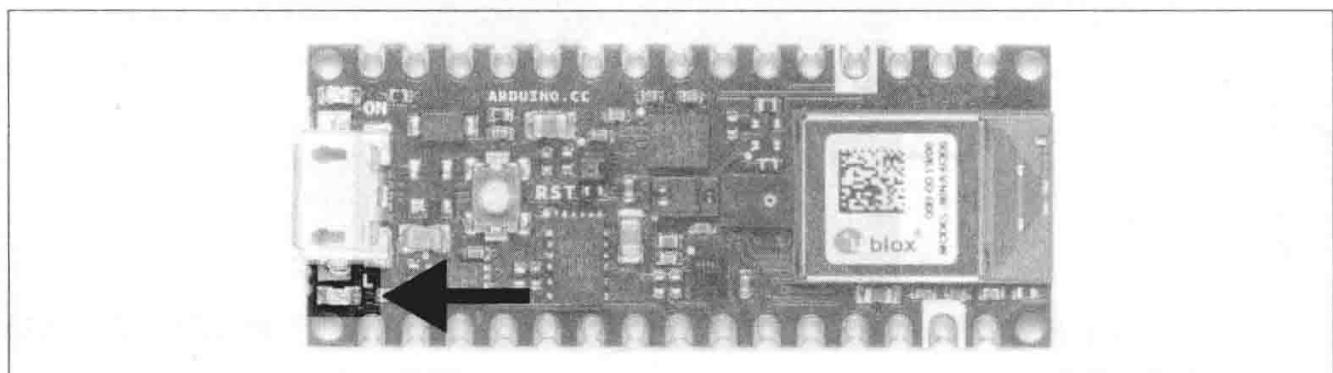


图 6-2：Arduino Nano 33 BLE Sense 开发板和其中的 LED

6.2.1 在 Arduino 上处理输出

因为我们只有一个 LED 可以工作，所以让我们发挥一些创造力，思考如何使用它。一种选择是根据最近预测的正弦值来改变 LED 的亮度。假定该值的范围是 $-1 \sim 1$ ，我们可以用完全熄灭的 LED 表示 0，用完全点亮的 LED 表示 -1 和 1 ，用 LED 部分变暗表示任何中间值。当程序在循环中运行推断时，LED 也将反复地点亮和熄灭。

我们可以使用 `kInferencesPerCycle` 常量改变在整个正弦波周期内执行推断的次数。由于运行一个推断需要一定的时间，因此调整 `constants.cc` 中定义的 `kInferencesPerCycle` 值可以调整 LED 的变暗速度。

Arduino 定制版本的 `constants.cc` 文件位于 `hello_world/arduino/constants.cc`。由于该文件与 `hello_world/constants.cc` 文件名相同，因此在编译 Arduino 应用程序时，会使用该文件而不是原始实现。

我们可以使用一种称为脉冲宽度调制（Pulse Width Modulation，PWM）的技术，来调暗开发板上内置的 LED。如果我们非常快速地打开和关闭输出引脚，那么引脚的输出电压与其关闭和开启状态持续时间的比例有关：如果引脚的每种状态持续 50% 的时间，那么它的输出电压将为最大电压的 50%；如果开启状态持续 75% 的时间，关闭状态持续 25% 的时间，则它的电压将为最大电压的 75%。

PWM 只适用于特定的 Arduino 设备的特定引脚，但它非常容易使用：我们只需调用一个函数即可设置所需的引脚输出电压。

实现 Arduino 输出处理的代码位于 `hello_world/arduino/output_handler.cc`，编译时它将代替原始文件 `hello_world/output_handler.cc`。

让我们来浏览一下源代码：

```
#include "tensorflow/lite/micro/examples/hello_world/output_handler.h"
#include "Arduino.h"
#include "tensorflow/lite/micro/examples/hello_world/constants.h"
```

首先，代码包含了一些头文件。我们的 `output_handler.h` 为此文件定义了接口。`Arduino.h` 提供了 Arduino 平台的接口，我们用它来控制开发板。因为我们需要访问 `kInferencesPerCycle`，所以还包含了 `constants.h`。

接下来，我们定义函数并指定它在首次运行时要执行的操作：

```
// 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;
    }
}
```

在 C++ 中，在函数内声明的静态 (`static`) 变量将在函数的多次运行中保留其值。在这里，我们使用 `is_initialized` 变量来跟踪下面的 `if(!is_initialized)` 代码块是否曾经运行过。

初始化块调用 Arduino 的 `pinMode()` 函数，它将告诉微控制器，给定的引脚应处于输入模式还是输出模式。在使用引脚之前都应先调用这个函数。我们使用两个 Arduino 平台定义的两个常量来调用 `pinMode()`：`LED_BUILTIN` 和 `OUTPUT`。`LED_BUILTIN` 表示连接到开发板的内置 LED 引脚，而 `OUTPUT` 表示输出模式。

将内置 LED 的引脚配置为输出模式后，`is_initialized` 被设置为 `true`，以此保证该代码块不会再次执行。

接下来，我们计算所需的 LED 亮度：

```
// 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));
```

Arduino 允许我们将 PWM 输出的电压设置为 0~255 的数字，其中 0 表示完全关闭，而 255 表示完全打开。我们的 `y_value` 是介于 -1 和 1 之间的数字。前面的代码将 `y_value` 映射到 0~255 的范围内，因此：当 `y = -1` 时，LED 完全熄灭；当 `y = 0` 时，LED 点亮一半；当 `y = 1` 时，LED 完全点亮。

下一步是实现设置 LED 的亮度：

```
// 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);
```

Arduino 平台的 `analogWrite()` 函数接收一个引脚号（我们提供的 `LED_BUILTIN`）和一个 0~255 的值。我们提供上一行中计算得到的亮度。调用此函数时，LED 将以对应的亮度被点亮。



不幸的是，在某些型号的 Arduino 开发板上，内置 LED 所连接的引脚无法使用 PWM。这意味着调用 `analogWrite()` 并不会改变它的亮度。相反，如果传递给 `analogWrite()` 的值大于 127，则 LED 会被点亮；如果小于或等于 126，则 LED 会被熄灭。也就是说，我们能看到 LED 闪烁，但不是以亮度渐变的形式。虽然不是很酷，但是它仍然可以一定程度上验证我们的正弦波预测。

最后，我们使用 `ErrorReporter` 实例记录亮度值：

```
// Log the current brightness value for display in the Arduino plotter  
error_reporter->Report("%d\n", brightness);
```

在 Arduino 平台上，`ErrorReporter` 被设置为通过串口（serial port）记录数据。串行（serial）是微控制器与主机通信的一种常见方式，通常用于调试。它是一种通信协议，通过打开和关闭一个输出引脚来传输一位数据。我们可以使用它来发送和接收任何内容，从原始二进制数据到文本和数字。

Arduino IDE 包含用于捕获和显示通过串口接收的数据的工具。其中一个工具——串口绘图器（Serial Plotter）——可以显示它通过串口接收的值的图形。通过在代码中输出亮度值流，我们将能看到它们的图形。图 6-3 展示了串口绘制器显示的图形。

我们将在本节稍后提供有关如何使用串口绘图器的说明。



你可能想知道 `ErrorReporter` 是如何通过 Arduino 的串口输出数据的。你可以在 `micro/arduino/debug_log.cc` 中找到具体的代码实现。就像覆盖 `output_handler.cc` 一样，这个文件也替换了 `micro/debug_log.cc` 中的原始实现。我们可以通过将 TensorFlow Lite for Microcontrollers 中的任何源文件添加到有特定平台名称的目录中，提供特定于平台的实现。

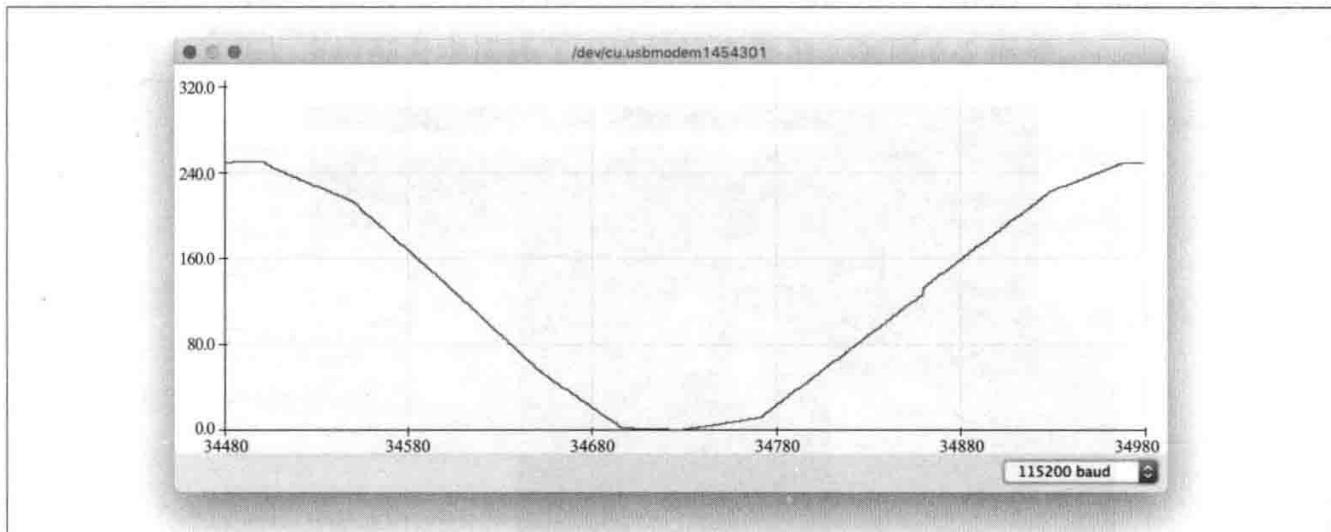


图 6-3：Arduino IDE 中的串口绘图器

6.2.2 运行示例

我们的下一个任务是编译 Arduino 项目，并将它部署到设备上。



与撰写本书时相比，编译过程有可能会发生变化，因此请查看 `README.md` 以获取最新说明。

以下是我们需要的所有东西：

- 支持的 Arduino 开发板（我们建议使用 Arduino Nano 33 BLE Sense）。
- 适当的 USB 线。
- Arduino IDE（在继续之前，你需要下载并安装它）。

本书中的项目是 TensorFlow Lite Arduino 库中的示例代码，你可以通过 Arduino IDE 轻松地安装所需的库。从“Tools”（工具）菜单中选择“Manage Libraries”（管理库），在出现的窗口中，搜索并安装名为 `Arduino_TensorFlowLite` 的库。你应该可以使用最新版本^{译注1}，但是如果最新版遇到问题，本书中测试的版本是 **1.14-ALPHA**。

译注 1：翻译本书时最新版本为 **1.15-ALPHA**。



你还可以从 TensorFlow Lite 团队网站下载 Arduino_TensorFlowLite 库，然后通过 `.zip` 文件安装，也可以使用 TensorFlow Lite for Microcontrollers Makefile 自行生成。如果你想这样做，请参阅附录 A。

安装完 Arduino_TensorFlowLite 库之后，`hello_world` 示例将显示在“File”（文件）菜单中的“Examples”（示例）→“Arduino_TensorFlowLite”下拉列表中，如图 6-4 所示。

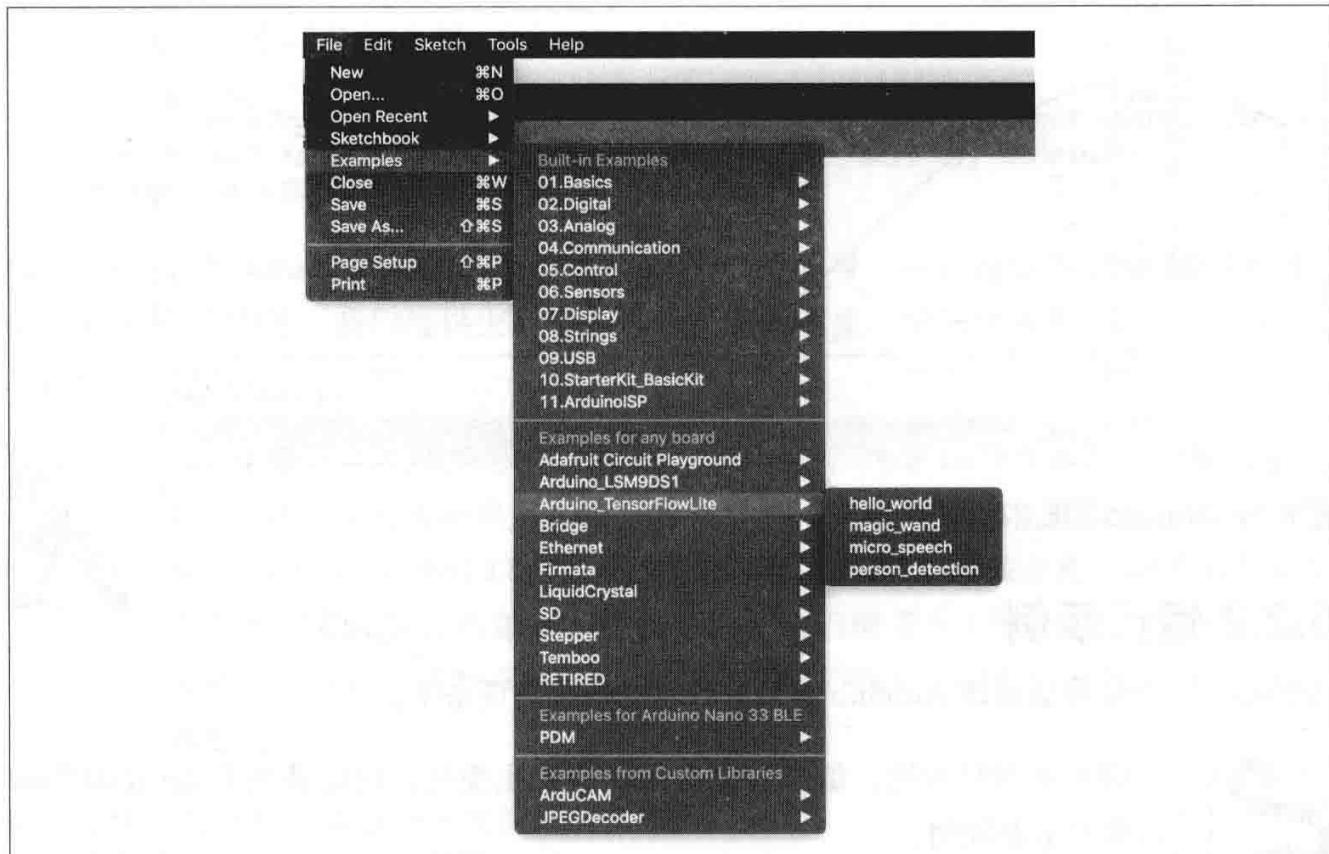


图 6-4：“Examples”菜单

单击“hello_world”以加载示例。它将显示一个新窗口，其中包含每个源文件的选项卡。第一个选项卡中的文件 `hello_world` 等同于我们之前介绍的 `main_functions.cc`。

Arduino 示例代码中的差异

在生成 Arduino 库时，会对代码进行一些细微更改，以使代码能与 Arduino IDE 配合得更好。也就是说，我们的 Arduino 示例和 TensorFlow GitHub 代码仓库中的代码之间会存在一些细微的差异。比如，在 `hello_world` 文件中，Arduino 环境会自动调用 `setup()` 和 `loop()` 函数，因此不需要 `main.cc` 文件和 `main()` 函数。

Arduino IDE 还希望源文件具有 `.cpp` 扩展名，而不是 `.cc`。此外，由于 Arduino IDE

不支持子文件夹，因此 Arduino 示例中的每个文件名都以其原始子文件夹名称为前缀。例如，*arduino_constants.cpp* 等同于我们讲解的名为 *arduino/constants.cc* 的文件。但是，除了一些细微的差异以外，大部分的代码都是一致的。

要运行示例，请通过 USB 插入你的 Arduino 设备。确保从“Tools”（工具）菜单中的“Board”（开发板）下拉列表中选择了正确的设备类型，如图 6-5 所示。

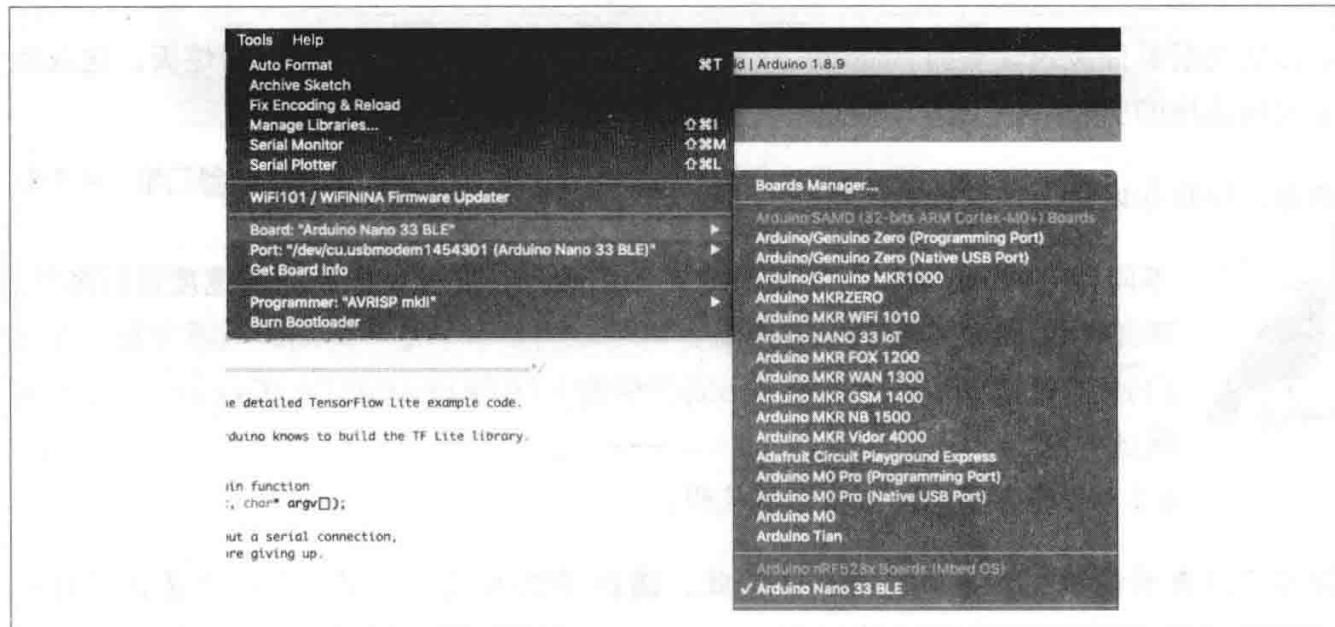


图 6-5：下拉列表中的 Arduino 开发板

如果你的设备没有显示在列表中，那么你需要安装它的支持包。为此，请单击“Boards Manager”（开发板管理器）。在出现的窗口中，搜索你的设备并安装相应的最新版本的支持包。

接下来，确保在“Port”（端口）下拉列表以及“Tools”（工具）菜单中选择了设备的端口，如图 6-6 所示。

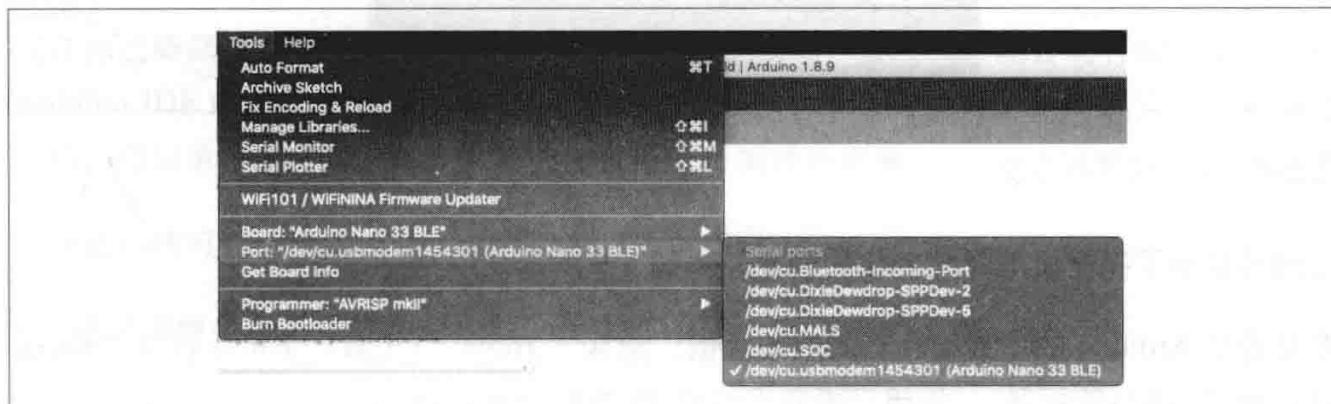


图 6-6：下拉列表中端口列表

最后，在 Arduino 窗口中，单击上传（upload）按钮（图 6-7 中白色突出显示的按钮），这会编译代码并将它上传到 Arduino 设备。

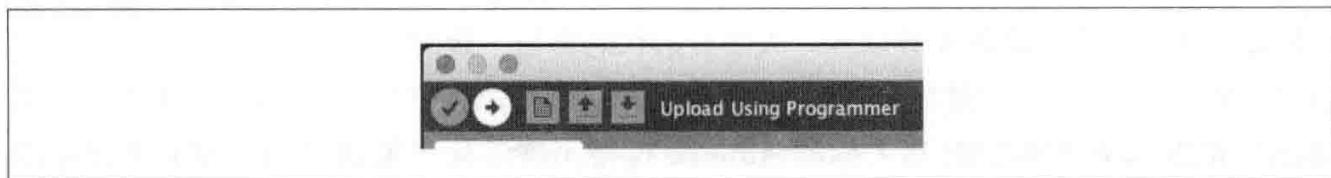


图 6-7：上传按钮为一个向右的箭头

上传成功后，你应该会看到 Arduino 开发板上的 LED 开始渐渐变亮或渐渐熄灭，这取决于它所连接的引脚是否支持 PWM。

恭喜，你正在设备上运行机器学习！



不同型号的 Arduino 开发板具有不同的硬件，并且将以不同的速度运行推断。如果你的 LED 闪烁或一直亮着，则可能需要增加每个周期的推断次数。你可以通过更改 `arduino_constants.cpp` 中的 `kInferencesPerCycle` 常数来实现这一点。

6.2.3 节将介绍如何编辑示例代码。

你还可以查看绘制在图上的亮度值。为此，请在“Tools”（工具）菜单中选择并打开 Arduino IDE 的串口绘图器，如图 6-8 所示。

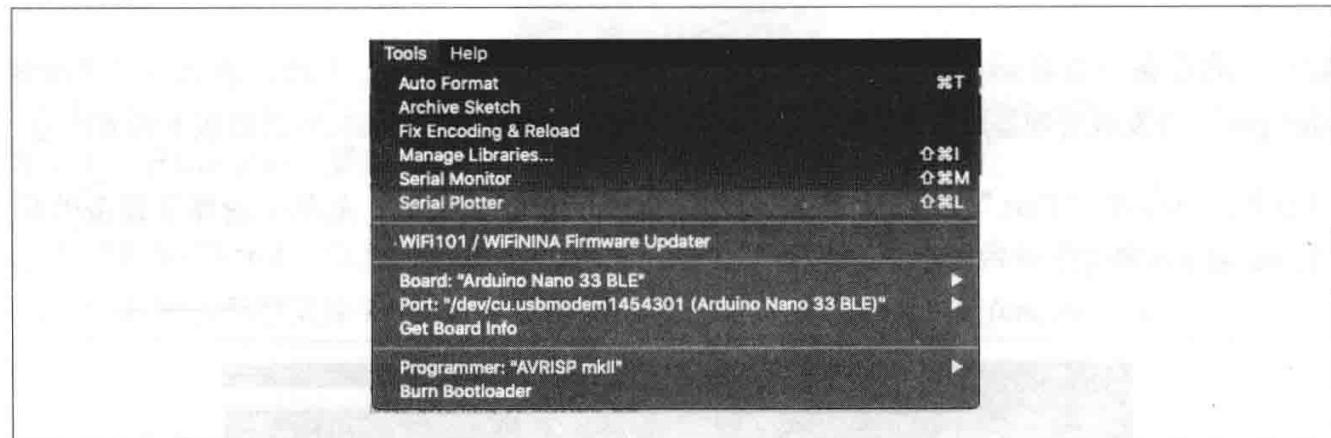


图 6-8：串口绘图器选项

绘图器显示了亮度值随时间变化的情况，如图 6-9 所示。

要查看从 Arduino 的串口接收到的原始数据，请从“Tools”（工具）菜单中打开“Serial Monitor”（串口监视器）。你将会看到大量数字掠过，如图 6-10 所示。

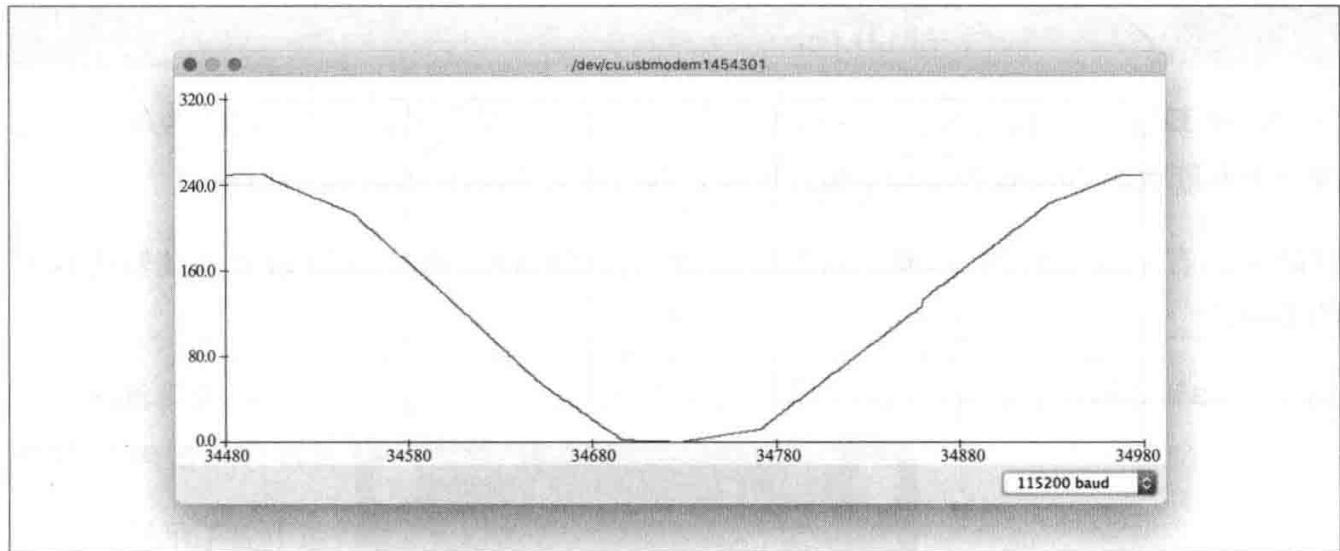


图 6-9：串口绘图器显示的值

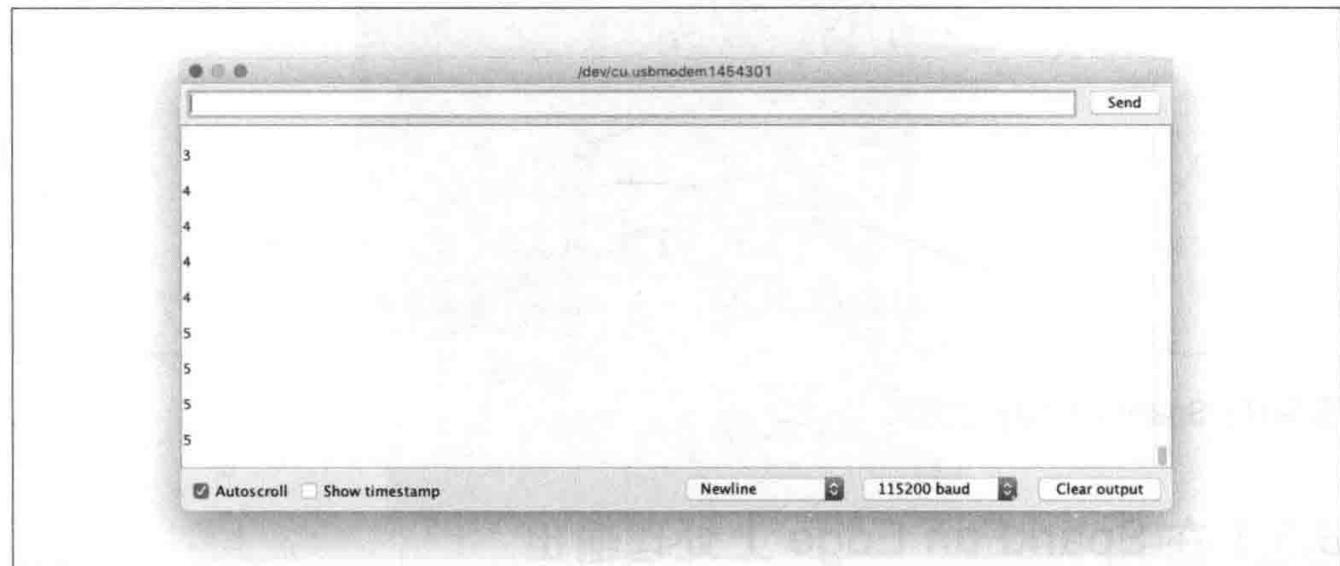


图 6-10：串口监视器打印的原始数据

6.2.3 尝试修改

现在你已经部署好了应用程序，那么尝试对代码进行一些更改可能会更有趣。你可以在 Arduino IDE 中编辑源文件。保存后，系统会提示你将示例重新保存到新位置。完成更改后，可以单击 Arduino IDE 中的上传按钮进行编译和部署。

以下是一些可以尝试的试验：

- 通过调整每个周期的推断次数，使 LED 闪烁的速度变慢或变快。
- 修改 *output_handler.cc* 以将基于文本的动画记录到串口。
- 使用正弦波来控制其他组件，例如其他 LED 或者发声器。

6.3 SparkFun Edge

SparkFun Edge 开发板是专门为在微型设备上进行机器学习试验而设计的开发平台。它有一个高能效的 Ambiq Apollo 3 微控制器，其内核是 Arm Cortex M4 处理器。

同时它还有一组（4个）LED，如图 6-11 所示。我们使用这些 LED 来直观地输出我们的正弦值

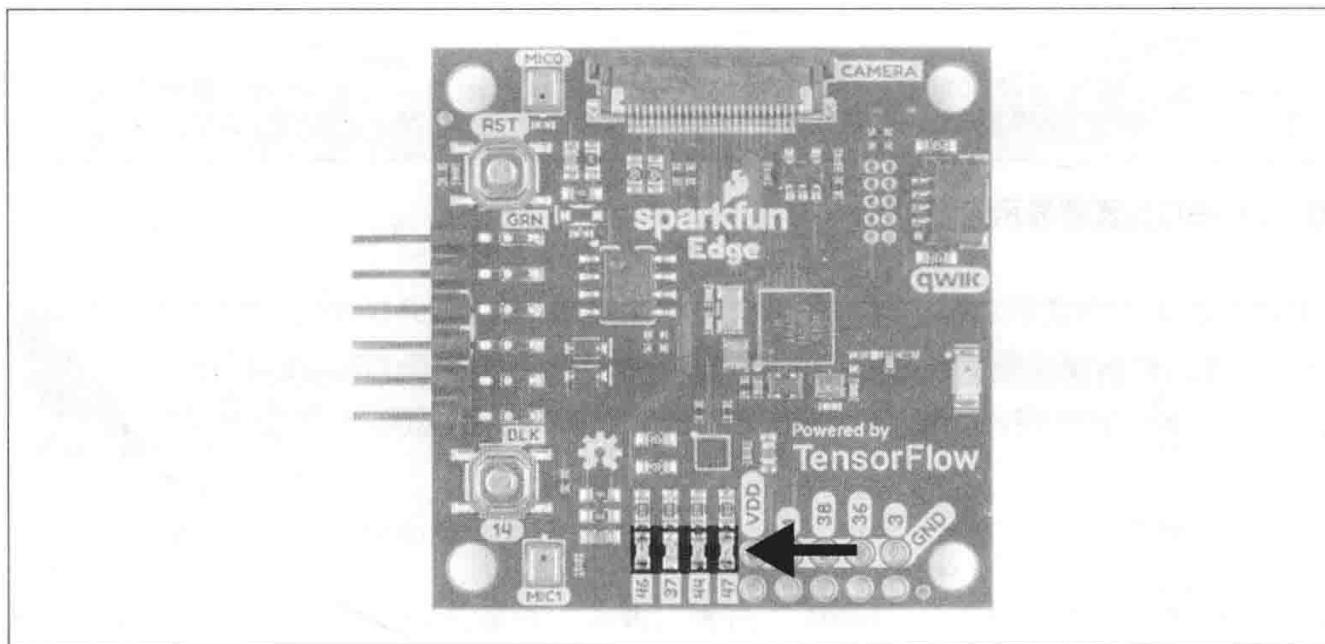


图 6-11：SparkFun Edge 上的 4 个 LED

6.3.1 在 SparkFun Edge 上处理输出

我们可以使用开发板上的 LED 来制作一个简单的动画，因为没有什么比指示灯阵列（blinkenlight）^{译注 2}更能体现先进的人工智能了。

LED（红色、绿色、蓝色和黄色）在物理上按以下顺序排列：

[R G B Y]

下面的表格说明了我们如何为不同的 y 值点亮 LED。

译注 2：blinkenlight 最早来源于早期许多计算机上印刷的最后一个单词，这是一些故意印刷的奇怪句子，以警告人们不要乱动这台机器，比如“ZO RELAXEN UND WATSCHEN DER BLINKENLICHTEN。”或者“Also: please keep still and only watchen astaunished the blinkenlights.”。同时，blinkenlight 也指早期计算机中的指示灯阵列，用以指示地址总线、数据总线和其他内部总线上以及不同寄存器中是否有值。

范 围	点亮 LED
$0.75 \leq y \leq 1$	[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]
$-1 \leq y \leq -0.75$	[1 1 0 0]

每次推断都需要花费一定的时间，因此调整在 *constants.cc* 中定义的 **kInferencesPerCycle** 常量将调整 LED 循环的速度。

图 6-12 展示了程序运行 gif 动画中的一帧图像。

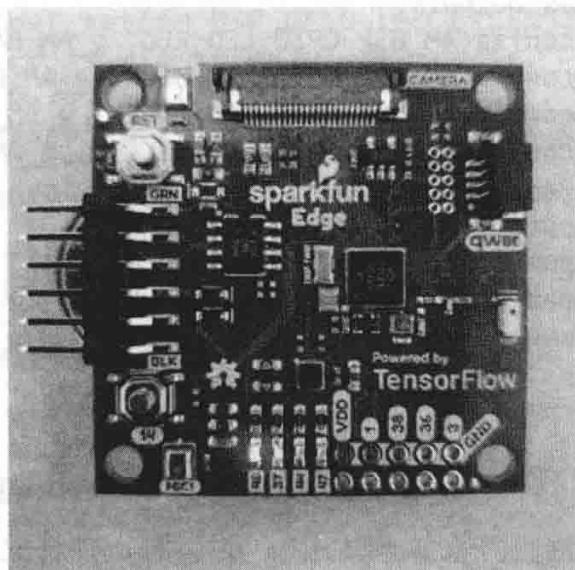


图 6-12：来自 SparkFun Edge 上的 LED 动画的静态图片

实现 SparkFun Edge 的输出处理的代码位于 *hello_world/sparkfun_edge/output_handler.cc* 文件，它替代了原始文件 *hello_world/output_handler.cc*。

让我们开始逐步介绍它：

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

首先，程序包含了一些头文件。我们的 *output_handler.h* 指定了此文件的接口。另一个文件 *am_bsp.h* 来自 *Ambiq Apollo3 SDK*。Ambiq 是 SparkFun Edge 中使用的微控制器 Apollo3 的制造商。SDK (Software Development Kit, 软件开发工具包) 是一系列源文件的集合，这些源文件定义了可用于控制微控制器功能的常量和函数。

由于我们计划控制开发板的 LED，因此我们需要能够打开和关闭微控制器的引脚。所以我们需要使用 Ambiq Apollo3 SDK。



当我们最终编译项目时，Makefile 将自动下载 SDK。如果你感到好奇，可以阅读更多有关它的内容，或者在 SparkFun 网站下载并查看源代码。

接下来，我们定义 HandleOutput() 函数并定义其首次运行时要执行的操作：

```
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;
    }
}
```

要设置的东西这么多！我们正在使用 *am_bsp.h* 提供的 *am_hal_gpio_pinconfig()* 函数，通过它可以配置连接至开发板内置 LED 的引脚，并将它设置为输出模式（用 *g_AM_HAL_GPIO_OUTPUT_12* 表示）。每个 LED 的引脚号都用一个常数表示，例如 *AM_BSP_GPIO_LED_RED*。

然后，我们使用 *am_hal_gpio_output_clear()* 清除所有输出，因此所有 LED 都会被熄灭。与 Arduino 中的实现一样，我们使用名为 *is_initialized* 的 *static* 变量来确保该块中的代码仅运行一次。接下来，如果 *y* 值为负，我们确定应该点亮哪些 LED：

```
// 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);
    }
```

首先，如果 y 值刚刚变为负值，我们清除两个用于表示正值的 LED。接下来，我们调用 `am_hal_gpio_output_set()` 来打开蓝色的 LED。蓝色的 LED 将在 y 值为负的时候一直点亮。最后，如果 y 值小于 -0.75，我们将打开红色 LED，否则关闭红色 LED。

接下来，我们对于值为正的 y 值做同样的事情：

```
// 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);
    }
}
```

对 LED 做的事情就这些。我们要做的最后一件事是将当前输出值记录到在串口上监听的任何组件：

```
// Log the current X and Y values
error_reporter->Report("x_value: %f, y_value: %f\n", x_value, y_value);
```



由于 `micro/sparkfun_edge/debug_log.cc` 的自定义实现替代了 `mmicro/debug_log.cc` 中的原始实现，因此我们的 `ErrorReporter` 能够通过 SparkFun Edge 的串口输出数据。

6.3.2 运行示例

现在，我们可以编译示例代码并将它部署到 SparkFun Edge。



自撰写本书时起，编译过程始终有可能发生变化，因此请查看 `README.md` 以获取最新说明。

为了编译和部署我们的代码，我们需要以下内容：

- SparkFun Edge 开发板。

- USB 编程器（我们建议使用 SparkFun Serial Basic Breakout，它有 micro-B USB 和 USB-C 两种型号）。
- 配套的 USB 线。
- Python 3 和一些依赖项。

Python 和依赖项

此过程会涉及运行一些 Python 脚本。在继续之前，你应确保已经安装了 Python 3。要检查系统上是否已经安装 Python，请打开一个终端并输入以下内容：

```
python --version
```

如果你安装了 Python 3，则会看到以下输出（其中 x 和 y 是次要版本号，具体值是什么并不重要）：

```
Python 3.x.y
```

如果运行成功，你可以在本节后面的部分中使用命令 `python` 来运行 Python 脚本。

如果看到不同的输出，请尝试以下命令：

```
python3 --version
```

你应该希望看到的输出与我们之前寻找的输出相同：

```
Python 3.x.y
```

如果这样做，则意味着可以在需要时使用命令 `python3` 来运行 Python 脚本。

否则，你需要在系统上安装 Python 3。你可以在网上搜索有关在特定操作系统上安装它的说明。

Python 3 安装完成后，你还需要安装一些依赖项。运行以下命令来执行此操作（如果你的 Python 命令是 `python3`，则应使用命令 `pip3` 而不是 `pip`）：

```
pip install pycrypto pyserial --user
```

依赖项安装完成后，就可以开始使用了。

首先，打开一个终端，克隆 TensorFlow 代码仓库，然后跳转到对应目录：

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

接下来，我们将编译二进制文件并运行一些命令，以准备将它下载到设备上。为了避免键入内容，可以从 *README.md* 复制并粘贴这些命令。

编译二进制文件

以下命令会下载所有必需的依赖项，然后为 SparkFun Edge 编译二进制文件：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
TARGET=sparkfun_edge hello_world_bin
```



二进制文件是一种包含程序的文件，它可以在 SparkFun Edge 硬件上直接运行。

二进制文件将以 *.bin* 为后缀，路径如下：

```
tensorflow/lite/micro/tools/make/gen/ \
sparkfun_edge_cortex-m4/bin/hello_world.bin
```

要检查文件是否存在，可以使用以下命令：

```
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"
```

如果运行此命令，你应该看到 “*Binary was successfully created*”（已成功编译二进制文件）被打印到控制台。

如果你看到 “*Binary is missing*”，则说明编译过程存在问题。如果是这样，你很可能可以在 *make* 命令的输出中找到一些有关错误的线索。

文件签名

二进制文件必须使用加密密钥（cryptographic key）签名后才能部署到设备。现在，让我们运行一些命令对二进制文件进行签名，以便能够将它烧录到 SparkFun Edge 上。此处使用的脚本来自 Ambiq SDK，它在运行 *Makefile* 时会被下载。

输入以下命令来设置一些可用于开发的虚拟加密密钥：

```
cp tensorflow/lite/micro/tools/make/downloads/AmbiqSuite-Rel2.0.0/ \
tools/apollo3_scripts/keys_info.py \
tensorflow/lite/micro/tools/make/downloads/AmbiqSuite-Rel2.0.0/ \
tools/apollo3_scripts/keys_info.py
```

接下来，运行以下命令以编译签名的二进制文件。如有必要，请用 *python* 替换 *python3*：

```
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
```

这将创建文件 *main_nonsecure_ota.bin*。现在运行这个命令来创建一个文件的最终版本，你可以用将在下一个步骤中使用的脚本来烧录你的设备：

```
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
```

现在，你运行命令的目录中应该有一个名为 *main_nonsecure_wire.bin* 的文件。这就是我们将要烧录到设备中的文件。

烧录二进制文件

SparkFun Edge 将当前运行的程序存储在它 1MB 的闪存中。如果你想让开发板运行一个新的程序，你需要把它发送到开发板，它会把程序存储在闪存中，覆盖之前保存的任何程序。

这个过程称为烧录（flash）。让我们一步步来做。

将编程器连接到开发板上。要将新的程序下载到开发板上，你将使用 SparkFun USB-C Serial Basic 串口编程器。它使你的计算机可以通过 USB 与微控制器通信。

要将此设备连接到开发板上，请执行以下步骤：

1. 在 SparkFun Edge 的侧面找到六引脚接口。
2. 将 SparkFun USB-C Serial Basic 插入这些引脚，确保每个设备上标记为 BLK 和 GRN 的引脚正确排列。

你可以在图 6-13 中看到正确的排列。

将编程器连接到计算机。接下来，通过 USB 将开发板连接到计算机。要对开发板进行编程，你需要确定计算机为设备指定的名称。最好的方法是在连接计算机之前和之后列出所有计算机设备，然后查看哪个是新设备。

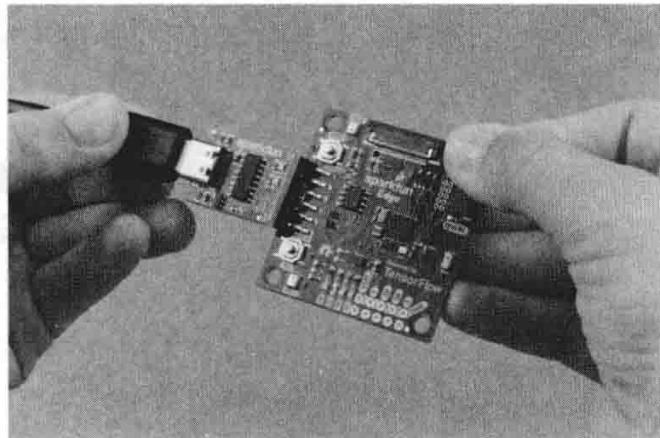


图 6-13：连接 SparkFun Edge 和 USB-C 串口（由 SparkFun 提供）



有些人已经报告了他们使用的操作系统的默认驱动程序的问题，因此，我们强烈建议你在继续之前先安装驱动程序。

在通过 USB 连接设备之前，请运行以下命令：

```
# macOS:  
ls /dev/cu*  
  
# Linux:  
ls /dev/tty*
```

这应该会输出已连接设备的列表，如下所示：

```
/dev/cu.Bluetooth-Incoming-Port  
/dev/cu.MALS  
/dev/cu.SOC
```

现在，将编程器连接到计算机的 USB 端口，然后再次运行命令：

```
# macOS:  
ls /dev/cu*  
  
# Linux:  
ls /dev/tty*
```

你应该会在输出中看到一个多出的项，如下所示。在你的计算机上，新列出的项可能具有不同的名称，它就是我们的设备名称：

```
/dev/cu.Bluetooth-Incoming-Port  
/dev/cu.MALS  
/dev/cu.SOC  
/dev/cu.wchusbserial-1450
```

这个名字将用于引用设备。但是，它可能会根据编程器所连接的 USB 端口而改变，因此，如果你将开发板从计算机上断开，然后重新连接，则可能需要再次查询设备名称。



一些用户报告说看到两个设备出现在列表中。如果你看到两个设备，正确的设备应该是以字母“wch”开头的设备，例如“/dev/wchusbserial-14410”。

确定设备名称后，将它存入一个 shell 变量以备后用：

```
export DEVICENAME=<your device name here>
```

在稍后运行需要设备名的命令时，可以使用这个变量。

运行脚本以烧录开发板。要烧录开发板，你需要将它置于特殊的 bootloader 状态，以使其准备好接收新的二进制文件。然后，你可以运行脚本以将二进制文件发送到开发板。

首先创建一个环境变量以指定波特率，波特率是将数据发送到设备的速度：

```
export BAUD_RATE=921600
```

现在，请将下面的命令粘贴到你的终端中，但是先不要按回车键！命令中的 \${DEVICENAME} 和 \${BAUD_RATE} 将被替换为你在上一节中设置的值。请记住，如有必要，请用 python 替换 python3：

```
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
```

接下来，将开发板重置为 bootloader 状态并烧录开发板。在开发板上，找到标记为 RST 和 14 的按键，如图 6-14 所示。

执行以下步骤：

1. 确保你的开发板已连接至编程器，且整个设备均已通过 USB 连接到你的计算机。
2. 在开发板上，按住标记为 14 的按键，不要松手。
3. 在长按标记为 14 的按键的同时，按标记为 RST 的按键以重置开发板。
4. 在计算机上按回车键运行脚本。继续长按按键 14。

现在，你应该在屏幕上看到如下内容：

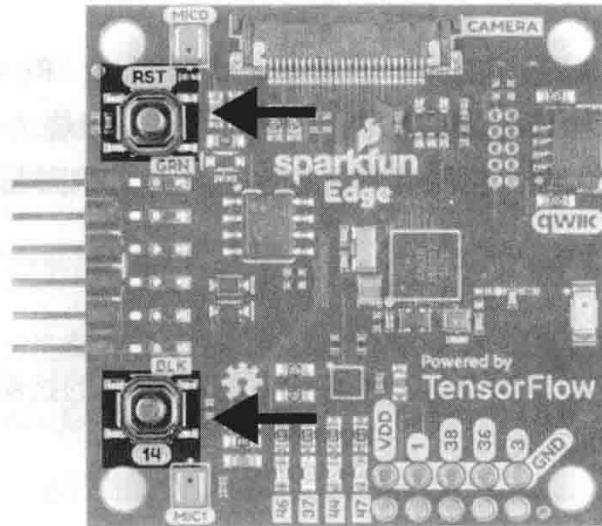


图 6-14: SparkFun Edge 上的按键

```
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...]
```

一直按住按键 14，直到你看到“Sending Data Packet of length 8180”。看到此消息后可以松开按键（但如果你一直按着它也没关系）。

该程序将继续在终端上打印日志。最终，你会看到类似以下内容：

```
[...lots more Sending Data Packet of length 8180...]
Sending Data Packet of length 8180
Sending Data Packet of length 6440
Sending Reset Command.
Done.
```

这表示烧录成功。



如果程序输出以错误结尾，请检查“**Sending Reset Command.**”是否被打印出来。如果是这样，尽管出现了错误，但烧录可能还是会成功；否则，烧录很可能会失败。尝试再次执行这些步骤（可以跳过设置环境变量）。

6.3.3 测试程序

现在，二进制文件已经被部署到设备上了。按下标有 RST 的按键以重新启动开发板。你应该看到设备的 4 个 LED 依次闪烁。干得不错！

如果失败了怎么办

以下是一些可能出现的问题以及调试方法：

问题：烧录时，脚本在“**Sending Hello.**”的地方停顿一段时间。然后打印错误。

解决方案：运行脚本时，需要按住标记为 14 的按键。按住按键 **14**，按 RST 按键，然后在保持按住按键 **14** 的同时运行脚本。

问题：烧录后，所有指示灯都不亮。

解决方案：尝试按住 RST 按键，或将开发板与编程器断开连接，然后重新连接。如果这些都不起作用，请尝试再次烧录开发板。

6.3.4 查看调试数据

程序运行时，调试信息由开发板记录。要查看它，我们可以使用 115200 的波特率监视开发板的串口输出。在 macOS 和 Linux 上，以下命令应该起作用：

```
screen ${DEVICENAME} 115200
```

你会看到大量输出掠过！要停止滚动，请按 Ctrl+A，然后立即按 Esc。然后，你可以使用箭头键来查看输出，其中包含对不同 x 值的推断结果：

```
x_value: 1.1843798*2^2, y_value: -1.9542645*2^-1
```

要停止在屏幕上查看调试输出，请按 Ctrl+A，紧接着按 K 键，然后按 Y 键。



screen 程序是用于连接到其他计算机的帮助程序。在本例中，我们使用它来监听 SparkFun Edge 开发板通过串口记录的数据。如果你使用的是 Windows，则可以尝试使用 CoolTerm 程序执行相同的操作。

6.3.5 尝试修改

现在，你已经部署了基本的应用程序，可以尝试并进行一些更改了。你可以在 `tensorflow/lite/micro/examples/hello_world` 文件夹中找到我们应用程序的代码。只需编辑并保存，然后重复前面的操作，就可以将修改后的代码部署到设备上。

以下是一些你可以尝试的事情：

- 通过调整每个周期的推断次数，使 LED 闪烁的速度变慢或变快。
- 修改 `output_handler.cc` 以将基于文本的动画记录到串口。
- 使用正弦波来控制其他组件，例如其他 LED 或发声器。

6.4 ST Microelectronics STM32F746G Discovery 套件

STM32F746G 是一个具有相对强大的 Arm Cortex-M7 处理器内核的微控制器开发板。

它运行 Arm 的 Mbed OS，这是一个嵌入式操作系统，旨在使编译和部署嵌入式应用程序更加容易。这意味着我们可以使用本节中的许多指令编译其他 Mbed 设备。

STM32F746G 带有一个附加的 LCD 屏幕，这将允许我们实现更加精细的视觉展示。

6.4.1 在 STM32F746G 上处理输出

现在我们有了一个完整的 LCD，我们可以绘制一个漂亮的动画。让我们使用屏幕的 x 轴表示推断次数， y 轴表示预测的当前值。

我们将 x 和 y 指定的地方绘制一个点，当我们在 $0 \sim 2\pi$ 的输入范围内循环时，它将在屏幕上移动。图 6-15 展示了它的线框图。

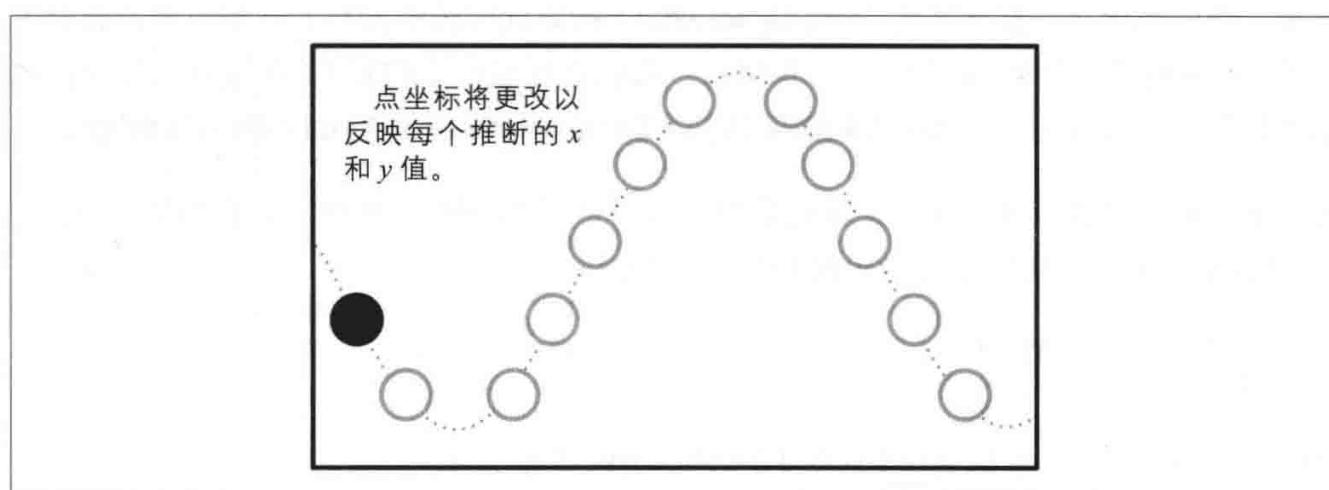


图 6-15：我们将在 LCD 显示屏上绘制的动画

每次推断都需要花费一定的时间，因此，调整 *constants.cc* 中定义的 **kInferencesPerCycle** 将调整点的运动速度和平滑度。

图 6-16 显示了程序运行 gif 动画中的一帧静止图像。

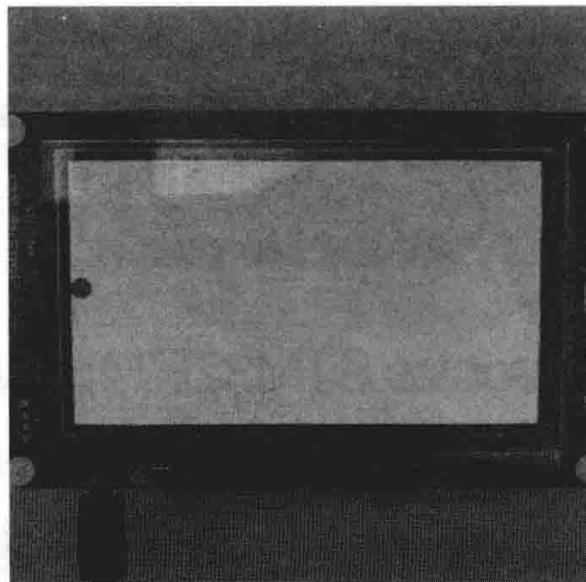


图 6-16：在具有 LCD 显示屏的 STM32F746G Discovery 套件开发板上运行的程序

处理 STM32F746G 输出的代码位于 *hello_world/disco_f746ng/output_handler.cc*，用于代替原始文件 *hello_world/output_handler.cc*。

让我们来看一下：

```
#include "tensorflow/lite/micro/examples/hello_world/output_handler.h"
#include "LCD_DISCO_F746NG.h"
#include "tensorflow/lite/micro/examples/hello_world/constants.h"
```

首先，我们包含了一些头文件。*output_handler.h* 指定此文件的接口。开发板制造商提供的 *LCD_DISCO_F746NG.h* 声明了一些接口，我们将用它们来控制 LCD 屏幕的界面。我们还包含了 *constants.h*，因为我们需要访问 **kInferencesPerCycle** 和 **kXrange**。

接下来，我们设置大量的变量。首先是 *LCD_DISCO_F746NG* 的实例，它在 *LCD_DISCO_F746NG.h* 中定义，提供了可用于控制 LCD 的方法：

```
// The LCD driver
LCD_DISCO_F746NG lcd;
```

Mbed 网站提供了有关 *LCD_DISCO_F746NG* 类的详细信息。

接下来，我们定义一些控制视觉效果的常量：

```
// The colors we'll draw
const uint32_t background_color = 0xFFFF4B400; // Yellow
const uint32_t foreground_color = 0xFFDB4437; // Red
// The size of the dot we'll draw
const int dot_radius = 10;
```

颜色以十六进制值的形式提供，例如 0xFFFF4B400。它的格式为 AARRGGBB，其中 AA 代表阿尔法（alpha）值，或不透明度，FF 表示完全不透明，RR、GG 和 BB 分别代表红色、绿色和蓝色的数量。



你可以通过一些练习学习从十六进制值读取颜色。0xFFFF4B400 是完全不透明的，且红色和绿色的值较大，这会得到漂亮的橙黄色。你还可以通过网上搜索来快速查找一个颜色的十六进制值。

然后，我们再声明一些变量来定义动画的形状和大小：

```
// 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;
```

声明完变量之后，我们定义 HandleOutput() 函数并告诉它在第一次运行时要做的事情：

```
// 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;
    }
}
```

其中包含很多内容！首先，我们使用 lcd 的方法来设置背景色和前景色。SetColor() 函数实际上设置了我们绘制的任何东西的颜色，而不仅仅是文本：

```
// Set the background and foreground colors
lcd.Clear(background_color);
lcd.SetTextColor(foreground_color);
```

接下来，我们计算实际可以绘制到屏幕的位置，以便知道应该在哪里绘制圆。如果我们弄错了，我们绘制的地方可能会超出屏幕的边缘，得到一些意料之外的结果：

```
width = lcdGetXSize() - (dot_radius * 2);
height = lcdGetYSize() - (dot_radius * 2);
```

之后，我们确定屏幕中间的位置，并在它的下方将绘制负的 y 值。我们还计算了多少个屏幕宽度像素代表 x 值的一个单位。请注意我们是如何使用 `static_cast` 确保获得浮点结果的：

```
// Calculate the y axis midpoint
midpoint = height / 2;
// Calculate fractional pixels per unit of x_value
x_increment = static_cast<float>(width) / kXrange;
```

就像我们之前所做的，使用名为 `is_initialized` 的静态变量来确保该块中的代码仅运行一次。

初始化完成后，就可以开始输出了。首先，清除之前的任何图形：

```
// Clear the previous drawing
lcd.Clear(background_color);
```

接下来，我们使用 `x_value` 来计算我们的点在显示器的 x 轴上的位置：

```
// 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);
```

然后，对 y 值做相同的处理。这有点复杂，因为我们要在 `midpoint` 上方绘制正值，在下方绘制负值：

```
// 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));
}
```

确定位置后，我们就可以画点了：

```
// Draw the dot  
lcd.FillCircle(x_pos, y_pos, dot_radius);
```

最后，我们使用 `ErrorReporter` 将 `x` 和 `y` 值记录到串口：

```
// Log the current X and Y values  
error_reporter->Report("x_value: %f, y_value: %f\n", x_value, y_value);
```



`micro/disco_f746ng/debug_log.cc` 中定义的特定 `ErrorReporter` 会取代原来的 `micro/debug_log.cc` 中的定义，实现通过 STM32F746G 的串口输出数据。

6.4.2 运行示例

接下来，让我们来编译项目！STM32F746G 运行 Arm 的 Mbed OS，因此我们将使用 Mbed 工具链将应用程序部署到设备上。



自从撰写本书时起，编译过程始终有可能发生变化，因此请查看 `README.md` 以获取最新说明。

在开始之前，我们需要以下内容：

- STM32F746G Discovery 套件开发板。
- mini-USB 线。
- Arm Mbed CLI（请遵循 Mbed 设置指南）。
- Python 3 和 pip。

像 Arduino IDE 一样，Mbed 也会要求以某种方式组织源文件。TensorFlow Lite for Microcontrollers Makefile 清楚如何为我们执行此操作，并且可以生成适合 Mbed 的目录。

为此，请运行以下命令：

```
make -f tensorflow/lite/micro/tools/make/Makefile \  
TARGET=mbed TAGS="CMSIS disco_f746ng" generate_hello_world_mbed_project
```

这会创建新的目录：

```
tensorflow/lite/micro/tools/make/gen/mbed_cortex-m4/prj/ \  
hello_world/mbed
```

此目录包含示例的所有依赖项，这些依赖项以正确的方式组织，以便 Mbed 能够编译它们。

首先，进入目录，以便在其中运行一些命令：

```
cd tensorflow/lite/micro/tools/make/gen/mbed_cortex-m4/prj/ \
hello_world/mbed
```

现在，你将使用 Mbed 下载依赖项并编译项目。

首先，使用以下命令向 Mbed 指定当前目录是 Mbed 项目的根目录：

```
mbed config root .
```

接下来，指示 Mbed 下载依赖项并准备编译：

```
mbed deploy
```

默认情况下，Mbed 将使用 C++ 98 构建项目。但是 TensorFlow Lite 需要 C++ 11。运行以下 Python 代码以修改 Mbed 配置文件，使其使用 C++ 11。你可以将它们键入或粘贴到命令行中：

```
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"))'
```

最后，运行以下命令进行编译：

```
mbed compile -m DISCO_F746NG -t GCC_ARM
```

这将在以下路径中生成二进制文件：

```
./BUILD/DISCO_F746NG/GCC_ARM/mbed.bin
```

使用支持 Mbed 的开发板（例如 STM32F746G）的好处之一就是部署非常简单。要部署程序，只需插入 STM 开发板，然后将文件复制到其中即可。在 macOS 上，可以使用以下命令执行此操作：

```
cp ./BUILD/DISCO_F746NG/GCC_ARM/mbed.bin /Volumes/DIS_F746NG/
```

或者，只需在文件浏览器中找到 DIS_F746NG 并将文件拖到上方即可。复制文件将启动烧录过程。完成此操作后，你应该在设备的屏幕上看到动画。

除此动画以外，程序运行时，开发板还会记录调试信息。要查看它，请使用 9600 的波特率建立与开发板的串行连接。

在 macOS 和 Linux 上，使用以下命令列出设备：

```
ls /dev/tty*
```

应该得到与以下类似的内容：

```
/dev/tty.usbmodem1454203
```

确定设备名词后，使用下面的命令连接到设备，将 `</dev/tty.devicename>` 替换为 `/dev` 中显示的设备名称：

```
screen /<dev/tty.devicename> 9600
```

你会看到大量输出掠过。要停止滚动，请按 `Ctrl+A`，然后立即按 `Esc`。然后，你可以使用箭头键浏览输出，其中将包含对各种 `x` 值进行推断的结果：

```
x_value: 1.1843798*2^2, y_value: -1.9542645*2^-1
```

要停止在屏幕上查看调试输出，请按 `Ctrl+A`，紧接着按 `K` 键，然后按 `Y` 键。

6.4.3 尝试修改

现在，你已经部署好了应用程序，是时候尝试你自己的修改了！你可以在 `tensorflow/lite/micro/tools/make/gen/mbed_cortex-m4/prj/hello_world/mbed` 文件夹中找到应用程序的代码。只需编辑并保存，然后重复前面的说明，就可以将修改后的代码部署到设备上。

以下是一些你可以尝试的事情：

- 通过调整每个周期的推断次数，使点移动得更快或者更慢。
- 修改 `output_handler.cc`，将基于文本的动画记录到串口。
- 使用正弦波来控制其他组件，例如 LED 或发声器。

6.5 小结

在过去的三章中，我们完成了一个完整的端到端的过程：训练模型，将模型转换为 TensorFlow Lite，围绕它编写应用程序，并将应用程序部署到微型设备上。在接下来的章节中，我们将探讨一些更复杂也更令人兴奋的例子，这些例子可以使嵌入式机器学习发挥更大的作用。

首先，我们将创建一个语音识别程序，它使用一个大小只有 18KB 的微型模型。

唤醒词检测：创建应用程序

TinyML 也许是一种新现象，但它最广泛的应用可能已经在你的家里、车里，甚至口袋里工作了。你能猜出都有哪些应用吗？

过去几年中我们见证了数字助手（digital assistant）的兴起。这些产品提供了语音用户接口（User Interface, UI），旨在让用户无须借助屏幕或键盘就可以立即访问信息。如 Google Assistant、苹果的 Siri 和亚马逊的 Alexa，这些数字助手几乎无处不在。从旗舰机型到专为新兴市场设计的语音优先设备，几乎所有手机都内置了某种数字助手。这些数字助手同样存在于智能扬声器、计算机以及车辆之中。

在大多数情况下，语音识别、自然语言处理以及对用户查询生成响应的繁重工作都是在云端完成的，由性能强大的服务器运行大型机器学习模型。当用户提出一个问题时，这个问题将以音频流的形式被发送到服务器。服务器将识别出这个音频流的含义，并查找出所有必需的信息，然后将适当的响应发送回去。

然而，数字助手的吸引力之一就是它们能够时刻保持工作状态，可以随时为你提供帮助。无须借助任何按键，仅仅通过说“Hey Google”或“Alexa”，你就可以唤醒助手并告诉它你想要哪些信息。这意味着它们需要全天候地监听你的声音，无论你是坐在客厅里、行驶在高速上，还是手持着手机在户外活动。

尽管在服务器上进行语音识别很容易，但是将恒定的音频流从设备发送到数据中心却是不可行的。从隐私的角度来看，将捕获到的每一秒音频都发送到远程服务器绝对是一场灾难。即使这是可能的，这样做也将需要大量带宽并且可能会在几小时内耗尽移动流量数据。除此之外，网络通信会消耗电量，发送稳定的数据流将会很快耗尽设备的电池电量。更重要的是，如果将每一个请求都发送到服务器并等待返回结果，你将会觉得语音助手反应十分缓慢。

数字助手真正需要的音频是紧随唤醒词（例如“Hey Google”）之后的声音。如果我们

能在不发送数据的情况下检测到这个唤醒词，并在听到唤醒词之后才开始音频流的传输会怎么样？我们将能够保护用户的隐私、节省电池电量和带宽，而且可以在没有网络的情况下唤醒助手。

这就是 TinyML 的用武之地。我们可以训练一个监听唤醒词的微型模型，并将其运行在低能耗的芯片上。如果我们将其嵌入手机，它就可以一直监听唤醒词。当听到唤醒词时，芯片会通知手机的操作系统（Operating System, OS），操作系统可以开始捕获音频并将其发送到服务器上。

唤醒词检测是 TinyML 的一个完美应用，它非常适合提供隐私保护、高效、迅速且离线的推断。在这种方法中，一个小而有效的模型会“唤醒”一个更大、更消耗资源的模型，这种方式称为级联（cascading）。

在本章中，我们将探讨如何使用预先训练的语音检测模型，在微控制器上提供时刻在线的唤醒词检测功能。在第 8 章中，我们将探索该模型是如何训练的，以及如何创建我们自己的模型。

7.1 我们要创建什么

我们将要创建一个嵌入式应用程序，该应用程序将使用 18KB 大小的模型（在语音命令数据集上进行过训练）对语音音频进行分类。该模型被训练来识别“yes”和“no”两个词，并且能够区分未知词和沉默或是背景噪声。

我们的应用程序将使用麦克风监听周围的环境，并根据设备的功能通过点亮 LED 或在屏幕上显示数据表明检测到了某个单词。理解此代码将使你能够使用语音命令控制任何电子项目。



与第 5 章一样，这个应用程序的源代码可以在 TensorFlow GitHub 代码仓库中找到。

我们将遵循与第 5 章类似的模式：首先详解测试，然后详解应用代码，最后详解使该示例在各种设备上运行的逻辑。

我们提供了将应用程序部署到以下设备的说明：

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



TensorFlow Lite 会定期添加对新设备的支持，因此如果你想要使用的设备未在此处列出，请查看该示例的 *README.md*。如果你在执行以下步骤时遇到麻烦，也可以在这里查看更新的部署说明。

这是一个比“Hello World”示例要复杂得多的应用程序，所以让我们从了解它的结构开始。

7.2 应用架构

在前面的几章中，你已经了解了机器学习应用会执行以下操作：

1. 获取输入。
2. 对输入进行预处理以提取适合输入模型的特征。
3. 对预处理后的输入进行推断。
4. 对模型的输出进行后处理以使其有意义。
5. 使用结果信息来使事情变为现实。

“Hello World”示例非常简单地遵循了这些步骤。这个示例使用一个简单的计数器生成的单个浮点数作为输入，并以另一个浮点数作为输出，我们直接将输出用于控制可视化的输出。

由于以下原因，我们的唤醒词检测应用要复杂一些：

- 唤醒词检测应用以音频数据作为输入。如你所见，在将音频数据输入模型之前，我们需要对数据进行大量处理。
- 唤醒词检测模型是分类器，它输出分类概率。因此我们需要解析并理解模型的输出。
- 唤醒词检测应用被设计成在实时数据上连续地进行推断。因此我们需要编写代码以理解各种推断。
- 唤醒词检测模型更大且更加复杂。因此我们会将我们的硬件推向极限。

由于这种复杂性很大程度上来自我们将要使用的模型，所以让我们来进一步了解它。

7.2.1 模型介绍

如前所述，我们在本章中使用的模型被训练为能够识别“yes”和“no”两个单词，并且能够区分未知词、沉默或背景噪声。

这个模型是在“语音命令数据集”（Speech Commands dataset）上训练的。这个数据集包含了 65 000 个由大众在线提供的 30 个单词的时长为一秒的发音。

尽管该数据集包含 30 个不同的单词，但是我们训练模型仅区分 4 个类别：单词“yes”、单词“no”、未知单词（数据集中的其他 28 个单词）和沉默。

我们的模型一次获取一秒的数据，并输出 4 个概率得分——每个类别一个得分，表示模型预测输入数据属于其中一个类别的可能性。

但是，该模型不包含原始音频样本数据，而是使用频谱图（spectrogram）。频谱图是由频率信息片组成的二维阵列，每个频率信息片均取自不同的时间窗口。

图 7-1 是从某人说“yes”的一秒音频剪辑生成的频谱图的直观表示，图 7-2 是单词“no”的频谱图。

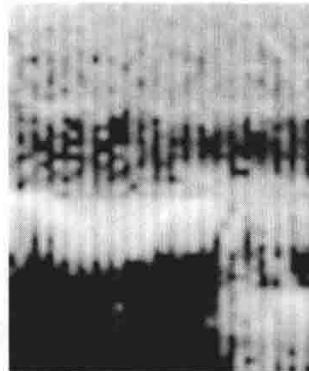


图 7-1：“yes”的频谱图

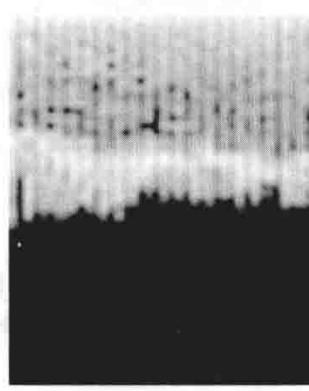


图 7-2：“no”的频谱图

通过在预处理过程中对频率信息进行分离，我们可以使模型的训练更加容易。在训练期间，模型无须学习如何解释原始音频数据；相反，模型可以直接在高层的抽象上提取出最有用的信息。

我们将在本章后面的内容中介绍频谱图的生成方式。现在，我们只需要知道模型将一个

频谱图作为输入即可。由于频谱图是二维数组，因此我们将其作为二维张量输入模型。有一种类型的神经网络结构专门用于处理多维张量，其中信息包含在相邻值组之间的关系中。这种类型的神经网络称为卷积神经网络（Convolutional Neural Network，CNN）。这类数据中最常见的例子是图像，其中一组相邻像素可能表示形状、图案或纹理。在训练过程中，CNN 可以识别这些特征并了解它们代表什么。

CNN 可以学习如何将简单的图像特征（如线条或边缘）组合成更复杂的特征（如眼睛或耳朵），以及如何将这些特征组合以形成输入图像，例如人脸的照片。这意味着 CNN 可以学习区分不同类别的输入图像，例如区分人的照片和狗的照片。

尽管 CNN 通常应用于二维像素网格的图像，但它也可以用于任何多维向量输入。事实证明，CNN 非常适合处理频谱图数据。

在第 8 章中，我们将了解这个模型是如何训练的。在此之前，让我们继续讨论我们应用程序的架构。

7.2.2 所有的功能组件

如前所述，我们的唤醒词应用程序比“Hello World”示例更为复杂。图 7-3 展示了组成该应用的组件。

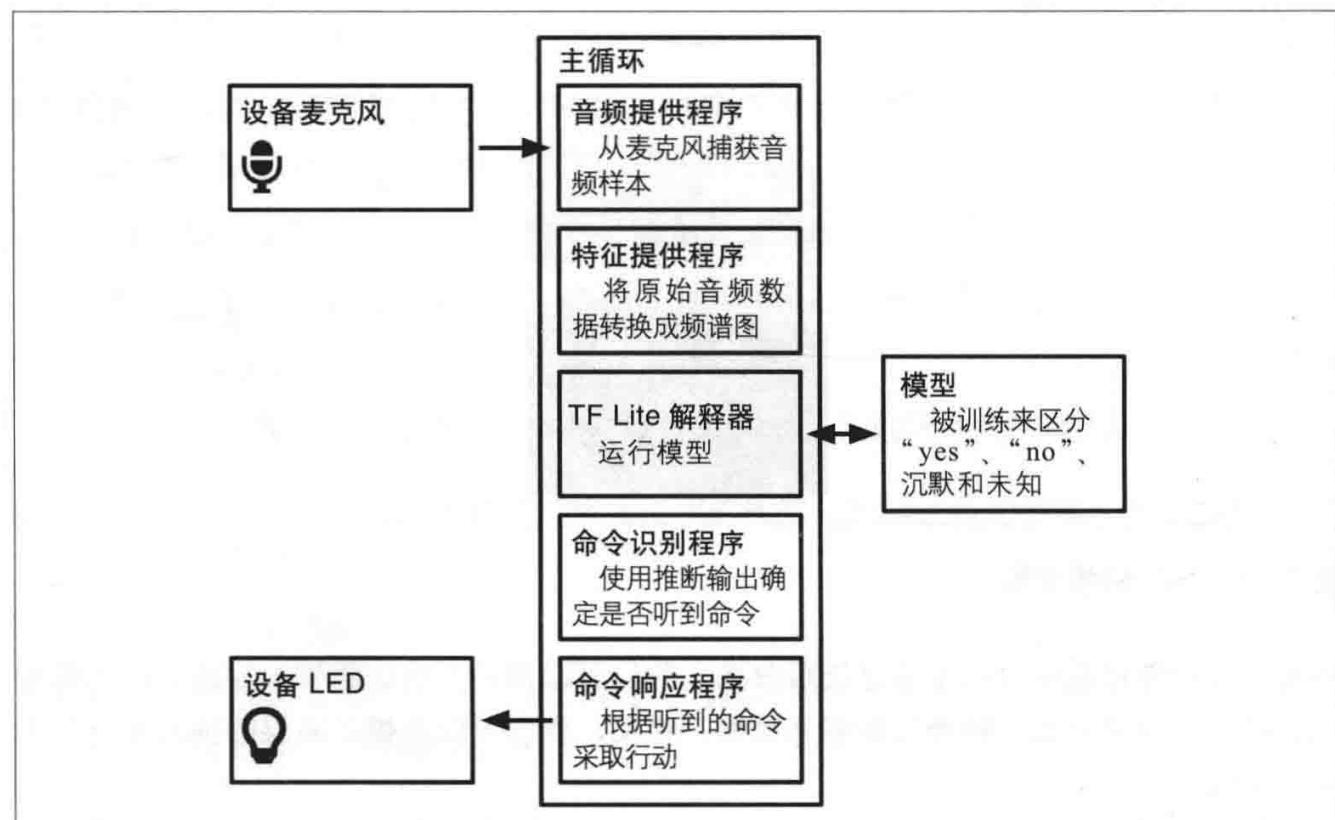


图 7-3：唤醒词检测应用的组件

让我们研究一下这些组件的作用：

主循环 (*main loop*)

与“Hello World”示例一样，我们的应用程序以连续循环的形式运行。所有的后续过程都包含在其中，并以微控制器可以运行它们的最快速度（每秒多次）连续执行。

音频提供程序 (*audio provider*)

音频提供程序从麦克风捕获原始音频数据。由于捕获音频的方法因设备而异，因此该组件可以被重写和自定义。

特征提供程序 (*feature provider*)

特征提供程序将原始音频数据转换为我们的模型所需的频谱图格式。它将作为主循环的一部分以滚动方式进行，为解释器提供了一系列重叠的一秒窗口。

TF Lite 解释器 (*TF Lite interpreter*)

解释器运行 TensorFlow Lite 模型，将输入的频谱图转换为一组概率。

模型 (*model*)

模型将作为一个数据组被包含在应用中，并由解释器运行。这个数组的位置是 `tiny_conv_micro_features_model_data.cc`。

命令识别程序 (*command recognizer*)

由于推断每秒运行多次，因此 `RecognizeCommands` 类将汇总所有结果并确定平均而言是否听到了一个已知的单词。

命令响应程序 (*command responder*)

如果听到命令，命令响应程序将使用设备的输出功能来通知用户。根据设备的不同，这可能意味着 LED 闪烁或在 LCD 显示器上显示数据。此部分可以针对不同的设备类型被重写。

GitHub 上的示例文件包含这些组件的测试。下一步我们将通过这些测试了解这些组件是如何工作的。

7.3 详解测试

与第 5 章一样，我们可以通过测试了解应用程序是如何工作的。由于我们已经介绍了很多 C++ 和 TensorFlow Lite 基础知识，因此我们不需要解释代码的每一行。相反，让我们专注于每个测试中最重要的部分，并解释发生了什么。

我们将探索以下测试，你可以在 GitHub 代码仓库中找到这些测试：

micro_speech_test.cc

演示如何对频谱图数据进行推断并解释结果。

audio_provider_test.cc

演示如何使用音频提供程序。

feature_provider_mock_test.cc

演示如何使用特征提供程序，使用音频提供程序的模拟（伪）实现来传递假数据。

recognize_commands_test.cc

演示如何解释模型的输出以确定是否检测到了命令。

command_responder_test.cc

演示如何调用命令响应程序来触发一个输出。

示例中还有更多测试，但是探索这几个测试将使我们能够对关键的功能部件有所了解。

7.3.1 基本流程

测试 *micro_speech_test.cc* 遵循与“Hello World”示例相同的基本流程：加载模型，设置解释器，并分配张量。

然而，这里有一个显著的区别。在“Hello World”示例中，我们使用 `AllOpsResolver` 加载运行模型所需的所有深度学习算子。这是一种可靠却很浪费的方法，因为 TensorFlow Lite 中可能有几种算子，但是给定的模型很可能不会使用全部算子。当部署到设备上时，这些不必要的算子将占用设备宝贵的内存，因此最好是只包含真正需要的算子。

为此，我们首先在测试文件的顶部定义模型需要的算子：

```
namespace tflite {
namespace ops {
namespace micro {
TfLiteRegistration* Register_DEPTHWISE_CONV_2D();
TfLiteRegistration* Register_FULLY_CONNECTED();
TfLiteRegistration* Register_SOFTMAX();
} // namespace micro
} // namespace ops
} // namespace tflite
```

接下来，我们像往常一样设置日志记录并加载模型：

```
// Set up logging.
tflite::MicroErrorReporter micro_error_reporter;
tflite::ErrorReporter* 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.
const tflite::Model* model =
    ::tflite::GetModel(g_tiny_conv_micro_features_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);
}
```

加载模型后，我们声明 `MicroMutableOpResolver` 并使用其方法 `AddBuiltin()` 添加前面列出的算子：

```
tflite::MicroMutableOpResolver micro_mutable_op_resolver;
micro_mutable_op_resolver.AddBuiltin(
    tflite::BuiltinOperator_DEPTHWISE_CONV_2D,
    tflite::ops::micro::Register_DEPTHWISE_CONV_2D());
micro_mutable_op_resolver.AddBuiltin(
    tflite::BuiltinOperator_FULLY_CONNECTED,
    tflite::ops::micro::Register_FULLY_CONNECTED());
micro_mutable_op_resolver.AddBuiltin(tflite::BuiltinOperator_SOFTMAX,
    tflite::ops::micro::Register_SOFTMAX());
```

你可能想知道我们是如何知道给定的模型包含了哪些算子的。一种方法是尝试使用 `MicroMutableOpResolver` 运行模型但是不调用 `AddBuiltin()`。推断将运行失败，附带的错误消息将通知我们缺少了哪些需要添加的算子。



`MicroMutableOpResolver` 在 `tensorflow/lite/micro/micro_mutable_op_resolver.h` 中定义，你需要将其添加到 `include` 语句中。

在设置完 `MicroMutableOpResolver` 之后，我们照常进行接下来的步骤，即设置解释器及其工作内存：

```
// Create an area of memory to use for input, output, and intermediate arrays.
const int tensor_arena_size = 10 * 1024;
uint8_t tensor_arena[tensor_arena_size];
// Build an interpreter to run the model with.
tflite::MicroInterpreter interpreter(model, micro_mutable_op_resolver, tensor_arena, tensor_arena_size, error_reporter);
interpreter.AllocateTensors();
```

在“Hello World”应用程序中，鉴于模型很小，我们仅为 `tensor_arena` 分配了 2×1024 字节。由于我们的语音模型更大，并且会处理更复杂的输入和输出，因此需要更多空间（ 10×1024 ）。这个大小是通过反复试验确定的。

接下来，我们检查输入张量的大小。不过，这次有点不同：

```
// Get information about the memory area to use for the model's input.  
TfLiteTensor* input = interpreter.input(0);  
// Make sure the input has the properties we expect.  
TF_LITE_MICRO_EXPECT_NE(nullptr, input);  
TF_LITE_MICRO_EXPECT_EQ(4, input->dims->size);  
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);  
TF_LITE_MICRO_EXPECT_EQ(49, input->dims->data[1]);  
TF_LITE_MICRO_EXPECT_EQ(40, input->dims->data[2]);  
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[3]);  
TF_LITE_MICRO_EXPECT_EQ(kTfLiteUInt8, input->type);
```

由于我们将频谱图作为输入，所以输入张量具有更多的维度——共 4 个。第一个维度只是一个包含单个元素的包装。第二个和第三个维度分别代表频谱图的“行”和“列”，恰好有 49 行 40 列。第四个维度，也就是输入张量最里面的维度（大小为 1）保存频谱图的每个“像素”。稍后，我们将详细介绍频谱图的结构。

接下来，我们获取存储在常量 `g_yes_micro_f2e59fea_nohash_1_data` 中的“yes”的样本频谱图。该常量在文件 `micro_features/yes_micro_features_data.cc` 中定义，该文件被包含在此测试中。频谱图以一维数组的形式存在，我们只需遍历它，将其复制到输入张量中：

```
// Copy a spectrogram created from a .wav audio file of someone saying "Yes"  
// into the memory area used for the input.  
const uint8_t* yes_features_data = g_yes_micro_f2e59fea_nohash_1_data;  
for (int i = 0; i < input->bytes; ++i) {  
    input->data.uint8[i] = yes_features_data[i];  
}
```

在给输入赋值之后，我们运行推断并检查输出张量的大小和形状：

```
// Run the model on this input and make sure 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);  
  
// Get the output from the model, and make sure it's the expected size and  
// type.  
TfLiteTensor* output = interpreter.output(0);  
TF_LITE_MICRO_EXPECT_EQ(2, output->dims->size);  
TF_LITE_MICRO_EXPECT_EQ(1, output->dims->data[0]);  
TF_LITE_MICRO_EXPECT_EQ(4, output->dims->data[1]);  
TF_LITE_MICRO_EXPECT_EQ(kTfLiteUInt8, output->type);
```

我们的输出有两个维度，第一维只是包装，第二维有四个元素。这种结构保留了四个类别（沉默、未知、“yes”和“no”）中每个类别匹配的概率。

下一部分代码将检查概率是否符合预期。因为输出张量中给定的一个元素总是代表一个确定的类别，所以我们知道用对应的索引去检查对应的元素。元素的顺序在训练期间被定义：

```
// There are four possible classes in the output, each with a score.  
const int kSilenceIndex = 0;  
const int kUnknownIndex = 1;  
const int kYesIndex = 2;  
const int kNoIndex = 3;  
  
// Make sure that the expected "Yes" score is higher than the other classes.  
uint8_t silence_score = output->data.uint8[kSilenceIndex];  
uint8_t unknown_score = output->data.uint8[kUnknownIndex];  
uint8_t yes_score = output->data.uint8[kYesIndex];  
uint8_t no_score = output->data.uint8[kNoIndex];  
TF_LITE_MICRO_EXPECT_GT(yes_score, silence_score);  
TF_LITE_MICRO_EXPECT_GT(yes_score, unknown_score);  
TF_LITE_MICRO_EXPECT_GT(yes_score, no_score);
```

由于我们传入了“yes”的频谱图，因此我们期望变量 `yes_score` 包含的概率高于 `silence_score`、`unknown_score` 和 `no_score`。

当我们对“yes”感到满意时，我们会对“no”频谱图执行相同的操作。首先，我们复制输入并运行推断：

```
// Now test with a different input, from a recording of "No".  
const uint8_t* no_features_data = g_no_micro_f9643d42_nohash_4_data;  
for (int i = 0; i < input->bytes; ++i) {  
    input->data.uint8[i] = no_features_data[i];  
}  
// Run the model on this "No" input.  
invoke_status = interpreter.Invoke();  
if (invoke_status != kTfLiteOk) {  
    error_reporter->Report("Invoke failed\n");  
}  
TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, invoke_status);
```

推断完成后，我们确认“no”获得了最高分：

```
// Make sure that the expected "No" score is higher than the other classes.  
silence_score = output->data.uint8[kSilenceIndex];  
unknown_score = output->data.uint8[kUnknownIndex];  
yes_score = output->data.uint8[kYesIndex];  
no_score = output->data.uint8[kNoIndex];  
TF_LITE_MICRO_EXPECT_GT(no_score, silence_score);  
TF_LITE_MICRO_EXPECT_GT(no_score, unknown_score);  
TF_LITE_MICRO_EXPECT_GT(no_score, yes_score);
```

这样就完成了！

要运行此测试，请从 TensorFlow 代码仓库的根目录运行以下命令：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
      test_micro_speech_test
```

接下来，让我们看一下所有音频数据的来源：音频提供程序。

7.3.2 音频提供程序

音频提供程序将设备的麦克风与我们的代码相连。每个设备都有着不同的机制来捕获音频。因此，*audio_provider.h* 定义了一个用于请求音频数据的接口，开发人员可以为他们想要支持的任何平台编写自己的实现。



该示例包括 Arduino、STM32F746G、SparkFun Edge 和 macOS 的音频提供程序实现。如果你希望此示例支持新设备，可以阅读现有的实现以了解如何实现新的支持。

音频提供程序的核心部分是一个名为 `GetAudioSamples()` 的函数，该函数在 *audio_provider.h* 中定义。这个函数看起来像这样：

```
TfLiteStatus GetAudioSamples(tfLite::ErrorReporter* error_reporter,
                             int start_ms, int duration_ms,
                             int* audio_samples_size, int16_t** audio_samples);
```

正如 *audio_provider.h* 中所描述的那样，该函数应返回一个 16 位脉冲编码调制（Pulse Code Modulated，PCM）音频数据的数组。这是一种非常常见的数字音频格式。

我们要使用 `ErrorReporter` 实例、开始时间 (`start_ms`)、持续时间 (`duration_ms`) 和两个指针来调用该函数。

这些指针是为 `GetAudioSamples()` 提供数据的机制。调用方声明适当类型的变量，然后在调用函数时将指针传递给它们。在函数的实现内部，我们将取消对指针的引用，并设置变量的值。

第一个指针 `audio_samples_size` 将接收音频数据中 16 位样本的总数。第二个指针 `audio_samples` 将接收一个包含音频数据本身的数组。

通过查看测试，我们可以看到这一点。*audio_provider_test.cc* 中有两个测试，但是我们只需要查看第一个即可了解如何使用音频提供程序：

```
TF_LITE_MICRO_TEST(TestAudioProvider) {
    tfLite::MicroErrorReporter micro_error_reporter;
    tfLite::ErrorReporter* error_reporter = &micro_error_reporter;

    int audio_samples_size = 0;
    int16_t* audio_samples = nullptr;
```

```

TfLiteStatus get_status =
    GetAudioSamples(error_reporter, 0, kFeatureSliceDurationMs,
                    &audio_samples_size, &audio_samples);
TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, get_status);
TF_LITE_MICRO_EXPECT_LE(audio_samples_size, kMaxAudioSampleSize);
TF_LITE_MICRO_EXPECT_NE(audio_samples, nullptr);

// Make sure we can read all of the returned memory locations.
int total = 0;
for (int i = 0; i < audio_samples_size; ++i) {
    total += audio_samples[i];
}
}

```

这段测试展示了如何使用一些值和指针来调用 `GetAudioSamples()`，并且确认了在调用函数后已正确分配了指针。



你会注意到一些常量的使用，如 `kFeatureSliceDurationMs` 和 `kMaxAudioSampleSize`。这些是在训练模型时选择的值，你可以在 `micro_features/micro_model_settings.h` 中找到它们。

`audio_provider.cc` 的默认实现只是返回一个空数组。为了证明该数组的大小是正确的，测试只是针对期望的样本数量对其进行循环遍历。

除了 `GetAudioSamples()`，音频提供程序还包含一个名为 `LatestAudioTimestamp()` 的函数。这是为了返回上次捕获音频数据的时间（以毫秒为单位）。特征提供程序需要此信息来确定所要获取的音频数据。

要运行音频提供程序的测试，请使用以下命令：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
test_audio_provider_test
```

特征提供程序将音频提供程序用作新音频样本的来源，下面将对其进行介绍。

7.3.3 特征提供程序

特征提供程序将从音频提供程序获得的原始音频转换成可以输入我们模型的频谱图，它会在主循环中被调用。

特征提供程序的接口定义在 `feature_provider.h` 中，如下所示：

```

class FeatureProvider {
public:
    // Create the provider, and bind it to an area of memory. This memory should
    // remain accessible for the lifetime of the provider object, since subsequent
    // calls will fill it with feature data. The provider does no memory

```

```

// management of this data.
FeatureProvider(int feature_size, uint8_t* feature_data);
~FeatureProvider();

// Fills the feature data with information from audio inputs, and returns how
// many feature slices were updated.
TfLiteStatus PopulateFeatureData(tflite::ErrorReporter* error_reporter,
                                 int32_t last_time_in_ms, int32_t time_in_ms,
                                 int* how_many_new_slices);

private:
    int feature_size_;
    uint8_t* feature_data_;
// Make sure we don't try to use cached information if this is the first call
// into the provider.
    bool is_first_run_;
};

```

要了解其用法，可以查看 *feature_provider_mock_test.cc* 中的测试。

为了让特征提供程序能够使用音频数据，这些测试使用了音频提供程序的一个特殊的伪（模拟）实现，它被设置为提供音频数据。它在 *audio_provider_mock.cc* 中定义。



模拟音频提供程序代替了测试的构建说明中的真实实现，你可以在 **FEATURE_PROVIDER_MOCK_TEST_SRCS** 下的 *Makefile.inc* 中找到该部分。

feature_provider_mock_test.cc 文件中包含了两个测试。这是第一个测试：

```

TF_LITE_MICRO_TEST(TestFeatureProviderMockYes) {
    tflite::MicroErrorReporter micro_error_reporter;
    tflite::ErrorReporter* error_reporter = &micro_error_reporter;

    uint8_t feature_data[kFeatureElementCount];
    FeatureProvider feature_provider(kFeatureElementCount, feature_data);

    int how_many_new_slices = 0;
    TfLiteStatus populate_status = feature_provider.PopulateFeatureData(
        error_reporter, /* last_time_in_ms= */ 0, /* time_in_ms= */ 970,
        &how_many_new_slices);
    TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, populate_status);
    TF_LITE_MICRO_EXPECT_EQ(kFeatureSliceCount, how_many_new_slices);

    for (int i = 0; i < kFeatureElementCount; ++i) {
        TF_LITE_MICRO_EXPECT_EQ(g_yes_micro_f2e59fea_nohash_1_data[i],
                               feature_data[i]);
    }
}

```

为了创建 *FeatureProvider*，我们需要调用其构造函数，并传入 *feature_size* 和 *feature_data* 作为参数：

```
FeatureProvider feature_provider(kFeatureElementCount, feature_data);
```

第一个参数表明了频谱图中总共应该有多少个数据元素，第二个参数是指向数组的指针，我们希望使用频率图数据填充这个数组。

频谱图中的元素数量是在训练模型时确定的，并在 *micro_features/micro_model_settings.h* 中定义为 **kFeatureElementCount**。

为了获取音频的最后一秒的特征，**feature_provider.PopulateFeatureData()** 将被调用：

```
TfLiteStatus populate_status = feature_provider.PopulateFeatureData(  
    error_reporter, /* last_time_in_ms= */ 0, /* time_in_ms= */ 970,  
    &how_many_new_slices);
```

我们提供一个 **ErrorReporter** 实例、一个代表该方法上次调用时间的整数 (**last_time_in_ms**)、当前时间 (**time_in_ms**) 以及一个指向整数的指针，该整数将随着接收到的新特征切片 (feature slice) 的数量 (**how_many_new_slices**) 而更新。切片只是频谱图中的一行，代表一秒。

因为我们总是想要最后一秒的音频，因此特征提供程序将比较上次调用它的时间 (**last_time_in_ms**) 与当前时间 (**time_in_ms**)，并从在这段时间内捕获的音频中创建频谱图数据，然后更新 **feature_data** 数组以添加任何额外的切片，同时丢弃任何超过一秒的切片。

当 **PopulateFeatureData()** 运行时，它将从模拟音频提供程序中请求音频。该模拟程序将为其提供代表“yes”的音频，特征提供程序将对音频进行处理并提供结果。

在调用 **PopulateFeatureData()** 之后，我们将检查其结果是否符合预期。我们将生成的数据与已知的频谱图进行比较，该频谱图对于模拟音频提供程序给出的“yes”输入是正确的：

```
TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, populate_status);  
TF_LITE_MICRO_EXPECT_EQ(kFeatureSliceCount, how_many_new_slices);  
for (int i = 0; i < kFeatureElementCount; ++i) {  
    TF_LITE_MICRO_EXPECT_EQ(g_yes_micro_f2e59fea_nohash_1_data[i],  
                            feature_data[i]);  
}
```

模拟音频提供程序可以根据传入的开始时间和结束时间来提供“yes”或“no”的音频。*feature_provider_mock_test.cc* 中的第二个测试与第一个测试完全相同，只是音频表示的是“no”。

要运行测试，请使用以下命令：

```
make -f tensorflow/lite/micro/tools/make/Makefile \  
test_feature_provider_mock_test
```

特征提供程序如何将音频转换为频谱图

特征提供程序的实现位于 `feature_provider.cc`。下面让我们来介绍它的工作原理。

正如我们之前所讨论的，特征提供程序的工作是填充一个表示一秒音频频谱图的数组。该程序被设计成循环调用，因此为了避免不必要的工作，特征提供程序只会在其上次被调用的时间和当前时间之间生成新的特征。如果该程序在不到一秒之前刚被调用过，它将保留一些之前的输出并且仅生成缺失的部分。

在我们的代码中，每个频谱图都表示为一个具有 40 列 49 行的二维数组，其中每一行代表一个 30 毫秒 (ms)、被分为 43 个频率桶 (bucket) 的音频样本。

为了创建每一行数据，我们通过快速傅里叶变换 (Fast Fourier Transform, FFT) 算法处理一个 30 毫秒的音频输入切片。FFT 分析样本中音频的频率分布，并创建一个包含 256 个频率桶的数组，每个频率桶的值是 0~255 的一个数。这些频率桶被平均分为 6 组，每组约 43 个频率桶。

执行此操作的代码在文件 `micro_features/micro_features_generator.cc` 中，并由特征提供程序调用。

为了构建整个二维数组，我们将对 49 个连续的 30 毫秒音频切片运行 FFT 的结果进行组合，每个切片与前一个切片有 10 毫秒的重叠。图 7-4 展示了这是如何发生的。

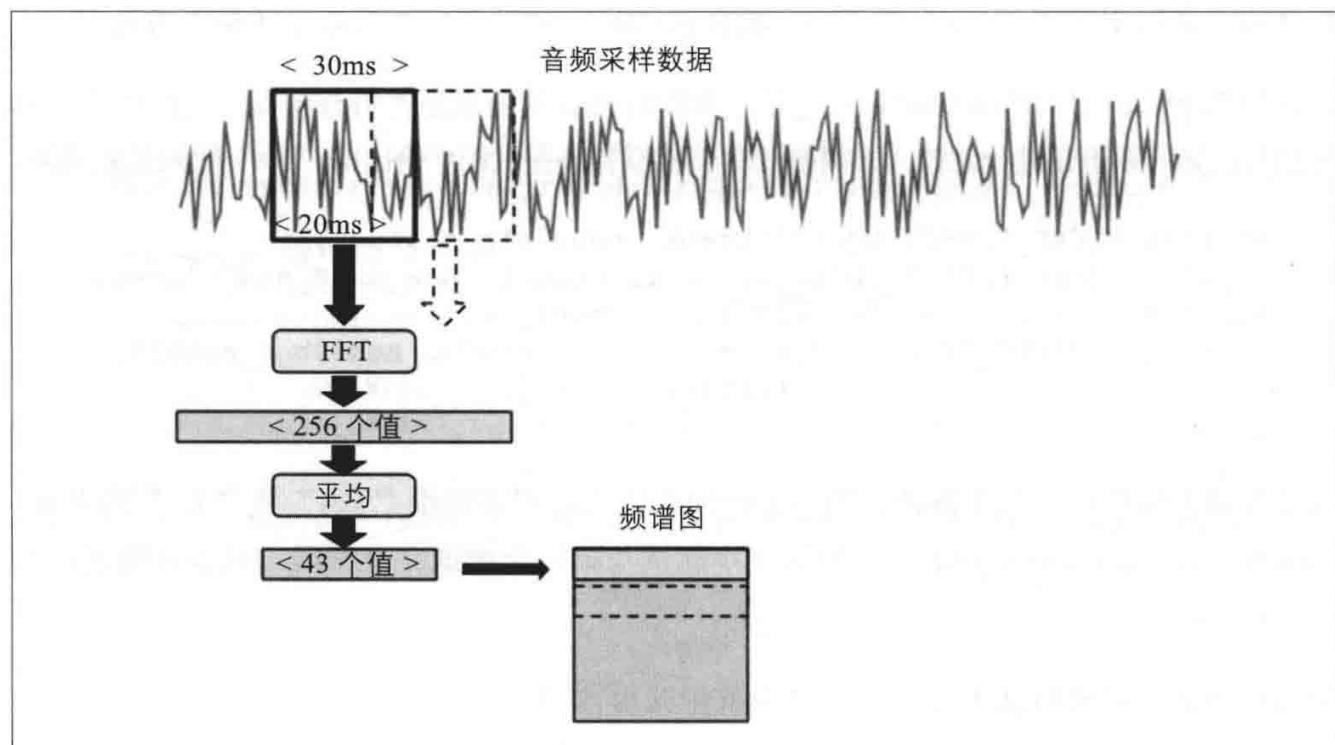


图 7-4：正在被处理的音频采样图

你可以看到 30 毫秒的采样窗口如何每次向前移动 20 毫秒，直到覆盖整个一秒的采样为止。至此，生成的频谱图已准备好传递到我们的模型中。

可以在 *feature_provider.cc* 中了解此过程的运作方式。首先，程序将根据上次调用 *PopulateFeatureData()* 的时间来确定实际需要生成的切片：

```
// Quantize the time into steps as long as each window stride, so we can
// figure out which audio data we need to fetch.
const int last_step = (last_time_in_ms / kFeatureSliceStrideMs);
const int current_step = (time_in_ms / kFeatureSliceStrideMs);

int slices_needed = current_step - last_step;
```

如果程序从未运行过，或者距离上次运行超过了一秒，它将生成所需的最大数量的切片：

```
if (is_first_run_) {
    TfLiteStatus init_status = InitializeMicroFeatures(error_reporter);
    if (init_status != kTfLiteOk) {
        return init_status;
    }
    is_first_run_ = false;
    slices_needed = kFeatureSliceCount;
}
if (slices_needed > kFeatureSliceCount) {
    slices_needed = kFeatureSliceCount;
}
*how_many_new_slices = slices_needed;
```

生成的数字将被写入 *how_many_new_slices*。

接下来，程序将计算应被保留的现有切片的数量，并移动数组中的数据来为任何新切片腾出空间：

```
const int slices_to_keep = kFeatureSliceCount - slices_needed;
const int slices_to_drop = kFeatureSliceCount - slices_to_keep;
// If we can avoid recalculating some slices, just move the existing data
// up in the spectrogram, to perform something like this:
// last time = 80ms           current time = 120ms
// +-----+           +-----+
// | data@20ms |           --> | data@60ms |
// +-----+           --   +-----+
// | data@40ms |           --> | data@80ms |
// +-----+           --   +-----+
// | data@60ms |           --   | <empty> |
// +-----+           --   +-----+
// | data@80ms |           --   | <empty> |
// +-----+
if (slices_to_keep > 0) {
    for (int dest_slice = 0; dest_slice < slices_to_keep; ++dest_slice) {
        uint8_t* dest_slice_data =
            feature_data_ + (dest_slice * kFeatureSliceSize);
```

```
    const int src_slice = dest_slice + slices_to_drop;
    const uint8_t* src_slice_data =
        feature_data_ + (src_slice * kFeatureSliceSize);
    for (int i = 0; i < kFeatureSliceSize; ++i) {
        dest_slice_data[i] = src_slice_data[i];
    }
}
}
```



如果你是一位经验丰富的 C++ 程序员，你可能会想知道为什么我们不使用标准库来做像复制数据之类的事情。我们这么做的原因是试图避免不必要的依赖关系，以保持较小的二进制文件。由于嵌入式平台的内存很少，因此较小的应用程序二进制文件意味着我们有空间容纳更大、更准确的深度学习模型。

在移动数据之后，程序将开始一个循环，该循环将为其需要的每个切片迭代一次。在这个循环中，程序首先使用 `GetAudioSamples()` 从音频提供程序请求该片段的音频：

```
for (int new_slice = slices_to_keep; new_slice < kFeatureSliceCount;
      ++new_slice) {
    const int new_step = (current_step - kFeatureSliceCount + 1) + new_slice;
    const int32_t slice_start_ms = (new_step * kFeatureSliceStrideMs);
    int16_t* audio_samples = nullptr;
    int audio_samples_size = 0;
    GetAudioSamples(error_reporter, slice_start_ms, kFeatureSliceDurationMs,
                    &audio_samples_size, &audio_samples);
    if (audio_samples_size < kMaxAudioSampleSize) {
        error_reporter->Report("Audio data size %d too small, want %d",
                               audio_samples_size, kMaxAudioSampleSize);
        return kTfLiteError;
    }
}
```

为了完成循环迭代，程序将数据传递到在 `micro_features/micro_features_generator.h` 中定义的 `GenerateMicroFeatures()` 中，这是执行 FFT 并返回音频信息的函数。

函数还传递了一个指针 `new_slice_data`，该指针指向应该写入新数据的内存位置：

```
uint8_t* new_slice_data = feature_data_ + (new_slice * kFeatureSliceSize);
size_t num_samples_read;
TfLiteStatus generate_status = GenerateMicroFeatures(
    error_reporter, audio_samples, audio_samples_size, kFeatureSliceSize,
    new_slice_data, &num_samples_read);
if (generate_status != kTfLiteOk) {
    return generate_status;
}
}
```

在对每个切片的完成这一过程之后，我们得到了一秒的最新频谱图。



生成 FFT 的函数是 `GenerateMicroFeatures()`。如果你有兴趣，可以在 `micro_features/micro_features_generator.cc` 中查看它的定义。如果你正在创建使用频谱图的应用程序，你可以直接重用此代码。在训练模型时，你需要使用相同的代码将数据预处理成频谱图。

一旦有了频谱图，我们就可以使用模型对其进行推断。在这之后，我们需要对推断的结果进行解释。该任务属于我们接下来探讨的 `RecognizeCommands` 类。

7.3.4 命令识别程序

在我们的模型输出一组已知单词在音频最后一秒出现的概率之后，`RecognizeCommands` 类的工作就是确定这是否表示检测成功。

这似乎很简单：如果一个给定类别的概率大于某个阈值，那么检测结果就是该类别代表的词。但是在现实世界中，事情往往会更加复杂。

正如我们之前所看到的，我们的程序每秒运行多个推断，每个推断都运行在一个一秒的数据窗口之上。这意味着我们将在多个窗口中多次对任何给定单词运行推断。

在图 7-5 中，你会看到说出“noted”一词的波形，周围有一个代表正在捕获的一秒窗口的方框。

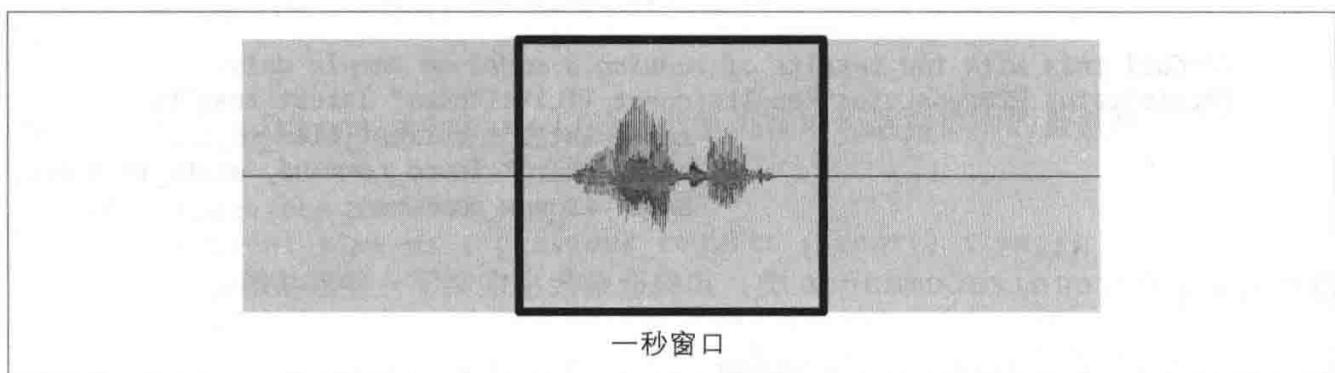


图 7-5：“noted”在窗口中被捕获

我们的模型被训练来检测“no”这个词，它能理解“noted”与“no”不同。如果我们在这一秒的窗口上进行推断，则（希望）输出“no”一词的可能性很小。但是，如果窗口在音频流中稍早出现，如图 7-6 所示，我们该怎么办？

在这种情况下，窗口中出现的单词“noted”的唯一部分是它的第一个音节。由于“noted”的第一个音节听起来像“no”，因此模型将其解释为“no”的可能性很高。

这些问题意味着我们不能依靠一个单一的推断来告诉我们是否出现了一个词。这就是 `RecognizeCommands` 发挥作用的地方！

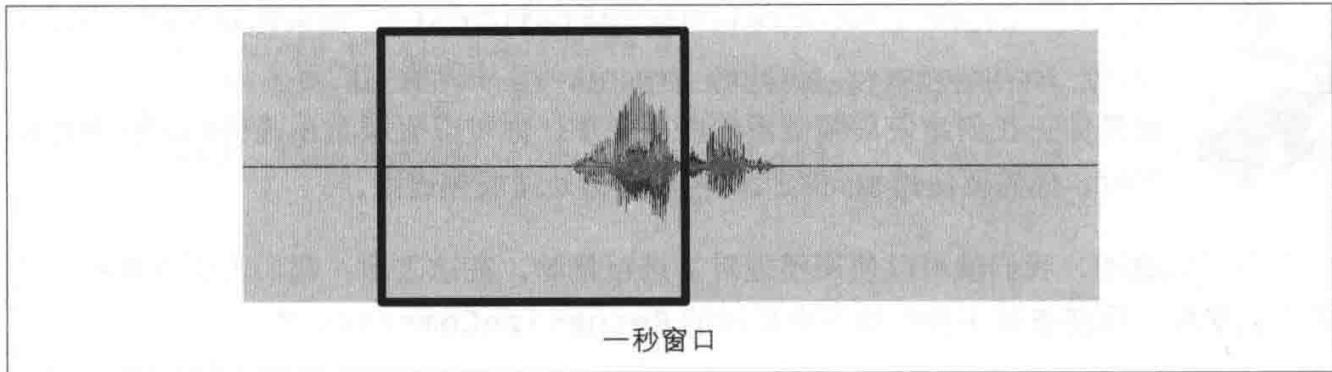


图 7-6: “noted” 的一部分在窗口被捕获

命令识别程序会计算过去几个推断中每个单词的平均分数，并决定这些分数是否足够高，以致可以算作一次检测。要做到这一点，我们要将每个推断结果都输入命令识别程序中。

你可以在 `recognize_commands.h` 中看到命令识别程序的接口，这里是部分代码：

```
class RecognizeCommands {
public:
    explicit RecognizeCommands(tfLite::ErrorReporter* error_reporter,
                               int32_t average_window_duration_ms = 1000,
                               uint8_t detection_threshold = 200,
                               int32_t suppression_ms = 1500,
                               int32_t minimum_count = 3);

    // Call this with the results of running a model on sample data.
    TfLiteStatus ProcessLatestResults(const TfLiteTensor* latest_results,
                                      const int32_t current_time_ms,
                                      const char** found_command, uint8_t* score,
                                      bool* is_new_command);
}
```

这里定义了 `RecognizeCommands` 类，其构造函数也定义了一些默认值：

- 平均窗口的时长 (`average_window_duration_ms`)。
- 被视为检测出命令的最低平均分数 (`detection_threshold`)。
- 听到第一个命令后，在识别第二个命令之前所等待的时间 (`suppression_ms`)。
- 窗口中计算结果所需的最小推断次数 (`minimum_count`)。

`RecognizeCommands` 类有一个叫作 `ProcessLatestResults()` 的方法，该方法会接收一个指向 `TFLite Tensor` 用来接收模型输出的指针 (`latest_results`)，并且必须使用当前时间 (`current_time_ms`) 来调用该方法。

除此之外，该方法还需要三个用来输出的指针。第一个指针给出了所检测到的单词的名称 (`found_command`)，第二个指针提供了所检测到命令的平均分数 (`score`)，第三

个指针表明该命令是新命令还是在以前某个时间段内的推断中听到过的命令。

在处理时序数据时，对多个推断的结果取平均值是一种有用且通用的技术。接下来我们将遍历 `recognize_commands.cc` 中的代码，并了解其工作原理。你不需要了解每一行代码，但是可以深入了解一下那些可能会对你的项目有所帮助的部分。

首先，确保输入张量的形状和类型是正确的：

```
TfLiteStatus RecognizeCommands::ProcessLatestResults(
    const TfLiteTensor* latest_results, const int32_t current_time_ms,
    const char** found_command, uint8_t* score, bool* is_new_command) {
    if ((latest_results->dims->size != 2) ||
        (latest_results->dims->data[0] != 1) ||
        (latest_results->dims->data[1] != kCategoryCount)) {
        error_reporter_->Report(
            "The results for recognition should contain %d elements, but there are "
            "%d in an %d-dimensional shape",
            kCategoryCount, latest_results->dims->data[1],
            latest_results->dims->size);
        return kTfLiteError;
    }

    if (latest_results->type != kTfLiteUInt8) {
        error_reporter_->Report(
            "The results for recognition should be uint8 elements, but are %d",
            latest_results->type);
        return kTfLiteError;
    }
}
```

接下来，检查 `current_time_ms` 以验证其是否在平均值窗口中的最新结果之后：

```
if ((!previous_results_.empty()) &&
    (current_time_ms < previous_results_.front().time_)) {
    error_reporter_->Report(
        "Results must be fed in increasing time order, but received a "
        "timestamp of %d that was earlier than the previous one of %d",
        current_time_ms, previous_results_.front().time_);
    return kTfLiteError;
}
```

之后，将最新的结果添加到将取平均值的结果列表中：

```
// Add the latest results to the head of the queue.
previous_results_.push_back({current_time_ms, latest_results->data.uint8});
// Prune any earlier results that are too old for the averaging window.
const int64_t time_limit = current_time_ms - average_window_duration_ms_;
while (!previous_results_.empty() &&
       previous_results_.front().time_ < time_limit) {
    previous_results_.pop_front();
```

如果平均窗口中的结果数量少于最小值（由 `minimum_count_` 定义，默认为 3），则

我们无法提供有效的平均值。在这种情况下，我们会设置输出指针来表明 `found_command` 是最近一个已检测出的命令，同时分数为 0，且当前命令不是一个新命令：

```
// If there are too few results, assume the result will be unreliable and
// bail.
const int64_t how_many_results = previous_results_.size();
const int64_t earliest_time = previous_results_.front().time_;
const int64_t samples_duration = current_time_ms - earliest_time;
if ((how_many_results < minimum_count_) ||
    (samples_duration < (average_window_duration_ms_ / 4))) {
    *found_command = previous_top_label_;
    *score = 0;
    *is_new_command = false;
    return kTfLiteOk;
}
```

否则，将对窗口中的所有分数求平均值：

```
// Calculate the average score across all the results in the window.
int32_t average_scores[kCategoryCount];
for (int offset = 0; offset < previous_results_.size(); ++offset) {
    PreviousResultsQueue::Result previous_result =
        previous_results_.from_front(offset);
    const uint8_t* scores = previous_result.scores_;
    for (int i = 0; i < kCategoryCount; ++i) {
        if (offset == 0) {
            average_scores[i] = scores[i];
        } else {
            average_scores[i] += scores[i];
        }
    }
}
for (int i = 0; i < kCategoryCount; ++i) {
    average_scores[i] /= how_many_results;
}
```

现在，我们有足够的信息来确定哪一个类别胜出。确定该结果的过程很简单：

```
// Find the current highest scoring category.
int current_top_index = 0;
int32_t current_top_score = 0;
for (int i = 0; i < kCategoryCount; ++i) {
    if (average_scores[i] > current_top_score) {
        current_top_score = average_scores[i];
        current_top_index = i;
    }
}
const char* current_top_label = kCategoryLabels[current_top_index];
```

最后一段逻辑将确定结果是否是一次有效的检测。为此，我们将确保结果的得分高于检测阈值（默认为 200），并且不是在上次的有效检测后很快发生的——这种情况可能表明结果有误：

```

// If we've recently had another label trigger, assume one that occurs too
// soon afterwards is a bad result.
int64_t time_since_last_top;
if ((previous_top_label_ == kCategoryLabels[0]) ||
    (previous_top_label_time_ == std::numeric_limits<int32_t>::min())) {
    time_since_last_top = std::numeric_limits<int32_t>::max();
} else {
    time_since_last_top = current_time_ms - previous_top_label_time_;
}
if ((current_top_score > detection_threshold_) &&
    ((current_top_label != previous_top_label_) ||
     (time_since_last_top > suppression_ms_))) {
    previous_top_label_ = current_top_label;
    previous_top_label_time_ = current_time_ms;
    *is_new_command = true;
} else {
    *is_new_command = false;
}
*found_command = current_top_label;
*score = current_top_score;

```

如果结果是有效的，`is_new_command`会被设为 `true`。调用者可以使用该值来确定是否真的检测到了单词。

`recognize_commands_test.cc` 中的测试对存储在平均窗口中的输入和结果进行各种不同的组合。

让我们看一下其中的一项测试 `RecognizeCommandsTestBasic`，该测试演示了 `RecognizeCommands` 的用法。首先，我们创建该类的一个实例：

```

TF_LITE_MICRO_TEST(RecognizeCommandsTestBasic) {
    tflite::MicroErrorReporter micro_error_reporter;
    tflite::ErrorReporter* error_reporter = &micro_error_reporter;

    RecognizeCommands recognize_commands(error_reporter);

```

接下来，我们创建一个包含了一些假推断结果的张量，`ProcessLatestResults()` 将使用该张量来确定是否听到了命令：

```

TfLiteTensor results = tflite::testing::CreateQuantizedTensor(
    {255, 0, 0, 0}, tflite::testing::IntArrayFromInitializer({2, 1, 4}),
    "input_tensor", 0.0f, 128.0f);

```

然后，我们设置一些变量，这些变量将通过 `ProcessLatestResults()` 的输出进行设置：

```

const char* found_command;
uint8_t score;
bool is_new_command;

```

最后，我们调用 `ProcessLatestResults()`，提供指向以上变量的指针，以及包含着推断结

果的张量。我们断言 (assert) 该函数将返回 kTfLiteOk，表明输入已成功处理：

```
TF_LITE_MICRO_EXPECT_EQ(  
    kTfLiteOk, recognize_commands.ProcessLatestResults(  
        &results, 0, &found_command, &score, &is_new_command));
```

该文件中的其他测试将执行更详尽的检查，以确保函数正常运行。你可以通读代码以了解更多信息。

要运行所有测试，请使用以下命令：

```
make -f tensorflow/lite/micro/tools/make/Makefile \  
    test_recognize_commands_test
```

一旦确定检测到了命令，就可以与世界分享我们的结果（至少可以通过我们的板载 LED）。命令响应程序可以实现这部分功能。

7.3.5 命令响应程序

我们难题的最后一个部分是命令响应程序，它的功能是产生一个输出，让我们知道一个单词被检测到了。

命令响应程序被设计为可以针对不同类型的设备进行重写。我们将在本章后面探讨特定于设备的实现。

现在，让我们看一下其非常简单的参考实现，该参考实现仅将检测结果记录为文本。你可以在文件 *command_responder.cc* 中找到其代码：

```
void RespondToCommand(tflite::ErrorReporter* error_reporter,  
                      int32_t current_time, const char* found_command,  
                      uint8_t score, bool is_new_command) {  
    if (is_new_command) {  
        error_reporter->Report("Heard %s (%d) @%dms", found_command, score,  
                               current_time);  
    }  
}
```

仅此而已！这个文件仅实现一个函数：`RespondToCommand()`。该函数需要 `error_reporter`、当前时间 (`current_time`)、最后检测到的命令 (`found_command`)、该命令的分数 (`score`) 以及是否刚刚听到该命令 (`is_new_command`)。

请务必注意，在程序的主循环中，即使未检测到命令，每次执行推断时都会调用此函数。这意味着我们应该检查 `is_new_command` 以确定是否需要做任何事情。

command_responder_test.cc 中对此函数的测试同样简单。我们只是调用该函数，因为我们无法测试生成的输出是否正确：

```
TF_LITE_MICRO_TEST(TestCallability) {
    tflite::MicroErrorReporter micro_error_reporter;
    tflite::ErrorReporter* error_reporter = &micro_error_reporter;

    // This will have external side-effects (like printing to the debug console
    // or lighting an LED) that are hard to observe, so the most we can do is
    // make sure the call doesn't crash.
    RespondToCommand(error_reporter, 0, "foo", 0, true);
}
```

要运行此测试，请向终端输入以下内容：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
test_command_responder_test
```

就是这些！我们已经遍历了应用程序的所有组件。现在，让我们看看这些组件如何在程序中被组合在一起。

7.4 监听唤醒词

你可以在 `main_functions.cc` 中找到以下代码，该代码定义了 `setup()` 和 `loop()` 函数，这两个函数是我们程序的核心。让我们一起了解它们吧！

由于你现在已经是一个经验丰富的 TensorFlow Lite 专家了，很多代码应该看起来都很熟悉。所以让我们试着把注意力放在新的方面。

首先，列出要使用的算子：

```
namespace tflite {
namespace ops {
namespace micro {
TfLiteRegistration* Register_DEPTHWISE_CONV_2D();
TfLiteRegistration* Register_FULLY_CONNECTED();
TfLiteRegistration* Register_SOFTMAX();
} // namespace micro
} // namespace ops
} // namespace tflite
```

接下来，设置全局变量：

```
namespace {
tflite::ErrorReporter* error_reporter = nullptr;
const tflite::Model* model = nullptr;
tflite::MicroInterpreter* interpreter = nullptr;
TfLiteTensor* model_input = nullptr;
FeatureProvider* feature_provider = nullptr;
RecognizeCommands* recognizer = nullptr;
int32_t previous_time = 0;

// Create an area of memory to use for input, output, and intermediate arrays.
```

```

// The size of this will depend on the model you're using, and may need to be
// determined by experimentation.
constexpr int kTensorArenaSize = 10 * 1024;
uint8_t tensor_arena[kTensorArenaSize];
} // namespace

```

请注意除了常见的 TensorFlow 对象之外我们是如何声明 `FeatureProvider` 和 `RecognizeCommands` 的。我们还声明了一个名为 `g_previous_time` 的变量，该变量跟踪我们最近一次收到新音频采样的时间。

接下来，在 `setup()` 函数中，我们加载模型、设置解释器、添加算子并分配张量：

```

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_tiny_conv_micro_features_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;
    }

    // Pull in only the operation implementations we need.
    static tflite::MicroMutableOpResolver micro_mutable_op_resolver;
    micro_mutable_op_resolver.AddBuiltin(
        tflite::BuiltinOperator_DEPTHWISE_CONV_2D,
        tflite::ops::micro::Register_DEPTHWISE_CONV_2D());
    micro_mutable_op_resolver.AddBuiltin(
        tflite::BuiltinOperator_FULLY_CONNECTED,
        tflite::ops::micro::Register_FULLY_CONNECTED());
    micro_mutable_op_resolver.AddBuiltin(tflite::BuiltinOperator_SOFTMAX,
                                         tflite::ops::micro::Register_SOFTMAX());

    // Build an interpreter to run the model with.
    static tflite::MicroInterpreter static_interpreter(
        model, micro_mutable_op_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;
    }
}

```

分配张量后，检查输入张量的形状和类型是否正确：

```

// Get information about the memory area to use for the model's input.
model_input = interpreter->input(0);
if ((model_input->dims->size != 4) || (model_input->dims->data[0] != 1) ||
    (model_input->dims->data[1] != kFeatureSliceCount) ||
    (model_input->dims->data[2] != kFeatureSliceSize) ||
    (model_input->type != kTfLiteUInt8)) {
    error_reporter->Report("Bad input tensor parameters in model");
    return;
}

```

接下来是一些很有趣的东西。首先，实例化 FeatureProvider，将其指向输入张量：

```

// Prepare to access the audio spectrograms from a microphone or other source
// that will provide the inputs to the neural network.
static FeatureProvider static_feature_provider(kFeatureElementCount,
                                              model_input->data.uint8);
feature_provider = &static_feature_provider;

```

然后，创建一个 RecognizeCommands 实例并初始化 previous_time 变量：

```

static RecognizeCommands static_recognizer(error_reporter);
recognizer = &static_recognizer;

previous_time = 0;
}

```

接下来，是时候使用 loop() 函数了。与前面的示例一样，该函数将无限期地反复调用。在循环中，我们首先使用特征提供程序创建一个频谱图：

```

void loop() {
    // Fetch the spectrogram for the current time.
    const int32_t current_time = LatestAudioTimestamp();
    int how_many_new_slices = 0;
    TfLiteStatus feature_status = feature_provider->PopulateFeatureData(
        error_reporter, previous_time, current_time, &how_many_new_slices);
    if (feature_status != kTfLiteOk) {
        error_reporter->Report("Feature generation failed");
        return;
    }
    previous_time = current_time;
    // If no new audio samples have been received since last time, don't bother
    // running the network model.
    if (how_many_new_slices == 0) {
        return;
    }
}

```

如果自上次迭代以来没有新数据，我们就不必费心运行推断。

当得到输入之后，我们只需要调用解释器：

```

// Run the model on the spectrogram input and make sure it succeeds.
TfLiteStatus invoke_status = interpreter->Invoke();

```

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

模型的输出张量现在被每个类别的概率填充。为了解释模型输出，我们使用了 `RecognizeCommands` 实例。我们准备一个指向输出张量的指针，然后设置一些变量以接收 `ProcessLatestResults()` 的输出：

```
// Obtain a pointer to the output tensor
TfLiteTensor* output = interpreter->output(0);
// Determine whether a command was recognized based on the output of inference
const char* found_command = nullptr;
uint8_t score = 0;
bool is_new_command = false;
TfLiteStatus process_status = recognizer->ProcessLatestResults(
    output, current_time, &found_command, &score, &is_new_command);
if (process_status != kTfLiteOk) {
    error_reporter->Report("RecognizeCommands::ProcessLatestResults() failed");
    return;
}
```

最后，我们调用命令响应程序的 `RespondToCommand()` 方法，以便可以在检测到单词时通知用户：

```
// Do something based on the recognized command. The default implementation
// just prints to the error console, but you should replace this with your
// own function for a real application.
RespondToCommand(error_reporter, current_time, found_command, score,
    is_new_command);
}
```

以上，就是这样！调用 `RespondToCommand()` 是循环中要做的最后一件事。从特征生成开始的所有操作都将无休止地重复进行，以检查音频中的已知单词并在确认一个单词时产生一些输出。

`setup()` 和 `loop()` 函数由 `main.cc` 中定义的 `main()` 函数调用，该函数在应用程序启动时开始循环：

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

运行我们的应用

该示例包含与 macOS 兼容的音频提供程序。如果你可以使用 Mac，则可以在开发计算

机上运行该示例。首先，使用以下命令开始编译：

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

在编译完成后，你可以使用以下命令运行示例：

```
tensorflow/lite/micro/tools/make/gen/osx_x86_64/bin/micro_speech
```

你可能会看到一个弹出窗口要求访问麦克风。如果是，批准该权限，程序就会启动。

尝试说“yes”和“no”，你应该看到如下所示的输出：

```
Heard yes (201) @4056ms
Heard no (205) @6448ms
Heard unknown (201) @13696ms
Heard yes (205) @15000ms
Heard yes (205) @16856ms
Heard unknown (204) @18704ms
Heard no (206) @21000ms
```

每个检测到的单词后面的数字是其得分。默认情况下，命令识别程序组件仅在匹配分数大于 200 时才认为匹配有效，因此你看到的所有分数都将至少为 200。

得分之后的数字是自程序启动以来的毫秒数。

如果你看不到任何输出，请确保在 Mac 的“声音”(Sound) 菜单中选择了 Mac 的内置麦克风，并且将输入音量调得足够高。

我们已经确定这个程序可以在 Mac 上运行。现在，让我们在一些嵌入式硬件上运行该程序。

7.5 部署到微控制器

在本节中，我们会将代码部署到三个不同的设备：

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

我们将逐一介绍每种设备的编译和部署过程。

由于每种设备都有其自己的捕获音频的机制，因此每种设备都有单独的 *audio_provider.cc* 实现。输出也是如此，因此每个输出也都有 *command_responder.cc* 的变体。

由于 *audio_provider.cc* 实现复杂且特定于设备，与机器学习没有直接关系，因此本章将

不介绍它们。但是，附录 B 中有 Arduino 变体的演练。如果你需要在自己的项目中捕获音频，欢迎在自己的代码中重复使用这些实现。

除了对部署的说明以外，我们还将逐步介绍每台设备的 *command_responder.cc* 实现。首先，让我们介绍如何使用 Arduino 进行部署。

7.5.1 Arduino

由于在撰写本书时，唯一带有内置麦克风的 Arduino 开发板是 Arduino Nano 33 BLE Sense，因此我们将在本节中使用这个型号的开发板。如果你使用其他 Arduino 开发板并连接自己的麦克风，则需要实现自己的 *audio_provider.cc*。

Arduino Nano 33 BLE Sense 同时也有一个内置的 LED，我们可以用它来表明一个单词已经被识别。

图 7-7 展示了一个标出 LED 的开发板。

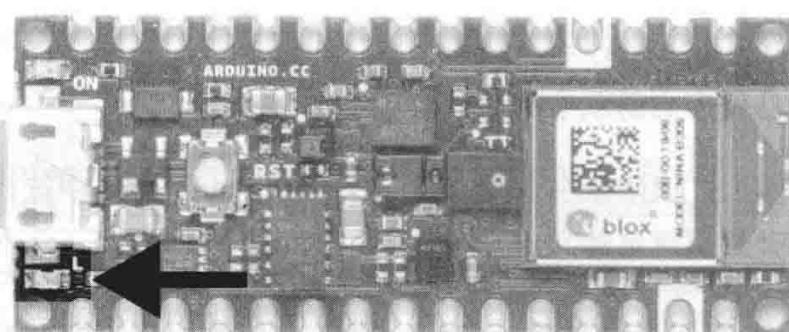


图 7-7：标出 LED 的 Arduino Nano 33 BLE Sense 开发板

现在，让我们看看如何使用该 LED 指示已检测到单词。

在 Arduino 上响应指令

每个 Arduino 开发板都有一个内置的 LED，还有一个方便的常量 `LED_BUILTIN`，我们可以用它来获取 LED 的引脚号，该常数在各个板上都不同。为了保持代码的可移植性，我们将限制自己只使用这个 LED 作为输出。

我们将做以下这些事情：为了展示推断正在运行，我们将在每次推断时通过打开或关闭 LED 来使其闪烁。但是，当听到单词“yes”时，我们将保持 LED 常亮一段时间。

那么“no”这个单词呢？好吧，由于这只是一个演示，所以我们不需要关心太多。但是，我们确实会将所有检测到的命令记录到串口，所以我们可以连接到设备来查看每一次匹配。

Arduino 的命令响应程序位于 `arduino/command_responder.cc`。让我们来看看它的代码。首先，包含命令响应程序的头文件和 Arduino 平台的库头文件：

```
#include "tensorflow/lite/micro/examples/micro_speech/command_responder.h"
#include "Arduino.h"
```

接下来，开始进行函数实现：

```
// Toggles the LED every inference, and keeps it on for 3 seconds if a "yes"
// was heard
void RespondToCommand(tflite::ErrorReporter* error_reporter,
                      int32_t current_time, const char* found_command,
                      uint8_t score, bool is_new_command) {
```

下一步是将内置 LED 的引脚置于输出模式，以便可以将其打开和关闭。我们会在一个 `if` 语句中做这件事。通过一个静态布尔变量 `is_initialized`，该语句将仅被触发一次。请记住，静态变量会在不同的函数调用之间保持其状态。

```
static bool is_initialized = false;
if (!is_initialized) {
    pinMode(LED_BUILTIN, OUTPUT);
    is_initialized = true;
}
```

接下来，设置另外两个静态变量以跟踪上次检测到“yes”的时间，以及已执行的推断次数：

```
static int32_t last_yes_time = 0;
static int count = 0;
```

有趣的事情来了。如果 `is_new_command` 参数为 `true`，那么我们就知道我们听到了一些东西，所以我们可以使用 `ErrorReporter` 实例来进行记录。但是，如果听到的是“yes”（通过检查 `found_command` 字符数组的第一个字符来确定），我们将存储当前时间并打开 LED：

```
if (is_new_command) {
    error_reporter->Report("Heard %s (%d) @%dms", found_command, score,
                           current_time);
    // If we heard a "yes", switch on an LED and store the time.
    if (found_command[0] == 'y') {
        last_yes_time = current_time;
        digitalWrite(LED_BUILTIN, HIGH);
    }
}
```

接下来，实现在几秒钟后——准确地说是 3 秒——关闭 LED 的行为：

```
// If last_yes_time is non-zero but was >3 seconds ago, zero it
// and switch off the LED.
if (last_yes_time != 0) {
```

```

if (last_yes_time < (current_time - 3000)) {
    last_yes_time = 0;
    digitalWrite(LED_BUILTIN, LOW);
}
// If it is non-zero but <3 seconds ago, do nothing.
return;
}

```

当 LED 熄灭时，我们还将 `last_yes_time` 设置为 0，因此直到下一次听到“yes”时我们才会再次进入 `if` 语句。`return` 语句很重要：如果我们最近听到了“yes”，该语句会阻止任何其他的输出代码运行，因此 LED 可以保持稳定点亮。

到目前为止，当听到“yes”时，我们的实现将点亮 LED 约 3 秒。下一部分实现将在每次推断时切换 LED 的开和关，除非我们处于“yes”模式，因为上述的 `return` 语句会阻止我们达到这一点。

这是最后的代码块：

```

// Otherwise, toggle the LED every time an inference is performed.
++count;
if (count & 1) {
    digitalWrite(LED_BUILTIN, HIGH);
} else {
    digitalWrite(LED_BUILTIN, LOW);
}

```

通过在每次运行推断时递增 `count` 变量，我们可以跟踪已经运行过的推断的总次数。在 `if` 条件中，我们使用 `&` 运算符对 `count` 变量和数字 `1` 进行二进制与（AND）运算。

通过对 `count` 变量和 `1` 进行与运算，我们过滤出了 `count` 变量所有的比特位中除了最小位之外的其他比特位。如果最小比特位为 `0`，则意味着 `count` 是一个偶数。在 C++ 的 `if` 语句中，这意味着 `false`。否则，结果将为 `1`，表明这是一个奇数。因为 `1` 表示为 `true`，所以 LED 将在 `count` 为奇数时打开，为偶数时关闭。这就是 LED 闪烁的原因。

就是这样！现在，我们已经为 Arduino 实现了命令响应程序。让我们运行该程序来看看它运行的情况。

运行示例

为了部署该示例，我们需要以下设备和软件：

- Arduino Nano 33 BLE Sense 开发板。
- micro-USB 数据线。
- Arduino IDE。



从撰写本书时起，编译过程有可能会发生变化，因此请查看 *README.md* 以获取最新说明。



你还可以从 *.zip* 文件安装该库。你可以从 TensorFlow Lite 团队网站下载该文件，也可以使用 TensorFlow Lite for Microcontrollers Makefile 自行生成。如果你希望使用后者，请参阅附录 A。

在安装完成 TensorFlow Lite Arduino 库后，*micro_speech* 示例将显示在“File”（文件）菜单中的“Examples”（示例）→“Arduino_TensorFlowLite”，如图 7-8 所示。



图 7-8：“Examples”菜单

单击“*micro_speech*”以加载示例。该示例会将显示为新窗口，其中包含每个源文件的选项卡。第一个选项卡中的 *micro_speech* 文件与我们之前介绍的 *main_functions.cc* 等效。



由于 6.2.2 节已经解释了 Arduino 示例的结构，所以我们在此不再赘述。

要运行示例，请通过 USB 插入 Arduino 设备。确保从“Tools”（工具）菜单的“Board”（开发板）下拉列表中选择了正确的设备类型，如图 7-9 所示。

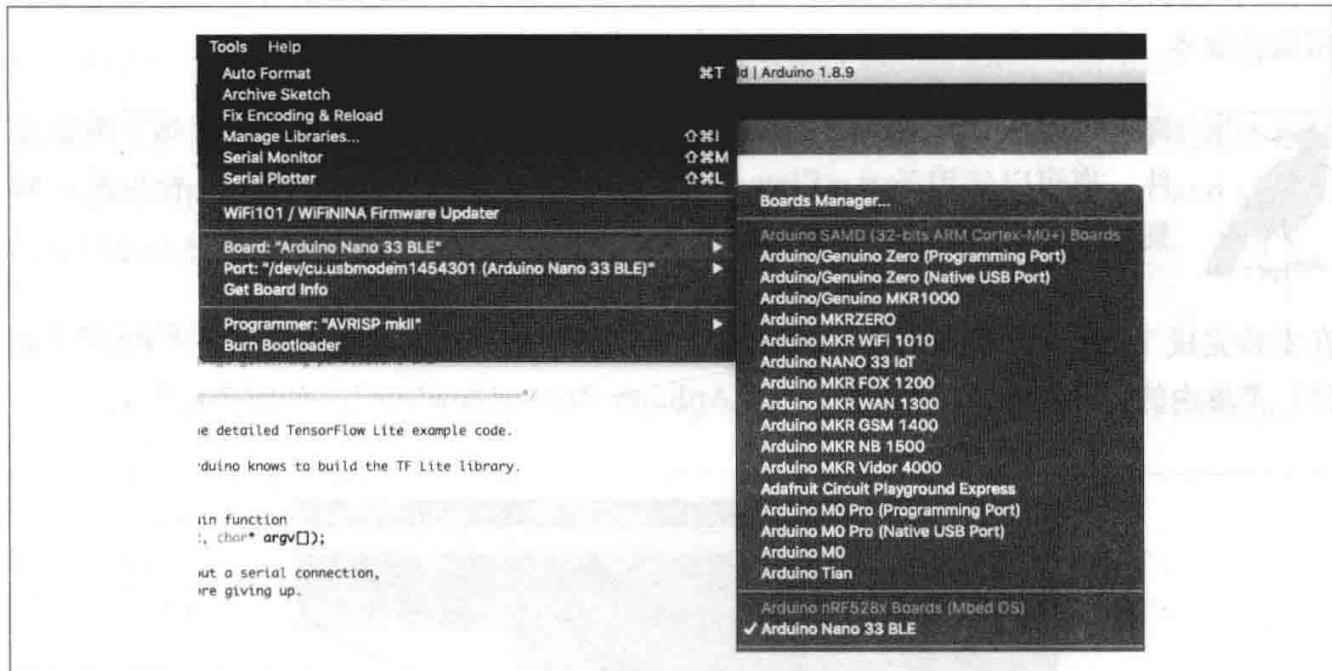


图 7-9：“Board”下拉列表

如果你的设备名称未显示在列表中，则需要安装其支持包。为此，请单击“Boards Manager”（开发板管理器）。在出现的窗口中，搜索你的设备，然后安装相应支持包的最新版本。接下来，请确保在同样位于“Tools”（工具）菜单的“Port”（端口）下拉列表中选择了设备的端口，如图 7-10 所示。

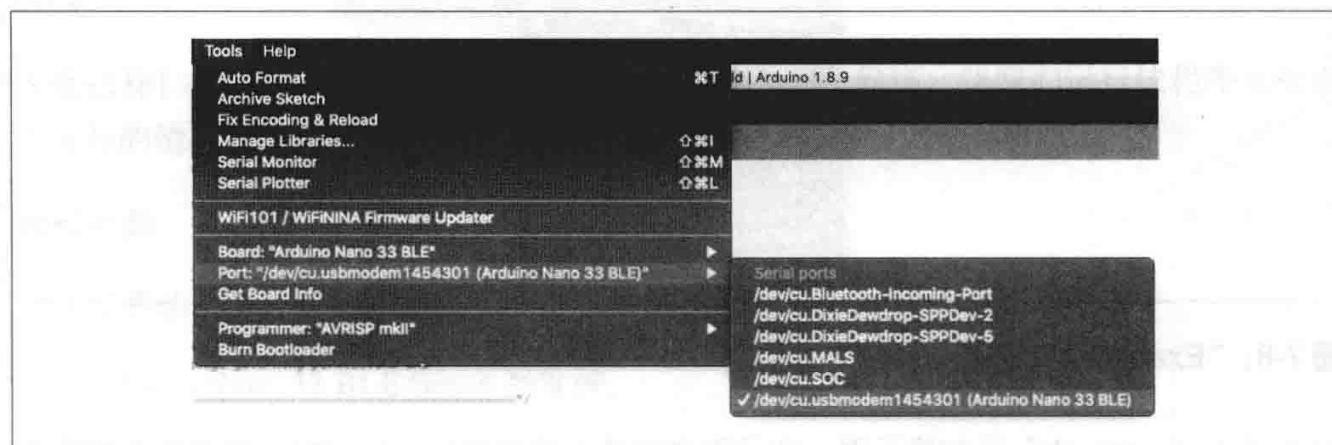


图 7-10：“Port”下拉列表

最后，在 Arduino 窗口中，单击上传按钮（图 7-11 中以白色突出显示）以编译代码并将其上传到 Arduino 设备。

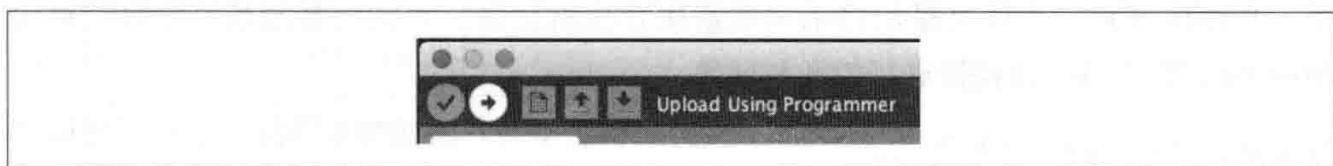


图 7-11：上传按钮，一个向右箭头

上传完成后你应该会看到 Arduino 板上的 LED 开始闪烁。

要测试该程序，请尝试说“yes”。当程序检测到“yes”时，LED 将持续点亮大约 3 秒。



如果程序无法识别你说的“yes”，请尝试连续说几遍。

你还可以通过 Arduino 串口监视器来查看推断的结果。为此，请从“Tools”（工具）菜单中打开“Serial Monitor”（串口监视器）。现在，请尝试说“yes”“no”和其他词语。你应该能看到类似图 7-12 所示的内容。

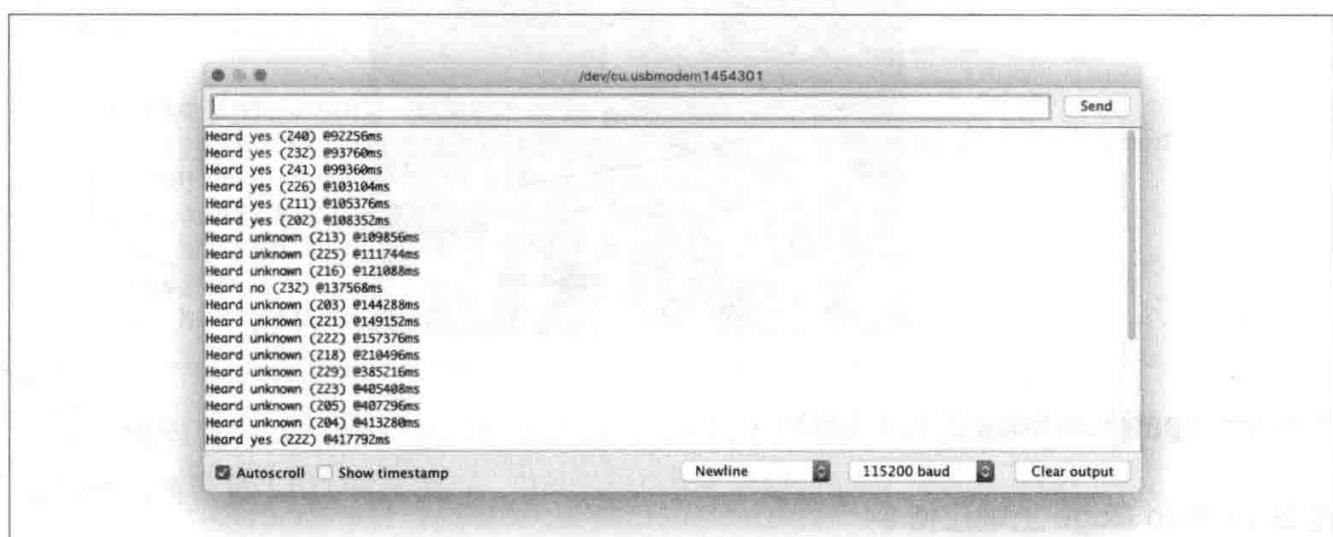


图 7-12：显示一些匹配项的串口监视器



由于我们使用的模型很小且不完善，你可能会注意到检测“yes”的效果比“no”更好。这是一个示例，说明了针对微小模型进行优化会如何导致准确率问题。我们会在第 8 章中讨论这个话题。

尝试修改

现在你已经部署了应用程序，请尝试使用这些代码！你可以在 Arduino IDE 中编辑源文件。当你保存时，系统会提示将示例重新保存到新位置。在做出更改后，你可以单击 Arduino IDE 中的上传按钮进行编译和部署。

以下是一些你可以尝试的想法：

- 改变示例，以在说出“no”而不是“yes”时点亮 LED。
- 使应用程序响应特定的“yes”和“no”命令序列，就像一个密码短语。
- 使用“yes”和“no”命令来控制其他组件，例如额外的 LED 或伺服装置。

7.5.2 SparkFun Edge

SparkFun Edge 有一个麦克风和一行 4 种颜色的 LED——红色、蓝色、绿色与黄色。这将让显示结果更加简单。图 7-13 展示了标出 LED 的 SparkFun Edge 开发板。

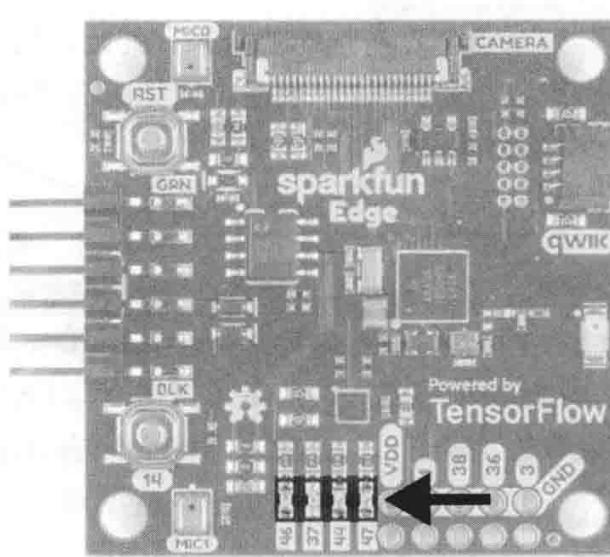


图 7-13：SparkFun Edge 的 4 个 LED

在 SparkFun Edge 上响应指令

为了清楚地表明我们的程序正在运行，我们将在每次运行推断时点亮或熄灭蓝色 LED。当听到“yes”时，将点亮黄色 LED；当听到“no”时，将点亮红色 LED；当听到未知命令时，将点亮绿色 LED。

SparkFun Edge 的命令响应程序实现位于 *sparkfun_edge/command_responder.cc*。该文件以下列 `include` 语句开始：

```
#include "tensorflow/lite/micro/examples/micro_speech/command_responder.h"
#include "am_bsp.h"
```

command_responder.h 是与这个文件相关的头文件。*am_bsp.h* 是 Ambiq Apollo3 SDK 的头文件，你在上一章中已经见过了。

在函数定义中，我们要做的第一件事是将连接到 LED 的引脚设置为输出：

```
// This implementation will light up the LEDs on the board in response to
// different commands.
void RespondToCommand(tflite::ErrorReporter* error_reporter,
                      int32_t current_time, const char* found_command,
                      uint8_t score, bool is_new_command) {
    static bool is_initialized = false;
    if (!is_initialized) {
        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);
        is_initialized = true;
    }
}
```

我们调用 Apollo3 SDK 中的 *am_hal_gpio_pinconfig()* 函数来将所有的 4 个 LED 引脚设置为输出模式，输出模式由常数 *g_AM_HAL_GPIO_OUTPUT_12* 表示。我们使用名为 *is_initialized* 的静态变量来确保我们只初始化一次。

接下来是点亮和熄灭蓝色 LED 的代码。我们使用 *count* 变量来执行此操作，这里与 Arduino 实现中的方法相同：

```
static int count = 0;
// Toggle the blue LED every time an inference is performed.
++count;
if (count & 1) {
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_BLUE);
} else {
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_BLUE);
}
```

这部分代码使用 *am_hal_gpio_output_set()* 和 *am_hal_gpio_output_clear()* 函数来切换蓝色 LED 的开关。

通过在每次运行推断时递增 *count* 变量，我们可以跟踪已执行的推断总数。在 *if* 条件中，我们使用 *&* 运算符对 *count* 变量和数字 1 进行二进制与运算。

通过对 *count* 和 1 进行与运算，我们可以滤除 *count* 除最小位以外的所有位。如果最低位为 0，则意味着 *count* 是偶数，且运算结果为 0。在 C++ 的 *if* 语句中，这意味着 *false*。否则，结果将为 1，表示 *count* 是一个奇数。由于 1 代表 *true*，因此 LED 将在 *count* 为奇数时点亮，*count* 为偶数时熄灭。这就是 LED 闪烁的原因。

接下来，我们根据刚刚听到的单词点亮适当的 LED。默认情况下，我们会熄灭所有 LED，因此，如果最近没有听到任何单词，那么 LED 将全部熄灭：

```
am_hal_gpio_output_clear(AM_BSP_GPIO_LED_RED);
am_hal_gpio_output_clear(AM_BSP_GPIO_LED_YELLOW);
am_hal_gpio_output_clear(AM_BSP_GPIO_LED_GREEN);
```

然后，我们将根据听到的命令，使用一些简单的 if 语句点亮相应的 LED：

```
if (is_new_command) {
    error_reporter->Report("Heard %s (%d) @%dms", found_command, score,
                           current_time);
    if (found_command[0] == 'y') {
        am_hal_gpio_output_set(AM_BSP_GPIO_LED_YELLOW);
    }
    if (found_command[0] == 'n') {
        am_hal_gpio_output_set(AM_BSP_GPIO_LED_RED);
    }
    if (found_command[0] == 'u') {
        am_hal_gpio_output_set(AM_BSP_GPIO_LED_GREEN);
    }
}
```

正如我们在前面所看到的，仅当真正的新命令调用 `RespondToCommand()` 时，`is_new_command` 才为 `true`。因此，如果未听到新命令，LED 指示灯将保持熄灭状态。否则，我们将使用 `am_hal_gpio_output_set()` 函数点亮相应的 LED。

运行示例

现在，我们已经遍历了示例代码并了解了如何点亮 SparkFun Edge 上的 LED。接下来，让我们启动并运行该示例。



自撰写本书时起，编译过程可能已经发生了变化，因此请查看 `README.md` 以获取最新说明。

为了部署该示例，我们需要以下设备和软件：

- SparkFun Edge 开发板。
- USB 编程器（建议使用 SparkFun Serial Basic Breakout，可在 micro-B USB 和 USB-C 变体中使用）。
- 配套的 USB 数据线。
- Python3 以及一些依赖项。



在第 6 章中，我们展示了如何确认你是否安装了正确版本的 Python。如果你已经做到了，那就太好了。如果没有，那么请回顾 6.3.2 节。

在你的终端中克隆 TensorFlow 代码仓库，然后转到其目录：

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

接下来，我们将编译二进制文件并运行一些命令以做好将其下载到设备上的准备。为避免键入错误，你可以从 *README.md* 复制并粘贴这些命令。

编译二进制文件。以下命令将下载所有必需的依赖项，然后为 SparkFun Edge 编译二进制文件：

```
make -f tensorflow/lite/micro/tools/make/Makefile \  
TARGET=sparkfun_edge TAGS=cmsis-nn micro_speech_bin
```

二进制文件在以下位置被创建为 *.bin* 文件：

```
tensorflow/lite/micro/tools/make/gen/ \  
sparkfun_edge_cortex-m4/bin/micro_speech.bin
```

要检查文件是否存在，你可以使用以下命令：

```
test -f tensorflow/lite/micro/tools/make/gen/ \  
sparkfun_edge_cortex-m4/bin/micro_speech.bin \  
&& echo "Binary was successfully created" || echo "Binary is missing"
```

如果运行该命令，你应该能看到终端中打印出的“*Binary was successfully created*”。如果你看到“*Binary is missing*”，这意味着编译过程存在问题。如果是这样，那么可以从 *make* 命令的输出找到可能在哪里出问题的线索。

给二进制文件签名。二进制文件必须使用加密密钥签名才能被部署到设备。现在，我们运行一些命令来对二进制文件进行签名，以便将其烧录到 SparkFun Edge。此处使用的脚本来自 Ambiq SDK，该文件在运行 *Makefile* 时下载。

输入以下命令来设置一些可用于开发的虚拟加密密钥：

```
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
```

接下来，运行以下命令来创建一个被签名的二进制文件。如有必要，用 *python* 替换 *python3*：

```
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/micro_speech.bin \
  --load-address 0xC000 \
  --magic-num 0xCB -o main_nonsecure_ota \
  --version 0x0
```

这将创建文件 *main_nonsecure_ota.bin*。现在运行以下命令来创建文件的最终版本，你可以使用下一个步骤中用到的脚本将文件的最终版本烧录到设备中：

```
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
```

现在，在你运行命令的目录中应该有一个名为 *main_nonsecure_wire.bin* 的文件。这就是你要烧录到设备中的文件。

烧录二进制文件。 SparkFun Edge 将其当前正在运行的程序存储在其 1MB 的闪存中。如果要让开发板运行新程序，则需要将新程序发送到开发板上，然后将其存储在闪存中，并覆盖以前保存的所有程序。

将编程器连接到开发板。 为了将新程序下载到开发板上，你需要使用 SparkFun USB-C Serial Basic 串行编程器。该设备使你的计算机可以通过 USB 与微控制器通信。

要将此设备连接到开发板上，请执行以下步骤：

1. 在 SparkFun Edge 的侧面找到六引脚接口。
2. 将 SparkFun USB-C Serial Basic 插入这些针脚，确保每个设备上标有 BLK 和 GRN 的针脚正确排列，如图 7-14 所示。

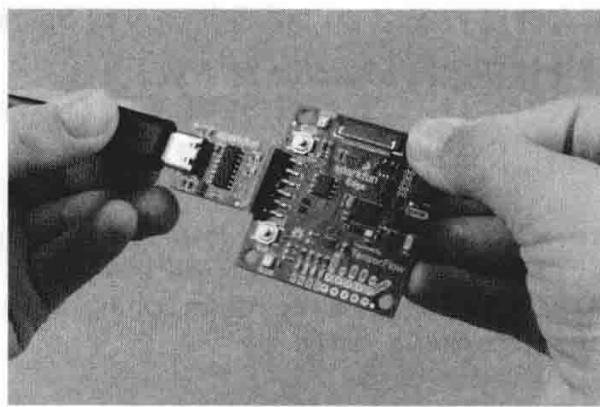


图 7-14：连接 SparkFun Edge 和 USB-C Serial Basic (SparkFun 提供)

将编程器连接到计算机。你可以通过 USB 将开发板连接到计算机。要对开发板进行编程，你需要找出计算机为设备指定的名称。最好的方法是在连接计算机之前和之后列出计算机的所有设备，并对比查看哪个是新设备。



有人报告了他们的操作系统的默认驱动程序有问题，因此，我们强烈建议你在继续之前安装驱动程序。

在通过 USB 连接开发板之前，请运行以下命令：

```
# macOS:  
ls /dev/cu*  
  
# Linux:  
ls /dev/tty*
```

这应该会输出已连接设备的列表，如下所示：

```
/dev/cu.Bluetooth-Incoming-Port  
/dev/cu.MALS  
/dev/cu.SOC
```

现在，将编程器连接到计算机的 USB 端口，然后再次运行命令：

```
# macOS:  
ls /dev/cu*  
  
# Linux:  
ls /dev/tty*
```

如下面的示例所示，你应该能在输出中看到一个额外的项目。你的项目可能具有不同的名称。此新项目就是设备的名称：

```
/dev/cu.Bluetooth-Incoming-Port  
/dev/cu.MALS  
/dev/cu.SOC  
/dev/cu.wchusbserial-1450
```

该名称将被用于引用设备。但是，该名称可能会根据编程器所连接的 USB 端口的不同而变化。因此，如果你将开发板从计算机上断开，然后重新连接，则可能需要再次查询设备名称。



一些用户报告列表中出现了两个设备。如果你看到两个设备，则正确的设备应以字母“wch”开头，例如“/dev/wchusbserial-14410”。

确定设备名称后，将其放入一个 shell 变量中以备后用：

```
export DEVICENAME=<your device name here>
```

在后面的流程中，当运行需要设备名称的命令时，你可以使用此变量。

运行脚本以烧录开发板。要烧录开发板，必须将其置于特殊的 bootloader 状态，以使其准备好接收新的二进制文件。然后，你将运行一个脚本，将二进制文件发送到开发板。

首先创建一个环境变量以指定波特率，波特率是将数据发送到设备的速度：

```
export BAUD_RATE=921600
```

现在，将以下命令粘贴到你的终端中，但先不要按回车键！命令中的 \${DEVICENAME} 和 \${BAUD_RATE} 将被替换为你在上面设置的值。请记住，如有必要，请用 python 替换 python3：

```
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
```

接下来，将开发板重置为 bootloader 状态并烧录开发板。在开发板上找到标记为 RST 和 14 的按键，如图 7-15 所示。执行以下步骤：

1. 确保你的开发板已连接至编程器，并且整个设备均已通过 USB 连接至你的计算机。
2. 在开发板上，按住标记为 14 的按键不要松手。
3. 在长按标记为 14 的按键的同时，按下标记为 RST 的按键以重置开发板。
4. 在计算机上按回车键以运行脚本。继续按住按键 14。

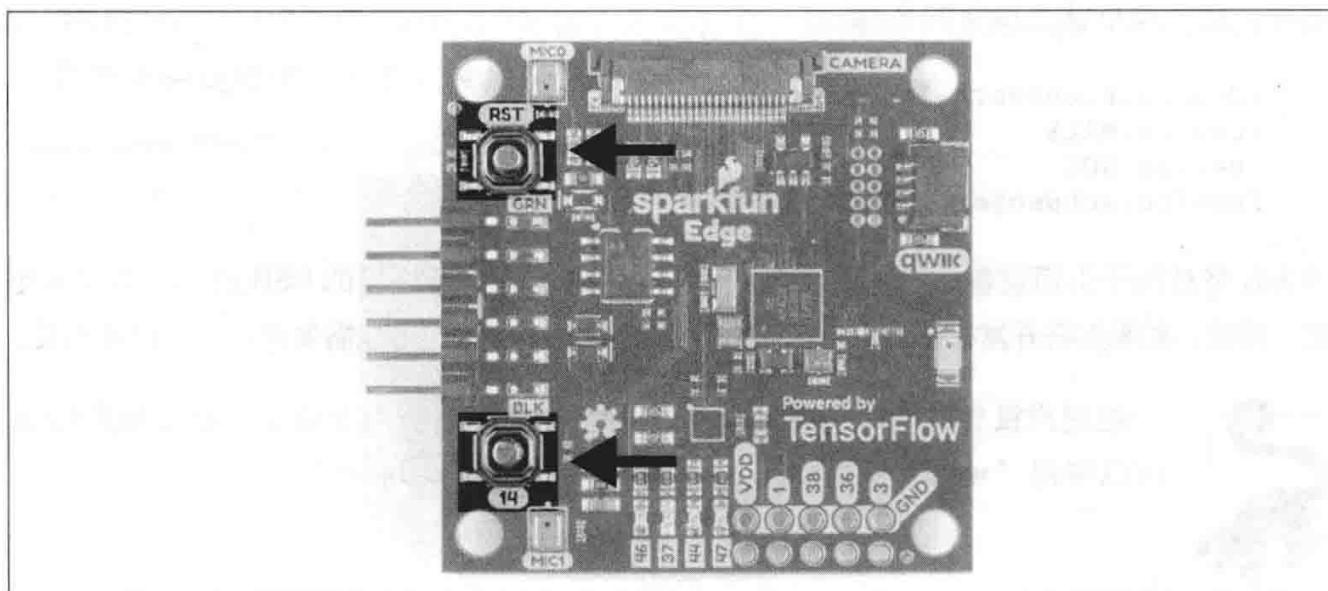


图 7-15：SparkFun Edge 上的按键

现在，你应该能在屏幕上看到类似以下的内容：

```
Connecting with Corvette over serial port /dev/cu.usbserial-1440...
Sending Hello.
Received response for Hello
Received Status
length = 0x58
version = 0x3
Max Storage = 0x4ffao
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...]
```

继续按住按键 14，直到你看到“**Sending Data Packet of length 8180**”。你可以在看到该信息后松开按键（但一直按住它也可以）。程序将继续在终端上打印信息。最终，你会看到类似以下的内容：

```
[...lots more Sending Data Packet of length 8180...]
Sending Data Packet of length 8180
Sending Data Packet of length 6440
Sending Reset Command.
Done.
```

这表明烧录成功。



如果程序输出以错误结尾，请检查是否打印了“**Sending Reset Command.**”。如果是这样，尽管有错误，但烧录仍可能成功。否则，烧录可能已经失败了。这时可以尝试再次执行这些步骤（可以跳过设置环境变量的步骤）。

测试程序

要确保程序正在运行，请按一下 RST 按键。现在你应该能看到蓝色 LED 在闪烁。

要测试该程序，请尝试说“yes”。当程序检测到“yes”时，黄色 LED 将闪烁。该模型还被训练以识别“no”以及未知单词。红色 LED 闪烁表示检测到“no”，绿色 LED 闪

烁表示检测到未知单词。

如果无法让程序识别“yes”，请尝试连续说几遍。

由于我们使用的模型很小而且并不完善，你可能会发现比起识别“no”，模型更善于识别“yes”，而“no”经常被识别为未知单词。这是一个示例，说明针对微小的模型进行优化是如何导致准确率问题的。我们会在第8章中讨论这个话题。

如果不起作用怎么办

以下是一些可能的问题以及调试方法。

问题：烧录时，脚本在“`Sending Hello.`”时卡住一会儿，然后打印错误。

解决方案：你需要在运行脚本时按住标记为**14**的按键。长按按键**14**，按下按键**RST**，然后运行脚本，同时始终按住按键**14**。

问题：烧录后，所有指示灯均不亮。

解决方案：尝试按下按键**RST**，或将开发板与编程器断开连接，然后重新连接。如果这些都不起作用，请尝试再次烧录开发板。

查看调试数据

该程序还会将成功的识别记录到串口。要查看此数据，可以使用115200的波特率监视开发板的串行端口输出。在macOS和Linux上，以下命令应该起作用：

```
screen ${DEVICENAME} 115200
```

最初，你应该看到类似于以下内容的输出：

```
Apollo3 Burst Mode is Available
```

```
Apollo3 operating in Burst Mode (96MHz)
```

尝试通过说“yes”或“no”来发出一些命令。你应该能看到开发板对于每个命令打印的调试信息：

```
Heard yes (202) @65536ms
```

要停止在屏幕上查看调试输出，请按住Ctrl+A，然后立即按K键，接着按Y键。

尝试修改

现在，你已经部署了基本应用程序，可以尝试进行一些更改。你可以在*tensorflow/lite/*

`micro/examples/micro_speech` 文件夹中找到应用程序的代码。只需编辑并保存文件，然后重复上述说明，即可将修改后的代码部署到设备上。

你可以尝试以下几种方案：

- `RespondToCommand()` 的 `score` 参数表示预测的分数。你可以使用 LED 作为仪表来显示匹配度。
- 使应用程序响应特定的“yes”和“no”命令序列，例如密码短语。
- 使用“yes”和“no”命令来控制其他组件，例如附加的 LED 或伺服装置。

7.5.3 ST Microelectronics STM32F746G Discovery 套件

由于 STM32F746G 带有精美的 LCD 显示屏，所以我们可以用它来显示检测到的任何唤醒词，如图 7-16 所示。

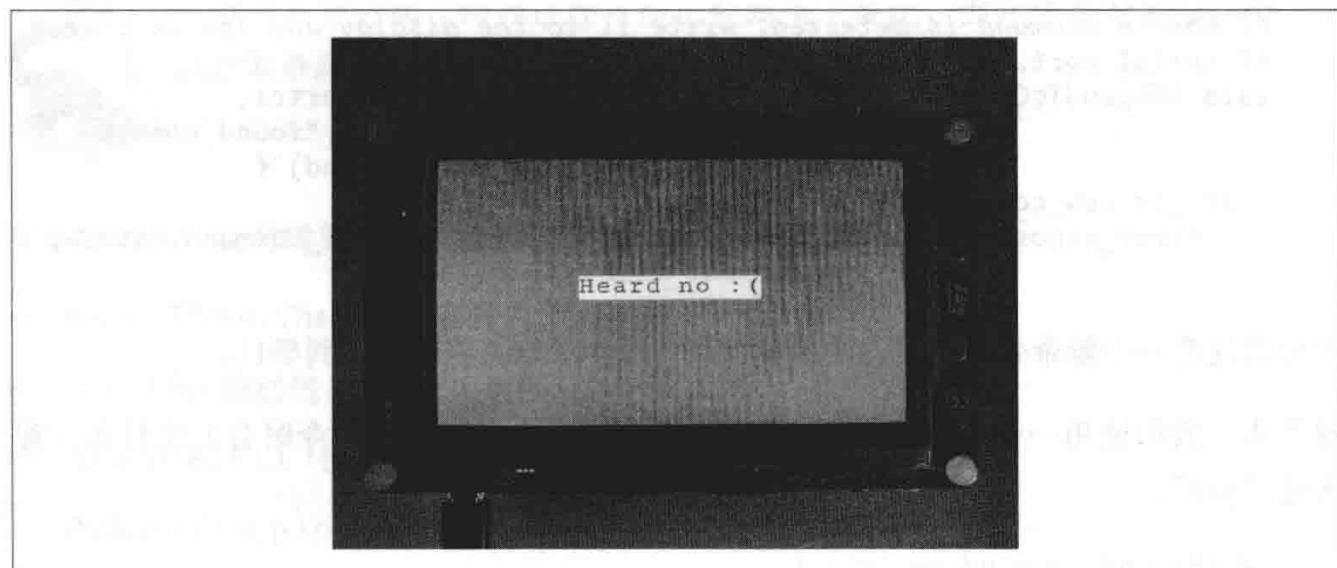


图 7-16：STM32F746G 正在显示“no”

在 STM32F746G 上响应指令

STM32F746G 的 LCD 驱动给我们提供了一些方法，使得可以在屏幕上显示信息。在本练习中，我们将使用其来显示以下消息之一，具体取决于听到的命令：

- “Heard yes!”
- “Heard no :(”
- “Heard unknown”
- “Heard silence”

我们还将根据听到的命令给屏幕设置不同的背景颜色。

首先，引入一些头文件：

```
#include "tensorflow/lite/micro/examples/micro_speech/command_responder.h"
#include "LCD_DISCO_F746NG.h"
```

第一个头文件 *comand_responder.h* 仅声明了这个文件的接口。第二个头文件 *LCD_DISCO_F746NG.h* 提供了一个接口来控制设备上 LCD 的显示，你可以在 Mbed 网站上阅读有关它的更多信息。

接下来，实例化一个由于 *LCD_DISCO_F746NG* 对象，该对象包含了用于控制 LCD 的方法：

```
LCD_DISCO_F746NG lcd;
```

在接下来的几行中，我们声明了 *RespondToCommand()* 函数，并将检查是否由于新的语音指令而调用了该函数：

```
// When a command is detected, write it to the display and log it to the
// serial port.
void RespondToCommand(tflite::ErrorReporter *error_reporter,
                      int32_t current_time, const char *found_command,
                      uint8_t score, bool is_new_command) {
    if (is_new_command) {
        error_reporter->Report("Heard %s (%d) @%dms", found_command, score,
                               current_time);
```

当知道这是一个新命令时，我们使用 *error_reporter* 将其记录到串口。

接下来，我们使用一个包含较多条件的 *if* 语句来决定遇到每个命令时会发生什么。首先是“yes”：

```
if (*found_command == 'y') {
    lcd.Clear(0xFF0F9D58);
    lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard yes!", CENTER_MODE);
```

我们使用 *lcd.Clear()* 来清除屏幕上先前显示的内容，并设置新的背景颜色，就像重新涂上一层油漆一样。*0xFF0F9D58* 是一种漂亮且饱满的绿色。

在绿色背景上，我们使用 *lcd.DisplayStringAt()* 来绘制一些文本。第一个参数指定 *x* 坐标，第二个参数指定 *y* 坐标。为了将文本大致定位在屏幕的中间，我们使用辅助函数 *LINE()* 来确定屏幕上第五行文本相对应的 *y* 坐标。

第三个参数是我们将显示的文本字符串，第四个参数确定文本的对齐方式。在这里，我们使用常量 *CENTER_MODE* 来设置文本居中对齐。

继续执行 *if* 语句，以涵盖剩余的三种可能性：“no”“未知”和“沉默”（由 *else* 语句捕获）：

```
    } else if (*found_command == 'n') {
        lcd.Clear(0xFFDB4437);
        lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard no :(", CENTER_MODE);
    } else if (*found_command == 'u') {
        lcd.Clear(0xFFFF4B400);
        lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard unknown", CENTER_MODE);
    } else {
        lcd.Clear(0xFF4285F4);
        lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard silence", CENTER_MODE);
    }
```

以上，就是这些！由于LCD库使我们可以轻松地对显示器进行高层次的控制，因此我们无须花费太多代码即可输出结果。让我们部署这个示例，根据它实际运行的情况观察这一切是如何起作用的。

运行示例

现在，我们可以使用Mbed工具链将应用程序部署到设备上。



自撰写本书时起，编译过程可能已经发生了变化，因此请查看 *README.md* 以获取最新说明。

在开始之前，我们需要以下设备和软件：

- STM32F746G Discovery 套件开发板。
- mini-USB 数据线。
- Arm Mbed CLI（请遵循 Mbed 设置指南）。
- Python3 以及 pip。

与 Arduino IDE 一样，Mbed 要求以某种方式对源文件进行结构化。TensorFlow Lite for Microcontrollers Makefile 知道如何为我们执行此操作，并且可以生成适合 Mbed 的目录格式。

为此，请运行以下命令：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
TARGET=mbed TAGS="cmsis-nn disco_f746ng" generate_micro_speech_mbed_project
```

这将导致创建一个新目录：

```
tensorflow/lite/micro/tools/make/gen/mbed_cortex-m4/prj/ \
micro_speech/mbed
```

这个目录包含了这个示例的所有依赖项，这些依赖项是以正确的方式结构化的，以便 Mbed 能够编译它们。

首先，切换到目录，以便可以在其中运行一些命令：

```
cd tensorflow/lite/micro/tools/make/gen/mbed_cortex-m4/prj/micro_speech/mbed
```

接下来，你将使用 Mbed 下载依赖项并编译项目。

首先，使用以下命令通知 Mbed 当前的目录是 Mbed 项目的根目录：

```
mbed config root .
```

接下来，指示 Mbed 下载依赖项并准备编译：

```
mbed deploy
```

默认情况下，Mbed 使用 C++ 98 构建项目。但是，TensorFlow Lite 需要 C++ 11。运行以下 Python 代码片段以修改 Mbed 配置文件，使其使用 C++ 11。你只需在命令行中键入或粘贴以下内容：

```
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\\\"))'
```

最后，运行以下命令以进行编译：

```
mbed compile -m DISCO_F746NG -t GCC_ARM
```

这将在以下路径中生成二进制文件：

```
./BUILD/DISCO_F746NG/GCC_ARM/mbed.bin
```

STM32F746G 开发板的优点之一是部署非常简单。要进行部署，只需插入 STM 开发板，然后将文件复制到其中即可。在 macOS 上，你可以使用以下命令来执行此操作：

```
cp ./BUILD/DISCO_F746NG/GCC_ARM/mbed.bin /Volumes/DIS_F746NG/
```

或者，只需在文件浏览器中找到 DIS_F746NG 卷并将文件拖到上方即可。

复制文件将启动烧录过程。

测试程序

完成以上操作后，尝试说“yes”。你应该能看到适当的文本出现在显示屏上，并且背景颜色随之发生变化。

如果你无法让程序识别你说的“yes”，请尝试连续说几遍。

由于我们使用的模型很小且不完善，你可能会发现比起识别“no”，模型更善于识别

“yes”，而“no”经常被识别为未知单词。这是一个示例，说明针对微小的模型进行优化是如何导致准确率问题的。我们会在第8章中讨论这个话题。

查看调试数据

程序还将成功的识别记录到串口。要查看输出，请使用9600的波特率建立到开发板的串行连接。

在macOS和Linux上，当你使用以下命令时应该能看到设备列表：

```
ls /dev/tty*
```

将会显示类似以下的内容：

```
/dev/tty.usbmodem1454203
```

在确定设备后，你可以使用以下命令连接到设备，请将`</dev/tty.devicename>`替换为`/dev`中显示的设备名称：

```
screen /dev/<tty.devicename 9600>
```

尝试通过说“yes”或“no”来发出一些命令。你应该看到开发板对于每个命令所打印的调试信息：

```
Heard yes (202) @65536ms
```

要停止在屏幕上查看调试输出，请按住Ctrl+A，然后立即按K键，接着按Y键。



如果你不确定如何在你的平台上建立串行连接，可以尝试使用在Windows、macOS和Linux上都能运行的CoolTerm。该开发板应该会显示在CoolTerm的“Port”（端口）下拉列表中。请确保将波特率设置为9600。

尝试修改

既然你已经部署了该应用程序，那么进行一些更改可能会很有趣。你可以在目录`tensorflow/lite/micro/tools/make/gen/mbed_cortex-m4/prj/micro_speech/mbed`中找到应用的代码。只需编辑并保存，然后重复上述说明即可将修改后的代码部署到设备上。

以下是一些你可以尝试的想法：

- `RespondToCommand()`的`score`参数表示预测的分数。你可以在LCD显示屏上创建分数的可视化指示器。
- 使应用程序响应特定的“yes”和“no”命令序列，例如密码短语。
- 使用“yes”和“no”命令来控制其他组件，例如附加的LED或伺服装置。

7.6 小结

我们浏览过的应用程序代码主要涉及从硬件捕获数据，然后提取适合推断的特征。实际上将数据输入模型并进行推断的部分代码量相对较小，并且与第 6 章介绍的示例非常相似。

这在机器学习项目中非常典型。模型已经训练好了，因此我们的工作只是为模型提供适当类型的数据。作为一个使用 TensorFlow Lite 的嵌入式开发人员，你将花费大部分的编程时间来捕获传感器数据、将其处理为特征并响应模型的输出。运行推断的部分既快速又简单。

但是嵌入式应用程序只是软件包的一部分，真正有趣的部分是模型。在第 8 章中，你将学习如何训练自己的语音模型来识别不同的单词。你还将了解有关模型工作原理的更多信息。

唤醒词检测：训练模型

在第 7 章中，我们围绕一个模型构建了一个应用程序，该模型被训练为可以识别“yes”和“no”。在本章中，我们将训练一个可以识别不同单词的新模型。

我们的应用程序代码相当通用，它所做的只是捕获和处理音频，将其输入 TensorFlow Lite 模型，并根据输出执行一些操作。应用程序基本上不在乎模型在识别什么单词。这意味着如果我们训练一个新的模型，我们可以直接将它放入我们的应用程序中，新模型应该可以立即工作。

在训练新模型时，我们需要考虑以下几点：

输入

新模型必须使用与应用程序代码相同的预处理，使用相同形状和格式的输入数据进行训练。

输出

新模型的输出必须采用相同的格式：一个概率张量，其中每个值分别对应一种类别的概率。

训练数据

无论选择哪一个新的单词，都需要很多人的录音，这样才可以训练新模型。

优化

模型必须经过优化才能在内存有限的微控制器上高效运行。

幸运的是，我们现有的模型是使用 TensorFlow 团队发布的公开脚本进行训练的，所以我们可以使用此脚本来训练新模型。我们还可以使用一个免费的语音数据集作为训练数据。

在 8.1 节，我们将逐步介绍使用此脚本训练模型的过程。然后，在 8.2 节，我们将把新模型合并到现有的应用程序代码中。之后，在 8.3 节，你将学习模型的实际工作方式。最后，在 8.4 节，你将看到如何使用自己的数据集训练模型。

8.1 训练我们的新模型

我们使用的模型是使用 TensorFlow Simple Audio Recognition 脚本进行训练的，该脚本设计用于演示如何使用 TensorFlow 构建和训练音频识别模型。

这个脚本使得训练音频识别模型非常容易。该脚本允许我们做以下工作：

- 下载包含 20 个口语单词的音频数据集。
- 选择要训练模型的单词子集。
- 指定要在音频上使用的预处理类型。
- 从几种不同类型的模型架构中进行选择。
- 用量化方法优化微控制器的模型。

当我们运行脚本时，它将下载数据集、训练模型并输出代表已训练模型的文件。然后，我们使用其他一些工具将此文件转换为 TensorFlow Lite 的正确格式。



模型作者通常会创建这些类型的训练脚本。这使得他们可以轻松地试验模型架构和超参数的不同组合变体，并与他人共享他们的工作。

使用训练脚本的最简单方法是在 Colaboratory (Colab) 笔记本中运行，我们将在下一小节中执行此操作。

在 Colab 中训练模型

Google Colab 是训练模型的好地方。该平台提供了对云端强大计算资源的访问，也提供了可用来监视训练过程的工具。同时该平台是完全免费的。

在本节中，我们将使用 Colab 笔记本来训练我们的新模型。我们使用的笔记本可在 TensorFlow 代码仓库中找到。

打开笔记本并单击“Run in Google Colab”（在 Google Colab 中运行）按钮，如图 8-1 所示。

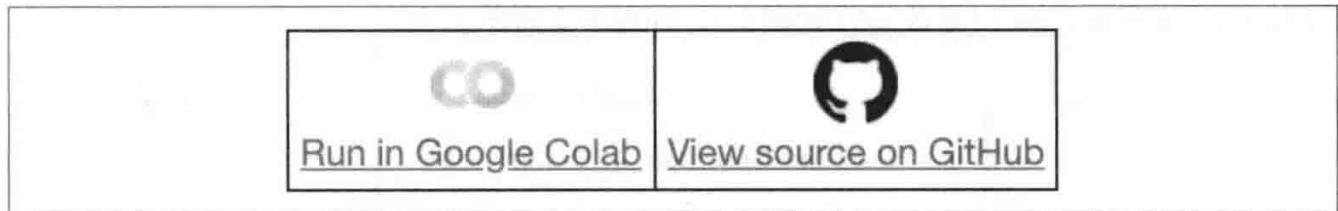


图 8-1：“Run in Google Colab” 按钮



在撰写本文时，GitHub 中存在一个错误，在显示 Jupyter 笔记本时会导致间歇性错误消息。如果你在尝试访问笔记本时看到“Sorry, something went wrong. Reload?”（对不起，出了点问题。重新加载？），请按照 4.3 节中的说明进行操作。

该笔记本将指导我们完成一个模型的训练过程。它通过以下步骤运行：

- 配置参数。
- 安装正确的依赖项。
- 使用 TensorBoard 监控训练过程。
- 运行训练脚本。
- 将训练输出转换成我们可以使用的模型。

启用 GPU 训练

在第 4 章中，我们用少量的数据训练了一个十分简单的模型。而我们现在正在训练的模型要复杂得多，数据集也要大得多，并且训练时间会更长。在一般的现代计算机 CPU 上进行训练，需要三到四个小时。

为了减少训练模型所需的时间，我们可以使用称为 GPU 加速（GPU acceleration）的方法训练。GPU，又称图形处理单元（Graphics Processing Unit），这是一种旨在帮助计算机快速处理图像数据的硬件，使计算机可以平滑地呈现用户界面和视频游戏等内容。大部分计算机都有一个 GPU。

图像处理涉及并行运行许多任务，训练深度学习网络也是如此。这意味着可以使用 GPU 硬件来加速深度学习训练。通常，在 GPU 上训练要比在 CPU 上训练快 5 到 10 倍。

由于在训练过程中需要对音频进行预处理，这意味着我们不会看到特别大的提速，但是我们的模型在 GPU 上的训练速度仍然要快得多，总共需要一到两个小时。

对我们来说幸运的是，Colab 支持通过 GPU 进行训练。虽然默认情况下未启用 GPU，但是很容易就可以启用它。为此，请转到 Colab 的“Runtime”（运行时）菜单，然后单击

“Change runtime type”（更改运行时类型），如图 8-2 所示。

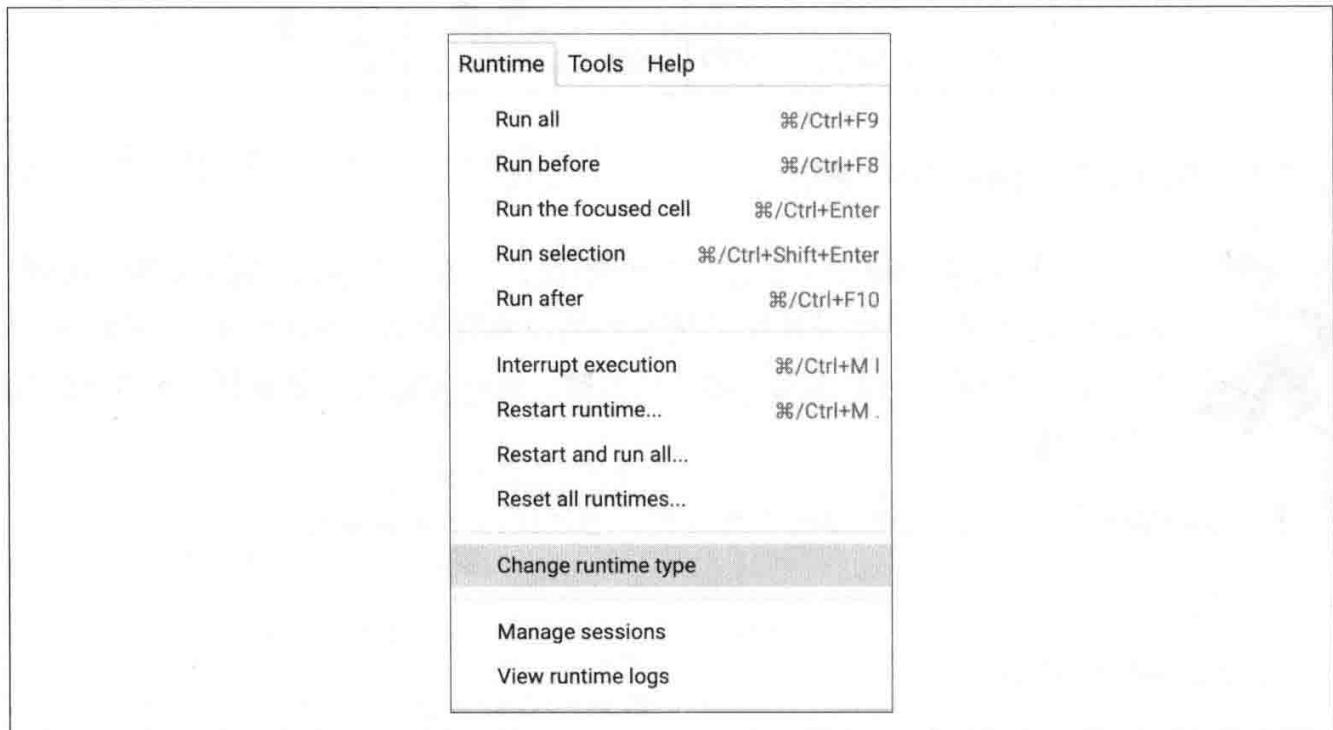


图 8-2：Colab 中的“Change runtime type”选项

当你选择此选项时，将打开图 8-3 所示的“Notebook settings”（笔记本设置）框。

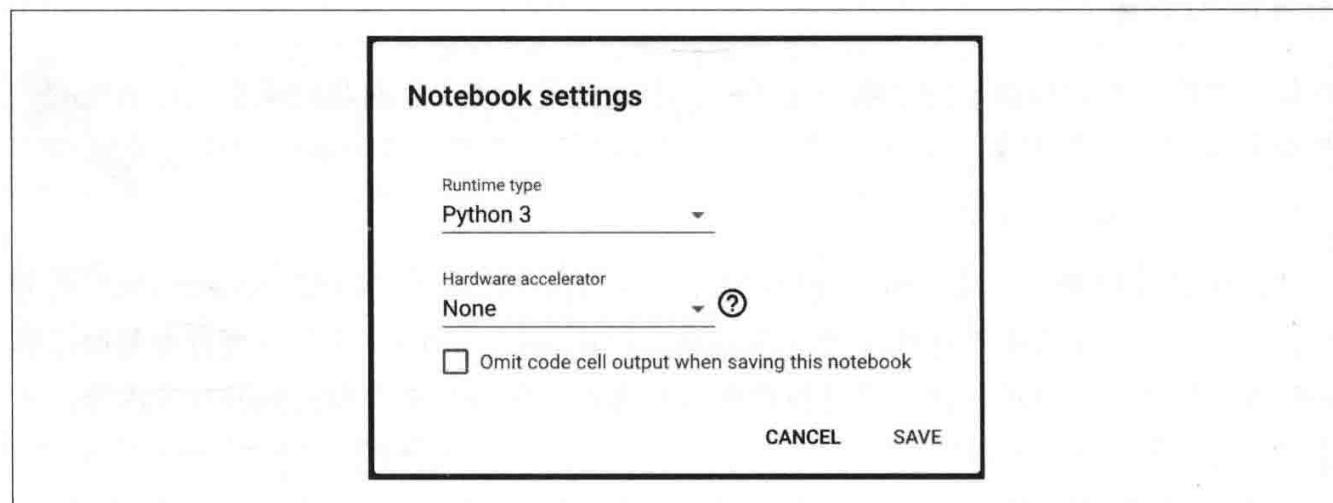


图 8-3：“Notebook settings”框

从“Hardware accelerator”（硬件加速器）下拉列表中选择 GPU，如图 8-4 所示，然后单击“Save”（保存）。

Colab 现在将在具有 GPU 的后台计算机（称为运行时）上运行 Python。

下一步是为笔记本配置我们要训练的单词。

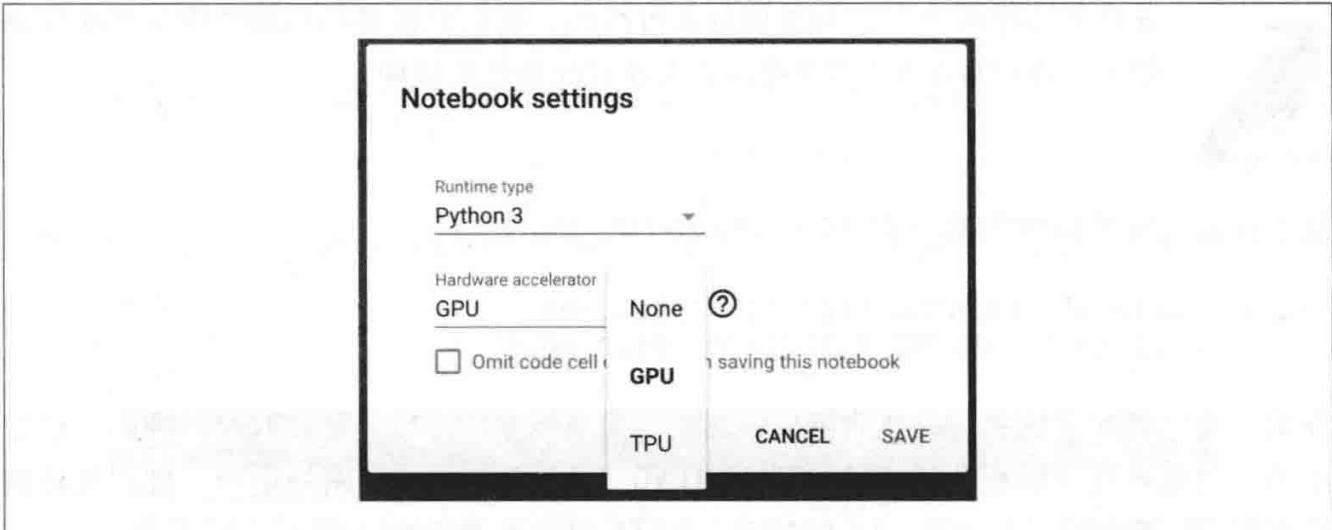


图 8-4: “Hardware accelerator” 下拉列表

配置训练

训练脚本是通过一堆命令行标志配置的，这些标志控制着从模型的架构到将要通过训练来区分的单词等所有的选项。

为了简化脚本的运行，笔记本的第一个单元格在环境变量中存储了一些重要的值。这些值在运行时将被替换为脚本的命令行标志。

第一个变量 `WANTED_WORDS`，使我们能够选择训练模型所使用的单词：

```
os.environ["WANTED_WORDS"] = "yes,no"
```

默认情况下所选单词为“yes”和“no”，但是我们可以提供以下单词的任意组合，所有这些单词都存在于数据集中：

- 常用命令：yes、no、up、down、left、right、on、off、stop、go、backward、forward、follow、learn。
- 数字 0~9：zero、one、two、three、four、five、six、seven、eight、nine。
- 随机单词：bed、bird、cat、dog、happy、house、Marvin、Sheila、tree、wow。

我们可以将要选择的单词包括在以逗号分隔的列表中。让我们选择“on”和“off”这两个词来训练我们的新模型：

```
os.environ["WANTED_WORDS"] = "on,off"
```

在训练模型时，列表中未包含的所有单词都将归为“未知”类别。



在这里选择两个以上的单词也是可行的，我们只需要稍微调整应用程序代码即可。我们将在 8.2 节中提供有关执行此操作的说明。

还要注意变量 TRAINING_STEPS 和 LEARNING_RATE：

```
os.environ["TRAINING_STEPS"]="15000,3000"  
os.environ["LEARNING_RATE"]="0.001,0.0001"
```

在第 3 章中我们了解到，模型的权重和偏差是逐步调整的，因此随着时间的推移，模型的输出将更接近于匹配所需的值。TRAINING_STEPS 是指通过网络运行一批训练数据并更新其权重和偏差的次数。LEARNING_RATE 将设置调整的比率。

学习率（learning rate）越高，每次迭代将会对权重和偏差进行的调整越多，这意味着收敛速度将越快。但是，这些巨大的调整意味着参数将难以达到理想的值，因为我们可能会不断跳过它们。当学习率较低时，这种跳跃比较小，因此需要更多的训练步（step）才能收敛，但是最终结果可能会更好。对于给定模型的最佳学习率是通过反复试验确定的。

在上述变量中，训练步和学习率被定义为逗号分隔的列表，这些列表定义了每个训练阶段的学习率。根据我们刚刚看过的值，该模型将以 0.001 的学习率训练 15 000 步，然后以 0.0001 的学习率训练 3000 步。步骤总数将为 18 000。

这意味着我们将以较高的学习率先进行一堆迭代，从而使网络快速收敛。然后，我们将以较低的学习率进行较少的迭代，从而对权重和偏差进行微调。

现在，我们将保持这些值不变，但是我们已经知道它们的用途。运行单元格，你会看到以下输出：

```
Training these words: on,off  
Training steps in each stage: 15000,3000  
Learning rate in each stage: 0.001,0.0001  
Total number of training steps: 18000
```

这是对我们的模型将如何训练的摘要。

安装依赖项

接下来，我们要获取运行脚本所需的一些依赖项。

运行接下来的两个单元格，以执行以下操作：

- 安装特定版本的 TensorFlow pip 软件包，其中包括训练模型所需的算子。
- 克隆 TensorFlow GitHub 代码仓库的相应版本，以便可以访问训练脚本。

加载 TensorBoard

为了监控训练过程，我们将使用TensorBoard。这是一个用户界面，通过图表、统计数据以及其他可视化的信息向我们展示有关训练进展的情况。

训练完成后，用户界面内容类似于图 8-5 中的屏幕截图。你将在本章稍后了解所有这些图表的含义。



图 8-5：训练完成后 TensorBoard 的截图

运行下一个单元以加载TensorBoard。该界面会出现在 Colab 中，但是直到我们开始训练时，它才会显示一些有趣的内容。

开始训练

以下单元格运行用以开始训练的脚本。你可以看到其中有很多命令行参数：

```
!python tensorflow/tensorflow/examples/speech_commands/train.py \
--model_architecture=tiny_conv --window_stride=20 --preprocess=micro \
--wanted_words=${WANTED_WORDS} --silence_percentage=25 --unknown_percentage=25 \
--quantize=1 --verbosity=WARNING --how_many_training_steps=${TRAINING_STEPS} \
--learning_rate=${LEARNING_RATE} --summaries_dir=/content/retrain_logs \
--data_dir=/content/speech_dataset --train_dir=/content/speech_commands_train
```

其中一些参数（例如 `--wanted_words=${WANTED_WORDS}`）使用我们先前定义的环境变量来配置我们要创建的模型。其他参数设置脚本的输出，例如 `--train_dir=`

`/content/speech_commands_train`, 该参数定义将训练后的模型保存在何处。

保留参数不变, 然后运行单元格。你将开始看到一些输出流。在下载语音命令数据集时, 数据流会暂停片刻:

```
>> Downloading speech_commands_v0.02.tar.gz 18.1%
```

完成此操作后, 将出现更多输出。其中可能会有一些警告, 但是只要单元格继续运行, 你就可以忽略它们。此时, 你可以向上滚动到TensorBoard, 希望其看起来如图 8-6 所示。如果你看不到任何图标, 请单击“SCALARS”标签。

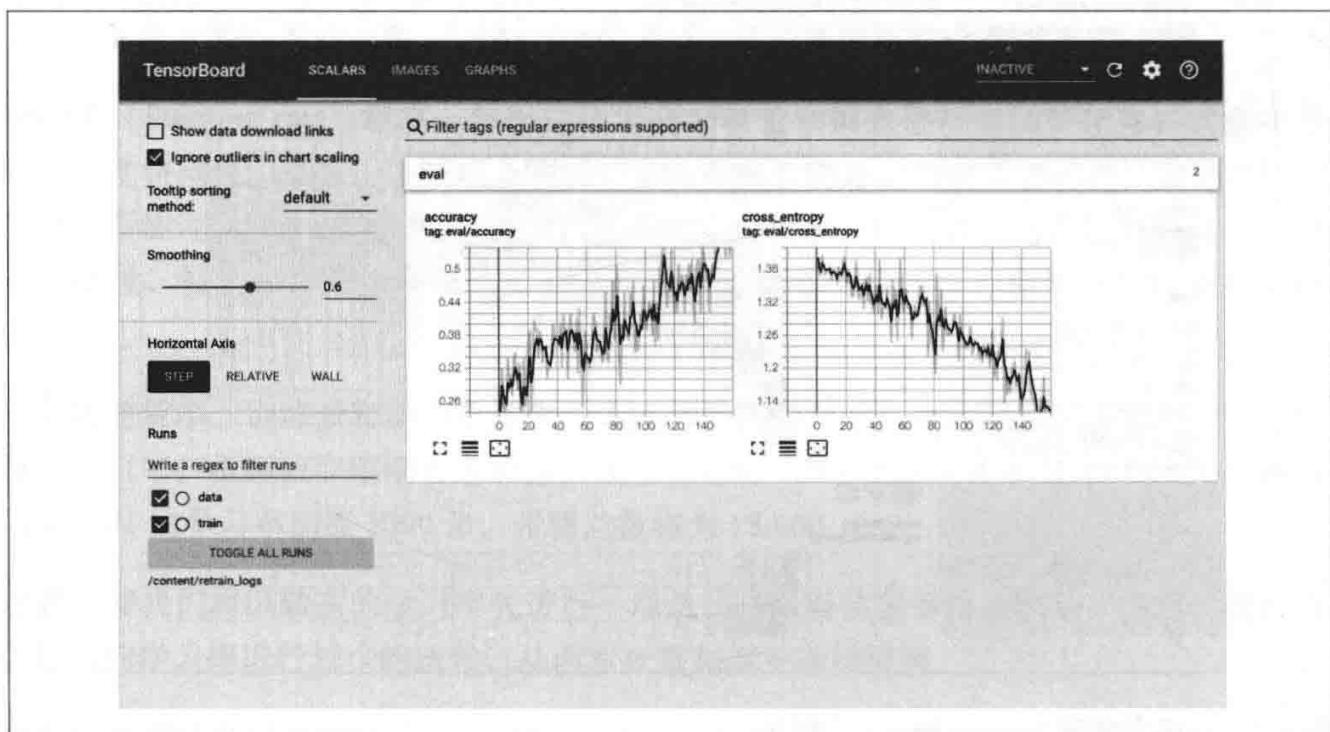


图 8-6: 在训练开始时 TensorBoard 的截图

好极了! 这意味着训练已经开始了。你刚刚运行的单元将在训练期间继续执行, 最多可能需要两个小时才能完成。该单元格将不再输出日志, 但有关训练运行的数据将显示在 TensorBoard 中。

你可以看到 TensorBoard 显示了两张图表, “accuracy” 和 “cross_entropy”, 如图 8-7 所示。两张图表都用 x 轴表示当前进行的步骤数。“accuracy” 图表用 y 轴表示模型准确率, 表示该模型在当前时间正确检测单词的能力。“cross_entropy” 图表展示了模型的损失值, 量化了模型的预测值与正确值之间的差距。



在进行分类任务的机器学习模型中, 交叉熵 (cross entropy) 是一种常用的测量损失值的方法, 其目标是预测输入属于哪一类别。

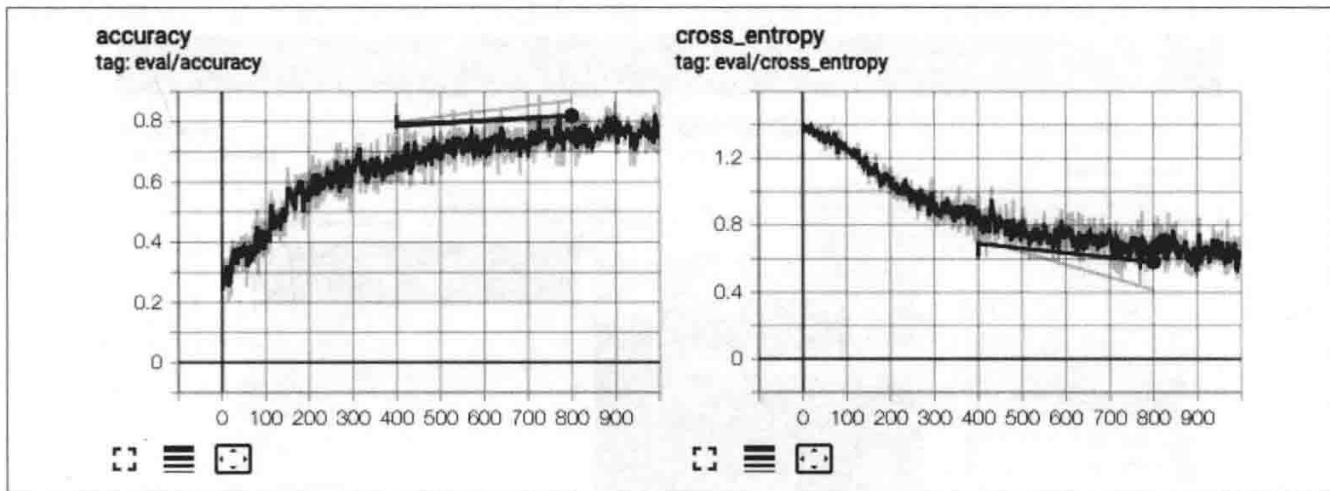


图 8-7：“accuracy” 和 “cross_entropy” 图表

图表中的锯齿线对应于训练数据集的表现，而直线反映了验证数据集的表现。验证是定期进行的，因此图形上的验证数据点较少。

随着时间的推移，新的数据将出现在图表中，但是要显示这些新数据，你需要调整图表的比例以适应数据。你可以通过单击每个图表下面的第三个按钮来执行此操作，如图 8-8 所示。



图 8-8：单击此按钮可以调整图表的比例以适应所有可用数据

你也可以单击图 8-9 中所示的按钮来放大每个图表。



图 8-9：单击此按钮可以放大图表

除了图表以外，TensorBoard 还可显示输入到模型中的输入。单击 IMAGES 选项卡，将显示类似图 8-10 的视图。这是在训练期间输入到模型的频谱图的示例。

等待训练完成

训练模型将花费一到两个小时，因此我们现在要做的是耐心等待。幸运的是，我们可以用 TensorBoard 的漂亮图表来自娱自乐。

随着训练的进行，你会注意到指标往往会在一定范围内跳跃。这是正常现象，但这会使图表显得模糊且难以阅读。为了更轻松地了解训练进行的情况，我们可以使用 TensorFlow 提供的平滑（smoothing）功能。

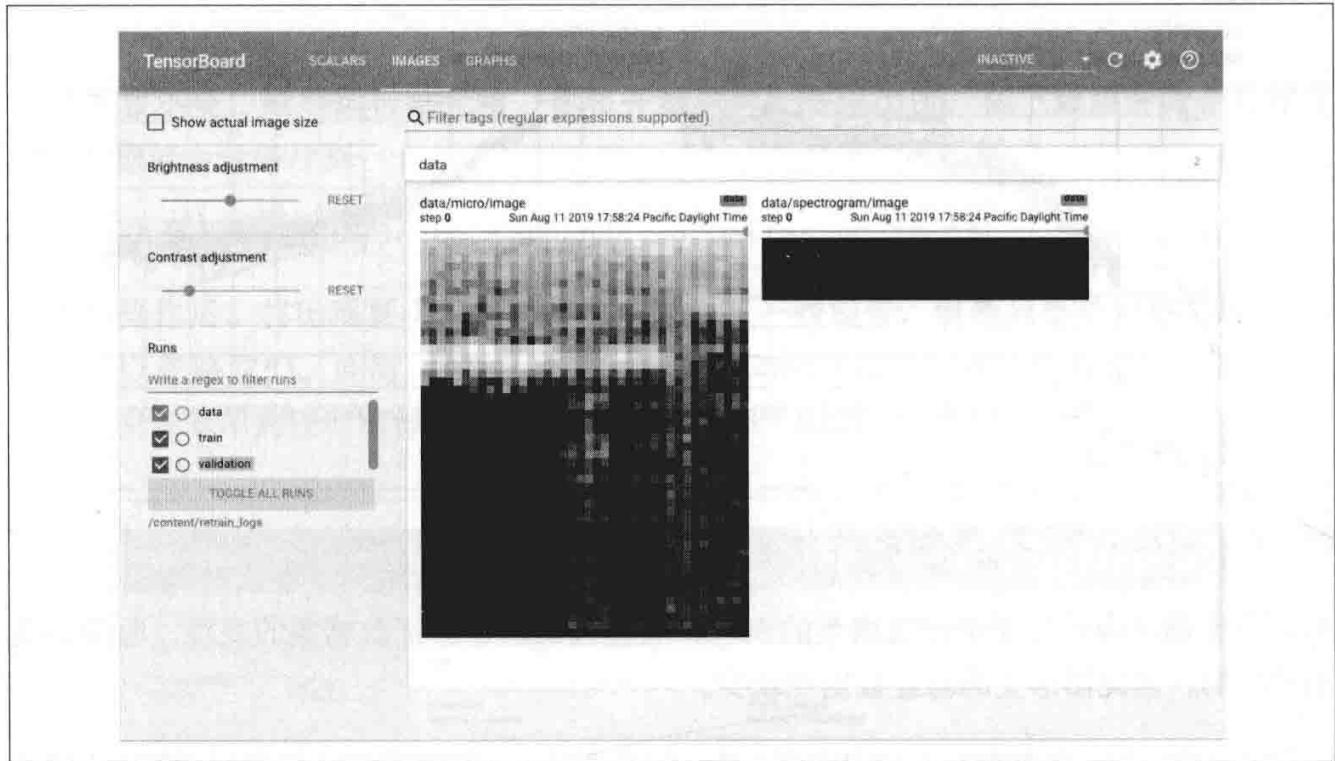


图 8-10：TensorBoard 的 IMAGE 选项卡

图 8-11 展示了应用了默认平滑度的图表，请注意这些图表有多模糊。

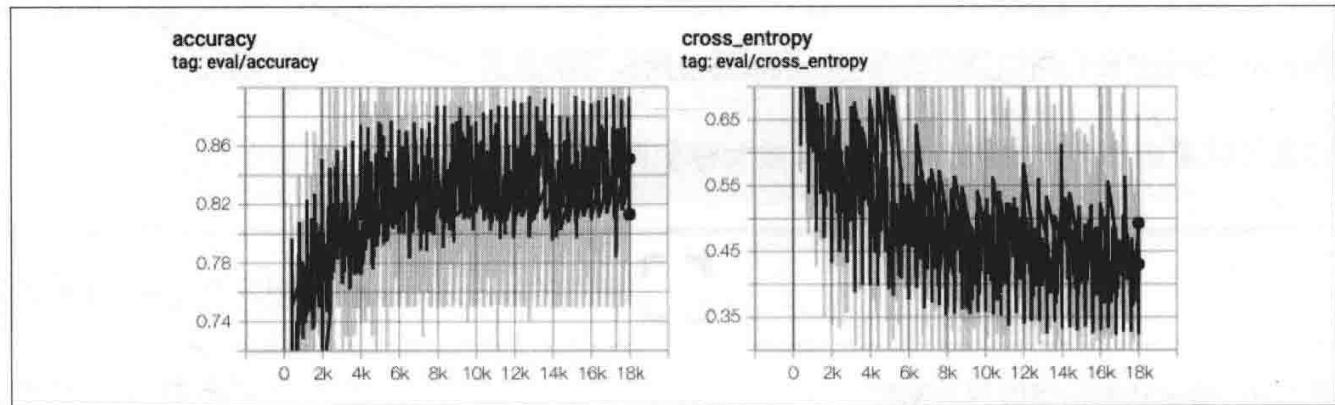


图 8-11：使用默认平滑度的训练图表

通过调整“Smoothing”（平滑度）滑块，我们可以增加平滑度，以使趋势更加明显，如图 8-12 所示。

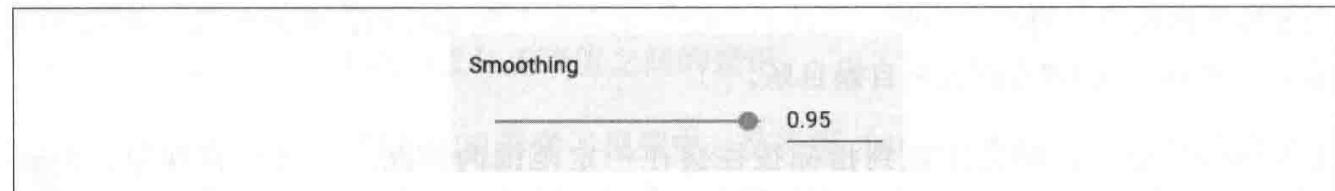


图 8-12：TensorBoard 的“Smoothing”滑块

图 8-13 展示了具有较高平滑度的相同图表。原始数据以较浅的颜色可见。

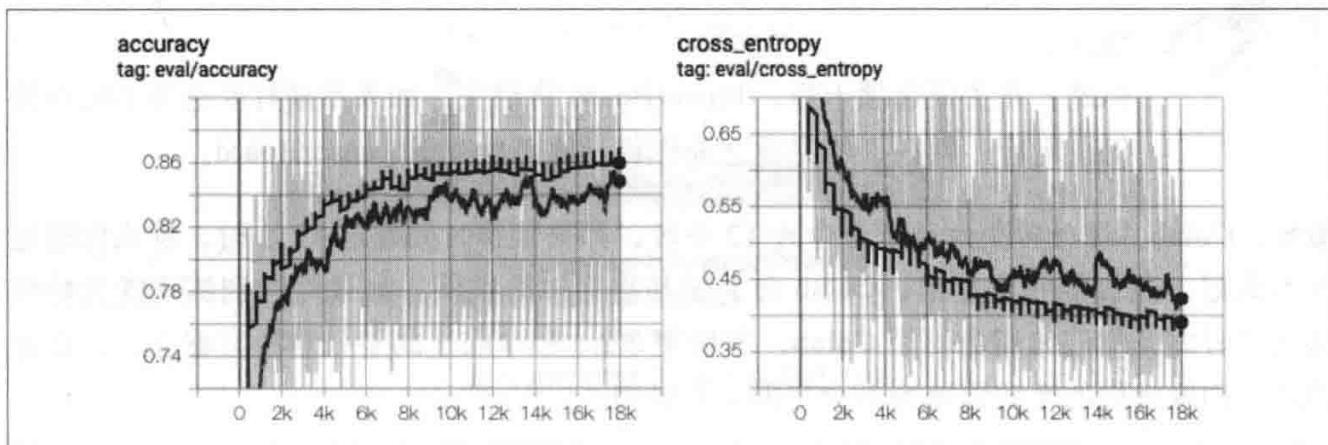


图 8-13：拥有更高平滑度的训练图表

保持 Colab 运行。为了防止废弃的项目占用资源，Colab 会在其没有被使用时关闭你的“运行时”。由于训练需要一段时间，所以我们需要防止这种情况的发生。我们需要考虑以下几件事。

首先，如果我们没有与 Colab 浏览器选项卡进行积极的交互，Web 用户界面将与正在执行训练脚本的后端“运行时”断开连接。这将在几分钟后发生，并将导致你的TensorBoard 图表停止更新最新的训练指标。如果发生这种情况，请不必惊慌，你的训练仍在后台进行。

如果你的“运行时”已断开连接，你将会在 Colab 的用户界面中看到一个“Reconnect”（重新连接）按钮，如图 8-14 所示。单击此按钮以重新连接你的“运行时”。

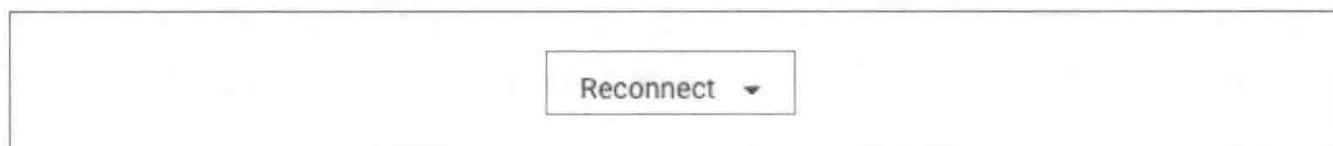


图 8-14：Colab 的“Reconnect”按钮

与“运行时”断开连接并不是一个很大的问题，但是 Colab 的下一个超时值得注意。如果你连续 90 分钟未与 Colab 进行交互，那么你的“运行时”实例将被回收。这会是一个问题：你将失去所有训练进度以及实例中存储的所有数据！

为避免这种情况发生，你只需要至少每 90 分钟与 Colab 进行一次交互。打开选项卡，确保已连接“运行时”，然后看看漂亮的图表。只要你在达到 90 分钟之前完成此操作，连接就会保持打开状态。



即使你关闭了 Colab 选项卡，其“运行时”仍将在后台继续运行长达 90 分钟。只要你在浏览器中打开原始 URL，就可以重新连接到“运行时”并继续。

但是，在关闭选项卡后，TensorBoard 将消失。如果重新打开选项卡时训练仍在运行，则在训练完成之前你将无法再次查看 TensorBoard。

最后，Colab “运行时”的最长寿命为 12 小时。如果训练时间超过 12 小时，那么你的运气不太好，在训练完成之前，Colab 会关闭并重置你的实例。如果你的训练可能会持续这么长时间，你应该避免使用 Colab，并使用 8.2.4 节中描述的替代解决方案之一。幸运的是，训练我们的唤醒词模型不会花那么长时间。

当图表显示了 18 000 个步骤的数据时，训练就完成了！我们现在必须运行更多的命令来准备部署模型。别担心，这部分要快得多。

冻结计算图

正如你在本书前面所学到的，训练是一个反复调整模型权重和偏差的过程，直到其产生有用的预测。训练脚本将这些权重和偏差写入 checkpoint 文件。每执行 100 个训练步将会写一次 checkpoint。这意味着，如果训练中途失败，可以从最近的 checkpoint 重新启动，而不会失去进度。

我们在调用 `train.py` 脚本时使用了一个参数 `--train_dir`，该参数指定了将在何处写入这些 checkpoint 文件。在我们的 Colab 中，该参数被设置为 `/content/speech_commands_train`。

你可以通过打开 Colab 的左边面板（其中有一个文件浏览器）来查看 checkpoint 文件。为此，单击图 8-15 所示的按钮。



图 8-15：打开 Colab 侧边栏的按钮

在此面板中，单击 Files 选项卡以查看运行时的文件系统。如果你打开 `speech_commands_train/` 目录，就会看到 checkpoint 文件，如图 8-16 所示。每个文件名中的数字表示保存 checkpoint 所在的步骤。

TensorFlow 模型包含两个主要内容：

- 训练产生的权重和偏差。
- 将模型的输入与这些权重和偏差相结合以产生模型的输出的算子图。

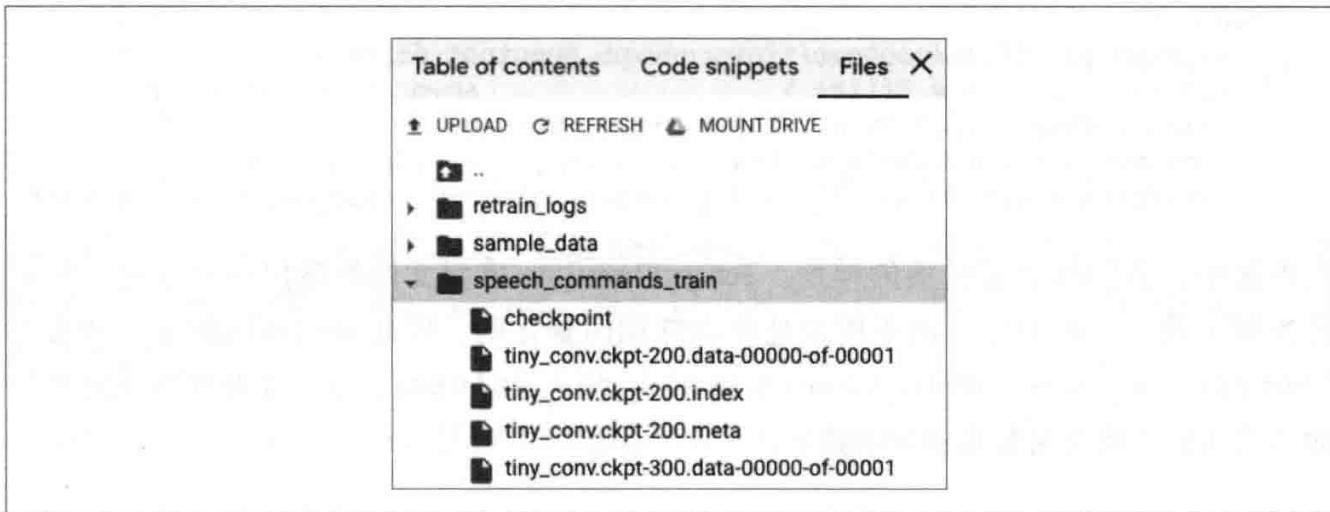


图 8-16：正在显示 checkpoint 文件列表的 Colab 文件浏览器

目前，我们的模型算子是在 Python 脚本中定义的，其受过训练的权重和偏差在最新的 checkpoint 文件中。我们需要将两者合并成具有特定格式的单个模型文件，以使用其运行推断。创建此模型文件的过程称为冻结（freezing）——我们正在创建计算图的静态表示，并将权重冻结在其中。

要冻结我们的模型，需要运行一个脚本。你将在下一个单元格的“Freeze the graph”（冻结计算图）部分找到它。该脚本的调用方式如下：

```
!python tensorflow/tensorflow/examples/speech_commands/freeze.py \
--model_architecture=tiny_conv --window_stride=20 --preprocess=micro \
--wanted_words=${WANTED_WORDS} --quantize=1 \
--output_file=/content/tiny_conv.pb \
--start_checkpoint=/content/speech_commands_train/tiny_conv. \
ckpt-${TOTAL_STEPS}
```

为了使脚本指向正确的要冻结的计算图，我们传递了一些与训练中使用的相同的参数。我们还将传递最终 checkpoint 文件所在的路径，该文件的文件名以训练步骤总数结尾。

运行此单元格以冻结计算图。冻结的计算图将被输出到名为 *tiny_conv.pb* 的文件。

该文件是经过全面训练的 TensorFlow 模型，它可以由 TensorFlow 加载并用于运行推断。很好，但该文件仍采用的是常规 TensorFlow 而非 TensorFlow Lite 所使用的格式。我们的下一步是将模型转换为 TensorFlow Lite 格式。

转换成 TensorFlow Lite

转换也是一个简单的步骤：我们只需要运行一个命令。由于现在我们有了一个冻结计算图文件，我们将使用 `toco`，即 TensorFlow Lite 转换器的命令行接口。

在“Convert the model”（转换模型）部分，运行第一个单元格：

```
!toco
--graph_def_file=/content/tiny_conv.pb --output_file= \
/content/tiny_conv.tflite \
--input_shapes=1,49,40,1 --input_arrays=Reshape_2 \
--output_arrays='labels_softmax' \
--inference_type=QUANTIZED_UINT8 --mean_values=0 --std_dev_values=9.8077
```

在参数中，我们指定要转换的模型、TensorFlow Lite 模型文件的输出位置以及一些其他依赖于模型架构的值。由于模型是在训练期间量化的，因此我们还提供了一些参数 (`inference_type`、`mean_values` 和 `std_dev_values`)，这些参数将指示转换器如何将其低精度值映射为实际的数字。

你可能想知道为什么 `input_shapes` 参数在宽、高和通道参数之前有一个前导 1。这其实是批次大小。为了提高训练期间的效率，我们通常将许多输入一起发送。但是当在实时应用程序中运行时，我们一次只能处理一个样本，这就是批次大小固定为 1 的原因。

转换后的模型将被写入 `tiny_conv.tflite`。恭喜！这是一个完整的 TensorFlow Lite 模型！

要查看此模型到底有多小，请在下一个单元格中运行代码：

```
import os
model_size = os.path.getsize("/content/tiny_conv.tflite")
print("Model is %d bytes" % model_size)
```

输出显示模型非常小，为 18 208 字节。

我们的下一步是将该模型转换为可部署到微控制器的形式。

创建一个 C 数组

回到 4.5 节，我们使用 `xxd` 命令将 TensorFlow Lite 模型转换为 C 数组。在下一个单元格中，我们将做同样的事情：

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

输出的最后一部分将是文件的内容，它是一个 C 数组和一个保存其长度的整数，如下所示（你看到的具体值可能会略有不同）：

```
unsigned char _content_tiny_conv_tflite[] = {
    0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0xe0, 0x00, 0x18, 0x00, 0x04, 0x00, 0x08, 0x00, 0xc0, 0x00,
```

```
// ...
0x00, 0x09, 0x06, 0x00, 0x08, 0x00, 0x07, 0x00, 0x06, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x04
};

unsigned int _content_tiny_conv_tflite_len = 18208;
```

该代码也被写入了一个文件，文件名是 *tiny_conv.cc*，你可以使用 Colab 的文件浏览器下载该文件。由于你的 Colab “运行时” 将在 12 小时后到期，因此最好立即将此文件下载到计算机上。

接下来，我们会将这个经过新训练的模型与 *micro_speech* 项目集成在一起，以便将其部署到某些硬件上。

8.2 在我们的项目中使用模型

要使用我们的新模型，我们需要做三件事情：

1. 在 *micro_features/tiny_conv_micro_features_model_data.cc* 中，用我们的新模型替换原始的模型数据。
2. 在 *micro_features/micro_model_settings.cc* 中，将标签名更换为新的“on”和“off”标签。
3. 更新特定于设备的 *command_responder.cc*，以执行我们要为新标签执行的操作。

8.2.1 替换模型

要替换模型，请在文本编辑器中打开 *micro_features/tiny_conv_micro_features_model_data.cc*。



如果你使用的是 Arduino 示例，则该文件将在 Arduino IDE 中显示为选项卡，其文件名为 *micro_features_tiny_conv_micro_features_model_data.cpp*。如果你使用的是 SparkFun Edge，则可以直接在 TensorFlow 代码仓库的本地拷贝中编辑文件。如果你使用的是 STM32F746G，则应在 Mbed 项目目录中编辑文件。

tiny_conv_micro_features_model_data.cc 文件包含一个数组声明，看起来像这样：

```
const unsigned char
g_tiny_conv_micro_features_model_data[] DATA_ALIGN_ATTRIBUTE = {
    0x18, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x0e, 0x00,
    0x18, 0x00, 0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00,
    //...
    0x00, 0x09, 0x06, 0x00, 0x08, 0x00, 0x07, 0x00, 0x06, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x04};

const int g_tiny_conv_micro_features_model_data_len = 18208;
```

你需要替换数组的内容以及常量 `g_tiny_conv_micro_features_model_data_len` 的值（如果数组长度发生变化）。

为此，请打开在上一节末尾下载的 `tiny_conv.cc` 文件。将数组的内容（而不是其定义）复制并粘贴到 `tiny_conv_micro_features_model_data.cc` 定义的数组中。请确保你覆盖的是数组的内容，而不是数组的声明。

在 `tiny_conv.cc` 的底部，你会找到 `_content_tiny_conv_tflite_len` 变量，该变量的值表示数组的长度。返回到 `tiny_conv_micro_features_model_data.cc` 中，用此变量的值替换 `g_tiny_conv_micro_features_model_data_len` 的值。然后保存文件，至此你已完成更新。

8.2.2 更新标签

接下来，打开 `micro_features/micro_model_settings.cc`。该文件包含一个类别标签的数组：

```
const char* kCategoryLabels[kCategoryCount] = {  
    "silence",  
    "unknown",  
    "yes",  
    "no",  
};
```

为了针对新模型进行调整，我们只需将“yes”和“no”分别替换为“on”和“off”即可。由于标签与模型的输出张量元素是按顺序进行匹配的，因此，请按照将标签提供给训练脚本的顺序列出这些标签，这一点很重要。

这是预期的代码：

```
const char* kCategoryLabels[kCategoryCount] = {  
    "silence",  
    "unknown",  
    "on",  
    "off",  
};
```

如果你训练的模型带有两个以上的标签，只需将它们全部添加到列表中即可。

现在，我们完成了模型切换。剩下的唯一步骤就是更新使用标签的输出代码。

8.2.3 更新 command_responder.cc

该项目包含针对 Arduino、SparkFun Edge 和 STM32F746G 的特定于设备的 `command_responder.cc` 实现。下面我们将展示如何更新这些内容。

Arduino

arduino/command_responder.cc 中的 Arduino 命令响应程序在听到“yes”一词时会将 LED 点亮 3 秒。让我们对其进行更新以使其在听到“on”或“off”时点亮 LED。在文件中，找到以下 if 语句：

```
// If we heard a "yes", switch on an LED and store the time.  
if (found_command[0] == 'y') {  
    last_yes_time = current_time;  
    digitalWrite(LED_BUILTIN, HIGH);  
}
```

if 语句判断命令的首字母是否为“y”（代表“yes”）。如果将“y”直接更改为“o”，那么在听到“on”或“off”时 LED 都会亮起，因为它们都以“o”开头：

```
if (found_command[0] == 'o') {  
    last_yes_time = current_time;  
    digitalWrite(LED_BUILTIN, HIGH);  
}
```

项目构想

说“off”来打开 LED 没有多大意义。请尝试更改代码，以便你可以通过说“on”打开 LED，并通过说“off”关闭 LED。

你可以使用通过 `found_command[1]` 访问的每个命令的第二个字母来区分“on”和“off”：

```
if (found_command[0] == 'o' && found_command[1] == 'n') {
```

完成这些代码更改后，你可以将代码部署到设备上并进行尝试。

SparkFun Edge

位于 *sparkfun_edge/command_responder.cc* 的 SparkFun Edge 命令响应程序根据听到的是“yes”还是“no”来点亮不同的 LED。在文件中，找到以下 if 语句：

```
if (found_command[0] == 'y') {  
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_YELLOW);  
}  
if (found_command[0] == 'n') {  
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_RED);  
}  
if (found_command[0] == 'u') {  
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_GREEN);  
}
```

更新这些内容以使得“on”和“off”分别打开不同的LED很简单：

```
if (found_command[0] == 'o' && found_command[1] == 'n') {
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_YELLOW);
}
if (found_command[0] == 'o' && found_command[1] == 'f') {
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_RED);
}
if (found_command[0] == 'u') {
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_GREEN);
}
```

由于两个命令都以相同的字母开头，所以我们需要查看它们的第二个字母以消除歧义。现在，当说出“on”时，黄色的LED会被点亮，而当说出“off”时，红色的LED会被点亮。

项目构想

尝试更改代码，以使得你可以通过说“on”持续点亮LED，并通过说“off”熄灭LED。

当你完成更改后，请按照7.5.2节中的相同步骤来部署和运行代码。

STM32F746G

位于*disco_f746ng/command_responder.cc*的STM32F746G命令响应程序根据听到的命令显示不同的单词。在文件中，找到以下if语句：

```
if (*found_command == 'y') {
    lcd.Clear(0xFF0F9D58);
    lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard yes!", CENTER_MODE);
} else if (*found_command == 'n') {
    lcd.Clear(0xFFDB4437);
    lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard no :(", CENTER_MODE);
} else if (*found_command == 'u') {
    lcd.Clear(0xFFFF4B400);
    lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard unknown", CENTER_MODE);
} else {
    lcd.Clear(0xFF4285F4);
    lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard silence", CENTER_MODE);
}
```

只需要进行简单的修改，就可以使代码响应“on”和“off”：

```
if (found_command[0] == 'o' && found_command[1] == 'n') {
    lcd.Clear(0xFF0F9D58);
    lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard on!", CENTER_MODE);
} else if (found_command[0] == 'o' && found_command[1] == 'f') {
```

```
lcd.Clear(0xFFDB4437);
lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard off", CENTER_MODE);
} else if (*found_command == 'u') {
    lcd.Clear(0xFFFF4B400);
    lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard unknown", CENTER_MODE);
} else {
    lcd.Clear(0xFF4285F4);
    lcd.DisplayStringAt(0, LINE(5), (uint8_t *)"Heard silence", CENTER_MODE);
}
```

同样，由于两个命令均以相同的字母开头，因此我们将查看它们的第二个字母以消除歧义。现在，我们为每个命令显示合适的文本。

项目构想

尝试更改代码，以便你可以通过说“on”显示一条秘密消息，并通过说“off”隐藏它。

8.2.4 运行脚本的其他方法

如果你无法使用 Colab，则建议使用另外两种方法来训练模型：

- 在带有 GPU 的云虚拟机 (Virtual Machine, VM) 上训练。
- 在你本地的工作站上训练。

基于 GPU 的训练所需的驱动程序仅在 Linux 上可用。没有 Linux，训练将花费大约四个小时。因此，我们建议使用带 GPU 的云虚拟机或具有类似配置的 Linux 工作站。

设置你的虚拟机或工作站超出了本书的范围。但是，我们确实有一些建议。如果你使用的是虚拟机，则可以启动 Google Cloud Deep Learning VM Image，该映像已预先配置了 GPU 训练所需的所有依赖项。如果你使用的是 Linux 工作站，TensorFlow GPU Docker 映像可以提供你所需的一切。

要训练模型，你需要安装每晚编译的 TensorFlow。要卸载任何现有版本并将其替换为经验证可以正常使用的版本，请使用以下命令：

```
pip uninstall -y tensorflow tensorflow_estimator
pip install -q tf-estimator-nightly==1.14.0.dev2019072901 \
tf-nightly-gpu==1.15.0.dev20190729
```

接下来，打开命令行并切换到你用于存放代码的目录。使用以下命令克隆 TensorFlow 并打开已确认可以正常工作的特定提交：

```
git clone -q https://github.com/tensorflow/tensorflow
git -c advice.detachedHead=false -C tensorflow checkout 17ce384df70
```

现在，你可以运行 `train.py` 脚本来训练模型。这将训练一个模型来识别“yes”和“no”，并将 checkpoint 文件输出到 `/tmp`：

```
python tensorflow/tensorflow/examples/speech_commands/train.py \
--model_architecture=tiny_conv --window_stride=20 --preprocess=micro \
--wanted_words="on,off" --silence_percentage=25 --unknown_percentage=25 \
--quantize=1 --verbosity=INFO --how_many_training_steps="15000,3000" \
--learning_rate="0.001,0.0001" --summaries_dir=/tmp/retrain_logs \
--data_dir=/tmp/speech_dataset --train_dir=/tmp/speech_commands_train
```

完成训练后，运行以下脚本以冻结模型：

```
python tensorflow/tensorflow/examples/speech_commands/freeze.py \
--model_architecture=tiny_conv --window_stride=20 --preprocess=micro \
--wanted_words="on,off" --quantize=1 --output_file=/tmp/tiny_conv.pb \
--start_checkpoint=/tmp/speech_commands_train/tiny_conv.ckpt-18000
```

接下来，将模型转换为 TensorFlow Lite 格式：

```
toco
--graph_def_file=/tmp/tiny_conv.pb --output_file=/tmp/tiny_conv.tflite \
--input_shapes=1,49,40,1 --input_arrays=Reshape_2 \
--output_arrays='labels_softmax' \
--inference_type=QUANTIZED_UINT8 --mean_values=0 --std_dev_values=9.8077
```

最后，将文件转换为可以编译进嵌入式系统的 C 源文件：

```
xxd -i /tmp/tiny_conv.tflite > /tmp/tiny_conv_micro_features_model_data.cc
```

8.3 模型的工作方式

现在你知道了如何训练自己的模型，让我们探索一下模型是如何工作的。到目前为止，我们已经把机器学习模型当作一个黑匣子：我们将训练数据输入其中，最终模型会自己找出如何预测结果。理解使用模型时在幕后发生了什么并不重要，但是这可以帮助调试问题，而且理解模型本身也很有趣。本节将为你提供一些有关模型如何进行预测的见解。

8.3.1 可视化输入

图 8-17 展示了实际输入到神经网络中的是什么。这是一个具有单个通道的二维矩阵，因此我们可以将其可视化为单色图像。我们正在处理的是 16kHz 音频样本数据，那么我们如何从音频数据源获得这种表示呢？这个过程是机器学习中被称为“特征生成”（feature generation）的一个例子，其目标是将更难处理的输入格式（在本例中，16 000 个数字值代表一秒音频）转换为更容易让机器学习模型理解的格式。如果你以前研究过用于深度学习的机器视觉用例，则可能没有遇到过这种情况，因为图像通常能相对更容易地被网络处理，不需要大量预处理。但是在许多其他领域，例如音频识别和自然语言处理，在将输入提供给模型之前先进行转换是很普遍的事情。



图 8-17：TensorBoard 的 IMAGES 选项卡

为了弄清为什么我们的模型更易于处理预处理过的输入，让我们看一下某些录音的原始表示形式，如图 8-18~图 8-21 所示。

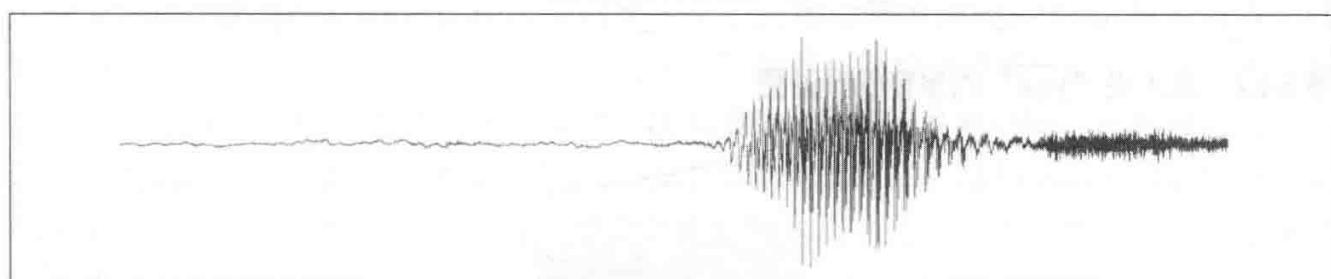


图 8-18：某人说“yes”的录音的波形图

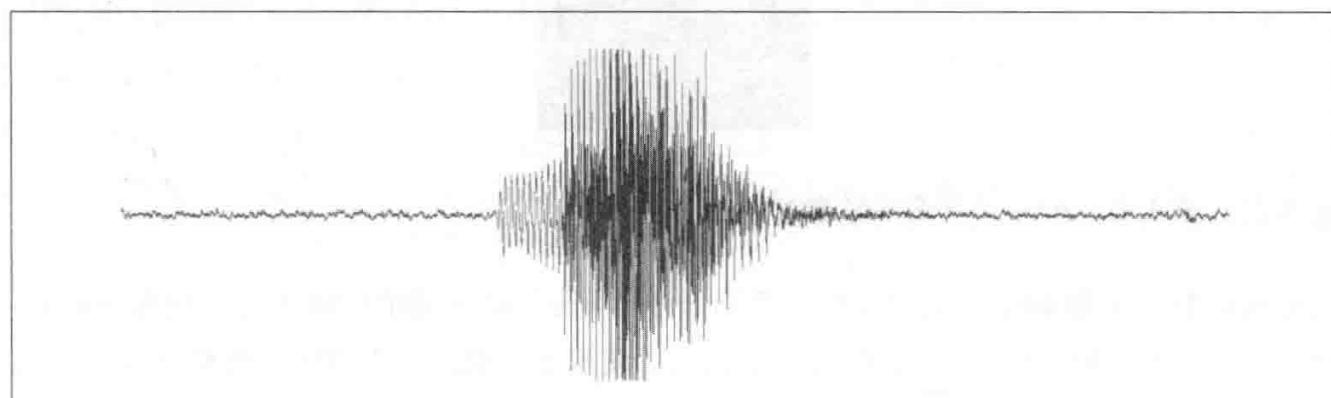


图 8-19：某人说“no”的录音的波形图



图 8-20：另一个人说“yes”的录音的波形图



图 8-21：另一个人说“no”的录音的波形图

如果没有标签，你将很难分辨出哪些波形对代表相同的单词。现在看看图 8-22~图 8-25，它们展示了通过特征生成处理相同的一秒录音的结果。

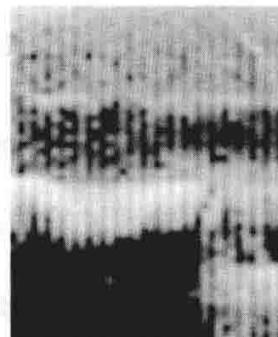


图 8-22：某人说“yes”的录音的频谱图



图 8-23：某人说“no”的录音的频谱图

虽然这些仍然很难解读，但你应该能看到两个“yes”频谱图的形状都有点像颠倒的 L，而“no”特征则显示了不同的形状。相比原始波形图，我们可以更容易地辨别频谱图之间的差异，并且模型也应该更容易区分它们。



图 8-24：另一个人说“yes”的录音的频谱图



图 8-25：另一个人说“yes”的录音的频谱图

另一个方面，生成的频谱图比样本数据要小得多。每个频谱图由 1960 个数值组成，而波形图具有 16 000 个数值。频谱图是音频数据的摘要，可减少神经网络必须完成的工作量。事实上，一个经过特别设计的模型，比如 DeepMind 的 WaveNet，是有可能把原始的样本数据作为输入的，但是其最终得到的模型往往要比我们使用的人工设计特征的神经网络组合需要更多的计算，所以对于资源受限的环境，比如嵌入式系统，我们更喜欢这里使用的方法。

8.3.2 特征生成是如何工作的

如果你有使用音频处理的经验，则可能熟悉梅尔频率倒谱系数（Mel-Frequency Cepstral Coefficients, MFCC）等方法。这是生成我们正在使用的频谱图的常用方法，但是我们的示例实际上使用了一种相关但不同的方法。该方法和 Google 生产环境中使用的方法是一样的，这意味着这种方法已经经过了许多实际验证，但其尚未在研究文献中发表。在这里，我们大致描述了这种方法的工作方式，但是对于细节，最好的参考是代码本身。

我们的处理方法是首先针对给定的时间片（在我们的情况下为 30 毫秒的音频数据）生

成傅里叶变换（也称为快速傅里叶变换（FFT））。该 FFT 是根据经过 Hann 窗口（Hann window）过滤的数据生成的，Hann 窗口是一种钟形函数，可减少 30 毫秒窗口两端对采样的影响。傅里叶变换会为每个频率产生具有实部和虚部的复数，但我们关心的是总能量，因此我们将两个分量的平方求和，再开平方来获得每个频率桶的幅度。

给定 N 个样本，傅里叶变换会产生关于 $N/2$ 个频率的信息。以每秒 16 000 次采样的速度，30 毫秒需要 480 次采样。由于我们的 FFT 算法需要两个输入的幂，所以我们用 0 填充 512 个样本，从而得到 256 个频率桶。然而这比我们需要的样本量要大，因此为了缩小样本数，我们将相邻的频率样本平均分成 40 个降采样的桶。不过，这种降采样不是线性的，而是使用基于人类感知的梅尔频率（mel frequency）标度来赋予较低的频率更多的权重，以便较低的频率有更多可用的桶，而较高的频率则会被合并为更宽泛的桶。图 8-26 给出了该过程的示意图。

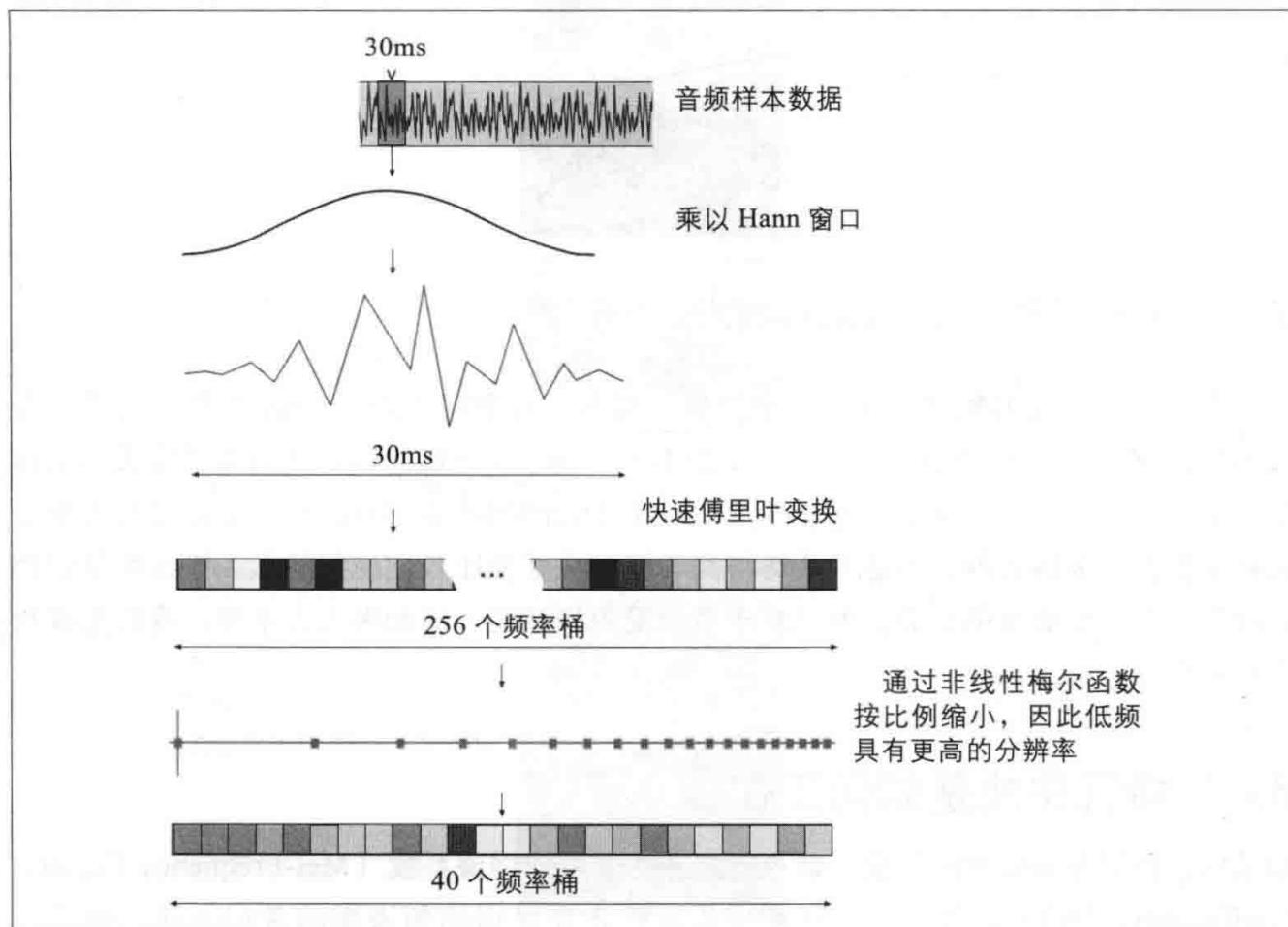


图 8-26：特征生成过程的示意图

这个特征生成器的一个不寻常的方面是其随后包括了降噪步骤。其工作原理是保持一个滑动窗口，计算每个频率桶中的平均值，然后从当前值中减去该平均值。这个想法的原理是，背景噪声将随时间变化相当恒定，并在特定频率下出现。通过减去这个随时计算的平均值，我们有很大的机会消除噪声的某些影响，并且留下我们感兴趣的、变化更

快的语音。棘手的部分是，特征生成器确实保留了跟踪每个桶的运行平均值的状态，因此，如果你试图为给定的输入重现相同的频谱图输出，就像我们在测试时尝试的那样，则需要将该状态重置为正确的值。

最初让我们吃惊的另一个降噪实现是对奇偶频率桶使用了不同的系数。这会产生独特的梳齿图案，你可以在最终生成的特征图像中看到（图 8-22~图 8-25）。最初，我们认为这是一个错误，但是在与原始实现者交谈时，我们了解到实际上这部分实现是刻意添加的，以帮助提高性能。Yuxuan Wang 等人在“Trainable Frontend for Robust and Far-Field Keyword Spotting”的 4.3 节中对此方法进行了广泛的讨论，其中还包括此特征生成流水线中进行的其他一些设计决策的背景。我们还用我们的模型对其进行实证检验，在消除奇偶桶处理的差异后，评估的准确率明显降低。

然后，我们使用基于通道的幅度标准化（Per-Channel Amplitude Normalization, PCAN）自动增强来增强基于运行平均噪声的信号。最后，我们将对数标度（log scale）应用于所有桶值，以使相对较大的频率不会淹没频谱中较安静的部分，这种标准化有助于后续模型使用这些特征。

这个过程总共重复了 49 次，一个 30 毫秒的窗口在每次迭代之间向前移动 20 毫秒，以覆盖整个一秒的音频输入数据。这将生成一个 40 个元素宽（每个频率桶一个）49 行高（每个时间片一行）的二维数组。

如果所有这些实现听起来很复杂，请不用担心。因为实现这些功能的代码都是开源的，所以欢迎你在自己的音频项目中重用这些代码。

8.3.3 理解模型架构

我们使用的神经网络模型被定义为一个小的算子计算图。你可以在 `create_tiny_conv_model()` 函数中找到在训练时定义该计算图的代码，图 8-27 展示了可视化后的结果。

该模型包括一个卷积层（convolutional layer），然后是一个全连接层，最后是一个 softmax 层。在图 8-27 中，卷积层被标记为“DepthwiseConv2D”，但这只是 TensorFlow Lite 转换器的一个“怪癖”（事实证明，具有单通道输入图像的卷积层也可以表示为深度卷积）。你还将看到一个标有“Reshape_1”的层，但这只是一个输入占位符，而不是实际算子。

卷积层用于识别输入图像中的二维图案。每个过滤器都是一个矩形的值阵列，该阵列作为滑动窗口在输入图像中移动，输出图像表示输入和过滤器在每个点上的匹配程度。你可以将卷积算子视为在图像上移动一系列矩形过滤器，每个过滤器的每个像素处的结果都与过滤器和图像中该块的相似程度相对应。在我们的示例中，每个过滤器的宽度为 8 像素，高度为 10 像素，总共有 8 个过滤器。图 8-28~图 8-35 展示了这些过滤器的外观。

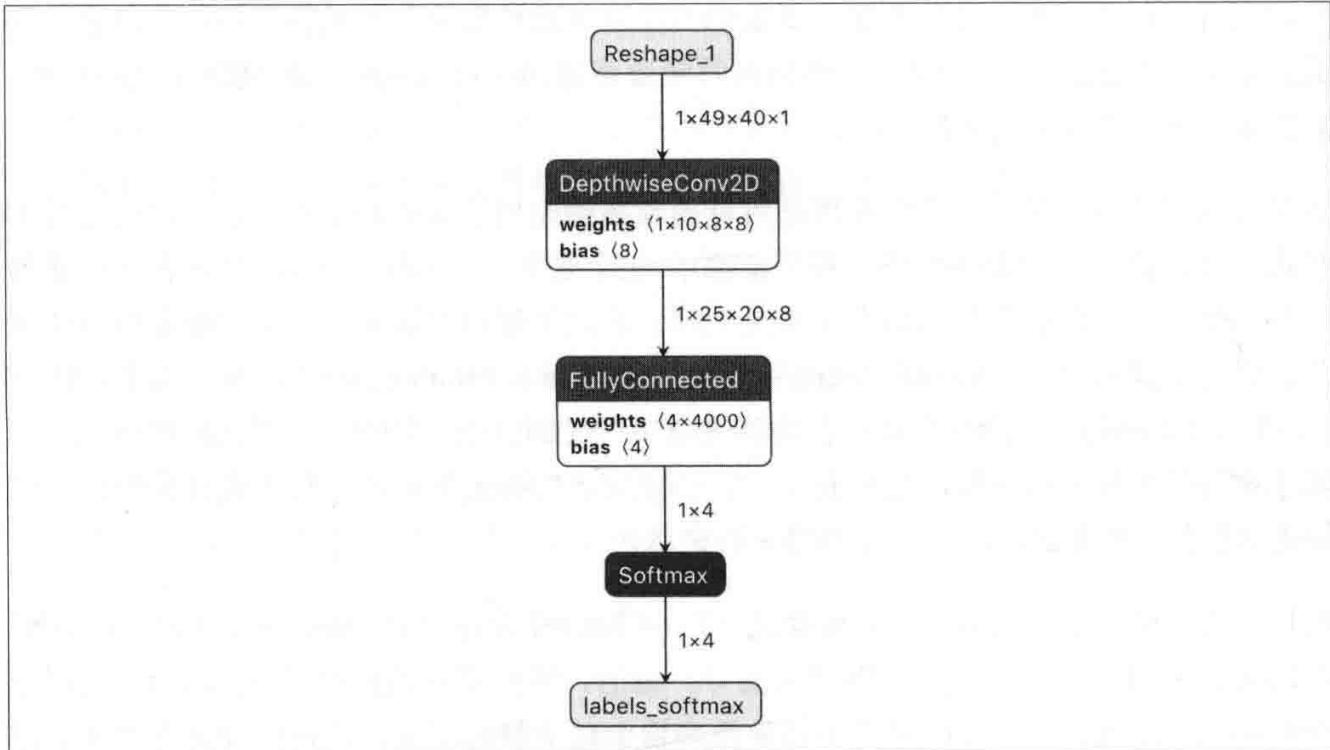


图 8-27：借助 Netron 工具生成的语音识别模型的图形可视化

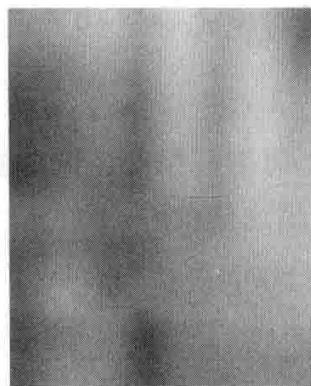


图 8-28：第一个过滤器的图片

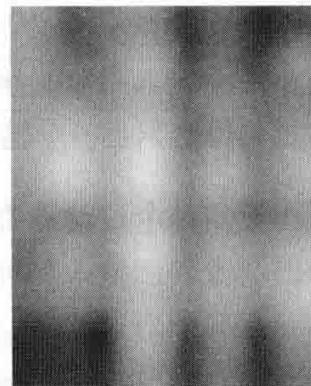


图 8-29：第二个过滤器的图片

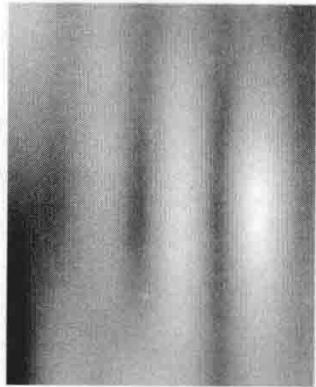


图 8-30：第三个过滤器的图片

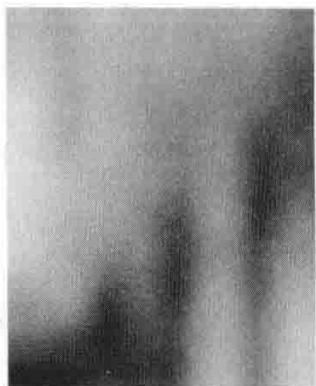


图 8-31：第四个过滤器的图片



图 8-32：第五个过滤器的图片

你可以将每一个这样的过滤器视为输入图像的一小部分。该算子会尝试将这一小部分图像与输入图像中看起来相似的部分进行匹配。如果图像的某部分与这一小部分相似，则会将较大的值写入输出图像的相应部分。直观地说，每个过滤器都是模型在训练输入图像中学习寻找的图案模式，以帮助区分其需要处理的不同类别。

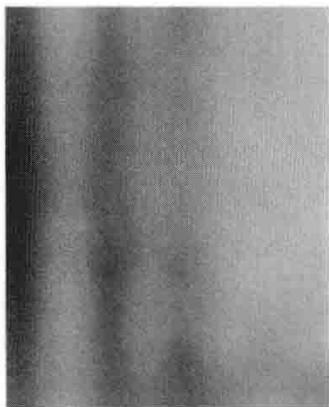


图 8-33：第六个过滤器的图片

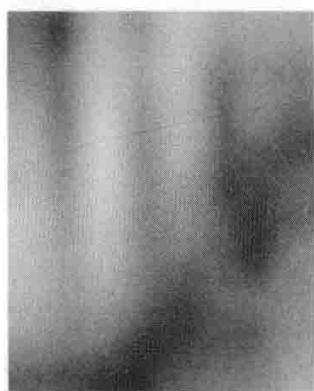


图 8-34：第七个过滤器的图片

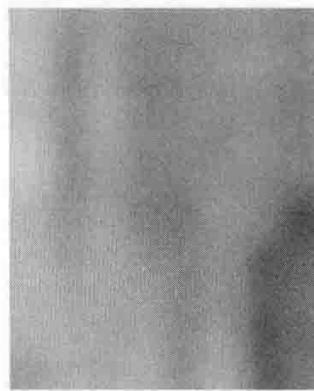


图 8-35：第八个过滤器的图片

由于我们有八个过滤器，所以会有八幅不同的输出图像，每幅图像对应于相应过滤器在输入图像中滑动时的匹配值。这些过滤器的输出实际上被组合成一个具有八个通道的输出图像。由于我们在两个方向上都将步幅（stride）设置为 2，这意味着在每次滑动中，过滤器将滑动两个像素而不是一个像素。由于我们跳过了这些像素，这意味着我们的输

出图像只有输入图像的一半大小。

你可以在可视化图中看到，输入图像高 49 像素、宽 40 像素，且只有一个通道，这正是我们在前一节讨论的特征频谱图时所看到的。由于当在输入图像上滑动卷积过滤器时，我们将在水平和垂直方向上跳过部分元素，因此，卷积的输出图像大小只有输入图像的一半，即 25 像素高 20 像素宽。但是由于有八个过滤器，所以图像变成了八个通道深。

下一个算子是全连接层，这是另一种模式匹配过程。在该过程中，输入张量中的每个值都有一个权重，而不是在输入图像上滑动一个小窗口。其结果表示的是在比较每个值之后，输入与权重的匹配程度。你可以将其视为一种全局的模式匹配，可以使用期望的结果作为输入，而输出其与实际输入的接近程度（体现在权重中）。我们模型中的每个类别都有自己的权重，因此对于“无声”“未知”“yes”和“no”都有一个理想的模式，并且会生成四个输出值。输入中有 4000 个值 ($25 * 20 * 8$)，因此每个类别都由 4000 个权重表示。

最后一层是 softmax。这一层有效地帮助增加了得分最高者与其最接近的竞争者之间的差异，这不会改变它们的相对顺序（在全连接层中得分最高的类别将继续保持最高得分），但是对产生更有用的得分很有帮助。该分数通常被非正式地称为概率（probability），但是严格来说，如果不对输入数据的实际组合进行更多校准，就无法可靠地使用该分数。例如，如果你在检测程序中有更多的单词，那么像“antidisestablishmentarianism”这样不常见的单词可能比“okay”这样的单词更不可能出现，但这取决于训练数据的分布，这些数据可能不会反映在原始分数中。

除了这些主要层之外，我们还在全连接层和卷积层的结果上添加了一些偏差，以帮助调整其输出，并在每层之后都提供了 ReLU 激励函数。ReLU 只是用来确保没有输出小于零，并将任何负数结果设置为最小值零。这种类型的激励函数是使深度学习变得更加有效的突破之一：它可以让训练过程更快收敛。

8.3.4 理解模型输出

模型的最终结果是 softmax 层的输出，即四个数字，分别代表“无声”“未知”“yes”和“no”。这些值是每个类别的得分，得分最高的是模型的预测结果，且得分代表的是模型对其预测的信心。例如，如果模型输出为 [10, 4, 231, 80]，则表明第三个分类——有着 231 分的“yes”是最有可能的结果。（我们以量化的形式给出这些值，这些值将介于 0 和 255 之间，但是由于这些只是相对得分，因此通常无须将其转换回它们的实际值。）

棘手的一件事是，该结果是基于对整段音频的最后一秒分析得出的。如果我们每秒只运行一次，那么最终可能得到一段话，其前一半位于上一秒，而后一半位于当前一秒。任

何模型都无法在仅听到一部分单词的情况下很好地识别单词，因此在这种情况下单词发现会失败。为了克服这一点，我们需要每秒运行模型一次以上，以便尽可能地在一秒的窗口中捕获整个单词。在实践中，我们发现必须每秒运行 10 或 15 次模型才能获得良好的效果。

如果我们能在很短的时间里获得所有这些结果，那么如何才能确定何时的得分是足够高的？我们实现了一个后处理类，该类可以对一段时间内的得分进行平均，当且仅当我们在短时间内对同一个单词获得多个高分时才触发识别。你可以在 `RecognizeCommands` 类中看到该实现。该后处理类将从模型中获取原始结果，然后使用累积和平均算法确定是否有任何类别超过了阈值。然后根据平台的输出能力，这些后处理的结果将被输入 `CommandResponder` 以采取行动。

由于模型参数都是从训练数据中学习的，而命令识别程序使用的算法是手动创建的，因此所有阈值——例如触发识别所需的得分值或所需的阳性结果的时间窗——都需要手动挑选。这意味着我们无法保证这些参数是最佳的，因此，如果你在自己的应用程序中看到较差的结果，则可能需要尝试自己进行调整。

更复杂的语音识别模型通常使用的是能够接收流数据（streaming data）的模型，例如递归神经网络（recursive neural network），而不是我们在本章中展示的单层卷积网络。将数据流纳入模型设计意味着你无须进行后处理即可获得准确的结果，尽管这样做确实会使训练变得更加复杂。

8.4 使用你自己的数据训练

你要构建的产品不太可能只需要回应“yes”和“no”，因此你需要训练一个对你所关心的音频敏感的模型。我们之前使用的训练脚本已经被设计成支持让你使用自己的数据创建自定义模型。该过程中最困难的部分通常是收集足够大的数据集，并确保该数据集适合你所面临的问题。我们将在第 16 章中讨论数据收集和清理的一般方法，但是本节将介绍一些训练你自己的音频模型的方法。

8.4.1 语音命令数据集

默认情况下，`train.py` 脚本下载 Speech Commands（语音命令）数据集。这是一个开源集合，包含超过 100 000 个一秒的 WAV 文件，涵盖了来自许多不同说话者的各种简短单词的语音。这个数据集是由 Google 发布的，但这些语音是从世界各地的志愿者那里收集的。Aakanksha Chowdhery 等人的“Visual Wake Words 数据集”提供了更多详细信息。

与 yes 和 no 一样，数据集还包含其他八个命令词（on、off、up、down、left、right、stop 和 go），以及从 0 到 9 的十个数字，每个单词都有数千个示例。另外还有一些其他单词，

例如 Marvin，这些单词的示例数量相对较少。该数据集旨在为每个命令词收集足够多的语音，以便让你能够训练一个合理的模型来识别它们。另一些词是用来填充“未知”类别的，这样一个模型就可以识别没有训练过的词，而不是误认其是命令词。

由于训练脚本使用此数据集，因此你可以轻松地使用具有许多示例的某些命令词来训练模型。如果使用训练集中以逗号分隔的单词列表更新 `--wanted_words` 参数并从头开始训练，你就可以创建一个有用的模型。需要注意的主要问题是，你需要将命令词或者数字的数量限制为十个及以下，否则你将没有足够的示例来精确地训练模型，并且如果你有两个以上想要的命令词，你就需要调低 `--silence_percentage` 和 `--unknown_percentage` 值。最后这两个参数控制在训练期间混入了多少无声样本和未知样本。无声的例子并不是真正的无声。相反，它们是从数据集的 `background` 文件夹中的 WAV 文件中随机选择的一秒记录的背景噪音片段。“未知”类别的样本是从训练集中不在 `wanted_words` 列表里的单词中挑选出来的。这就是为什么我们会从数据集中选择一些单词，并对这些单词选择数量相对较少的语音示例，因为这让我们有机会认识到，很多不同的单词实际上并不是我们要找的。这是语音和音频识别中的一个特殊问题，因为我们的产品通常需要在有着很多从未在训练中遇到的单词和噪音的环境中运行。在英语中有着成千上万个不同的单词，一个模型必须要能够忽略那些没有经过训练的单词才能发挥作用。这就是“未知”类别在实践中如此重要的原因。

以下是一个使用现有数据集训练不同单词的示例：

```
python tensorflow/examples/speech_commands/train.py \
--model_architecture=tiny_conv --window_stride=20 --preprocess=micro \
--wanted_words="up,down,left,right" --silence_percentage=15 \
--unknown_percentage=15 --quantize=1
```

8.4.2 在你自己的数据集上训练

训练脚本的默认设置是使用 Speech Commands 数据集进行训练，但如果你有自己的数据集，则可以通过指定 `--data_dir` 参数来使用它。你指向的目录应像语音命令一样组织，对于每个要被识别的类别都要创建一个子文件夹，每个子文件夹包含一组 WAV 文件。你还应该有一个特殊的 `background` 子文件夹，其中包含了应用程序可能会遇到的背景噪音的较长 WAV 录制。如果默认情况下一秒的识别持续时间不适用于你的用例，则还需要选择一个新的识别持续时间，然后通过 `--sample_duration_ms` 参数指定。然后，你可以使用 `--wanted_words` 参数设置要识别的类别。尽管名字是 `wanted_words`，这些类别可以是任意类型的语音事件，从玻璃的破碎声到笑声，只要对于每个类别有足够的 WAV 音频文件，训练过程就跟语音识别一样有效。

如果你在 `/tmp/my_wavs` 目录中有名为 `glass`（玻璃）和 `laughter`（笑声）的 WAV 文件夹，则可以使用以下方法来训练自己的模型：

```
python tensorflow/examples/speech_commands/train.py \
--model_architecture=tiny_conv --window_stride=20 --preprocess=micro \
--data_url="" --data_dir=/tmp/my_wavs/ --wanted_words="laughter,glass" \
--silence_percentage=25 --unknown_percentage=25 --quantize=1
```

通常，最困难的部分通常是找到足够的数据。举例来说，事实证明，玻璃破碎的真实声音与我们在电影中听到的音效有很大不同。这意味着你需要找到现有的录音，或者自己进行一些录音。因为训练过程可能需要每个类别有数千个示例，并且这些示例需要涵盖实际应用中可能发生的所有变化，所以数据收集的过程可能是令人沮丧、昂贵且耗时的。

对于图像模型来说，一个常见的解决方案是使用迁移学习（transfer learning）。在迁移学习中，你将获取一个已在大型公共数据集上训练过的模型，并使用其他数据微调其在不同类别上的权重。这种方法不需要辅助数据集有像从头开始训练那样多的示例，而且通常会产生高准确率的结果。但不幸的是，有关语音模型的迁移学习仍在研究中，但是请多留意这方面的研究。

8.4.3 如何录制你自己的音频

如果你需要捕获自己想要的单词的音频，那么拥有一个能提示说话者并将结果分割为带标签的文件的工具会容易得多。Speech Commands 数据集是使用 Open Speech Recording（开放语音录制）应用程序录制的，它是 Web 托管的应用程序，可让用户通过最常见的网络浏览器记录语音。作为用户，你会看到一个网页，该网页首先要求你同意被录音，这里使用的是默认的 Google 协议，该页面很容易更改。达成协议后，你将被转到一个具有录制控件的新页面。当你按下“录音”按钮时，单词将作为提示出现，并且你说出每个单词的音频都将被录制。记录完所有需要的单词后，系统会要求你将结果提交到服务器。

README 文件中有关于其在 Google Cloud 上运行的说明，但这是一个用 Python 编写的 Flask 应用程序，因此你应该可以将其移植到其他环境。如果你使用的是 Google Cloud，则需要更新 *app.yaml* 文件以指向自己的存储桶并提供自己的随机会话密钥（该密钥仅用于哈希，因此可以是任何值）。为了自定义要被录音的单词，你需要在客户端 JavaScript 中编辑几个数组：一个用于经常重复的主要单词，另一个用于辅助填充符。

录制的文件以 OGG 压缩音频的形式存储在 Google Cloud bucket 中，但是由于训练需要 WAV 格式，因此你需要对其进行转换。你的某些录音中还可能包含错误，例如忘记说出这个单词或声音太小，因此在可能的情况下自动过滤掉这些错误将有所帮助。如果你已经在 *BUCKET_NAME* 变量中设置了桶名称，则可以使用以下 bash 命令将文件复制到本地计算机上：

```
mkdir oggs
gsutil -m cp gs://${BUCKET_NAME}/* oggs/
```

压缩后的 OGG 格式的一个不错的特性是，安静或无声的音频会生成非常小的文件，因此一个好的第一步是删除任何特别小的文件，例如：

```
find ${BASEDIR}/oggs -iname "*.ogg" -size -5k -delete
```

我们发现将 OGG 文件转换为 WAV 文件的最简单方法是使用 FFmpeg 项目，该项目提供了命令行工具。以下是一组可以将整个目录中的 OGG 文件转换为我们所需的格式的命令：

```
mkdir -p ${BASEDIR}/wavs
find ${BASEDIR}/oggs -iname "*.ogg" -print0 | \
xargs -0 basename -s .ogg | \
xargs -I {} ffmpeg -i ${BASEDIR}/oggs/{}.ogg -ar 16000 ${BASEDIR}/wavs/{}.wav
```

开放语音录制应用程序为每个单词录制一秒以上的音频。这样可以确保即使用户说话的时间比我们预期的时间早或晚，也能捕获用户的话语。由于训练需要一秒的录音，所以当单词出现在每个录音中间的部分时效果最好。我们创建了一个小的开源实用程序来查看每次录制的音量，以尝试正确居中并修剪音频，使其仅为一秒钟。在终端中输入以下命令以使用该程序：

```
git clone https://github.com/petewarden/extract_loudest_section \
/tmp/extract_loudest_section_github
pushd /tmp/extract_loudest_section_github
make
popd
mkdir -p ${BASEDIR}/trimmed_wavs
/tmp/extract_loudest_section/gen/bin/extract_loudest_section \
${BASEDIR}'/wavs/*.wav' ${BASEDIR}/trimmed_wavs/
```

这将为你提供一个充满有着正确的格式和所需长度的文件的文件夹，但是训练过程需要按标签将 WAV 文件组织到子文件夹中。由于标签已经被编码在每个文件的文件名中，因此我们有一个示例 Python 脚本，该脚本使用这些文件名将这些文件分类到适当的文件夹中。

8.4.4 数据增强

数据增强（data augmentation）是扩展训练数据并提高准确率的另一种有效方法。在实践中，这意味着在将录音用于训练之前，我们将对这些录音进行音频转换。这些转换可以包括更改音量、混入背景噪音或稍微修剪录音的开始或结束部分。训练脚本在默认情况下会对输入数据应用所有这些转换，但是你可以使用命令行参数来调整使用这些转换的频率和强度。



这种增强确实有助于使小型数据集有所改善，但它不能创造奇迹。如果过分应用转换，最终可能会导致训练输入失真，使其无法被人识别，这可能会导致模型被与预期类别不相似的声音错误地触发。

你可以通过以下方式，使用其中一些命令行参数来控制增强：

```
python tensorflow/examples/speech_commands/train.py \
--model_architecture=tiny_conv --window_stride=20 --preprocess=micro \
--wanted_words="yes,no" --silence_percentage=25 --unknown_percentage=25 \
--quantize=1 --background_volume=0.2 --background_frequency=0.7 \
--time_shift_ms=200
```

8.4.5 模型架构

我们之前训练过的“yes” / “no”模型被设计得小巧且快速。该模型大小只有18KB，并且执行一次需要400 000次算术运算。为了适应这些约束，需要权衡准确率。如果你要设计自己的应用程序，则可能需要权衡取舍，尤其是当你尝试识别两个以上的类别时。你可以通过修改 *models.py* 文件，然后使用 *--model_architecture* 参数来指定自己的模型架构。你需要编写自己的模型创建函数，例如 *create_tiny_conv_model0*，并在其中指定你的模型所需的层。然后，你可以更新 *create_model0* 中的 *if* 语句来为你的架构命名，并在将该名称作为架构参数传入命令行时调用你的新模型创建函数。你可以查看一些现有的创建函数以获取灵感，包括如何处理 *dropout*。如果你添加了自己的模型代码，则可以通过以下方式调用：

```
python tensorflow/examples/speech_commands/train.py \
--model_architecture=my_model_name --window_stride=20 --preprocess=micro \
--wanted_words="yes,no" --silence_percentage=25 --unknown_percentage=25 \
--quantize=1
```

8.5 小结

在现实世界中使用有限的内存来识别人说出的某个单词是一个很困难的问题，要解决这个问题，我们需要使用比简单示例所需的更多、更复杂的组件。大多数产品环境中的机器学习应用程序都需要考虑诸如特征生成、模型架构选择、数据增强、寻找最合适的数据以及如何将模型的结果转化为可操作的信息之类的问题。

根据产品的实际需求，我们需要考虑很多折中方案，希望你现在能够理解一些从训练到部署的过程中可供权衡的选项。

在下一章中，我们将探讨如何对不同类型的数据运行推断，这些数据虽然看起来比音频复杂，但处理起来却非常容易。

行人检测：创建应用程序

如果你问人们人的哪种感觉对日常生活影响最大，许多人一定会回答“视觉”。^{注1}

视觉是一种非常有用的感觉。它使无数自然生物能够在环境中穿行、寻找食物来源，避免陷入危险。作为人类，视觉帮助我们识别我们的朋友、解释符号信息，使我们不必依靠近距离接触也能了解我们周围的世界。

直到不久之前，机器还没有视觉。大多数机器人只是在有限的范围内使用触摸和近距离传感器进行探测，从一系列碰撞中收集有关结构的信息。人类不需要与物体本身有任何的交互，只需看一眼就可以描述一个对象的形状、属性和用途。但是机器人没有这样的能力。对于机器人来说，视觉信息过于混乱，没有结构，而且难以理解。

随着卷积神经网络（Convolutional Neural Network, CNN）的发展，创建可以“看见”（see）的程序变得越来越容易。受到哺乳动物视觉皮层结构的启发，CNN 可以学习理解我们的视觉世界，将极其复杂的输入过滤到已知模型和形状的映射表中。这些碎片信息的精确组合可以告诉我们，给定的数字图像中存在哪些物体。

如今，视觉模型已经被用于许多不同的任务。自动驾驶汽车使用视觉来识别道路上的危险。工厂机器人使用相机找出有缺陷的零件。研究人员训练出的病例学模型可以根据医学图像诊断疾病。还有，你的智能手机很可能可以在照片中发现人脸，以确保它们能完美对焦。

有视力的机器可以帮助我们改造房屋和城市，使以前难以自动化的杂务自动化。但是视觉是一种亲密的感觉。我们大多数人都不喜欢有摄像头记录自己的行动，或者自己的生活被传输到云端——传统意义上进行机器学习推断的地方。

想象一下一个可以用内置摄像头“看到”周围世界的家用电器。它可能是可以发现入侵者的安全系统，或者是知道自己无人看管的烤箱，或者是在房间里没人的时候可以自动

注 1：在 2018 年 YouGov 的一项民意调查中，有 70% 的受访者表示，他们最不愿意失去的感觉是视觉。

关闭的电视。在每一种情境下，隐私都是至关重要的。即使不会有人观看这些录像，但嵌入始终开启的设备中、连接至互联网的摄像头所带来的安全隐患，也会使这些智能设备对大多数消费者失去吸引力。

但是，TinyML 改变了这一切。想象一个智能炉子，如果无人看管的时间过长，它将自动关闭。如果它无须连接到互联网，只通过微型控制器就可以“看到”附近的一个厨师，那么我们就可以在无须任何隐私权衡的情况下，享受智能设备带来的所有便利。

更重要的是，具备视觉能力的微型设备可以去到以前没有视力的机器不能去的任何地方。由于微控制器的能耗极小，因此基于微控制器的视觉系统可以依靠一块很小的电池持续运行几个月甚至几年。这些设备可以被植入丛林或珊瑚礁，无须联网，就可以记录和保存濒临灭绝的动物数量。

同样的技术可以使构建独立的视觉传感器电子组件成为可能。如果某个特定物体在视野中，传感器会输出 1，否则输出 0。但传感器不会与任何设备共享它的摄像头收集到的任何图像数据。这种类型的传感器可以被嵌入各种产品，从智能家居系统到私家车。当一辆汽车在你身后行驶时，你的自行车灯可能会闪烁。你的空调可以知道某人何时回家。而且，由于图像数据永远不会离开独立的传感器，因此即使产品已连接到互联网，也可以确保图像数据的安全。

我们将在本章中探讨的应用程序使用预先训练的行人检测模型，我们将在一个装有摄像头的微控制器上运行模型，以检测何时有行人出现在视线中。在第 10 章中，你将学习该模型的工作原理，以及如何训练自己的模型来检测任何你感兴趣的内容。

阅读本章后，你将了解如何在微控制器上使用摄像头数据，以及如何通过视觉模型进行推断并解释输出。你可能会惊讶于它实际上竟是如此简单！

9.1 我们在创建什么

我们将创建一个嵌入式应用程序，它使用机器学习模型对摄像头捕获的图像进行分类。模型被训练来识别何时有人出现在摄像头中。这意味着我们的应用程序将能够检测到某个人的存在与否，并产生相应的输出。

本质上，这就是我们之前介绍的智能视觉传感器。当检测到一个行人时，我们的示例代码将点亮一个 LED，但是你可以扩展它来控制各种项目。



与我们在第 7 章中处理过的应用程序一样，你可以在 TensorFlow GitHub 代码仓库中找到此应用程序的源代码。

与前面的章节一样，我们首先遍历所有的测试和应用程序代码，然后是示例在各种设备上的工作逻辑。

我们提供了将应用程序部署到以下微控制器平台的说明：

- Arduino Nano 33 BLE Sense
- SparkFun Edge



TensorFlow Lite 会定期添加对新设备的支持，因此，如果你想使用的设备未在此处列出，那么请查看示例的 *README.md*。如果在执行以下步骤时遇到麻烦，也可以查看更新的部署说明。

与前几章不同，你需要一些额外的硬件来运行本章的应用程序。由于这两块开发板都没有集成摄像头，因此我们建议购买一个摄像头模块（camera module）。你可以在每个设备的有关部分中找到此信息。

摄像头模块

摄像头模块是基于图像传感器（image sensor）的电子元件，它可以用数字的方式捕获图像数据。图像传感器将镜头与电子控制设备相结合，并且模块的制造方式使其易于与电子项目连接。

让我们开始浏览应用程序的结构。它会比你预期的要简单得多。

9.2 应用程序架构

到目前为止，我们已经大概了解创建嵌入式机器学习应用程序要做的事情：

1. 获取输入。
2. 预处理输入以提取适合输入模型的特征。
3. 对处理后的输入进行推断。
4. 对模型的输出进行后处理以使其有意义。
5. 使用后处理的结果达到我们需要的效果。

在第 7 章中，我们看到了基于此步骤创建具备唤醒词检测功能的应用程序，它使用音频作为输入。这一次，我们的输入是图像数据。这听起来可能比较复杂，但实际操作起来比音频要简单得多。

图像数据通常表示为像素值数组。我们将从嵌入式摄像头模块中获取图像数据，这些模块会以像素值数组的形式将数据提供给我们。同时，我们的模型本就期望它的输入为像素值数组。因此，在将数据输入模型之前，我们不需要进行太多的预处理。

考虑到我们无须进行太多的预处理，我们的应用程序将会非常简单。它从摄像头模块中获取数据快照，将数据输入模型，并确定检测到哪个输出类别。然后，程序以某种简单的方式显示结果。

在继续之前，让我们进一步了解将要使用的模型。

9.2.1 模型介绍

在第 7 章中，我们了解到卷积神经网络是用于处理多维张量的神经网络，这些多维张量的隐藏信息包含在相邻值之间的关系中。卷积神经网络特别适合处理图像数据。

我们的行人检测模型是在 Visual Wake Words 数据集上训练的卷积神经网络。这个数据集包含 115 000 张图像，每张图像都被标记了是否包含行人。

我们的模型大小为 250KB，比我们的语音模型大得多。除了占用更多内存外，它还意味着运行单次推断将会花费更长的时间。

我们的模型接收 96 像素 \times 96 像素的灰度图像作为输入。每个图像都是一个形状为 $(96, 96, 1)$ 的三维张量，其中最终维包含一个表示单个像素的 8 位数值。它表示像素的灰度值，范围从 0（全黑）到 255（全白）。

我们的摄像头模块可以返回各种分辨率的图像，因此我们需要确保将它的尺寸调整为 96 像素 \times 96 像素。我们还需要将全彩色的图像转换为灰度图，以便它们可以与模型一起使用。

你可能会认为 96 像素 \times 96 像素听起来是很小的分辨率，但是这已经足够让我们在每个图像中检测是否有行人。与图像相关的模型通常都接收小得惊人的小分辨率图像。增加模型的输入的大小会导致收益递减，同时，网络的复杂度也随着输入输出的增多而大大提高。因此，即使是最先进的图像分类模型，通常最多也只能处理 320 像素 \times 320 像素的图像。

我们的模型会输出两个概率：一个表示输入中存在一个人的概率，另一个表示输入中不存在行人的概率。概率的范围为 0~255。

我们的行人检测模型使用广为人知、久经考验的 MobileNet 架构，它被设计用于在移动电话等设备上进行图像分类。在第 10 章中，你将学习如何调整此模型以使其适合微

控制器，以及如何训练自己的模型。现在，让我们继续探索我们的应用程序是如何工作的。

9.2.2 所有的功能组件

图 9-1 展示了我们的行人检测应用程序的架构。

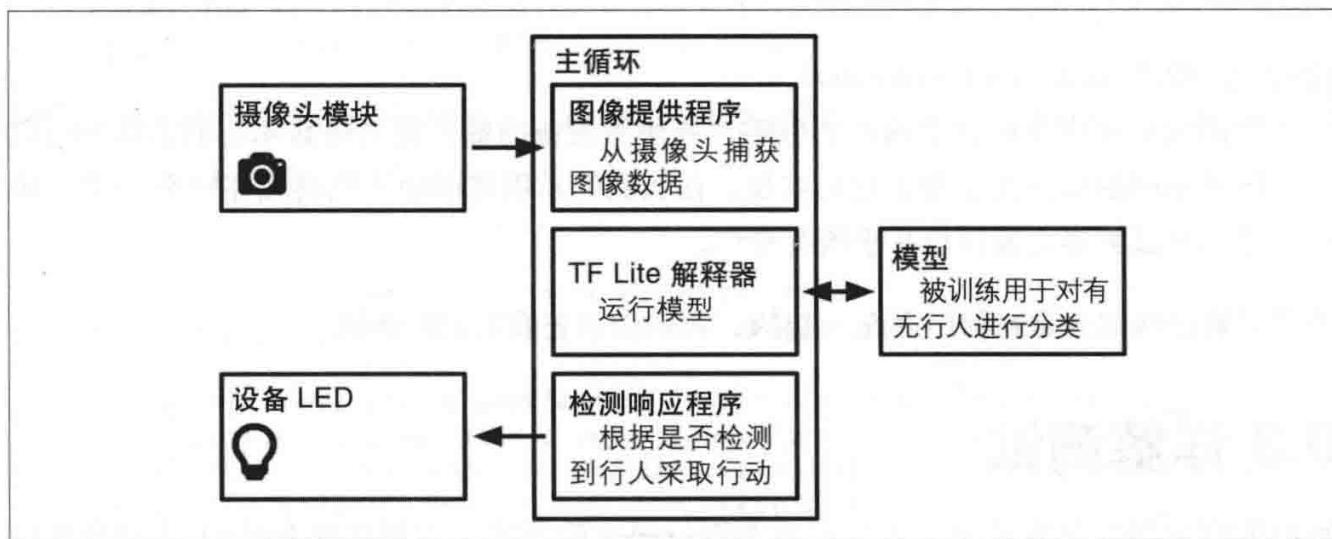


图 9-1：行人检测应用程序的组件

正如我们前面提到的，行人检测应用程序比唤醒词程序简单得多，因为我们可以将图像数据直接传递到模型中——无须进行预处理。

使事情变得简单的另外一方面是，我们不需要对模型的输出求平均值。我们的唤醒词模型每秒运行多次，因此我们必须对它的输出取平均以获得稳定的结果。但是我们的行人检测模型要大得多，并且运行推断所需的时间也要长得多，这就意味着我们没有必要平均它的输出。

我们的代码包含五个主要部分：

主循环 (*main Loop*)

与其他示例一样，我们的应用程序在一个连续循环中运行。但是，由于我们的模型更大也更复杂，因此进行推断需要更长的时间。根据设备的不同，我们可以预期程序每几秒进行一次推断，而不是每秒进行几次推断。

图片提供程序 (*image provider*)

这个部分从摄像头捕获图像数据并将它们写入输入张量。捕获图像的方法因设备而异，因此可以覆盖并自定义此部分的实现。

TensorFlow Lite 解释器 (*TensorFlow Lite interpreter*)

解释器运行 TensorFlow Lite 模型，将输入图像转换为一组概率。

模型 (*model*)

模型作为数据数组包含在内，并由解释器运行。我们的模型的大小为 250KB，由于它太大了，所以无法将其提交到 TensorFlow GitHub 代码仓库中。因此，在编译项目时，模型将由 Makefile 下载。如果你想看一下，可以在 *tf_lite_micro_person_data_grayscale.zip* 中自己下载。

检测响应程序 (*detection responder*)

检测响应程序获取模型输出的概率，并使用设备的输出能力来显示它们。我们可以针对不同的设备类型覆盖它的实现。在我们的示例代码中，它将点亮一个 LED，但是你可以扩展它来执行几乎所有操作。

为了了解这些部分是如何组合在一起的，我们先来查看它们的测试。

9.3 详解测试

我们的应用程序非常简单，因此只需要进行很少的测试。你可以在 GitHub 代码仓库中找到它们：

person_detection_test.cc

演示如何在表示单个图像的数组上运行推断。

image_provider_test.cc

演示如何使用图像提供程序捕获图像。

detection_responder_test.cc

演示如何使用检测响应程序输出检测结果。

让我们从探索 *person_detection_test.cc* 开始，看看如何对图像数据进行推断。这已经是我们介绍的第三个示例了，因此你应该对这段代码很熟悉。你很快就可以成为一个嵌入式机器学习开发者了！

9.3.1 基本流程

首先来看 *person_detection_test.cc*。我们先介绍模型需要的算子：

```
namespace tflite {
namespace ops {
namespace micro {
TfLiteRegistration* Register_DEPTHWISE_CONV_2D();
TfLiteRegistration* Register_CONV_2D();
```

```
TfLiteRegistration* Register_AVERAGE_POOL_2D();
} // namespace micro
} // namespace ops
} // namespace tflite
```

接下来，我们定义一个适当大小的张量 arena。与往常一样，这个数字是通过反复试验得出的：

```
const int tensor_arena_size = 70 * 1024;
uint8_t tensor_arena[tensor_arena_size];
```

然后，我们进行一些典型的设置，以使解释器准备就绪，其中包括使用 `MicroMutableOpResolver` 注册必要的算子：

```
// Set up logging.
tflite::MicroErrorReporter micro_error_reporter;
tflite::ErrorReporter* 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.
const tflite::Model* model = ::tflite::GetModel(g_person_detect_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);
}

// Pull in only the operation implementations we need.
tflite::MicroMutableOpResolver micro MutableOpResolver;
micro MutableOpResolver.AddBuiltin(
    tflite::BuiltInOperator_DEPTHWISE_CONV_2D,
    tflite::ops::micro::Register_DEPTHWISE_CONV_2D());
micro MutableOpResolver.AddBuiltin(tflite::BuiltInOperator_CONV_2D,
                                    tflite::ops::micro::Register_CONV_2D());
micro MutableOpResolver.AddBuiltin(
    tflite::BuiltInOperator_AVERAGE_POOL_2D,
    tflite::ops::micro::Register_AVERAGE_POOL_2D());

// Build an interpreter to run the model with.
tflite::MicroInterpreter interpreter(model, micro MutableOpResolver,
                                    tensor_arena, tensor_arena_size,
                                    error_reporter);
interpreter.AllocateTensors();
```

下一步是检查输入张量。我们检查它是否具有预期的维度数量，以及维度大小是否合适：

```
// Get information about the memory area to use for the model's input.
TfLiteTensor* input = interpreter.input(0);

// Make sure the input has the properties we expect.
TF_LITE_MICRO_EXPECT_NE(nullptr, input);
TF_LITE_MICRO_EXPECT_EQ(4, input->dims->size);
```

```
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);
TF_LITE_MICRO_EXPECT_EQ(kNumRows, input->dims->data[1]);
TF_LITE_MICRO_EXPECT_EQ(kNumCols, input->dims->data[2]);
TF_LITE_MICRO_EXPECT_EQ(kNumChannels, input->dims->data[3]);
TF_LITE_MICRO_EXPECT_EQ(kTfLiteUInt8, input->type);
```

从这里我们可以看出，输入是四维张量。第一维只是一个包含单个元素的包装器。随后的两个维度分别代表图像像素的行和列。最后一个维度保留用于表示每个像素的颜色通道的数量。

表示我们预期维度的常数 `kNumRows`、`kNumCols` 和 `kNumChannels` 在 `model_settings.h` 中定义。具体如下：

```
constexpr int kNumCols = 96;
constexpr int kNumRows = 96;
constexpr int kNumChannels = 1;
```

正如你所看到的，我们的模型将接收一个 96 像素 \times 96 像素的位图。图像是灰度的，每个像素只有一个颜色通道。

在代码的下一步，我们使用简单的 `for` 循环将测试图像复制到输入张量中：

```
// Copy an image with a person into the memory area used for the input.
const uint8_t* person_data = g_person_data;
for (int i = 0; i < input->bytes; ++i) {
    input->data.uint8[i] = person_data[i];
}
```

存储图像数据的变量 `g_person_data` 在 `person_image_data.h` 中定义。为了避免向代码仓库中添加更多的大型文件，首次运行测试时，数据本身会与模型一起作为 `tf-lite-micro_person_data_grayscale.zip` 的一部分被下载。

填充输入张量后，我们开始进行推断。就像以往一样简单：

```
// Run the model on this input and make sure 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);
```

现在，我们检查输出张量，以确保它具有预期的大小和形状：

```
TfLiteTensor* output = interpreter.output(0);
TF_LITE_MICRO_EXPECT_EQ(4, output->dims->size);
TF_LITE_MICRO_EXPECT_EQ(1, output->dims->data[0]);
TF_LITE_MICRO_EXPECT_EQ(1, output->dims->data[1]);
TF_LITE_MICRO_EXPECT_EQ(1, output->dims->data[2]);
TF_LITE_MICRO_EXPECT_EQ(kCategoryCount, output->dims->data[3]);
```

```
TF_LITE_MICRO_EXPECT_EQ(kTfLiteUInt8, output->type);
```

模型的输出有四个维度，前三个只是第四个的包装器，其中包含模型针对每个类别训练的一个元素。

类别总数是一个常量，即 `kCategoryCount`，它与其他一些有用的值一起在 `model_settings.h` 中定义：

```
constexpr int kCategoryCount = 3;
constexpr int kPersonIndex = 1;
constexpr int kNotAPersonIndex = 2;
extern const char* kCategoryLabels[kCategoryCount];
```

如 `kCategoryCount` 所示，输出中包含三个类别。第一个类别是一个未使用的类别，我们可以忽略它。从类别索引中可以看到，“person”（行人）类别排在第二位，它的索引存储在常量 `kPersonIndex` 中。“notperson”（非行人）类别排在第三位，它的索引由 `kNotAPersonIndex` 表示。

还有一个类别标签数组 `kCategoryLabels`，它在 `model_settings.cc` 中实现：

```
const char* kCategoryLabels[kCategoryCount] = {
    "unused",
    "person",
    "notperson",
};
```

冗余维度

输出张量的结构有一些冗余。为什么它明明只需要保存三个值，即每个类别一个概率就可以，却使用了四个维度？而且，我们只是试图区分“行人”和“非行人”，为什么会需要三种类别？

你会发现模型经常会具有一些古怪的输入和输出形状，或者似乎没有太大用处的多余类别。有时这些只是它们所使用的架构的特征，有时这些只是实现细节。无论出于什么原因，我们都不必为此担心。由于张量的数据内容是作为一个平面的内存数组存储的，因此是否将其包装在不必要的冗余维度中并没有太大区别。我们仍然可以通过给定的索引轻松地访问元素。

下一部分代码记录了“行人”和“非行人”的得分，并断言“行人”得分更高，这是因为我们传递了包含一个行人的图像：

```
uint8_t person_score = output->data.uint8[kPersonIndex];
uint8_t no_person_score = output->data.uint8[kNotAPersonIndex];
```

```
error_reporter->Report(
    "person data. person score: %d, no person score: %d\n", person_score,
    no_person_score);
TF_LITE_MICRO_EXPECT_GT(person_score, no_person_score);
```

由于输出张量的唯一数据内容是表示类别得分的三个 `uint8` 值，而第一个值是未使用的，因此我们可以使用 `output->data.uint8[kPersonIndex]` 和 `output->data.uint8[kNotAPersonIndex]` 直接访问得分。作为 `uint8` 类型值，它们的最小值为 0，最大值为 255。



如果“行人”和“非行人”的得分很接近，则表明模型对它的预测并不是很有信心。在这种情况下，你可以选择认为结果是不确定的。

接下来，我们测试一张没有人的图像，图像保存在 `g_no_person_data` 中：

```
const uint8_t* no_person_data = g_no_person_data;
for (int i = 0; i < input->bytes; ++i) {
    input->data.uint8[i] = no_person_data[i];
}
```

进行推断后，我们断言“非行人”的得分更高：

```
person_score = output->data.uint8[kPersonIndex];
no_person_score = output->data.uint8[kNotAPersonIndex];
error_reporter->Report(
    "no person data. person score: %d, no person score: %d\n", person_score,
    no_person_score);
TF_LITE_MICRO_EXPECT_GT(no_person_score, person_score);
```

正如你所看到的，这里并没有什么奇特的事情发生。我们的输入的是图像而不是标量或频谱图，但是推断的过程与我们之前看到的十分类似。

运行测试同样非常简单。只需从 TensorFlow 代码仓库的根目录运行以下命令：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
      test_person_detection_test
```

第一次运行测试时，将下载模型和图像数据。如果想查看下载的文件，你可以在 `tensorflow/lite/micro/tools/make/downloads/person_model_grayscale` 中找到它们。

接下来，我们检查图像提供程序的接口。

9.3.2 图像提供程序

图像提供程序负责从摄像头获取数据，并以适合写入模型输入张量的格式返回数据。文

件 *image_provider.h* 定义了图像提供程序的接口：

```
TfLiteStatus GetImage(tfLite::ErrorReporter* error_reporter, int image_width,  
                      int image_height, int channels, uint8_t* image_data);
```

由于它的实现是特定于平台的，因此在 *person_detection/image_provider.cc* 中有一个参考实现，返回无效的数据。

image_provider_test.cc 中的测试调用此参考实现以显示它的用法。我们的首要任务是创建一个数组来保存图像数据如下所示：

```
uint8_t image_data[kMaxImageSize];
```

常量 *kMaxImageSize* 定义在 *model_settings.h* 中。

设置完数组之后，我们可以调用 *GetImage()* 函数从摄像头捕获图像：

```
TfLiteStatus get_status =  
    GetImage(error_reporter, kNumCols, kNumRows, kNumChannels, image_data);  
TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, get_status);  
TF_LITE_MICRO_EXPECT_NE(image_data, nullptr);
```

我们使用 *ErrorReporter* 实例，我们想要的列、行和通道的数量，以及指向 *image_data* 数组的指针来调用 *GetImage()* 函数。该函数会将图像数据写入我们的数组。我们可以检查函数的返回值以确定捕获过程是否成功：如果有问题，返回值将为 *kTfLiteError*，否则返回 *kTfLiteOk*。

最后，测试遍历返回的数据，以证明所有内存位置都是可读的。虽然从技术上讲，图像具有行、列和通道，但实际上图像数据会被展平为一维数组：

```
uint32_t total = 0;  
for (int i = 0; i < kMaxImageSize; ++i) {  
    total += image_data[i];  
}
```

使用以下命令运行此测试：

```
make -f tensorflow/lite/micro/tools/make/Makefile \  
test_image_provider_test
```

在后面的章节中我们会仔细研究 *image_provider.cc* 中特定于平台的实现。但是现在，让我们先来看一下检测响应程序的接口。

9.3.3 检测响应程序

我们最后的测试展示了如何使用检测响应程序。这是负责传递推断结果的代码。它的接

口定义在 *detection_responder.h* 中，测试位于 *detection_responder_test.cc*。

它的接口非常简单：

```
void RespondToDetection(tflite::ErrorReporter* error_reporter,
                        uint8_t person_score, uint8_t no_person_score);
```

我们调用 `RespondToDetection()` 并传入“行人”和“非行人”类别的得分，它将通过得分来决定具体做什么。

detection_responder.cc 中的参考实现只是打印了这些值。*detection_responder_test.cc* 中的测试尝试使用不同的值调用函数：

```
RespondToDetection(error_reporter, 100, 200);
RespondToDetection(error_reporter, 200, 100);
```

使用以下命令运行测试并查看输出：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
      test_detection_responder_test
```

我们已经探索了所有的测试和它们所使用的接口。现在让我们来看看程序本身。

9.4 行人检测

应用程序的核心代码位于 *main_functions.cc*。它们短小精悍，我们在测试中已经看到了它们的很多逻辑。

首先，我们把模型需要的所有算子都放进来：

```
namespace tflite {
namespace ops {
namespace micro {
TfLiteRegistration* Register_DEPTHWISE_CONV_2D();
TfLiteRegistration* Register_CONV_2D();
TfLiteRegistration* Register_AVERAGE_POOL_2D();
} // namespace micro
} // namespace ops
} // namespace tflite
```

接下来，我们声明一组变量来保存重要的部分：

```
tflite::ErrorReporter* g_error_reporter = nullptr;
const tflite::Model* g_model = nullptr;
tflite::MicroInterpreter* g_interpreter = nullptr;
TfLiteTensor* g_input = nullptr;
```

之后，我们为张量运算分配一些工作内存：

```
constexpr int g_tensor_arena_size = 70 * 1024;
static uint8_t tensor_arena[kTensorArenaSize];
```

`setup()` 函数运行在其他任何操作发生之前。在 `setup()` 函数中，我们创建一个错误报告程序，加载我们的模型，设置一个解释器实例，并获取对模型的输入张量的引用：

```
void setup() {
    // Set up logging.
    static tflite::MicroErrorReporter micro_error_reporter;
    g_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.
    g_model = tflite::GetModel(g_person_detect_model_data);
    if (g_model->version() != TFLITE_SCHEMA_VERSION) {
        g_error_reporter->Report(
            "Model provided is schema version %d not equal "
            "to supported version %d.",
            g_model->version(), TFLITE_SCHEMA_VERSION);
        return;
    }

    // Pull in only the operation implementations we need.
    static tflite::MicroMutableOpResolver micro_mutable_op_resolver;
    micro_mutable_op_resolver.AddBuiltin(
        tflite::BuiltinOperator_DEPTHWISE_CONV_2D,
        tflite::ops::micro::Register_DEPTHWISE_CONV_2D());
    micro_mutable_op_resolver.AddBuiltin(tflite::BuiltinOperator_CONV_2D,
                                         tflite::ops::micro::Register_CONV_2D());
    micro_mutable_op_resolver.AddBuiltin(
        tflite::BuiltinOperator_AVERAGE_POOL_2D,
        tflite::ops::micro::Register_AVERAGE_POOL_2D());

    // Build an interpreter to run the model with.
    static tflite::MicroInterpreter static_interpreter(
        model, micro_mutable_op_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;
    }

    // Get information about the memory area to use for the model's input.
    input = interpreter->input(0);
}
```

代码的下一部分在程序的主循环中被不断地调用。它首先使用图像提供程序获取图像，将一个引用传递给输入张量，用来写入图像：

```
void loop() {
    // Get image from provider.
    if (kTfLiteOk != GetImage(g_error_reporter, kNumCols, kNumRows, kNumChannels,
        g_input->data.uint8)) {
        g_error_reporter->Report("Image capture failed.");
    }
}
```

然后进行推断，获得输出张量，并从中读取“行人”和“非行人”的得分。这些得分被传递到检测响应程序的 `RespondToDetection()` 函数中：

```
// Run the model on this input and make sure it succeeds.
if (kTfLiteOk != g_interpreter->Invoke()) {
    g_error_reporter->Report("Invoke failed.");
}

TfLiteTensor* output = g_interpreter->output(0);

// Process the inference results.
uint8_t person_score = output->data.uint8[kPersonIndex];
uint8_t no_person_score = output->data.uint8[kNotAPersonIndex];
RespondToDetection(g_error_reporter, person_score, no_person_score);
}
```

在 `RespondToDetection()` 完成输出结果后，`loop()` 函数将返回，准备再次被程序的主循环调用。

循环本身是在 `main.cc` 中程序的 `main()` 函数中定义的。它调用一次 `setup()` 函数，然后重复不停地调用 `loop()` 函数：

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

这就是整个程序！这个例子很不错，因为它很好地展示了使用复杂的机器学习模型可以很简单。模型隐含了所有的复杂性，而我们要做的只是把数据提供给它。

在我们继续之前，你可以尝试一下在本地运行程序。图像提供程序的参考实现只返回虚拟数据，因此你不会得到有意义的识别结果，但至少可以看到代码在工作。

首先，使用这个命令来编译程序：

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

一旦编译完成，你就可以使用以下命令运行示例：

```
tensorflow/lite/micro/tools/make/gen/osx_x86_64/bin/ \
person_detection
```

你将看到程序的滚动输出，直到按 Ctrl+C 使它终止：

```
person score:129 no person score 202
```

在下一节中，我们将遍历特定于设备的代码，这些代码将捕获摄像头图像并在每个平台上输出结果。我们还将展示如何部署和运行代码。

9.5 部署到微处理器

在本节中，我们将会把代码部署到两个熟悉的设备上：

- Arduino Nano 33 BLE Sense
- SparkFun Edge

这一次有一个很大的不同：因为这两款设备都没有内置摄像头，所以我们建议你为正在使用的任何设备购买一个摄像头模块。每个设备都有自己的 *image_provider.cc* 实现，它控制摄像头模块以捕获图像。*detection_response.cc* 中还有特定于设备的输出代码。

这样的组织很直观而且很简单，你可以把它当作模板，创建你自己的视觉相关的机器学习应用程序。

让我们先来探索 Arduino 的实现。

9.5.1 Ardunio

作为拥有完整生态系统的 Arduino 开发板，Arduino Nano 33 BLE Sense 可以访问和使用所有兼容 Andunio 的第三方硬件和库。我们使用的是一个第三方摄像头模块，它是专门针对 Arduino 设计的，同时还有两个 Arduino 库，使我们可以连接摄像头模块，并理解它输出的数据。

购买哪种摄像头模块

本例中我们使用了 Arducam Mini 2MP Plus 摄像头模块。它与 Arduino Nano 33 BLE Sense 的交互非常简单，可以由 Arduino 开发板的电源供电。它有一个大的镜头，能够捕捉 200 万像素的高质量图像——尽管我们将使用它自带的图像缩放功能来获得更小的分辨率。虽然它不是特别省电，但它捕获高质量图像的能力仍使得它成为很多应用程序的理想选择，比如记录野生动物。

在 Arduino 上拍摄图像

我们通过多个引脚将 Arducam 模块连接到 Arduino 开发板上。为了获取图像数据，我们通过 Arduino 开发板向 Arducam 发送命令，指示它捕获图像。Arducam 将会先把图像存储在内部数据缓冲区中，然后我们发送进一步的命令，从 Arducam 的内部缓冲区读取图像数据，并将数据存储在 Arduino 的内存中。为了完成所有这些操作，我们使用官方的 Arducam 库。

Arducam 摄像头模块有一个 200 万像素的图像传感器，分辨率为 1920×1080 。我们的行人检测模型的输入大小只有 $96 \text{ 像素} \times 96 \text{ 像素}$ ，所以我们并不需要摄像头捕获的所有数据。事实上，Arduino 本身也没有足够的内存来保存 200 万像素的图像，一张 200 万像素的图像大小大约为几兆字节。

幸运的是，Arducam 硬件能够将输出调整到更小的分辨率，即 $160 \text{ 像素} \times 120 \text{ 像素}$ 。在我们的代码中，我们可以很容易地将其裁剪为 96×96 的大小，只保留中心的 $96 \text{ 像素} \times 96 \text{ 像素}$ 。然而，更复杂的是，Arducam 调整大小后的输出使用 JPEG 编码，这是一种常见的图像压缩格式。我们的模型需要一个像素数组，而不是 JPEG 编码的图像，这意味着我们需要在使用之前解码 Arducam 的输出。我们可以使用一个开源库来实现这一点。

我们最后的任务是将 Arducam 输出的彩色图像转换成灰度图，因为行人检测模型期望的是灰度输入。我们将把灰度数据写到模型的输入张量中。

图像提供程序在 *arduino/image_provider.cc* 中实现。我们不会解释它的每一个细节，因为代码是特定于 Arducam 摄像头模块的。相反，让我们从更高的角度来逐步了解发生了什么。

`GetImage()` 函数是图像提供程序与外部世界的接口。它在应用程序的主循环中被调用，以获取图像数据的帧。第一次调用它时，我们需要先初始化摄像头。这是通过调用 `InitCamera()` 函数实现的，如下所示：

```
static bool g_is_camera_initialized = false;
if (!g_is_camera_initialized) {
    TfLiteStatus init_status = InitCamera(error_reporter);
    if (init_status != kTfLiteOk) {
        error_reporter->Report("InitCamera failed");
        return init_status;
    }
    g_is_camera_initialized = true;
}
```

`InitCamera()` 函数是在 *image_provider.cc* 中进一步定义的。我们不会在这里详细介绍它，因为它是特定于设备的，如果你想在自己的代码中使用它，你可以复制粘贴它。

它将设置 Arduino 的硬件与 Arducam 通信，然后确认通信正常，最后指示 Arducam 输出 160 像素 × 120 像素的 JPEG 图像。

`GetImage()` 调用的下一个函数是 `PerformCapture()`：

```
TfLiteStatus capture_status = PerformCapture(error_reporter);
```

我们也不会讨论这里的细节。它所做的就是向摄像头模块发送一个命令，指示摄像头捕获图像并将图像数据存储在内部缓冲区中，然后等待图像确认被捕获。此时，会有图像数据在 Arducam 的内部缓冲区中等待，但是 Arduino 本身还没有任何图像数据。

我们调用的下一个函数是 `ReadData()`：

```
TfLiteStatus read_data_status = ReadData(error_reporter);
```

`ReadData()` 函数通过更多的命令从 Arducam 获取图像数据。函数运行后，全局变量 `jpeg_buffer` 将被从摄像机捕获到的 JPEG 编码的图像数据填充。

有了 JPEG 编码的图像后，下一步是将其解码为原始图像数据。这些操作定义在 `DecodeAndProcessImage()` 函数中：

```
TfLiteStatus decode_status = DecodeAndProcessImage(
    error_reporter, image_width, image_height, image_data);
```

此函数使用一个名为 `JPEGDecoder` 的库来解码 JPEG 数据，并将解码后的数据直接写入模型的输入张量。在这个过程中，它会裁剪图像，丢弃 160×120 的数据，使用余下的大致在图像中心位置的 96 像素 × 96 像素的部分。它还将图像的 16 位彩色表示降低为 8 位灰度表示。

在图像被捕获并存储在输入张量中之后，我们就可以运行推断了。接下来，我们将展示如何显示模型的输出。

响应 Arduino 的检测结果

Arduino Nano 33 BLE Sense 有一个内置的 RGB LED，它是一个单独的组件，包含不同的红色、绿色和蓝色的 LED，可以分别进行控制。

每次运行推断时，检测响应程序的实现都会闪烁蓝色 LED。当检测到有行人时，它会点亮绿色的 LED；当检测到非行人时，它会点亮红色的 LED。

检测响应程序的实现代码位于 `arduino/image_provider.cc`。让我们来快速地浏览一下。

`RespondToDetection()` 函数接受两个得分，一个代表“行人”类别，另一个代表“非行人”类别。第一次调用时，它会设置蓝色、绿色和黄色的 LED 作为输出：

```
void RespondToDetection(tfLite::ErrorReporter* error_reporter,
                        uint8_t person_score, uint8_t no_person_score) {
    static bool is_initialized = false;
    if (!is_initialized) {
        pinMode(led_green, OUTPUT);
        pinMode(led_blue, OUTPUT);
        is_initialized = true;
    }
}
```

接下来，为了表示推断刚刚完成，我们将关闭所有的 LED 然后快速地闪烁蓝色 LED：

```
// Note: The RGB LEDs on the Arduino Nano 33 BLE
// Sense are on when the pin is LOW, off when HIGH.

// Switch the person/not person LEDs off
digitalWrite(led_green, HIGH);
digitalWrite(led_red, HIGH);

// Flash the blue LED after every inference.
digitalWrite(led_blue, LOW);
delay(100);
digitalWrite(led_blue, HIGH);
```

你会注意到，与 Arduino 的内置 LED 不同，这些 LED 是低开高关（on with LOW and off with HIGH）的。这其实仅仅取决于 LED 是如何连接到电路板的。

接下来，我们根据哪一类得分较高来打开和关闭相应的 LED：

```
// Switch on the green LED when a person is detected,
// the red when no person is detected
if (person_score > no_person_score) {
    digitalWrite(led_green, LOW);
    digitalWrite(led_red, HIGH);
} else {
    digitalWrite(led_green, HIGH);
    digitalWrite(led_red, LOW);
}
```

最后，我们使用 `error_reporter` 实例将得分输出到串口：

```
error_reporter->Report("Person score: %d No person score: %d",
                        person_score,
                        no_person_score);
}
```

就是这些！函数的核心是一个基本的 if 语句，你可以很轻松地使用类似的逻辑来控制其他类型的输出。将如此复杂的视觉输入转换为单一的布尔输出：“行人”或者“非行人”是一件非常令人兴奋的事情。

运行示例

运行当前的示例比运行其他 Arduino 示例要稍微复杂一些，因为我们需要将 Arducam 连

接到 Arduino 开发板。我们还需要安装和配置 Arducam 的接口，并解码其 JPEG 输出。不过别担心，总体来说还是很简单的！

要部署当前的示例，我们需要：

- Arduino Nano 33 BLE Sense 开发板。
- Arducam Mini 2MP Plus。
- 跨接电缆（可选面包板）。
- micro-USB 线。
- Arduino IDE。

我们的第一个任务是使用跨接电缆将 Arducam 连接到 Arduino 开发板。这不是一本电子方面的书籍，所以我们不会讨论使用电缆的细节。相反，表 9-1 展示了引脚的连接方式。每个设备上都贴有引脚标签。

表 9-1：Arducam Mini 2MP plus 到 Arduino Nano 33 BLE Sense 开发板的连接

Arducam 引脚	Arduino 引脚
CS	D7(未标示，紧邻 D6 右侧)
MOSI	D11
MISO	D12
SCK	D13
GND	GND (任何一个标为 GND 的引脚都可以)
VCC	3.3V
SDA	A4
SCL	A5

设置好硬件之后，就可以继续安装软件了。



与撰写本书时相比，编译过程有可能会发生变化，因此请查看 *README.md* 以获取最新说明。

本书中的项目可以作为 TensorFlow Lite Arduino 库中的示例代码获得。如果你还没有安装此库，可以打开 Arduino IDE 并从“Tools”（工具）菜单中选择“Manage Libraries”（管理库）。在出现的窗口中，搜索并安装名为 *Arduino_TensorFlowLite* 的库。你应该可以使用最新的版本，但是如果你遇到问题，也可以尝试本书使用的版本，即 1.14-ALPHA。



你还可以从 TensorFlow Lite 团队网站下载 Arduino_TensorFlowLite 库，然后通过 `.zip` 文件安装，也可以使用 TensorFlow Lite for Microcontrollers Makefile 自行生成。如果你想这样做，请参阅附录 A。

安装好 Arduino_TensorFlowLite 库之后，`person_detection` 示例将出现在“Examples”（示例）→“Arduino_TensorFlowLite”下的菜单中，如图 9-2 所示。



图 9-2：“Examples”菜单

单击“`person_detection`”以加载示例。它将显示为一个新的窗口，每个源文件都有一个选项卡。第一个选项卡中的 `person_detection` 文件相当于我们的 `main_functions.cc`。我们在之前已经介绍了它的内容。



6.2.2 节已经解释了 Arduino 示例的结构，所以这里不再赘述。

除了 TensorFlow 库，我们还需要安装另外两个库：

- Arducam 库，使得我们的代码可以与硬件连接。
- JPEGDecoder 库，使得我们可以解码 JPEG 编码的图像。

Arducam Arduino 库可以从 GitHub 上获取。要安装它，请下载或克隆代码仓库。接下来，将它的 *ArduCAM* 子目录复制到 *Arduino/libraries* 目录中。要在你的机器上找到 *libraries* 目录，请检查 Arduino IDE 的“Preferences”（选项）窗口中的 Sketchbook 位置。

下载库后，你需要编辑其中一个文件，以确保它是为 Arducam Mini 2MP Plus 配置的。打开 *Arduino/libraries/ArduCAM/memorysaver.h*。你应该会看到一堆 `#define` 语句。请确保注释掉所有除 `#define OV2640_MINI_2MP_PLUS` 之外的 `#define` 语句，如下所示：

```
//Step 1: select the hardware platform, only one at a time
//#define OV2640_MINI_2MP
//#define OV3640_MINI_3MP
//#define OV5642_MINI_5MP
//#define OV5642_MINI_5MP_BIT_ROTATION_FIXED
#define OV2640_MINI_2MP_PLUS
//#define OV5642_MINI_5MP_PLUS
//#define OV5640_MINI_5MP_PLUS
```

保存文件后，就完成了 Arducam 库的配置。



我们的示例是使用 Arducam 库的 #e216049 提交开发的。如果在库中遇到问题，可以尝试下载这个特定的提交，以确保使用完全相同的代码。

下一步是安装 JPEGDecoder 库。你可以在 Arduino IDE 中完成这项工作。在“Tools”（工具）菜单中，选择“Manage Libraries”（管理库）选项并搜索 JPEGDecoder。你应该安装该库的 1.8.0 版本。

库安装完成之后，你需要对其进行配置，以禁用一些与 Arduino Nano 33 BLE Sense 不兼容的可选组件。打开 *Arduino/libraries/JPEGDecoder/src/User_Config.h*，确保 `#define LOAD_SD_LIBRARY` 和 `#define LOAD_SD_FAT_LIBRARY` 都被注释掉，如下所示：

```
// Comment out the next #defines if you are not using an SD Card to store
// the JPEGs
// Commenting out the line is NOT essential but will save some FLASH space if
// SD Card access is not needed. Note: use of SdFat is currently untested!

#ifndef LOAD_SD_LIBRARY // Default SD Card library
#define LOAD_SD_FAT_LIBRARY // Use SdFat library instead, so SD Card SPI can
// be bit bashed
```

保存文件之后，就完成了库的安装。现在可以运行行人检测应用程序了！

首先，通过 USB 连接 Arduino 设备。确保从“Tools”（工具）菜单中的“Boards”（开发板）下拉列表中选择了正确的设备类型，如图 9-3 所示。

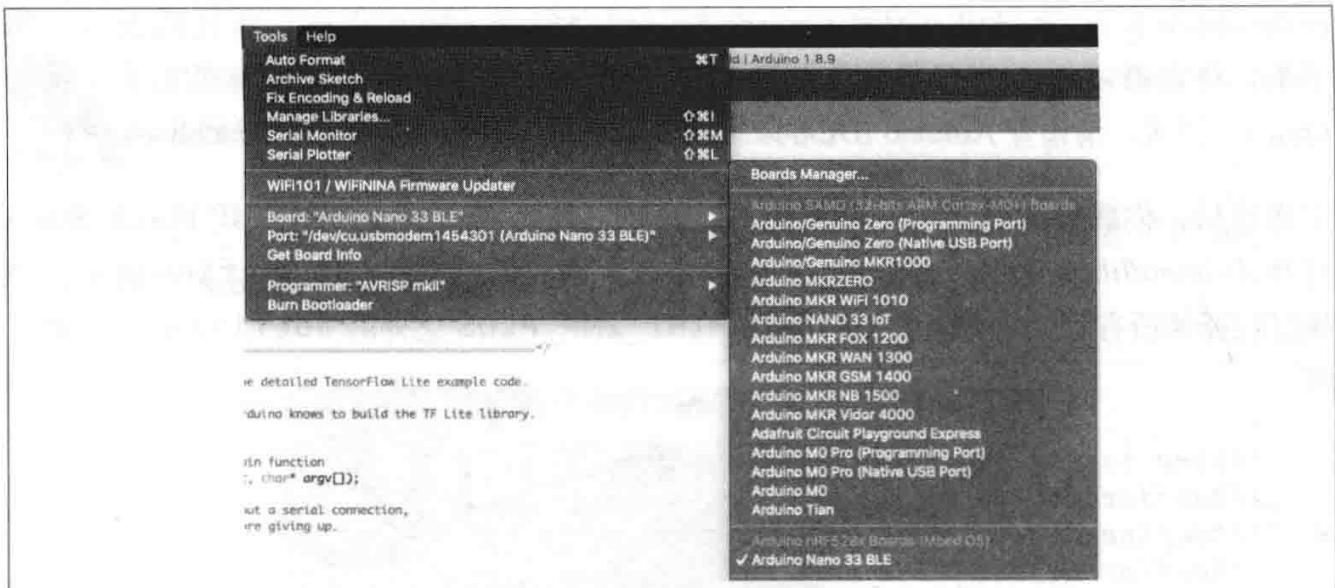


图 9-3：“Boards”下拉列表

如果你的设备的名字没有出现在列表中，你需要安装它的支持包。要做到这一点，需要单击“Boards Manager”（开发板管理器）。在出现的窗口中，搜索你的设备并安装相应支持包的最新版本。

同样，在“Tools”菜单中，确保在“Port”下拉列表中选择了设备的端口，如图 9-4 所示。

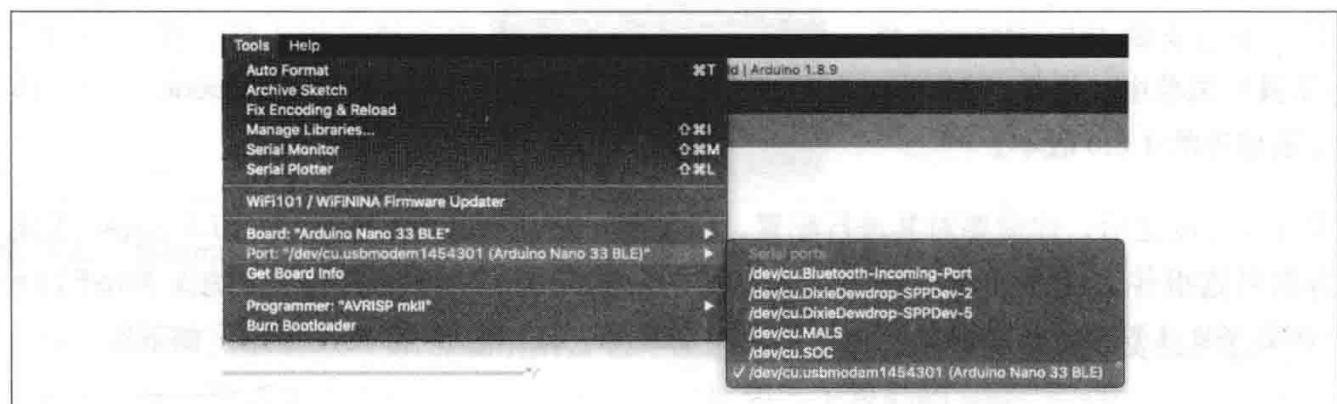


图 9-4：“Port”下拉列表

最后，在 Arduino 窗口中，点击上传（upload）按钮（图 9-5 中用白色高亮显示的按钮），将代码编译并上传到你的 Arduino 设备上。

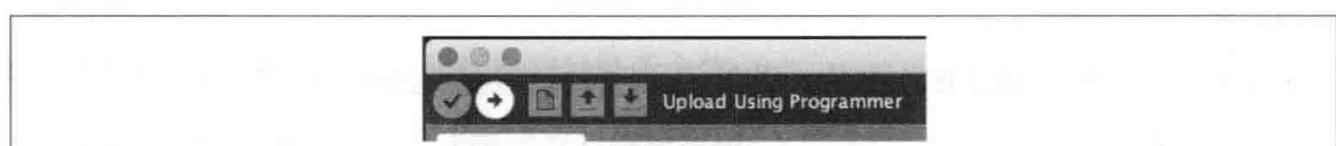


图 9-5：上传按钮

一旦上传成功，程序就会开始运行。

要测试它，首先要把设备的摄像头对准并非行人的任意物品，或者直接盖住镜头。当下一次蓝色 LED 闪烁时，设备将从摄像头捕获一帧图像并开始运行推断。由于我们的行人检测视觉模型相对比较大，所以可能会需要很长时间进行推断——撰写本书时尝试运行大约需要 19 秒，不过较之本书撰写的时间，现在的 TensorFlow Lite 可能已经变得更快了。

当推断完成时，结果将被转换成点亮另一个 LED。因为你将摄像机指向非行人的其他物品，所以应该看到红色的 LED 被点亮。

现在，试着把设备的摄像头对准你自己！当下一次蓝色 LED 闪烁时，设备将捕获另一张图像并开始运行推断。大约 19 秒后，绿色的 LED 灯就会被点亮。

请记住，图像数据会在每次推断之前被捕获为快照，也就是在每次蓝色 LED 闪烁的时候。无论摄像头在那个时刻面向什么，它都会被输入到模型中。在下一次捕捉到图像之前，也就是下次蓝色的 LED 闪烁之前，摄像头面向哪里并不重要。

如果你得到了看起来不太正确的结果，请首先确保你所处的环境光线良好。你还应该确保摄像头的方向是正确的，即引脚是向下的，这样它捕捉到的图像才是正确向上的——模型并没有被训练来识别颠倒的人。此外，最好始终记住这是一个很小的模型，它以准确率为代价换取了较小的模型尺寸。虽然它运行得很好，但并不能保证 100% 准确。

你还可以通过 Arduino 串口监视器查看推断结果。要执行此操作，从“Tools”菜单中打开“Serial Monitor”（串口监视器）。你将看到详细的日志，它显示应用程序运行时发生的事情。可以勾选“Show timestamp”（显示时间戳），这样你就可以看到每个部分需要多长时间：

```
14:17:50.714 -> Starting capture
14:17:50.714 -> Image captured
14:17:50.784 -> Reading 3080 bytes from ArduCAM
14:17:50.887 -> Finished reading
14:17:50.887 -> Decoding JPEG and converting to greyscale
14:17:51.074 -> Image decoded and processed
14:18:09.710 -> Person score: 246 No person score: 66
```

从这份日志中我们可以看到，从摄像头模块捕获和读取图像数据大约花费了 170 毫秒，解码 JPEG 并将其转换为灰度图需要 180 毫秒，而运行推断需要 18.6 秒。

尝试修改

现在你已经部署了基本的应用程序，可以试着修改一下代码。只需在 Arduino IDE 中编辑文件并保存，然后重复前面的说明，将修改后的代码部署到设备上。

以下是一些你可以尝试的事情：

- 修改检测响应程序，使其忽略模糊的输入，也就是“行人”和“非行人”得分之间没有太多差异的输入。
- 使用行人检测的结果来控制其他组件，如额外的 LED 或者伺服装置。
- 通过存储或者传输图像——仅限于包含行人的图像——来搭建智能安全摄像头。

9.5.2 SparkFun Edge

SparkFun Edge 开发板是以低功耗为优化目标而设计的开发板。当它与同样高效的摄像头模块搭配使用时，它会成为构建以电池为动力的视觉应用程序的理想平台。通过开发板的带状电缆适配器，可以很容易插入摄像头模块。

购买哪种摄像头模块

我们示例中使用的是 SparkFun 的 Himax HM01B0 breakout 摄像头模块。它基于一个 320 像素 × 320 像素的图像传感器，耗电量非常小：当以每秒 30 帧（30 FPS）的速度捕捉图像时，耗电量小于 2 毫瓦。

使用 SparkFun Edge 捕捉图像

要开始使用 Himax HM01B0 摄像头模块捕获图像，首先必须初始化摄像头。完成这一步后，每当需要新图像时，我们就可以从摄像头中读取帧（frame）。帧是一个字节数组，表示摄像头当前可以看到的内容。

使用摄像头需要大量使用 Ambiq Apollo3 SDK（会作为编译过程的一部分下载）和 HM01B0 驱动程序，后者位于 `sparkfun_edge/himax_driver`。

图像提供程序在 `sparkfun_edge/image_provider.cc` 中实现。我们不会解释它的每一个细节，因为代码是特定于 SparkFun 开发板和 Himax 摄像头模块的。相反，让我们从更高的角度来逐步了解程序中发生了什么。

`GetImage()` 函数是图像提供程序与外部世界的接口。它在应用程序的主循环中被调用，以获取图像数据的帧。在第一次调用它时，我们需要初始化摄像头。这是通过调用 `InitCamera()` 函数实现的，如下所示：

```
// Capture single frame. Frame pointer passed in to reduce memory usage. This
// allows the input tensor to be used instead of requiring an extra copy.
TfLiteStatus GetImage(tflite::ErrorReporter* error_reporter, int frame_width,
                      int frame_height, int channels, uint8_t* frame) {
    if (!g_is_camera_initialized) {
        TfLiteStatus init_status = InitCamera(error_reporter);
```

```
if (init_status != kTfLiteOk) {
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_RED);
    return init_status;
}
```

如果 `InitCamera()` 返回的不是 `kTfLiteOk`，我们就点亮开发板上的红色 LED（通过 `am_hal_gpio_output_set(AM_BSP_GPIO_LED_RED)` 实现）来表示初始化出现了问题，这会有助于我们调试程序。

`InitCamera()` 函数是在 `image_provider.cc` 中实现的。我们不会在这里详细介绍它，因为它是特定于设备的，如果你想在自己的代码中使用它，可以直接复制粘贴代码。

它调用 Apollo3 SDK 中的一系列函数来配置微控制器的输入和输出，以便能与摄像头模块通信。它还支持中断（interrupt），这是摄像头用来发送新图像数据的机制。当这一切都设置好后，它使用摄像头驱动程序来打开摄像头，并配置它来开始不断地捕捉图像。

摄像头模块有一个自动曝光（autoexposure）功能，当图像被捕获时会自动校准曝光设置。为了让它有机会在我们执行推断前校准，`GetImage()` 函数的下一部分使用摄像头驱动程序的 `hm01bo_blocking_read_oneframe_scaled()` 函数来捕获多个帧。我们对捕获的数据不做任何处理，这样做只是为了给摄像头模块的自动曝光功能提供一些可以用于校准的数据：

```
// Drop a few frames until auto exposure is calibrated.
for (int i = 0; i < kFramesToInitialize; ++i) {
    hm01bo_blocking_read_oneframe_scaled(frame, frame_width, frame_height,
                                             channels);
}
g_is_camera_initialized = true;
}
```

设置完成之后，`GetImage()` 函数的其余部分就非常简单了。它所做的就是调用 `hm01bo_blocking_read_oneframe_scale()` 来捕获图像：

```
hm01bo_blocking_read_oneframe_scaled(frame, frame_width, frame_height,
                                         channels);
```

在应用程序的主循环中调用 `GetImage()` 时，`frame` 变量是指向输入张量的指针，因此数据由摄像头驱动程序直接写入分配给输入张量的内存区域。我们还会指定所需的通道的宽度、高度和数量。通过这部分实现，我们就能够从摄像头模块捕获图像数据。

接下来，让我们看看如何响应模型的输出。

响应 SparkFun Edge 的检测结果

检测响应程序的实现与我们的唤醒词检测示例的命令响应程序非常相似。每次运行推断

时，它都会切换设备的蓝色 LED。当检测到行人时，它会点亮绿色 LED；当检测到非行人时，它会点亮黄色 LED。

具体实现位于 `sparkfun_edge/image_provider.cc`，让我们来快速浏览一下。

`RespondToDetection()` 函数接受两个得分，一个代表“行人”类别，另一个代表“非行人”类别。第一次调用时，它会设置蓝色、绿色和黄色的 LED 作为输出：

```
void RespondToDetection(tflite::ErrorReporter* error_reporter,
                        uint8_t person_score, uint8_t no_person_score) {
    static bool is_initialized = false;
    if (!is_initialized) {
        // Setup LED's as outputs. Leave red LED alone since that's an error
        // indicator for sparkfun_edge in image_provider.
        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);
        is_initialized = true;
    }
}
```

因为每次推断都会调用 `RespondToDection()` 函数，所以下面一段代码会使它在每次执行推断时打开和关闭蓝色 LED：

```
// Toggle the blue LED every time an inference is performed.
static int count = 0;
if (++count & 1) {
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_BLUE);
} else {
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_BLUE);
}
```

最后，如果检测到有行人，它会点亮绿色的 LED，如果不是行人，它会点亮黄色的 LED。它还使用 `ErrorReporter` 实例记录得分：

```
am_hal_gpio_output_clear(AM_BSP_GPIO_LED_YELLOW);
am_hal_gpio_output_clear(AM_BSP_GPIO_LED_GREEN);
if (person_score > no_person_score) {
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_GREEN);
} else {
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_YELLOW);
}

error_reporter->Report("person score:%d no person score %d", person_score,
                       no_person_score);
```

就是这样！函数的核心是一个基本的 `if` 语句，你可以很轻松地使用类似的逻辑来控制其他类型的输出。将如此复杂的视觉输入转换为单一的布尔输出（“行人”或“非行人”）是一件非常令人兴奋的事情。

运行示例

既然我们已经了解了 SparkFun Edge 实现的工作原理，让我们开始运行它。



与撰写本书时相比，编译过程有可能会发生变化，因此请查看 *README.md* 以获取最新说明。

要部署当前的示例，我们需要：

- SparkFun Edge 开发板与 Himax HM01B0 breakout 附件。
- USB 编程器（我们推荐 SparkFun Serial Basic Breakout，它有 micro-B USB 和 USB-C 两种版本）。
- 匹配的 USB 线。
- Python 3 和一些依赖项。



如果你不确定是否安装了正确的 Python 版本，请查看 6.3.2 节中有关于检查 Python 版本的说明。

在终端中，克隆 TensorFlow 代码仓库，并将其切换到它的目录下：

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

接下来，我们将编译二进制文件并运行一些命令，以便使其可以被下载到设备上。为了避免一些键入操作，你可以从 *README.md* 复制和粘贴这些命令。

编译二进制文件。以下命令会下载所有需要的依赖项，然后编译 SparkFun Edge 的二进制文件：

```
make -f tensorflow/lite/micro/tools/make/Makefile \  
TARGET=sparkfun_edge person_detection_bin
```

二进制文件被创建为 .bin 文件，位于：

```
tensorflow/lite/micro/tools/make/gen/  
sparkfun_edge_cortex-m4/bin/person_detection.bin
```

可以使用以下命令检查文件是否存在：

```
test -f tensorflow/lite/micro/tools/make/gen \  
/sparkfun_edge_cortex-m4/bin/person_detection.bin \  
&& echo "Binary was successfully created" || echo "Binary is missing"
```

运行该命令时，你应该看到“Binary was successfully created”被打印到控制台。

如果你看到“Binary is missing”，则说明编译过程存在问题。如果是这样，那么 make 命令的输出可能会提供一些错误的线索。

为二进制文件签名。必须在使用密钥对二进制文件签名后才能将其部署到设备上。现在让我们运行一些命令来对二进制文件进行签名，以便能够将其烧录到 SparkFun Edge 上。这里使用的脚本来自 Ambiq SDK，它在运行 Makefile 时下载。

输入以下命令来设置一些可以用于开发的虚拟密钥：

```
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
```

接下来，运行以下命令来创建签名的二进制文件。如有需要，请用 python 替代 python3：

```
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/person_detection.bin \
    --load-address 0xC000 \
    --magic-num 0xCB \
    -o main_nonsecure_ota \
    --version 0x0
```

它将创建 *main_nonsecure_ota.bin* 文件。现在运行此命令来创建文件的最终版本，你可以用将在下一个步骤中使用的脚本来烧录设备：

```
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
```

在运行命令的目录中应该有一个名为 *main_nonsecure_wire.bin* 的文件。这是你要烧录到设备上的文件。

烧录二进制文件。SparkFun Edge 将当前运行的程序存储在它 1MB 闪存中。如果你想让开发板运行一个新程序，你需要把程序发送到开发板，开发板会把接收到的程序存储在闪存，并覆盖之前保存的任何程序。

正如我们在前面提到的，这个过程称为烧录。

将编程器连接到开发板上。要将新的程序下载到开发板，你将使用 SparkFun USB-C Serial Basic 串口编程器。它将允许你的计算机通过 USB 与微控制器进行通信。

请执行以下操作，将编程器安装到你的开发板上：

1. 在 SparkFun Edge 的一侧找到六引脚接口。
2. 将 SparkFun USB-C Serial Basic 串口编程器插入这些引脚，确保每个设备上被标记为 BLK 和 GRN 的引脚正确对齐，如图 9-6 所示。

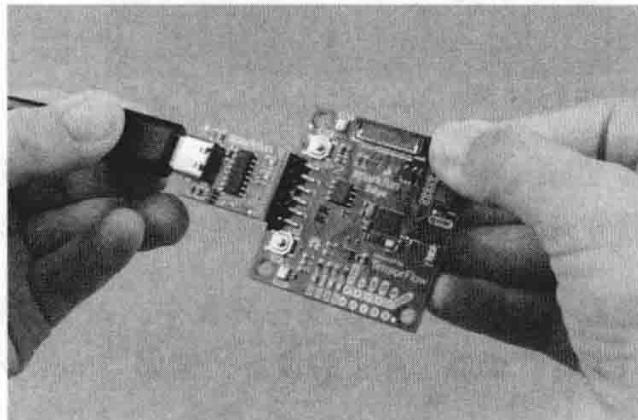


图 9-6：连接 SparkFun Edge 和 SparkFun USB-C Serial Basic

把编程器连接到你的计算机上。你可以通过 USB 接口把开发板连接到你的计算机上。要给开发板编程，你需要找出设备在计算机中的名称。最好的方法是列出连接编程器之前和之后计算机上所有的设备名称，然后看看哪个设备是新增的。



已经有人报告了他们在使用操作系统的默认驱动程序时遇到问题，所以我们强烈建议在继续之前安装驱动程序。

在通过 USB 连接设备之前，运行以下命令：

```
# macOS:  
ls /dev/cu*  
  
# Linux:  
ls /dev/tty*
```

运行后应该输出一个连接设备列表，类似于下面的输出：

```
/dev/cu.Bluetooth-Incoming-Port  
/dev/cu.MALS  
/dev/cu.SOC
```

现在，将编程器连接到你的计算机的 USB 端口，并再次运行以下命令：

```
# macOS:  
ls /dev/cu*  
  
# Linux:  
ls /dev/tty*
```

你应该在输出中看到一个额外的项，如下面的例子所示。在你的计算机上，新增的项可能有不同的名称。此例中设备的名称是：

```
/dev/cu.Bluetooth-Incoming-Port  
/dev/cu.MALS  
/dev/cu.SOC  
/dev/cu.wchusbserial-1450
```

此名称将用于引用设备。但是，它可能会随着编程器连接的 USB 端口的改变而改变，所以如果你从计算机上断开开发板，然后重新连接它，你可能需要再次查找它的名称。



一些用户报告说看到两个设备出现在列表中。如果你看到两个设备，正确的设备应该是以字母“wch”开头的设备，例如“/dev/wchusbserial-14410”。

确定设备名称后，将它存入 shell 变量以备后用：

```
export DEVICENAME=<your device name here>
```

在稍后运行需要设备名的命令时，可以使用这个变量。

运行脚本以烧录开发板。要烧录开发板，你需要将它置于特殊的 bootloader 状态，以使其准备好接收新的二进制文件。然后，你可以运行脚本以将二进制文件发送到开发板。

首先创建一个环境变量以指定波特率，波特率是将数据发送到设备的速度：

```
export BAUD_RATE=921600
```

现在，请将下面的命令粘贴到你的终端中，但是先不要按回车键！命令中的 \${DEVICENAME} 和 \${BAUD_RATE} 将被替换为你在之前设置的值。请记住，如有必要，请用 python 替换 python3：

```
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
```

接下来，将开发板重置为 bootloader 状态并烧录开发板。在开发板上，找到标记为 RST 和 14 的按键，如图 9-7 所示。

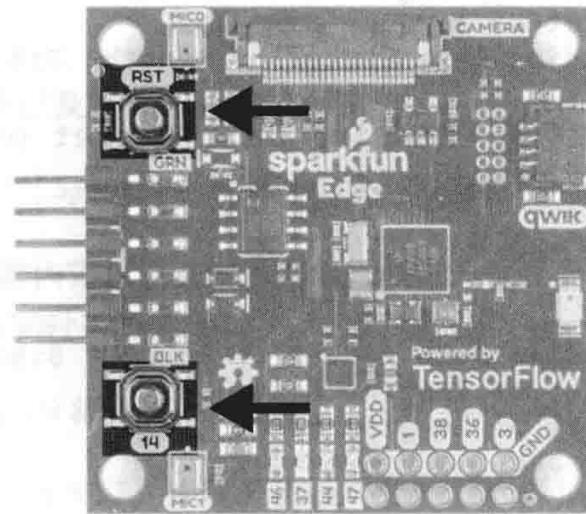


图 9-7: SparkFun Edge 上的按键

执行以下步骤：

1. 确保你的开发板已连接至编程器，且整个设备均已通过 USB 连接到你的计算机。
2. 在开发板上，按住按键 14，不要松手。
3. 在长按按键 14 的同时，按下按键 RST 以重置开发板。
4. 在计算机上按回车键以运行脚本。继续保持按住按键 14。

现在，你应该在屏幕上看到如下内容：

```
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
0xffff
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...]
```

一直按住按键 **14**，直到你看到“**Sending Data Packet of length 8180**”。看到此消息后可以松开按键（但如果你一直按着它也没关系）。

该程序将继续在终端上打印日志。最终，你会看到类似以下内容：

```
[...lots more Sending Data Packet of length 8180...]  
Sending Data Packet of length 8180  
Sending Data Packet of length 6440  
Sending Reset Command.  
Done.
```

这表示烧录成功。



如果程序输出以错误结尾，请检查“**Sending Reset Command.**”是否被打印出来。如果是这样，尽管出现了错误，但是最终烧录仍可能会成功；否则，烧录很可能会失败，你需要尝试再次执行这些步骤（可以跳过设置环境变量）。

测试程序

首先按下按键 **RST**，以确保程序正在运行。当程序运行时，每运行一次推断，蓝色的 LED 就会被点亮和关闭一次。由于我们使用的行人检测的视觉模型比较大，所以推断可能会需要很长时间——大约 6 秒。

首先，将设备的摄像头对准一些肯定不是行人的东西，或者遮挡住摄像头。当下一次蓝色 LED 闪烁时，设备将从摄像头捕捉一帧图像并开始运行推断。大约 6 秒后，推断结果将被转换成点亮另一个 LED。假设你将摄像头对准的东西不是行人，黄色的 LED 应该会被点亮。

现在，试着把设备的摄像头对准你自己。当下一次蓝色 LED 闪烁时，设备将捕获另一帧图像并开始运行推断。这时，应该会点亮绿色的 LED。

请记住，图像数据会在每次推断之前被捕获为快照，也就是每次蓝色 LED 闪烁的时候。无论摄像头在那个时刻面向了什么，它都会被输入模型。在下一次捕捉到图像之前，也就是下次蓝色 LED 闪烁之前，摄像头面向哪里并不重要。

如果你得到的结果看起来不太正确，请首先确保你所处的环境光线充足。还需要记住的是，这是一个很小的模型，它以准确率为代价换取了较小的模型尺寸。虽然它运行得很好，但并不是 100% 准确。

如果失败了怎么办

以下是一些可能的问题以及调试方法：

问题：烧录时，脚本在“*Sending Hello.*”的地方停顿一段时间。然后打印错误。

解决方案：运行脚本时，需要按住按键 **14**。先按住按键 **14**，然后按下按键 **RST**，在保持按住按键 **14** 的同时运行脚本。

问题：烧录后，所有 LED 都不亮。

解决方案：尝试按下按键 **RST**，或将开发板与编程器断开连接再重新连接。如果这些都不起作用，请尝试再次烧录开发板。

问题：烧录后，红色 LED 被点亮。

解决方案：红色 LED 被点亮表示摄像头模块有问题。确保摄像头模块连接正确，如果是这样，试着断开并重新连接。

查看调试数据

程序会将检测结果记录到串口。要查看它们，可以使用 115200 的波特率监视开发板的串口输出。在 macOS 和 Linux 上，可以运行以下命令：

```
screen ${DEVICENAME} 115200
```

刚开始你应该会看到类似如下的输出：

```
Apollo3 Burst Mode is Available
```

```
Apollo3 operating in Burst Mode (96MHz)
```

当捕获到图像帧并运行推断时，你应该看到如下调试信息被打印出来：

```
Person score: 130 No person score: 204
Person score: 220 No person score: 87
```

要在屏幕上停止查看调试输出，请按 **Ctrl+A**，紧接着按 **K** 键，然后按 **Y** 键。

尝试修改

现在你已经部署了基本的应用程序，可以试着修改一下代码。你可以在 *tensorflow/lite/micro/examples/person_detection* 文件夹中找到应用程序的代码。只需编辑并保存，然后重复前面的操作，将修改后的代码部署到设备上。

以下是一些你可以尝试的事情：

- 修改检测响应程序，使其忽略模糊的输入，也就是“行人”和“非行人”得分之间没有太多差异的输入。
- 使用行人检测的结果来控制其他组件，如额外的 LED 或者伺服装置。
- 通过存储或传输图像——仅限于包含行人的图像——来搭建智能安全摄像头。

9.6 小结

我们在本章中使用的视觉模型是很令人惊奇的。它接收原始而杂乱的输入，不需要任何预处理就可以给出一个漂亮而且简单的输出：有行人或者没有行人。这就是机器学习的魔力：它可以从噪声中过滤信息，只保留我们关心的信号。作为开发人员，你应该很容易利用这些简单的信号为用户创造令人惊叹的体验。

在创建机器学习应用程序时，我们常常会使用这种已经预先训练好的模型，这些模型已经包含了执行任务所需的知识。模型与代码库很相似，它封装了特定的功能，并且很容易实现在项目之间共享。你可能会经常发现自己在探索、评估或者寻找适合你的任务的模型。

在第 10 章中，我们将研究行人检测模型是如何工作的。你还将学习如何训练自己的视觉模型来识别不同类型的物体。

行人检测：训练模型

在第 9 章中，我们展示了如何部署一个预先训练好的模型来识别图像中的行人，但是我们并没有说明模型的来源。如果你的产品有不同的需求，你可能会希望能训练自己的模型，本章我们将介绍如何训练自己的模型。

10.1 选择机器

训练这个图像模型要使用比我们之前的示例多得多的算力（compute power），所以如果你希望训练可以在合理的时间内完成，那么你需要使用带有高端图形处理单元（GPU）的机器。除非你希望运行大量的训练工作，否则我们建议从租用云实例（而不是购买专用的机器）开始。不幸的是，我们在前几章中使用过的 Google 提供的 Colaboratory 服务只适用于较小的模型，你需要付费才能访问一台云实例。当然还有很多其他出色的服务提供商，但是我们将假定你正在使用 Google Cloud Platform，因为这是我们最熟悉的服务。如果你已经在使用 Amazon Web Services（AWS）或者 Microsoft Azure，它们也都支持 TensorFlow，而且训练步骤大致相同，但是你可能需要按照它们的教程进行设置。

10.2 配置 Google Cloud Platform 实例

你可以从 Google Cloud Platform 租用一台预装了 TensorFlow 和 NVIDIA 驱动程序并且支持 Jupyter Notebook Web 界面的虚拟机，这非常方便。但是设置这台机器的过程可能会麻烦一些。截至 2019 年 9 月，以下是创建机器所需的步骤：

1. 登录 console.cloud.google.com。如果你还没有 Google 账户，则需要创建一个，并且必须配置付款方式为创建的实例付费。如果你还没有项目，则需要创建一个新的项目。

2. 在屏幕的左上角，打开汉堡菜单（图标由三条水平线组成，如图 10-1 所示），然后向下滚动直到找到“Artificial Intelligence”（人工智能）部分。

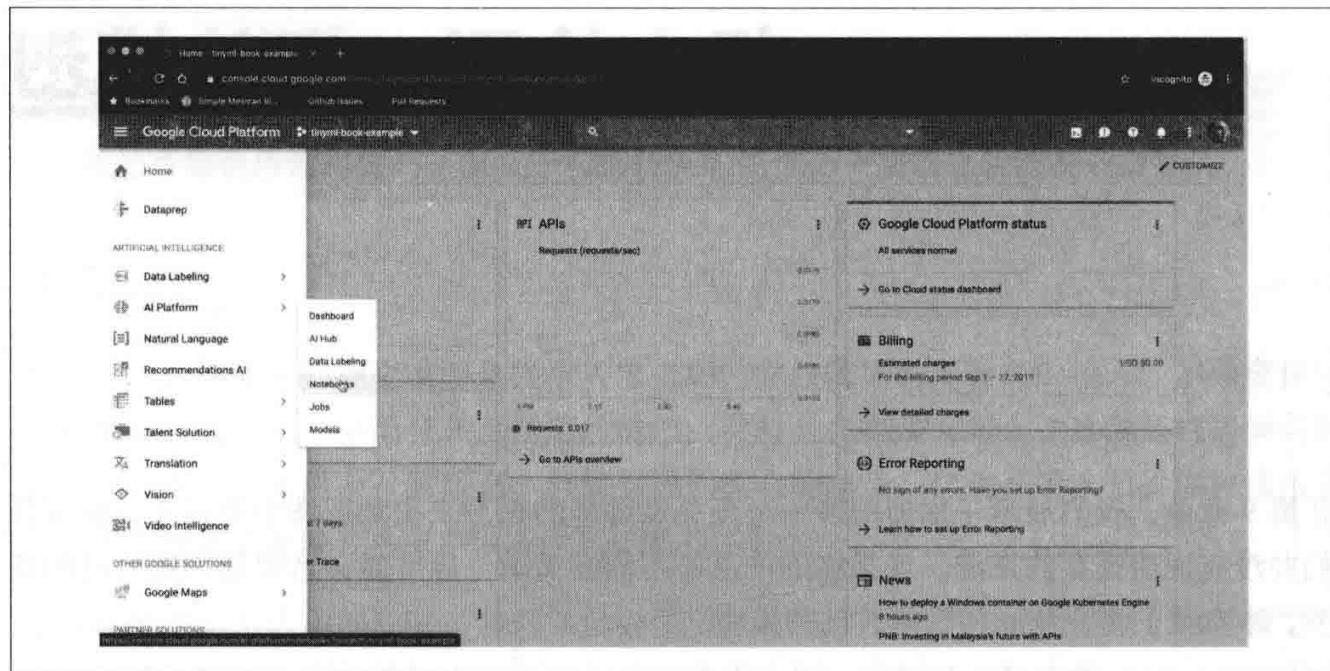


图 10-1：“AI Platform”菜单

3. 在该部分中，选择“AI Platform”（AI平台）→“Notebooks”（笔记本），如图 10-1 所示。
4. 你可能会看到一个提示，要求你继续启用“Compute Engine API”（计算引擎 API），如图 10-2 所示。选择启用它，这可能会需要几分钟的时间。

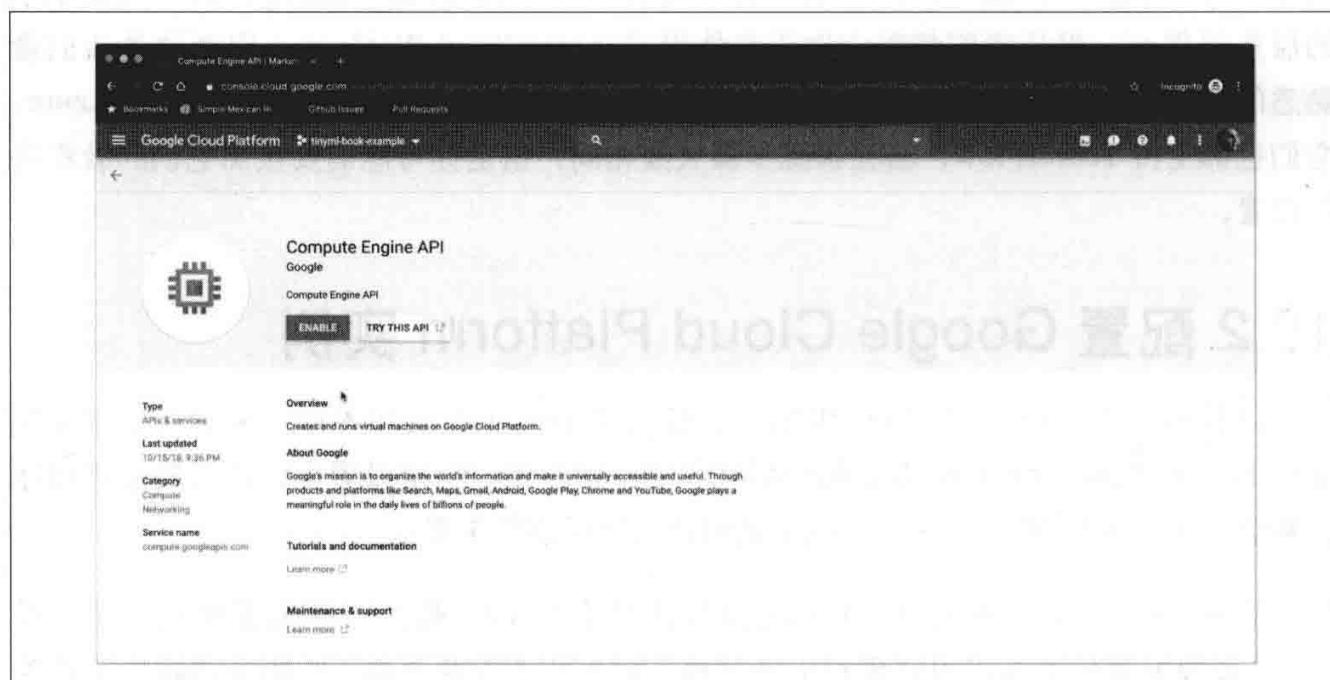


图 10-2：“Compute Engine API”页面

5. 接下来将打开一个“Notebook instances”（笔记本实例）页面。在顶部的菜单栏中，选择“NEW INSTANCE”（新建实例）。在打开的子菜单中，选择“Customize instance”（自定义实例），如图 10-3 所示。



图 10-3：新建实例菜单

6. 在“New notebook instance”（新建笔记本实例）页面的“instance name”（实例名称）框中，为你的机器命名，如图 10-4 所示，然后向下滚动以配置环境。

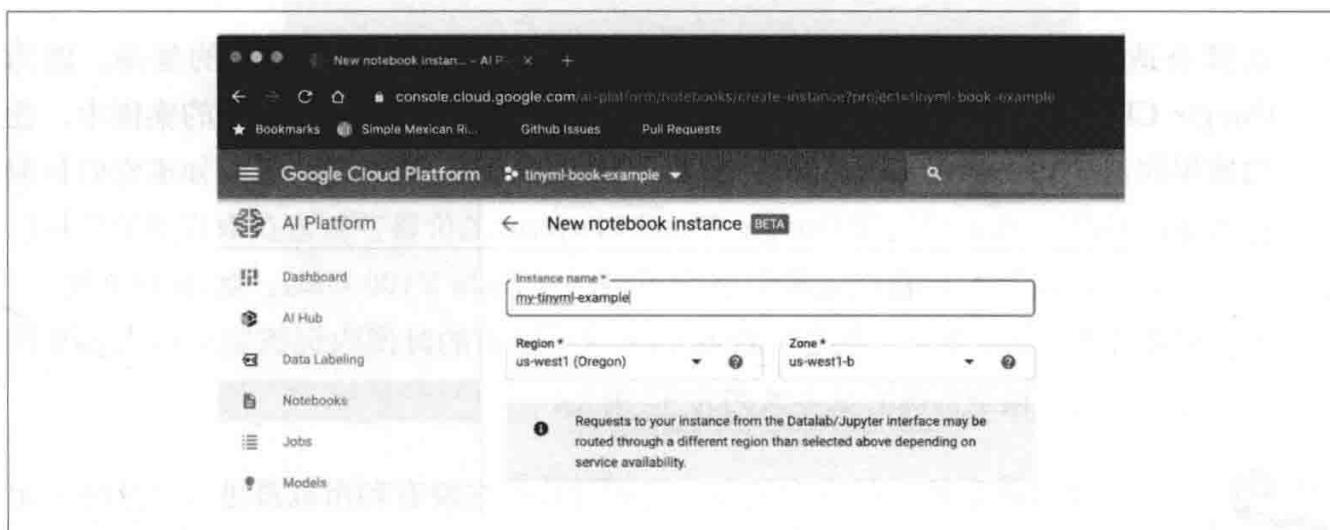


图 10-4：命名实例

7. 截至 2019 年 9 月，要选择的正确 TensorFlow 版本是 TensorFlow 1.14。在你阅读本书时，建议的版本可能已提高到 2.0 或者更高，但是可能会存在一些兼容问题，因此，如果仍然可以选择，建议从 1.14 或 1.x 分支中的其他版本开始。

8. 在“Machine configuration”（机器配置）部分，选择至少具有 4 个 CPU 和 15GB 的 RAM 的机器，如图 10-5 所示。

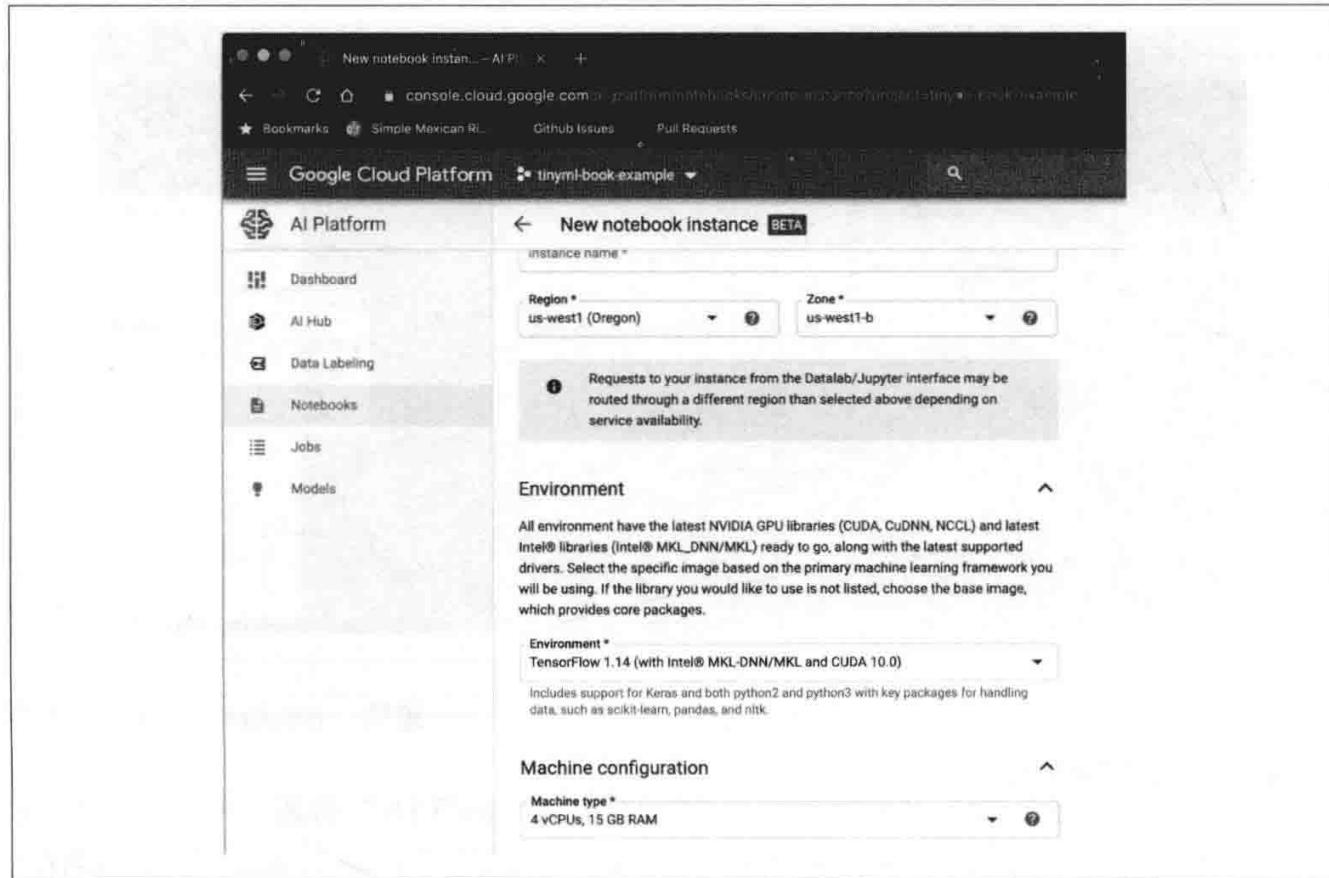


图 10-5：选择 CPU 以及版本

9. 选择合适的 GPU 将最大限度地提高你的训练速度。这可能比想象的复杂，因为 Google Cloud 并不一定会在所有区域都提供相同类型的硬件。在我们的案例中，我们使用的是“us-west1 (Oregon)”区和“us-west-1b”域，因为我们知道它们目前提供高端 GPU。你可以使用 Google Cloud Platform 的价格计算器获取详细的价格信息，但是在本示例中，我们选择了一个 NVIDIA Tesla V100 GPU，如图 10-6 所示。每月需要花费 1300 美元，但它允许我们在一天左右的时间内训练完成行人检测模型，也就是说，模型训练的成本大约为 45 美元。



这些高端机器的运行成本很高，因此请确保在没有利用机器进行训练时关闭机器。否则，你将为一台闲置的机器付费。

10. 自动安装 GPU 驱动程序会使配置工作轻松很多，因此请确保选择该选项，如图 10-7 所示。

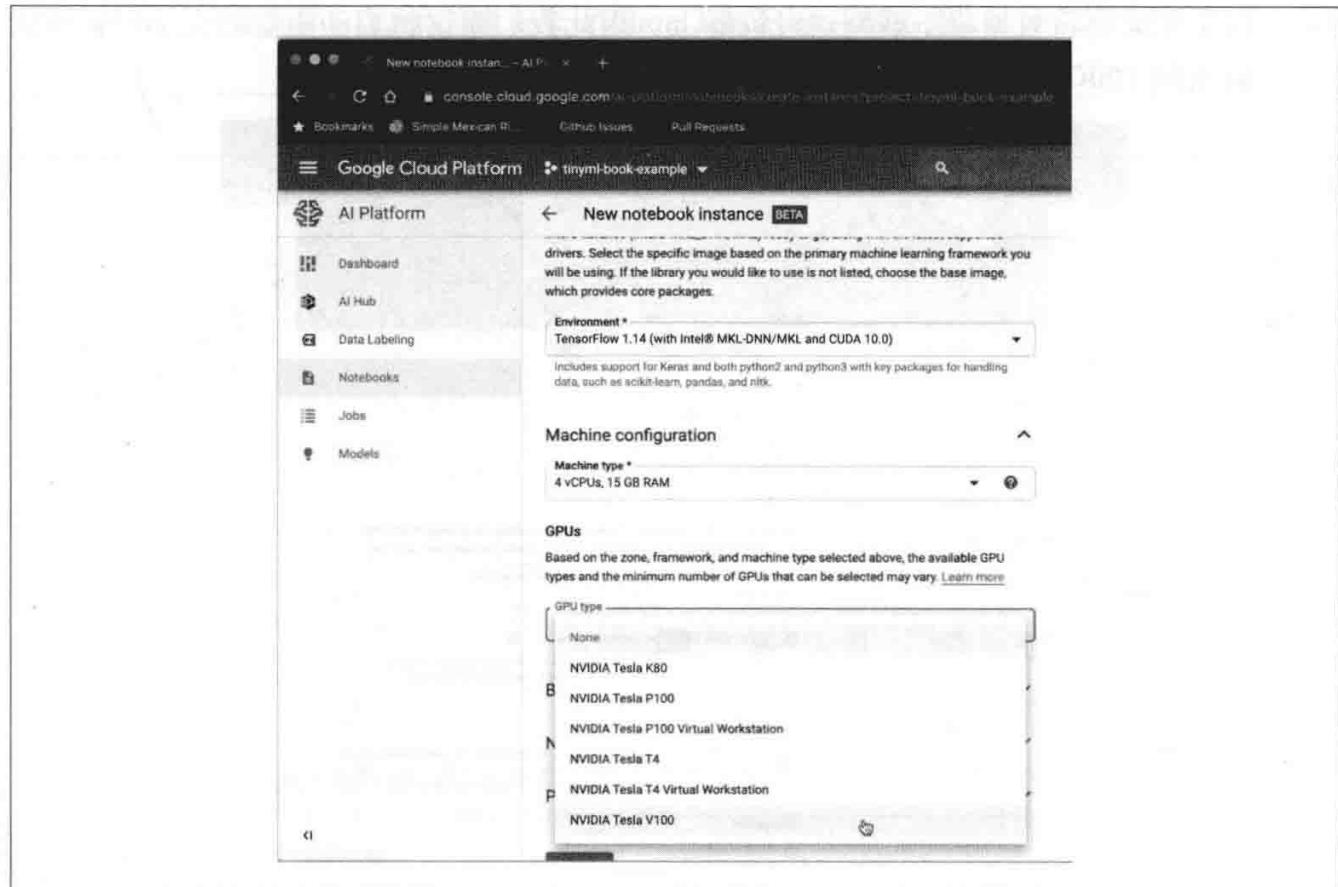


图 10-6：选择 GPU

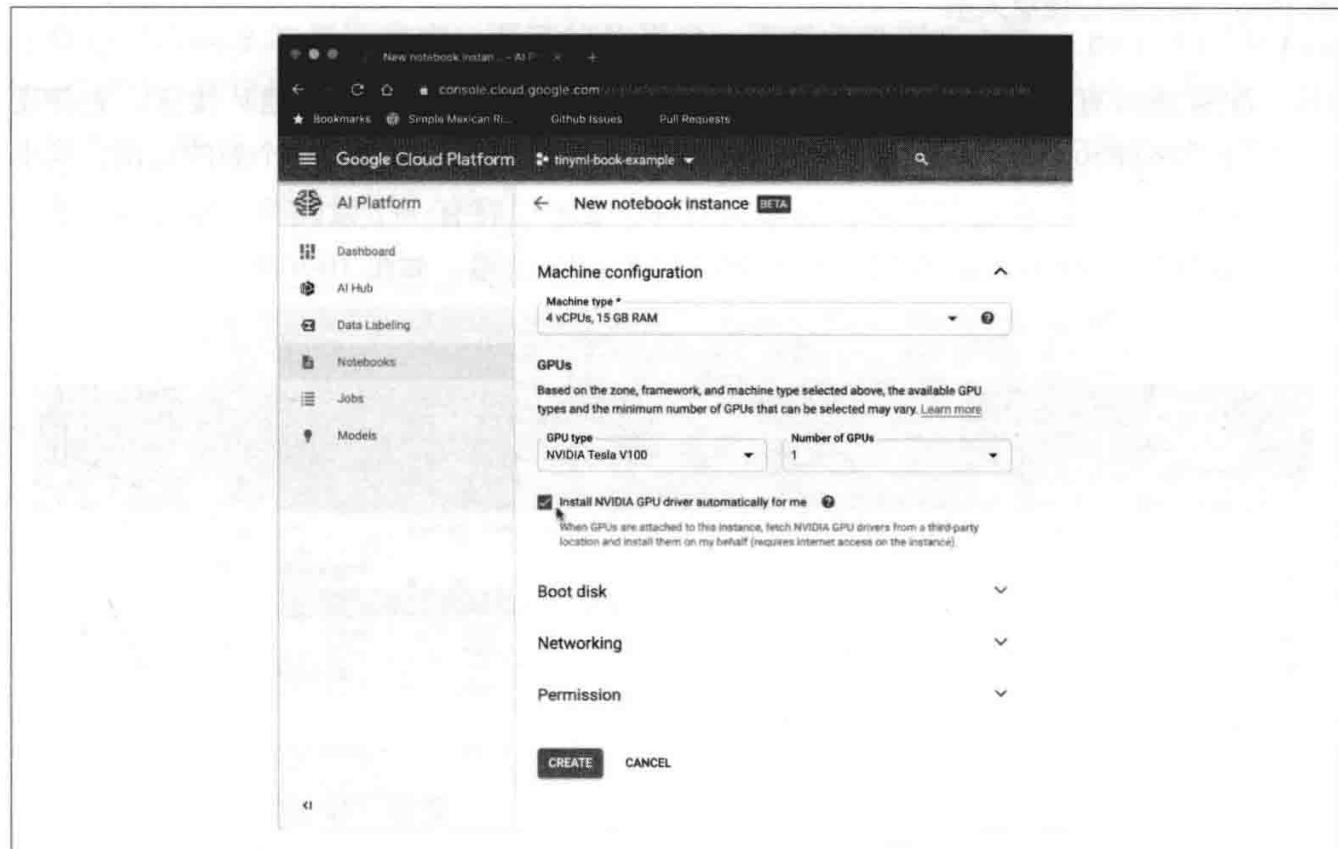


图 10-7：GPU 驱动

11. 由于你需要将数据集下载到这台机器上，因此我们建议将启动磁盘的大小设置为比默认的 100GB 大一点，可能需要高达 500GB，如图 10-8 所示。

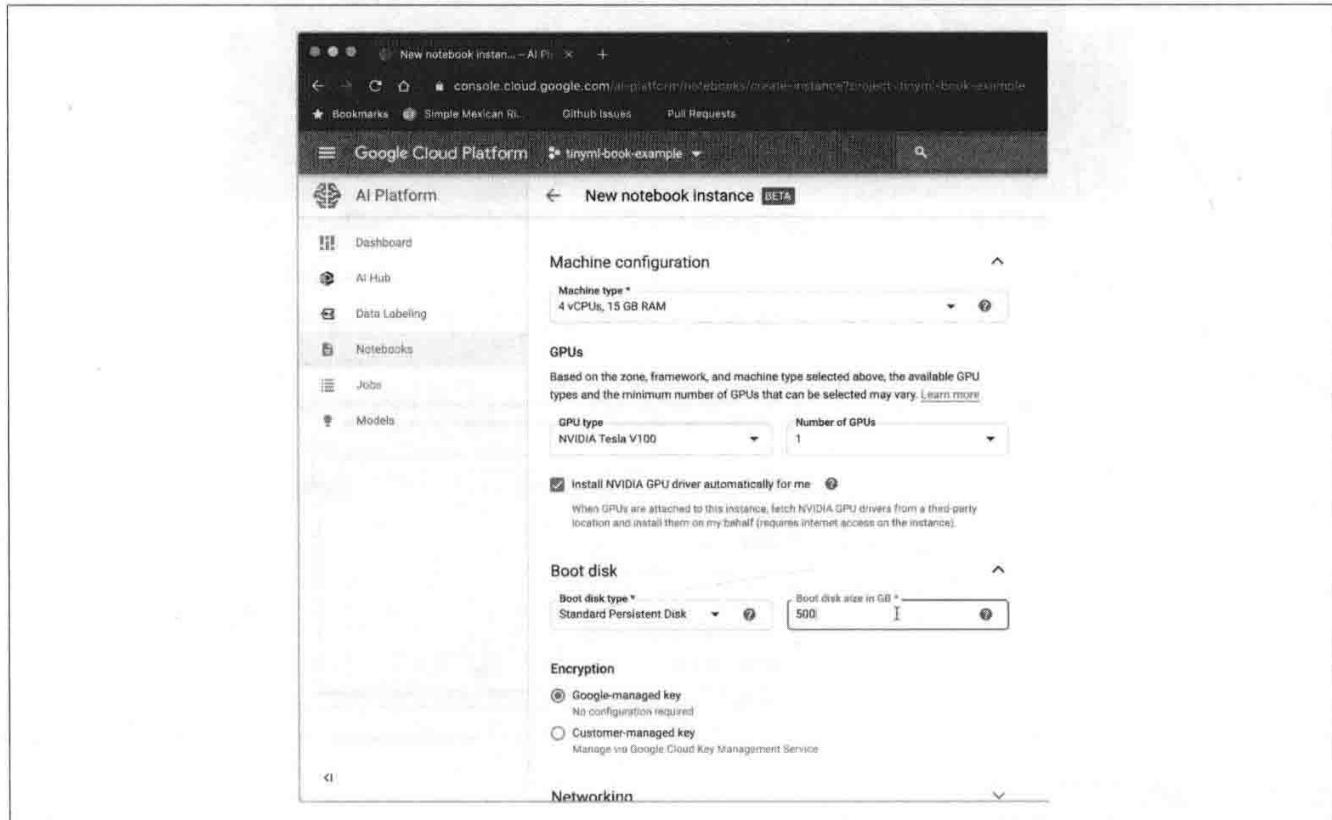


图 10-8：增加启动磁盘大小

12. 配置完所有这些选项后，在页面底部，单击“CREATE”（创建）按钮，它将返回“Notebook instances”（笔记本实例）页面。列表中应该有一个新的实例，其中包含你为机器指定的名称。设置实例时，它旁边将有一个微调器。完成后，单击“OPEN JUPYTERLAB”（打开 JUPYTERLAB）链接，如图 10-9 所示。



图 10-9：实例页面

13. 在打开的页面中，选择创建一个 Python 3 笔记本（参见图 10-10）。

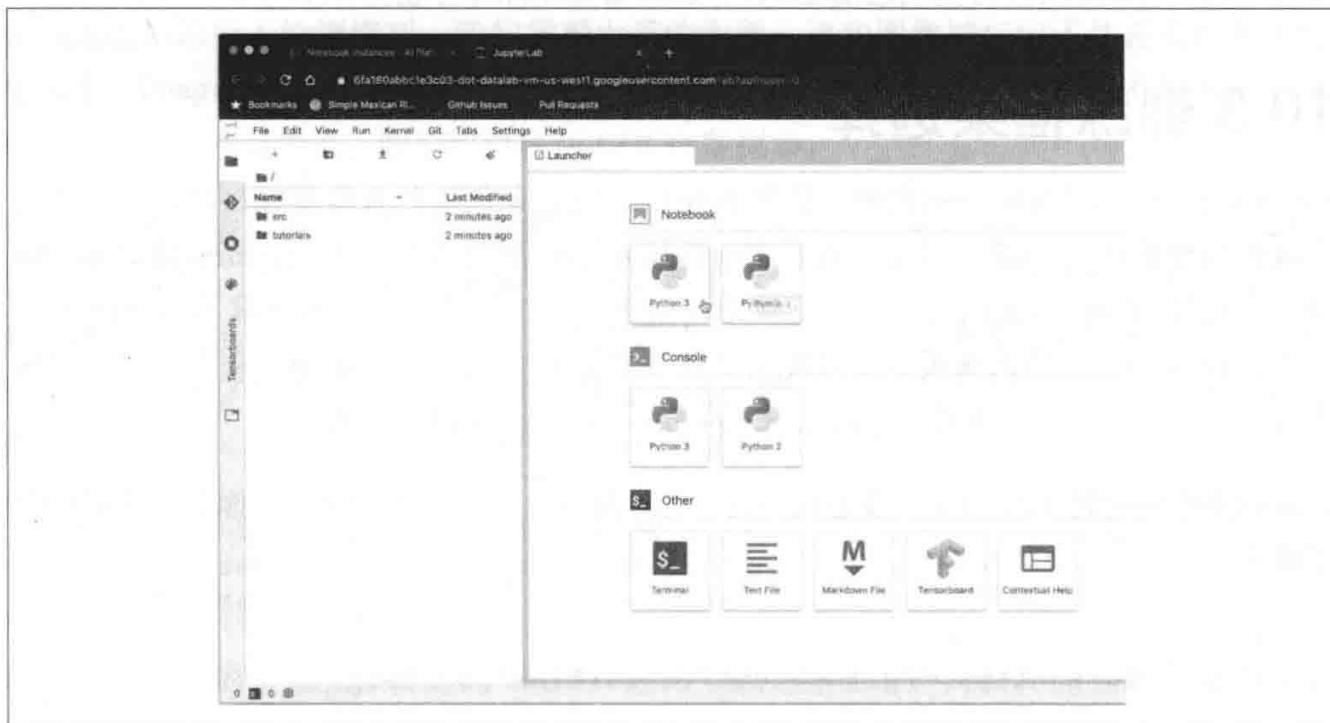


图 10-10：选择笔记本页面

这将为你创建一个连接到实例的 Jupyter 笔记本。如果你不熟悉 Jupyter，它将为你提供一个不错的 Web 界面，以帮助你在机器上运行 Python 解释器，并将命令和结果存储在可以共享的笔记本中。要开始使用它，请在右侧面板中输入 `print("Hello World")`，然后按 Shift+Return。你应该会在下面看到“Hello World！”被打印出来，如图 10-11 所示。如果是，那么说明你已经成功设置了机器实例。我们将在本教程的其余部分使用这个笔记本来输入命令。

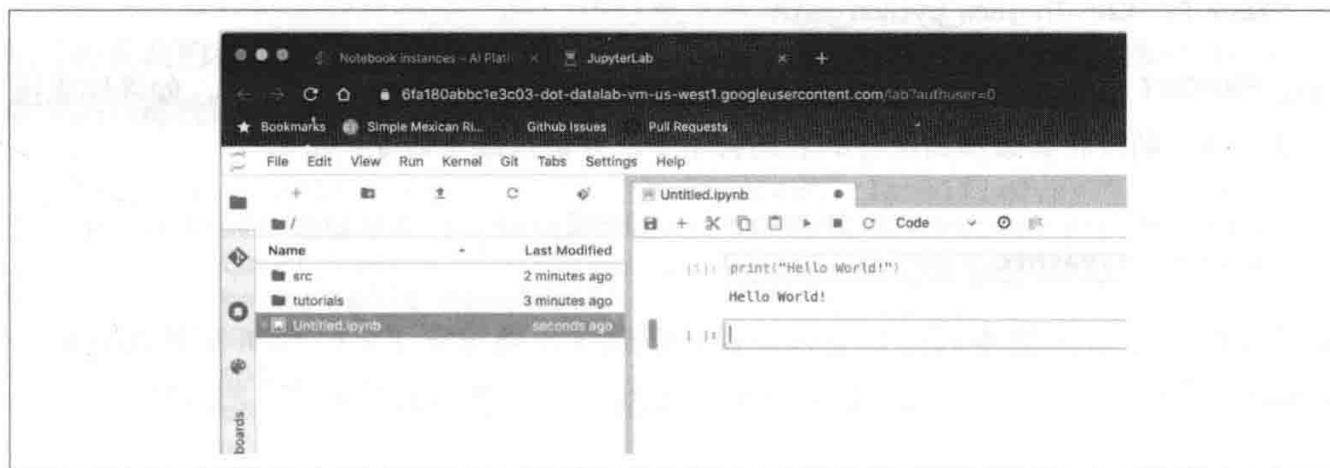


图 10-11：“Hello World”示例

后面的许多命令都假定你是在 Jupyter 笔记本上运行的，因此它们以！开头，这表示它

们作为 shell 命令而不是 Python 语句运行。如果你是直接在终端运行命令（例如，在打开 Secure Shell 连接并与实例通信之后），那么需要删除命令前的！符号。

10.3 训练框架选择

Keras 是在 TensorFlow 中构建模型的推荐接口，但是在创建行人检测模型时，它还不支持我们需要的所有功能。因此，我们将向你展示如何使用稍旧一些的 *tf.slim* 接口来训练模型。这些旧的接口虽然仍然被广泛使用，但它们现在已经被标记为弃用（deprecated）。因此，TensorFlow 的未来版本中可能会不再支持这种方法。我们希望将来在互联网上发布 Keras 版的说明，请查看 tinymlbook.com/persondetector 以获取更新。

Slim 的模型定义是 TensorFlow 模型代码仓库的一部分，因此在开始之前，你需要从 GitHub 下载它：

```
! cd ~  
! git clone https://github.com/tensorflow/models.git
```



后续的指南都假定你从主目录执行操作，因此模型代码仓库位于 `~/models`。除非另有说明，否则所有命令都是从主目录运行。你可以将代码仓库下载到其他位置，但是需要更新对它的所有引用。

要使用 Slim，你需要确保 Python 可以找到其模块并安装依赖项。在 iPython Notebook^{译注 1} 中执行此操作的方法如下：

```
! pip install contextlib2  
import os  
new_python_path = (os.environ.get("PYTHONPATH") or '') + ":models/research/slim"  
%env PYTHONPATH=$new_python_path
```

通过 EXPORT 语句更新 `PYTHONPATH` 仅适用于当前的 Jupyter 会话，因此，如果你直接使用 bash，则应将其添加到保存的启动脚本中，并运行以下命令：

```
echo 'export PYTHONPATH=$PYTHONPATH:models/research/slim' >> ~/.bashrc  
source ~/.bashrc
```

如果你在运行 Slim 脚本时看到导入错误，请确保正确设置了 `PYTHONPATH` 并安装了 `contextlib2`。你可以在代码仓库的 README 中找到更多有关 *tf.slim* 的信息。

译注 1：iPython 是一种基于 Python 的交互式解释器，较于原生的 Python Shell 提供了更为强大的编辑和交互功能。iPython Notebook 后来更名为 Jupyter Notebook。

10.4 构建数据集

要训练我们的行人检测模型，我们需要大量的图像，这些图像被标记了是否有行人出现在其中。ImageNet 1000-class 数据集被广泛用于训练图像分类器，但它并不包含是否有行人的标签。幸运的是，我们可以使用 COCO 数据集。

COCO 数据集被设计用于训练定位模型，因此图像并没有被标记我们需要的“行人”“非行人”的类别。相反，数据集中每个图像都带有它包含的所有对象的边框列表。“行人”是这些对象类别之一，因此要获得我们所需的类别标签，我们需要查找带有“行人”边框的图像。为了确保它们不会太小而导致无法识别，我们还需要排除掉一些过小的边框。Slim 包含一个方便的脚本，可以下载数据并将边框转换为标签：

```
! python download_and_convert_data.py \
--dataset_name=visualwakewords \
--dataset_dir=data/visualwakewords
```

这是一个很大的下载文件，约为 40GB，因此下载会需要一些时间，而且你需要确保驱动器上至少还有 100GB 的可用空间，以便为解压和进一步处理预留出空间。整个过程大约需要 20 分钟才能完成。下载和解压完成后，你将在 *data/visualwakewords* 中拥有一组 TFRecords，其中包含标记的图像信息。COCO 数据集由 Aakanksha Chowdhery 创建，被称为视觉唤醒词数据集（Visual Wake Words dataset）。它被设计用于对嵌入式计算机视觉进行基准评测和测试，因为它代表了在资源紧张的情况下需要完成的一项非常常见的任务。我们希望它能推动设计出更好的模型，以完成类似的任务。

10.5 训练模型

使用 *tf.slim* 进行训练的好处之一是，我们通常需要修改的参数可以作为命令行参数使用，因此我们可以调用标准的 *train_image_classifier.py* 脚本来训练模型。你可以使用此命令来构建我们在示例中使用的模型：

```
! python models/research/slim/train_image_classifier.py \
--train_dir=vwv_96_grayscale \
--dataset_name=visualwakewords \
--dataset_split_name=train \
--dataset_dir=data/visualwakewords \
--model_name=mobilenet_v1_025 \
--preprocessing_name=mobilenet_v1 \
--train_image_size=96 \
--use_grayscale=True \
--save_summaries_secs=300 \
--learning_rate=0.045 \
--label_smoothing=0.1 \
--learning_rate_decay_factor=0.98 \
```

```
--num_epochs_per_decay=2.5 \
--moving_average_decay=0.9999 \
--batch_size=96 \
--max_number_of_steps=1000000
```

在单个 GPU V100 实例上，需要花费几天时间才能完成全部 100 万步，但是如果你想尽早进行试验，几个小时后你应该就能获得一个相当准确的模型。以下是一些其他注意事项：

- `checkpoint` 和摘要将被保存在 `--train_dir` 参数指定的文件夹中，这是你查找结果的地方。
- `--dataset_dir` 参数应与你从 Visual Wake Words 构建脚本中保存 TFRecords 的参数相匹配。
- 我们使用的架构由 `--model_name` 参数指定。`mobilenet_v1` 前缀表示脚本使用 MobileNet 的第一个版本。我们曾经尝试过使用更高的版本，但是这些版本在中间层激活缓冲区中使用了更多的 RAM，因此现在我们仍坚持使用原始的版本。`025` 是要使用的深度缩放因子 (depth multiplier)，它主要影响权重参数的数量。将这个值设置为较小的值 (`025`) 可以确保模型适合大小在 250KB 之内的闪存。
- `--preprocessing_name` 用以控制在将输入图像输入模型之前如何对其进行修改。`mobilenet_v1` 版本将图像的宽度和高度缩小到 `--train_image_size` 指定的大小 (在我们的示例中为 96 像素，因为我们想减少计算量)。它还会将像素值从 0~255 范围内的整数缩放为 -1.0~+1.0 的浮点数 (尽管我们会在训练完成之后对它们进行量化)。
- 我们在 SparkFun Edge 开发板上使用的 HM01B0 摄像头是单通道的，因此为了获得最佳效果，我们需要在黑白图像上训练模型。我们传入 `--use_grayscale` 标志以启用灰度预处理。
- `--learning_rate`、`--label_smoothing`、`--learning_rate_decay_factor`、`--num_epochs_per_decay`、`--moving_average_decay` 和 `--batch_size` 参数都用以控制如何在训练过程中更新权重。训练深度网络仍然是一项隐秘的技术，因此我们通过不断地试验发现针对此特定模型的具体值。你可以试着调整它们以加快训练速度或者提高准确率，但是我们无法提供更多有关如何更改这些参数的指导意见，而且很可能训练准确率最终无法收敛。
- `--max_number_of_steps` 指定训练应持续多长时间。没有很好的方法提前设置此阈值——你需要不断试验以确定模型的准确率从何时开始不再提升，以了解何时需要终止模型训练。在我们的例子中，我们默认为 100 万步，因为对于这种特定模型，我们知道 100 万步通常是个不错的停止点。

启动脚本后，你应该看到如下所示的输出：

```
INFO:tensorflow:global step 4670: loss = 0.7112 (0.251 sec/step)
I0928 00:16:21.774756 140518023943616 learning.py:507] global step 4670: loss
= 0.7112 (0.251 sec/step)
INFO:tensorflow:global step 4680: loss = 0.6596 (0.227 sec/step)
I0928 00:16:24.365901 140518023943616 learning.py:507] global step 4680: loss
= 0.6596 (0.227 sec/step)
```

不用担心有重复的行出现：这只是 TensorFlow 日志打印与 Python 交互方式的副作用。每行都有关于训练过程的两个关键信息。一个关键信息是全局步（global step），它是对训练进行程度的计数。由于我们将训练限制为 100 万步，因此在本例中，我们完成了接近 5% 的训练。结合每秒步数来估算会很有用，你可以使用它来估算整个训练过程大致需要的时间。在我们的例子中，每秒完成约 4 个训练步，因此，100 万步将花费大约 70 个小时，即接近 3 天的时间。另一个关键信息是损失值（loss）。这是对部分训练的模型的预测与正确值的接近程度的一种度量，损失值越小越好。这将反映出模型准确性的变化，如果模型正在学习，损失值应该在训练期间逐渐减小。由于噪声很大，因此损失值可能会在很短的时间内反弹很多，但是如果一切正常，等待一个小时左右再来检查一下，你应该会看到明显的下降。在图表中更容易看到这种变化，这是尝试使用 TensorBoard 的主要原因之一。

10.6 TensorBoard

TensorBoard 是一个网络应用程序，它允许你查看 TensorFlow 训练过程中的数据可视化结果，默认情况下，大多数云实例中都包含 TensorBoard。如果你使用的是 Google Cloud AI Platform，则可以通过在笔记本界面的左侧选项卡中打开命令面板，然后向下滚动以选择“Create a new tensorboard”（创建新的 tensorboard）来启动新的 TensorBoard 会话。然后，系统会提示你输入摘要日志的位置，也就是在训练脚本中输入用于`--train_dir` 的路径。在上一个示例中，我们的摘要日志在名为 `www_96_grayscale` 的文件夹中。需要注意的是，一个常见错误是在路径末尾添加斜线，这将导致 TensorBoard 无法找到目录。

如果你是在其他环境中从命令行启动 TensorBoard，则需要将此路径作为`--logdir` 参数传递给 TensorBoard 命令行工具，并将浏览器指向 `http://localhost:6006`（或正在运行的机器的地址）。

导航到 TensorBoard 地址或通过 Google Cloud 打开会话后，你应该看到一个类似于图 10-12 的页面。考虑到脚本每五分钟才保存一次摘要，因此可能需要一些时间才能在图形中看到有用的内容。图 10-12 展示了训练超过一天后的结果。最重要的图是“clone_loss”，它展示了输出日志中损失值的变化。正如你在本例中所看到的，它的波动很大，但是总体趋势还是随着时间的推移而下降的。如果经过几个小时的训练后仍旧看不到损失值

下降，那么这表明你的模型可能无法收敛到一个很好的结果。你可能需要调试数据集或训练参数来看是哪里出了问题。

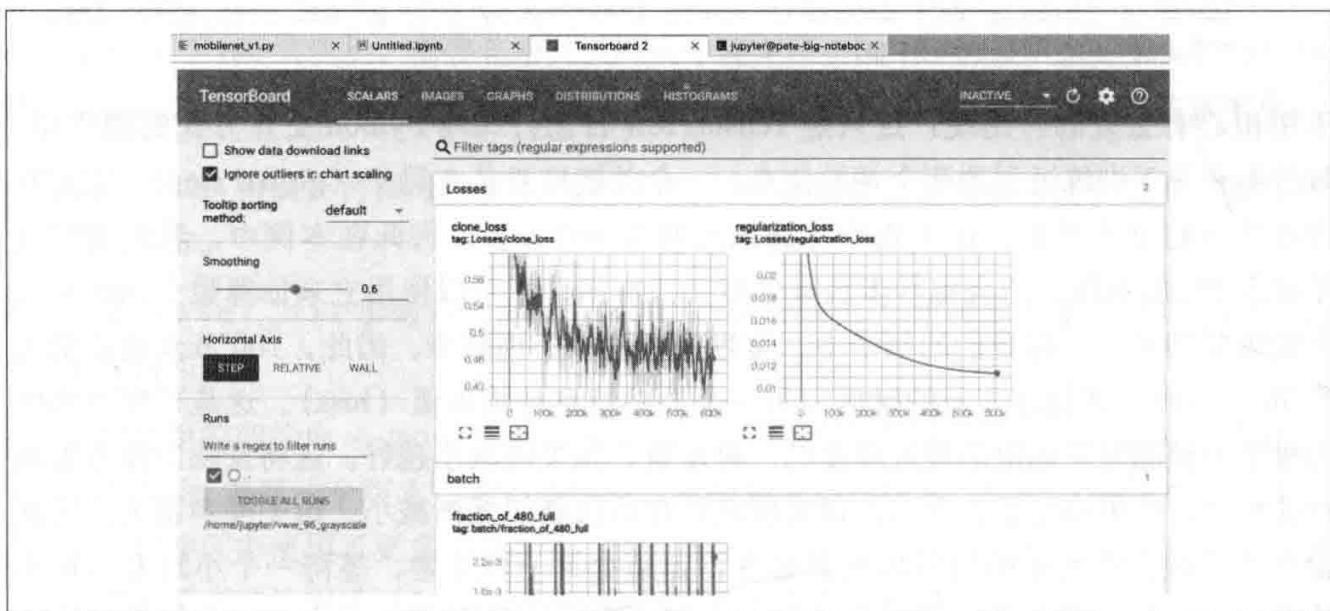


图 10-12: TensorBoard 中的图形

TensorBoard 打开时，默认情况下为“SCALARS”（标量）选项卡，但是在训练过程中可能有用的另一部分是“IMAGES”（图像）（图 10-13）。这显示了当前正在训练模型的图像的随机选择，包括任何形变或者其他预处理。在图 10-13 中，你可以看到图像已经被翻转，并且在送入模型之前已经被转换为灰度图。这些信息并不像损失图那样重要，但是有利于确保数据集是你所期望的，而且随着训练的进行，图像不断更新的过程也很有趣。



图 10-13: TensorBoard 中的图像

10.7 评估模型

损失函数（loss function）与模型训练的好坏有关，但它并不是一个直接的、容易理解的指标。我们真正关心的是我们的模型能够正确检测到多少行人，但是要计算它，我们需要运行一个单独的脚本。你无须等到模型完全训练好，就可以在 `--train_dir` 文件夹中检查所有 checkpoint 的准确率。为此，可以运行以下命令：

```
! python models/research/slim/eval_image_classifier.py \
--alsologtosterr \
--checkpoint_path=vw_96_grayscale/model.ckpt-698580 \
--dataset_dir=data/visualwakewords \
--dataset_name=visualwakewords \
--dataset_split_name=val \
--model_name=mobilenet_v1_025 \
--preprocessing_name=mobilenet_v1 \
--use_grayscale=True \
--train_image_size=96
```

你需要确保 `--checkpoint_path` 指向一组有效的 checkpoint 数据。checkpoint 存储在三个单独的文件中，因此需要指定它们通用的前缀。例如，如果你有一个名为 `model.ckpt-5179.data-00000-of-00001` 的 checkpoint 文件，则其前缀为 `model.ckpt-5179`。该脚本应产生类似如下所示的输出：

```
INFO:tensorflow:Evaluation [406/406]
I0929 22:52:59.936022 140225887045056 evaluation.py:167] Evaluation [406/406]
eval/Accuracy[0.717438412]eval/Recall_5[1]
```

这里最重要数字是准确率（Accuracy）。它显示了被正确分类的图像的比例，在转换为百分比后，我们的准确率为 72%。如果你遵循示例脚本，那么应该期望经过 100 万步全面训练的模型能达到接近 84% 的准确率，且损失值大约为 0.4。

10.8 将模型导出到 TensorFlow Lite

当模型训练到一个你比较满意的准确率时，你需要将 TensorFlow 训练环境中得到的结果转换为可以在嵌入式设备上运行的形式。如前几章中提到的，这可能会是一个复杂的过程，而且 `tf.slim` 也为这个过程增加了一些特有的难度。

10.8.1 导出到 GraphDef Protobuf 文件

每次运行脚本，Slim 都会从 `model_name` 生成架构，因此要在 Slim 之外使用模型，需要以一种通用格式保存。我们将使用 GraphDef Protobuf 序列化格式，因为 Slim 和 TensorFlow 的其余部分都可以理解这种格式：

```
! python models/research/slim/export_inference_graph.py \
--alsologtostderr \
--dataset_name=visualwakewords \
--model_name=mobilenet_v1_025 \
--image_size=96 \
--use_grayscale=True \
--output_file=vww_96_grayscale_graph.pb
```

如果这个脚本运行成功，那么你的主目录中应该会出现一个新的名为 *vww_96_grayscale_graph.pb* 的文件。它包含模型中算子的布局（layout），但尚未包含任何权重数据。

10.8.2 冻结权重

将训练好的权重与算子计算图（operation graph）一起存储的过程称为冻结。这一过程将从 checkpoint 文件加载权重的值，然后将算子计算图中的所有变量转换为相应的权重常量。后面的命令使用 100 万训练步中的 checkpoint，但是你可以提供任何有效的 checkpoint 路径。冻结图的脚本存储在主 TensorFlow 代码仓库中，因此你需要在运行以下命令之前从 GitHub 下载该脚本：

```
! git clone https://github.com/tensorflow/tensorflow
! python tensorflow/tensorflow/python/tools/freeze_graph.py \
--input_graph=vww_96_grayscale_graph.pb \
--input_checkpoint=vww_96_grayscale/model.ckpt-1000000 \
--input_binary=true --output_graph=vww_96_grayscale_frozen.pb \
--output_node_names=MobilenetV1/Predictions/Reshape_1
```

运行这段脚本之后，你应该看到一个名为 *vww_96_grayscale_frozen.pb* 的文件。

10.8.3 量化并转换到 TensorFlow Lite

量化（quantization）是一个棘手且复杂的过程，它至今仍然是一个非常活跃的研究领域。因此，要把我们迄今为止训练过的浮点数图转换成 8 位实体需要相当多的代码。在第 15 章中，你可以找到更多有关什么是量化及其工作原理的解释，但这里我们将向你展示如何在训练的模型中使用量化。大部分代码都在准备将要被输入训练网络的图像，以便可以测量典型使用中的激活层范围。我们依靠 `TFLiteConverter` 类来处理量化并转换为推断引擎所需的 TensorFlow Lite FlatBuffer 文件：

```
import tensorflow as tf
import io
import PIL
import numpy as np

def representative_dataset_gen():
    record_iterator = tf.python_io.tf_record_iterator
    (path='data/visualwakewords/val.record-00000-of-00010')

    count = 0
```

```

for string_record in record_iterator:
    example = tf.train.Example()
    example.ParseFromString(string_record)
    image_stream = io.BytesIO(
        (example.features.feature['image/encoded'].bytes_list.value[0]))
    image = PIL.Image.open(image_stream)
    image = image.resize((96, 96))
    image = image.convert('L')
    array = np.array(image)
    array = np.expand_dims(array, axis=2)
    array = np.expand_dims(array, axis=0)
    array = ((array / 127.5) - 1.0).astype(np.float32)
    yield([array])
    count += 1
    if count > 300:
        break

converter = tf.lite.TFLiteConverter.from_frozen_graph \
    ('vw_96_grayscale_frozen.pb', ['input'], ['MobilenetV1/Predictions/ \
    Reshape_1'])
converter.inference_input_type = tf.lite.constants.INT8
converter.inference_output_type = tf.lite.constants.INT8
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset_gen

tflite_quant_model = converter.convert()
open("vw_96_grayscale_quantized.tflite", "wb").write(tflite_quant_model)

```

10.8.4 转换为 C 源文件

转换器会写出一个文件，但大多数嵌入式设备没有文件系统。要从程序访问序列化的数据，我们必须将它编译为可执行文件并将其存储在闪存中。最简单的方法是将文件转换为 C 数据数组，就像我们在前几章中所做的那样：

```

# Install xxd if it is not available
! apt-get -qq install xxd
# Save the file as a C source file
! xxd -i vw_96_grayscale_quantized.tflite > person_detect_model_data.cc

```

现在，你可以将现有的 *person_detect_model_data.cc* 文件替换为经过训练的版本，这样就能够在嵌入式设备上运行自己的模型了。

10.9 训练其他类别

COCO 数据集中有 60 多种不同的对象类型，因此训练自定义模型的一个简单方法就是在创建训练数据集时选择另外一种对象类型而非行人。以下脚本是训练寻找车辆的例子：

```
! python models/research/slim/datasets/build_visualwakewords_data.py \
--logtostderr \
--train_image_dir=coco/raw-data/train2014 \
--val_image_dir=coco/raw-data/val2014 \
--train_annotations_file=coco/raw-data/annotations/instances_train2014.json \
--val_annotations_file=coco/raw-data/annotations/instances_val2014.json \
--output_dir=coco/processed_cars \
--small_object_area_threshold=0.005 \
--foreground_class_of_interest='car'
```

你应该能够按照与行人检测程序相同的步骤进行操作，用新的 `coco/processed_cars` 路径代替之前所有的 `data/visualwakewords`。

如果 COCO 中没有你感兴趣的对像类别，则可以使用迁移学习来帮助你在自己收集的、哪怕是很小的数据集上进行训练。虽然我们还没有例子可以分享，但是你可以在 tinymlbook.com 上查看有关此方法的更新。

10.10 理解架构

MobileNets 有一系列的架构，旨在使用尽可能少的权重参数和算术运算提供良好的准确性。MobileNets 现在有多个版本，但在我们的例子中，我们使用的是原始的 v1 版本，因为它在运行时需要的 RAM 最少。v1 架构背后的核心概念是深度可分离卷积 (depthwise separable convolution)。这是经典二维卷积的一种变体，它的工作效率更高，同时不会牺牲太多准确性。常规卷积基于在输入的所有通道上应用特定大小的过滤器来计算输出值。这意味着每个输出中涉及的计算数量是过滤器的宽度乘高度，再乘输入通道的数量。深度卷积将这个庞大的计算分解为单独的部分。首先，每个输入通道由一个或多个矩形过滤器过滤以产生中间值。然后使用逐点卷积 (pointwise convolution) 将这些值组合在一起。这大大减少了所需的计算量，并且在实践中得到了与常规卷积相似的结果。

MobileNet v1 是由 14 个深度可分离的卷积层和一个平均池 (average pool) 组成的堆栈，然后是一个全连接层，最后是 softmax 层。我们将宽度缩放因子 (width multiplier) 指定为 0.25，与标准模型相比，通过将每个激活层中的通道数减少 75%，可以将每次推断的计算量减少到大约 6000 万次。从本质上讲，它在操作上与普通卷积神经网络非常相似，每个层的学习模式都在输入中。较靠前的层更像边缘识别过滤器，在图像中发现低级结构，而较靠后的层则将这些信息合成为更抽象的模式，从而帮助最终的对象分类。

10.11 小结

使用机器学习进行图像识别需要海量数据和强大的算力。在本章中，你学习了如何从零

开始、仅仅提供数据集以训练一个模型，以及如何将模型转换为针对嵌入式设备优化的格式。

这种经验应该能为你解决产品所需的机器视觉问题打下良好的基础。计算机能够看到并了解周围的世界，这看上去多么不可思议，我们迫不及待地想看看你能利用它创造出什么有趣的产品！

魔杖：创建应用程序

到目前为止，我们的示例应用程序已经处理了各种可以被人类轻松理解的数据。我们大脑的大部分区域都致力于理解语音和图像，所以对我们来说，解释视觉数据或者音频数据并对正在发生的事情有具体的概念并不困难。

然而，还有很多数据并不那么容易理解。机器及其传感器会产生大量的信息，这些信息无法轻松映射到人类的感官上。即使是以可视化的方式呈现，我们的大脑也很难掌握数据中的趋势和模式。

例如，图 11-1 和图 11-2 展示了放置在正在运动的人的口袋中的手机传感器捕获的数据。该传感器是加速度传感器（accelerometer），它可以在三个维度上测量加速度（我们将在后面详细讨论）。图 11-1 中的图形展示了来自一个慢跑者的加速度传感器的数据，而图 11-2 中的图形展示了同一个人走下楼梯的数据。

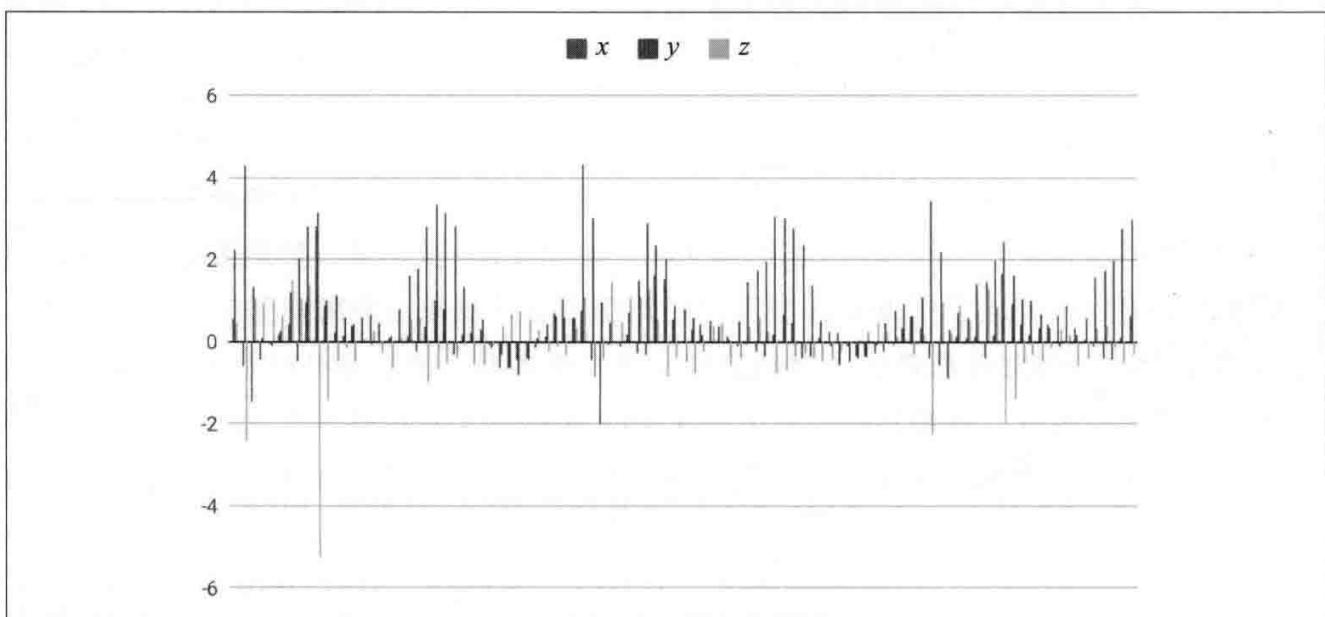


图 11-1：显示了一个人慢跑的数据的图像（MotionSense 数据集）

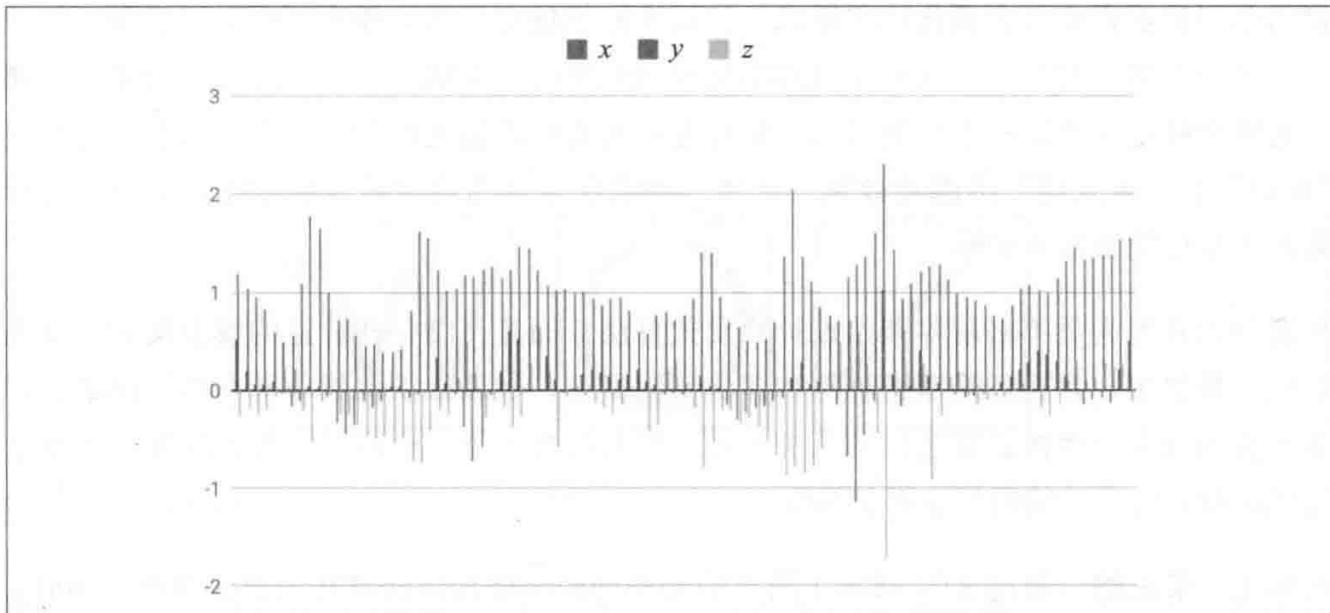


图 11-2：显示了一个人下楼梯的数据的图像（MotionSense 数据集）

如你所见，即使数据表示的是简单且可关联的活动，也很难区分这两个活动。想象一下，我们试图区分复杂的工业机器的运行状态，其中可能有数百个传感器测量得到的各种晦涩的特征。

通常，可以编写能理解此类数据的人工算法（handcrafted algorithm）。例如，步态专家可以识别走楼梯的迹象，并能够将这些知识在代码中用函数表达。这类函数称为启发式函数（heuristic），被广泛用于从工业自动化到医疗设备的各种应用中。

要创建启发式算法，有两个要求。第一个要求是领域知识（domain knowledge）。启发式算法表达了人类的知识和理解，因此要编写一种启发式算法，你需要了解数据的含义。为了理解这一点，考虑一种可以根据体温确定一个人是否发烧的启发式算法。无论是谁创建这种算法，他都必须了解与识别发烧有关的温度变化的知识。

创建启发式算法的第二个要求是编程和数学专业知识。尽管确定某人的温度是否过高相当容易，但其他问题可能要复杂得多。基于多个数据流中的复杂模式来识别系统状态可能需要了解一些高级技术，如统计分析或信号处理。例如，假设要创建一种启发式算法来基于加速度传感器数据区分步行和跑步。为此，你可能需要了解如何对加速度传感器数据进行数学滤波，以获得对迈步频率的估算值。

启发式算法可能非常有用，但它需要领域知识和编程专业知识，这意味着构建启发式算法可能非常困难。首先，领域知识并非总是轻易可得的。例如，一家小公司可能没有足够的资源来进行必要的基础研究，以了解一个状态与另一个状态的区别。同样，即使具备特定领域的知识，也不是每个人都有在代码中设计和实现此类启发式算法所需的编程背景。

机器学习使我们有机会简化这些要求。使用有标签数据训练的模型可以学习识别表示一个或另一个类别的信号，这意味着对深度领域知识的要求减少了。例如，一个模型无须知道哪个特定温度比较重要就可以了解标志着人发烧的温度波动——它所需的只是标有“发烧”和“未发烧”的温度数据。此外，与实现复杂的启发式算法相比，机器学习所需的工程技能更容易掌握。

机器学习开发人员不必从零开始设计启发式算法，而是可以寻找合适的模型架构，收集和标记数据集，并迭代地通过训练和评估创建模型。领域知识仍然非常有用，但是它可能不再是使某些事情正常运行的先决条件。在某些情况下，机器学习得到的模型实际上可能比最好的手工编码算法更加精确。

实际上，最近的一篇论文^{注1}展示了简单的卷积神经网络如何能够从一次心跳中以 100% 的准确率检测患者的充血性心力衰竭。这比以前的任何诊断技术都更为精确。即使你不了解每个细节，这篇论文也很有趣。

通过训练深度学习模型来理解复杂数据并将其嵌入微控制器程序，我们可以创建智能传感器，它能够理解环境的复杂性，并从更高的维度告诉我们发生了什么。这对许多领域都具有巨大的影响。以下是一些潜在的应用：

- 通信环境较差的偏远地区的环境监测。
- 针对问题实时调整的自动化工业流程。
- 能对复杂的外部刺激做出反应的机器人。
- 不需要医疗专业人员即可进行疾病诊断。
- 了解身体运动的计算机接口。

在本章中，我们将从最后一个类别创建一个项目：一个数字“魔杖”（magic wand），它的主人可以挥舞它来施展各种“咒语”。它需要复杂的多维传感器数据作为输入，这些数据对人类来说可能毫无意义。它的输出将是一个简单的类别，如果最近发生了几类移动中的一种，它就会提醒我们。我们将看到深度学习如何将奇怪的数字数据转换为有意义的信息，并产生神奇的效果。

11.1 我们要创建什么

我们的“魔杖”可以用来施展几种类型的“魔法咒语”。要施展“咒语”，持杖者只需要

注 1：Mihaela Porumb 等人，“A convolutional neural network approach to detect congestive heart failure”
生物医学信号处理与控制（2020 年 1 月）。<https://oreil.ly/4HBFT>

以三个手势之一挥动魔杖，即“画翅膀”（wing）、“画圆”（ring）和“画角”（slope），如图 11-3 所示。

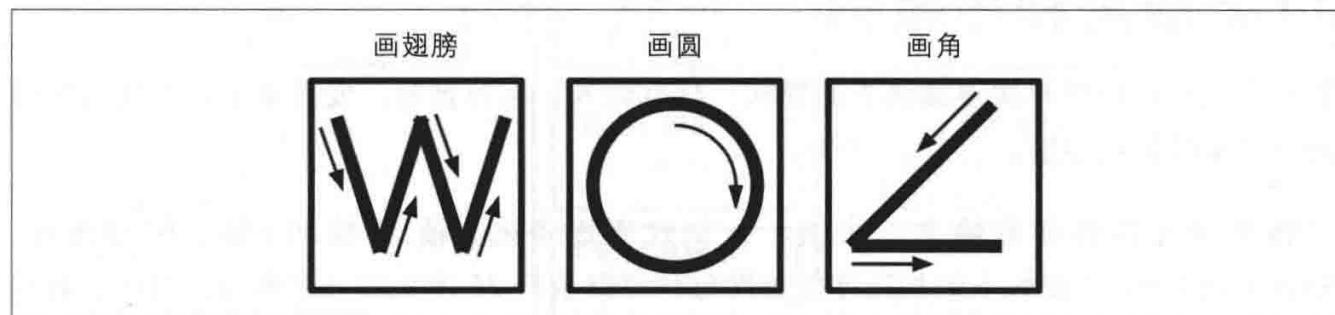


图 11-3：三种魔杖手势

魔杖将通过点亮 LED 对每个“咒语”做出反应。如果 LED 的“魔力”不够大，它还会将信息输出至可用于控制连接的计算机的串口。

为了理解物理手势，魔杖应用程序使用设备的加速度传感器来收集手在空间中的运动信息。加速度传感器可测量当前的加速度。举例来说，假设我们为一辆在红灯处停下并即将驶离的汽车安装了加速度传感器。当绿灯亮起时，汽车开始向前行驶，并提升速度直至达到最高限速。在此期间，加速度传感器将输出一个值，表示汽车的加速度。当汽车速度稳定后，它就不再加速，所以加速度传感器将输出零。

SparkFun Edge 和 Arduino Nano 33 BLE Sense 开发板都配备了三轴加速度传感器，这些加速度传感器被包含在焊接到每块开发板上的组件中。它们会在三个方向上测量加速度，这意味着它们可用于跟踪设备在三维空间中的运动状态。为了制造魔杖，我们将微控制器开发板连接到棍子的末端，这样就可以像施展咒语一样挥动魔杖。然后，我们将加速度传感器捕获的输出输入深度学习模型，该模型将对加速度数据进行分类以告诉我们是否检测到了已知手势。

我们提供了有关如何将该应用程序部署到以下微控制器平台的说明：

- Arduino Nano 33 BLE Sense
- SparkFun Edge

由于 ST Microelectronics STM32F746G Discovery 套件开发板不包含加速度传感器（而且开发板太大，无法连接至魔杖的末端），因此本节的示例将不再使用它。



TensorFlow Lite 会定期添加对新设备的支持，因此，如果你想使用的设备未在此处列出，那么请查看示例的 *README.md*。如果在执行后面的步骤时遇到麻烦，也可以查看更新的部署说明。

在下一节中，我们将研究应用程序的架构，并详细了解模型是如何工作的。

11.2 应用程序架构

我们的应用程序将再次遵循熟悉的模式：获取输入，运行推断，处理输出，并使用所得的信息来做某些事情。

三轴加速度传感器会输出三个值，分别代表设备的 x 轴、 y 轴和 z 轴上的加速度。SparkFun Edge 开发板上的加速度传感器每秒可以执行 25 次采样（速率为 25Hz）。我们的模型直接将这些值作为输入，这意味着我们不需要进行任何预处理。

捕获数据并运行推断后，我们的应用程序将确定是否检测到一个有效手势、打印一些输出到终端并点亮 LED。

11.2.1 模型介绍

我们的手势检测模型是一个大小约为 20KB 的卷积神经网络。它接收原始加速度传感器的值作为输入，一次可以接收 128 组 x 、 y 和 z 值，以 25Hz 的速率采集数据，相当于一次可以处理超过 5 秒的数据。输入数据的每个值都是一个 32 位浮点数，表示该方向上的加速度。

我们的模型是在多人分别执行四种手势的数据上训练的。它输出四种手势类别的概率得分：其中三个类别分别代表每种手势（“画翅膀”、“画圆”和“画角”），第四个类别代表未识别到手势。概率得分总和为 1，高于 0.8 的得分被认为是置信的。

由于我们每秒将运行多次推断，因此我们需要确保在执行手势时，单次错误的推断不会影响结果。我们的做法是，仅在经过一定数量的推断确认后才认为已经检测到手势。由于执行每种手势花费的时间不同，每个手势所需的推断次数也会有差别，最优的推断次数是通过试验确定的。同样，我们在不同设备上以不同的速率运行推断，因此也为每个设备设置了自己的阈值。

在第 12 章中，我们将探索如何根据自己的手势数据训练模型，并更深入地研究模型的工作原理。在此之前，让我们继续介绍我们的应用程序。

11.2.2 所有的功能组件

图 11-4 展示了魔杖应用程序的架构。如你所见，它几乎与行人检测应用程序一样简单。我们的模型接受原始的加速度传感器数据，这意味着我们不需要做任何预处理。

代码的六个主要部分与我们的行人检测示例结构类似，让我们依次介绍一下它们：

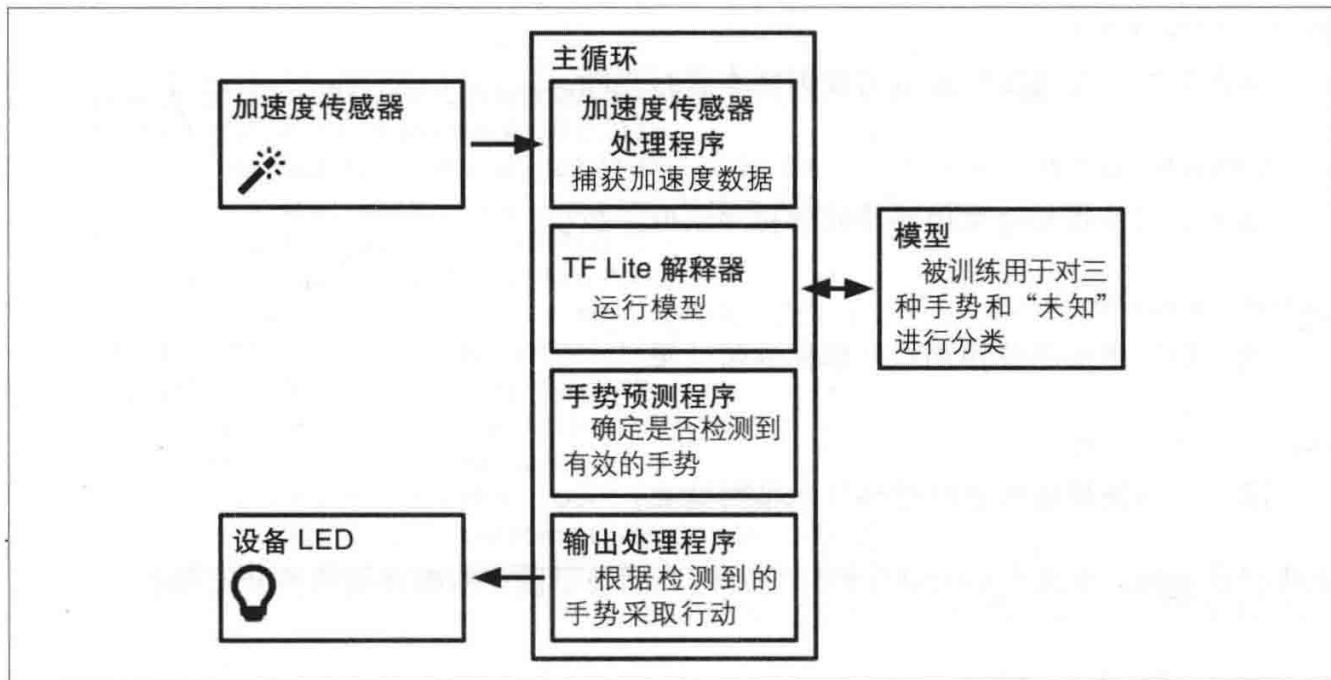


图 11-4：魔杖应用程序的主要组件

主循环 (*main loop*)

我们的应用程序在一个连续的循环中运行。由于我们的模型小而简单，并且不需要任何预处理，所以它每秒可以运行多次推断。

加速度传感器处理程序 (*accelerometer handler*)

该组件从加速度传感器捕获数据并将其写入模型的输入张量。它使用缓冲区保存数据。

TFlow Lite 解释器 (*TF Lite interpreter*)

同我们之前的示例一样，解释器运行 TensorFlow Lite 模型。

模型 (*model*)

模型作为数据数组包含在程序内，并由解释器运行。它的体积很小，只有 19.5KB。

手势预测程序 (*gesture predictor*)

该组件接收模型的输出，并根据概率阈值和连续的肯定预测数量来判断是否检测到某种手势。

输出处理器 (*output handler*)

输出处理器根据识别的手势点亮 LED 并将输出打印到串行端口。

11.3 详解测试

你可以在 GitHub 代码仓库中找到应用程序的测试：

magic_wand_test.cc

演示如何对加速度传感器的数据样本进行推断。

accelerometer_handler_test.cc

演示如何使用加速度传感器处理程序获取新数据。

gesture_predictor_test.cc

演示如何使用手势预测程序解释推断结果。

output_handler_test.cc

演示如何使用输出处理程序显示推断结果。

让我们从 *magic_wand_test.cc* 开始，它将向我们展示模型的端到端的推断过程。

11.3.1 基本流程

我们将介绍 *magic_wand_test.cc* 中的基本流程。

首先，我们列出模型需要的算子：

```
namespace tflite {
namespace ops {
namespace micro {
TfLiteRegistration* Register_DEPTHWISE_CONV_2D();
TfLiteRegistration* Register_MAX_POOL_2D();
TfLiteRegistration* Register_CONV_2D();
TfLiteRegistration* Register_FULLY_CONNECTED();
TfLiteRegistration* Register_SOFTMAX();
} // namespace micro
} // namespace ops
} // namespace tflite
```

测试本身（像往常一样）以设置推断所需的所有内容并获取指向模型输入张量的指针作为开始：

```
// Set up logging
tflite::MicroErrorReporter micro_error_reporter;
tflite::ErrorReporter* 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.
const tflite::Model* model =
    ::tflite::GetModel(g_magic_wand_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);
```

```

}

static tflite::MicroMutableOpResolver micro_mutable_op_resolver;
micro_mutable_op_resolver.AddBuiltin(
    tflite::BuiltinOperator_DEPTHWISE_CONV_2D,
    tflite::ops::micro::Register_DEPTHWISE_CONV_2D());
micro_mutable_op_resolver.AddBuiltin(
    tflite::BuiltinOperator_MAX_POOL_2D,
    tflite::ops::micro::Register_MAX_POOL_2D());
micro_mutable_op_resolver.AddBuiltin(
    tflite::BuiltinOperator_CONV_2D,
    tflite::ops::micro::Register_CONV_2D());
micro_mutable_op_resolver.AddBuiltin(
    tflite::BuiltinOperator_FULLY_CONNECTED,
    tflite::ops::micro::Register_FULLY_CONNECTED());
micro_mutable_op_resolver.AddBuiltin(tflite::BuiltinOperator_SOFTMAX,
                                     tflite::ops::micro::Register_SOFTMAX());

// 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 = 60 * 1024;
uint8_t tensor_arena[tensor_arena_size];

// Build an interpreter to run the model with
tflite::MicroInterpreter interpreter(model, micro_mutable_op_resolver, tensor_arena,
                                    tensor_arena_size, error_reporter);

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

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

```

然后我们检查输入张量，以确保它的形状和我们预期的一样：

```

// Make sure the input has the properties we expect
TF_LITE_MICRO_EXPECT_NE(nullptr, input);
TF_LITE_MICRO_EXPECT_EQ(4, input->dims->size);
// The value of each element gives the length of the corresponding tensor.
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);
TF_LITE_MICRO_EXPECT_EQ(128, input->dims->data[1]);
TF_LITE_MICRO_EXPECT_EQ(3, input->dims->data[2]);
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[3]);
// The input is a 32 bit floating point value
TF_LITE_MICRO_EXPECT_EQ(kTfLiteFloat32, input->type);

```

我们输入的形状是 (1, 128, 3, 1)。第一维只是第二维的封装，包含 128 个三轴加速度传感器读数。每个读数都有三个值，分别对应每个轴的加速度，每个值都被封装在一个元素张量中。输入均为 32 位浮点值。

确认输入形状后，我们将一些数据写入输入张量：

```

// Provide an input value
const float* ring_features_data = g_circle_micro_f9643d42_nohash_4_data;
error_reporter->Report("%d", input->bytes);
for (int i = 0; i < (input->bytes / sizeof(float)); ++i) {
    input->data.f[i] = ring_features_data[i];
}

```

常量 `g_circle_micro_f9643d42_nohash_4_data` 定义在 `circle_micro_features_data.cc` 中，它包含一个浮点值数组，代表一个人正在做出画圆的手势。在 `for` 循环中，我们逐步遍历数据并将每个值写入输入。我们只写入输入张量可以容纳的浮点值。

接下来，我们以熟悉的方式运行推断：

```

// 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);

```

之后，我们检查输出张量，以确保它的形状和我们预期的一样：

```

// Obtain a pointer to the output tensor and make sure it has the
// properties we expect.
TfLiteTensor* output = interpreter.output(0);
TF_LITE_MICRO_EXPECT_EQ(2, output->dims->size);
TF_LITE_MICRO_EXPECT_EQ(1, output->dims->data[0]);
TF_LITE_MICRO_EXPECT_EQ(4, output->dims->data[1]);
TF_LITE_MICRO_EXPECT_EQ(kTfLiteFloat32, output->type);

```

输出张量是二维的，包含一个单元素封装和一组四个值，这些值表示四种类别（“画翅膀”“画圆”“画角”和未知手势）。每个值都是一个 32 位浮点数。

然后，我们可以测试我们的数据，以确保推断结果符合我们的预期。我们输入了“画圆”手势的数据，因此我们希望“画圆”得分最高：

```

// There are four possible classes in the output, each with a score.
const int kWingIndex = 0;
const int kRingIndex = 1;
const int kSlopeIndex = 2;
const int kNegativeIndex = 3;

// Make sure that the expected "Ring" score is higher than the other
// classes.
float wing_score = output->data.f[kWingIndex];
float ring_score = output->data.f[kRingIndex];
float slope_score = output->data.f[kSlopeIndex];
float negative_score = output->data.f[kNegativeIndex];
TF_LITE_MICRO_EXPECT_GT(ring_score, wing_score);
TF_LITE_MICRO_EXPECT_GT(ring_score, slope_score);
TF_LITE_MICRO_EXPECT_GT(ring_score, negative_score);

```

然后，我们使用“画角”手势的数据重复整个过程：

```
// Now test with a different input, from a recording of "Slope".
const float* slope_features_data = g_angle_micro_f2e59fea_nohash_1_data;
for (int i = 0; i < (input->bytes / sizeof(float)); ++i) {
    input->data.f[i] = slope_features_data[i];
}

// Run the model on this "Slope" input.
invoke_status = interpreter.Invoke();
if (invoke_status != kTfLiteOk) {
    error_reporter->Report("Invoke failed\n");
}
TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, invoke_status);

// Make sure that the expected "Slope" score is higher than the other classes.
wing_score = output->data.f[kWingIndex];
ring_score = output->data.f[kRingIndex];
slope_score = output->data.f[kSlopeIndex];
negative_score = output->data.f[kNegativeIndex];
TF_LITE_MICRO_EXPECT_GT(slope_score, wing_score);
TF_LITE_MICRO_EXPECT_GT(slope_score, ring_score);
TF_LITE_MICRO_EXPECT_GT(slope_score, negative_score);
```

就是这样！我们已经看到了如何对原始加速度传感器数据进行推断。如同之前的示例一样，由于不需要进行预处理，事情就变得简单很多。

使用以下命令运行测试：

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

11.3.2 加速度传感器处理程序

我们的下一个测试演示加速度传感器处理程序的接口。这个组件的任务是用加速度传感器的数据为推断填充输入张量。

由于这两部分都取决于设备上加速度传感器的工作原理，因此为每个单独的设备都提供了不同的加速度传感器处理程序的实现。

稍后我们将逐步介绍这些实现，但是现在，我们来看看位于 *accelerometer_handler_test.cc* 中的测试，它展示了如何调用处理程序。

第一个测试非常简单：

```
TF_LITE_MICRO_TEST(Setup) {
    static tflite::MicroErrorReporter micro_error_reporter;
    TfLiteStatus setup_status = SetupAccelerometer(&micro_error_reporter);
    TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, setup_status);
}
```

`SetupAccelerometer()` 函数执行一次性的设置，以便从加速度传感器获取数据。这个测试展示了如何（使用指向 `ErrorReporter` 的指针）调用这个函数。函数返回 `TfLiteStatus`，表示是否设置成功。

下一个测试展示了如何使用加速度传感器处理程序来数据填充输入张量：

```
TF_LITE_MICRO_TEST(TestAccelerometer) {
    float input[384] = {0.0};
    tflite::MicroErrorReporter micro_error_reporter;
    // Test that the function returns false before insufficient data is available
    bool inference_flag =
        ReadAccelerometer(&micro_error_reporter, input, 384, false);
    TF_LITE_MICRO_EXPECT_EQ(inference_flag, false);

    // Test that the function returns true once sufficient data is available to
    // fill the model's input buffer (128 sets of values)
    for (int i = 1; i <= 128; i++) {
        inference_flag =
            ReadAccelerometer(&micro_error_reporter, input, 384, false);
    }
    TF_LITE_MICRO_EXPECT_EQ(inference_flag, true);
}
```

首先，我们准备一个名为 `input` 的浮点 (`float`) 数组来模拟模型的输入张量。

由于有 128 个三维读数，所以它的总大小为 384 个浮点数 (128×3)，1536 字节 (384×4)。我们将数组中的每个值都初始化为 0.0。

然后，我们调用 `ReadAccelerometer()`。我们传入一个 `ErrorReporter` 实例，向其中 (`input`) 写入数据的数组以及要获取的数据总量 (384 个浮点数)。最后一个参数是一个布尔值，它告诉 `ReadAccelerometer()` 是否在读取更多数据之前清除缓冲区，因为我们需要在成功识别手势之后清除缓冲区。

调用 `ReadAccelerometer()` 后，函数尝试将 384 个浮点数的数据写入传递给它的数组。如果加速度传感器才刚刚开始收集数据，那么可能还没有完整可用的 384 个浮点数的数据。在这种情况下，该函数将不执行任何操作并返回 `false`。如果没有数据可用，我们可以使用它来避免运行推断。

加速度传感器处理程序的虚拟实现位于 `accelerometer_handler.cc`，它模拟每次调用时可获得另一个读数。通过额外的 127 次调用，我们确保它已经积累了足够的数据以开始返回 `true`。

请使用以下命令运行测试：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
      test_gesture_accelerometer_handler_test
```

11.3.3 手势预测程序

进行推断后，我们的输出张量将被概率填充，这些概率会告诉我们检测到了（如果有）某种手势的概率。但是，由于机器学习不是一门精确的科学，因此任何单一的推断都有可能导致误报。

为了减少误报的影响，我们可以规定，要识别出一个手势，必须至少在一定数量的连续推断中检测到这个手势。假设每秒进行多次推断，那么我们可以快速确定结果是否有效。这就是手势预测程序的工作。

手势预测程序定义了一个函数 `PredictGesture()`，该函数将模型的输出张量作为输入。为了确定是否检测到有效手势，该函数会做以下两件事情：

1. 检查手势的概率是否达到最小阈值。
2. 检查是否在一定数量的推断中都始终检测到该手势。

每种手势所需的小推断次数都不相同，因为执行某些手势可能会比其他手势花费更长的时间。考虑到速度更快的设备能够更频繁地运行推断，每个设备的阈值也会有所不同。为 SparkFun Edge 开发板调整的阈值位于 `constants.cc`：

```
const int kConsecutiveInferenceThresholds[3] = {15, 12, 10};
```

这些值的定义顺序与手势在模型的输出张量中出现的顺序相同。其他平台，例如 Arduino，都有该文件的特定于设备的版本，其中包含了根据平台自身性能调整的值。

让我们看一下 `gesture_predictor.cc` 中的代码，看看这些值是如何使用的。

首先，我们定义一些变量，这些变量用于跟踪最后看到的手势以及连续多少次推断检测到了相同的手势：

```
// How many times the most recent gesture has been matched in a row
int continuous_count = 0;
// The result of the last prediction
int last_predict = -1;
```

接下来，我们定义 `PredictGesture()` 函数，并确定在最近的推断中，是否有某种手势类别在每次推断中得到的概率都大于 0.8：

```
// Return the result of the last prediction
// 0: wing("W"), 1: ring("O"), 2: slope("angle"), 3: unknown
int PredictGesture(float* output) {
    // Find whichever output has a probability > 0.8 (they sum to 1)
    int this_predict = -1;
    for (int i = 0; i < 3; i++) {
        if (output[i] > 0.8) this_predict = i;
    }
}
```

我们使用 `this_predict` 变量来存储所预测手势的索引。

变量 `continuous_count` 用于跟踪连续预测最近发现的手势的次数。如果所有手势类别均小于概率阈值 0.8，则将 `continuous_count` 设置为 0，将 `last_predict` 设置为 3（“未知”类别的索引）来重置任何正在进行的检测过程，这表明最近的结果是未知手势：

```
// No gesture was detected above the threshold
if (this_predict == -1) {
    continuous_count = 0;
    last_predict = 3;
    return 3;
}
```

接下来，如果最新的预测与先前的预测一致，则递增 `continuous_count`。否则，将其重置为 0。我们还将最新的预测存储在 `last_predict` 中：

```
if (last_predict == this_predict) {
    continuous_count += 1;
} else {
    continuous_count = 0;
}
last_predict = this_predict;
```

在 `PredictGesture()` 的下一部分，我们使用 `continuous_count` 来检查当前手势的概率是否已达到阈值。如果还没有，我们将返回 3，表示未知手势：

```
// If we haven't yet had enough consecutive matches for this gesture,
// report a negative result
if (continuous_count < kConsecutiveInferenceThresholds[this_predict]) {
    return 3;
}
```

如果超过了阈值，则意味着我们已经确认检测到了有效手势。在这种情况下，我们重置所有变量：

```
// Otherwise, we've seen a positive result, so clear all our variables
// and report it
continuous_count = 0;
last_predict = -1;
return this_predict;
}
```

该函数最终返回当前预测值。预测值将通过主循环传递到输出处理程序中，输出处理程序负责将结果显示给用户。

手势预测程序的测试位于 `gesture_predictor_test.cc`。第一个测试验证一次成功的预测：

```

TF_LITE_MICRO_TEST(SuccessfulPrediction) {
    // Use the threshold from the 0th gesture
    int threshold = kConsecutiveInferenceThresholds[0];
    float probabilities[4] = {1.0, 0.0, 0.0, 0.0};
    int prediction;
    // Loop just too few times to trigger a prediction
    for (int i = 0; i <= threshold - 1; i++) {
        prediction = PredictGesture(probabilities);
        TF_LITE_MICRO_EXPECT_EQ(prediction, 3);
    }
    // Call once more, triggering a prediction
    // for category 0
    prediction = PredictGesture(probabilities);
    TF_LITE_MICRO_EXPECT_EQ(prediction, 0);
}

```

向 `PredictGesture()` 函数提供一组概率，这些概率强烈表明很可能匹配到了第一种类别。但是，直到以该概率出现 `threshold` 次之前，它都返回 3，表示“未知”结果。在该概率连续出现 `threshold` 次之后，它将返回类别 0 的肯定预测。

下一个测试展示如果一个类别连续得到很好的概率，但是被另一个类别的高概率中断，则会发生什么情况：

```

TF_LITE_MICRO_TEST(FailPartWayThere) {
    // Use the threshold from the 0th gesture
    int threshold = kConsecutiveInferenceThresholds[0];
    float probabilities[4] = {1.0, 0.0, 0.0, 0.0};
    int prediction;
    // Loop just too few times to trigger a prediction
    for (int i = 0; i <= threshold - 1; i++) {
        prediction = PredictGesture(probabilities);
        TF_LITE_MICRO_EXPECT_EQ(prediction, 3);
    }
    // Call with a different prediction, triggering a failure
    probabilities[0] = 0.0;
    probabilities[2] = 1.0;
    prediction = PredictGesture(probabilities);
    TF_LITE_MICRO_EXPECT_EQ(prediction, 3);
}

```

在这个例子中，我们输入类别 0 的一组连续的高概率值，但数量不足以满足次数阈值。然后，我们更改数据，以使类别 2 的概率最高，从而得到类别 3（表示“未知”手势）的预测结果。

最后一个测试展示了 `PredictGesture()` 如何忽略低于阈值的概率。在一个循环中，我们将准确输入正确数量的预测，以满足类别 0 的阈值。但是，尽管类别 0 的可能性最高，但由于概率为 0.7，低于 `PredictGesture()` 的内部阈值 0.8，因此这仍将得到类别 3，即“未知”的预测结果：

```

TF_LITE_MICRO_TEST(InsufficientProbability) {
    // Use the threshold from the 0th gesture
    int threshold = kConsecutiveInferenceThresholds[0];
    // Below the probability threshold of 0.8
    float probabilities[4] = {0.7, 0.0, 0.0, 0.0};
    int prediction;
    // Loop the exact right number of times
    for (int i = 0; i <= threshold; i++) {
        prediction = PredictGesture(probabilities);
        TF_LITE_MICRO_EXPECT_EQ(prediction, 3);
    }
}

```

使用以下命令运行测试：

```

make -f tensorflow/lite/micro/tools/make/Makefile \
    test_gesture_predictor_test

```

11.3.4 输出处理程序

输出处理程序非常简单，它只负责接收 `PredictGesture()` 返回的类别索引，并将结果显示给用户。它的测试在 `output_handler_test.cc` 中，展示了如何使用它的接口：

```

TF_LITE_MICRO_TEST(TestCallability) {
    tflite::MicroErrorReporter micro_error_reporter;
    tflite::ErrorReporter* error_reporter = &micro_error_reporter;
    HandleOutput(error_reporter, 0);
    HandleOutput(error_reporter, 1);
    HandleOutput(error_reporter, 2);
    HandleOutput(error_reporter, 3);
}

```

使用以下命令运行测试：

```

make -f tensorflow/lite/micro/tools/make/Makefile \
    test_gesture_output_handler_test

```

11.4 检测手势

所有这些组件都被包含在 `main_functions.cc` 中，这个文件包含了我们程序的核心逻辑。

首先，它设置通常的变量以及一些额外的变量：

```

namespace tflite {
namespace ops {
namespace micro {
TfLiteRegistration* Register_DEPTHWISE_CONV_2D();
TfLiteRegistration* Register_MAX_POOL_2D();
TfLiteRegistration* Register_CONV_2D();
TfLiteRegistration* Register_FULLY_CONNECTED();
TfLiteRegistration* Register_SOFTMAX();
}
}

```

```

} // namespace micro
} // namespace ops
} // namespace tflite

// Globals, used for compatibility with Arduino-style sketches.
namespace {
tflite::ErrorReporter* error_reporter = nullptr;
const tflite::Model* model = nullptr;
tflite::MicroInterpreter* interpreter = nullptr;
TfLiteTensor* model_input = nullptr;
int input_length;

// Create an area of memory to use for input, output, and intermediate arrays.
// The size of this will depend on the model you're using, and may need to be
// determined by experimentation.
constexpr int kTensorArenaSize = 60 * 1024;
uint8_t tensor_arena[kTensorArenaSize];

// Whether we should clear the buffer next time we fetch data
bool should_clear_buffer = false;
} // namespace

```

`input_length` 变量存储模型的输入张量的长度，`should_clear_buffer` 变量标志是否应该在下次运行时清除加速度传感器处理程序的缓冲区。程序会在成功检测到某种手势之后清除缓冲区，以便为后续的推断提供更清晰的候选数据。

接下来，`setup()` 函数执行所有常规的设置，以便准备好运行推断：

```

void setup() {
    // Set up logging. Google style is to avoid globals or statics because of
    // lifetime uncertainty, but since this has a trivial destructor it's okay.
    static tflite::MicroErrorReporter micro_error_reporter; //NOLINT
    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_magic_wand_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;
    }

    // Pull in only the operation implementations we need.
    // This relies on a complete list of all the ops needed by this graph.
    // An easier approach is to just use the AllOpsResolver, but this will
    // incur some penalty in code space for op implementations that are not
    // needed by this graph.
    static tflite::MicroMutableOpResolver micro_mutable_op_resolver; // NOLINT
    micro_mutable_op_resolver.AddBuiltin(
        tflite::BuiltInOperator_DEPTHWISE_CONV_2D,

```

```

    tflite::ops::micro::Register_DEPTHWISE_CONV_2D());
micro MutableOpResolver.AddBuiltin(
    tflite::BuiltinOperator_MAX_POOL_2D,
    tflite::ops::micro::Register_MAX_POOL_2D());
micro MutableOpResolver.AddBuiltin(
    tflite::BuiltinOperator_CONV_2D,
    tflite::ops::micro::Register_CONV_2D());
micro MutableOpResolver.AddBuiltin(
    tflite::BuiltinOperator_FULLY_CONNECTED,
    tflite::ops::micro::Register_FULLY_CONNECTED());
micro MutableOpResolver.AddBuiltin(tflite::BuiltinOperator_SOFTMAX,
                                    tflite::ops::micro::Register_SOFTMAX());

// Build an interpreter to run the model with
static tflite::MicroInterpreter static_interpreter(model,
                                                 micro MutableOpResolver,
                                                 tensor Arena,
                                                 kTensorArenaSize,
                                                 error Reporter);

interpreter = &static_interpreter;

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

// Obtain pointer to the model's input tensor
model_input = interpreter->input(0);
if ((model_input->dims->size != 4) || (model_input->dims->data[0] != 1) ||
    (model_input->dims->data[1] != 128) ||
    (model_input->dims->data[2] != kChannelNumber) ||
    (model_input->type != kTfLiteFloat32)) {
    error_reporter->Report("Bad input tensor parameters in model");
    return;
}

input_length = model_input->bytes / sizeof(float);
TfLiteStatus setup_status = SetupAccelerometer(error_reporter);
if (setup_status != kTfLiteOk) {
    error_reporter->Report("Set up failed\n");
}

```

更有趣的事情发生在 `loop()` 函数中，它依然很简单：

```
void loop() {
    // Attempt to read new data from the accelerometer
    bool got_data = ReadAccelerometer(error_reporter, model_input->data.f,
                                       input_length, should_clear_buffer);
    // Don't try to clear the buffer again
    should_clear_buffer = false;
    // If there was no new data, wait until next time
    if (!got_data) return;
    // Run inference, and report any error
    TfLiteStatus invoke_status = interpreter->Invoke();
```

```

if (invoke_status != kTfLiteOk) {
    error_reporter->Report("Invoke failed on index: %d\n", begin_index);
    return;
}
// Analyze the results to obtain a prediction
int gesture_index = PredictGesture(interpreter->output(0)->data.f);
// Clear the buffer next time we read data
should_clear_buffer = gesture_index < 3;
// Produce an output
HandleOutput(error_reporter, gesture_index);
}

```

首先，我们尝试从加速度传感器读取一些值。读取到数据之后，我们将 `should_clear_buffer` 设置为 `false`，以确保暂时不会清除缓冲区。

如果获取新的数据失败，`ReadAccelerometer()` 会返回 `false`，我们将从 `loop()` 函数返回，以便下次调用该函数时可以再重新尝试。

如果 `ReadAccelerometer()` 返回 `true`，我们将对新填充的输入张量运行推断。我们将结果传递给 `PredictGesture()`，它将返回检测到的手势的索引。如果索引小于 3，则表示检测到了有效的手势，因此我们会设置 `should_clear_buffer` 标志为 `true`，以便在下次调用 `ReadAccelerometer()` 时清除缓冲区。然后，我们调用 `HandleOutput()` 将结果报告给用户。

在 `main.cc` 中，`main()` 函数负责启动我们的程序，运行 `setup()`，并在循环中调用 `loop()` 函数：

```

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

```

这些是全部内容了！请使用以下命令在你的开发计算机上编译该程序：

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

然后请使用以下命令运行该程序：

```
./tensorflow/lite/micro/tools/make/gen/osx_x86_64/bin/magic_wand
```

由于没有可用的加速度数据，所以程序不会产生任何输出，但是你可以确认它是否可以正常编译并运行。

接下来，我们将遍历各个平台的代码，这些代码负责捕获加速度传感器数据并产生输出。我们还将展示如何部署和运行应用程序。

11.5 部署到微处理器

在本节中，我们将会把代码部署到两种设备上：

- Arduino Nano 33 BLE Sense
- SparkFun Edge

让我们先来看看 Arduino 的实现。

11.5.1 Arduino

Arduino Nano 33 BLE Sense 开发板带有一个三轴加速度传感器而且支持蓝牙，它体积小、重量轻，是制造魔杖的理想选择。

蓝牙

本章的实现并没有演示如何使用蓝牙，但 Arduino 提供了一个带有示例代码的库，你可以参考这些代码来创建自己的实现。你可以在本节最后找到更详细的信息。

自本书出版以来，示例中也可能会添加蓝牙支持。请在 TensorFlow 代码仓库中检查最新版本。

让我们逐步介绍特定于 Arduino 的实现的一些关键文件。

Arduino 常量

文件 *arduino/constants.cc* 重新定义了常量 `kConsecutiveInferenceThresholds`：

```
// The number of expected consecutive inferences for each gesture type.  
// Established with the Arduino Nano 33 BLE Sense.  
const int kConsecutiveInferenceThresholds[3] = {8, 5, 4};
```

如本章前面所述，此常数存储了某种手势要被视为“检测到”所需的连续肯定推断的数量。这个数量取决于每秒能运行多少次推断，具体取决于每个设备。因为默认数量是针对 SparkFun Edge 开发板进行校准的，所以 Arduino 的实现需要一组自己的数量。你可以修改这些阈值使推断更难或更容易被触发，但是将它们设置得太低会导致很容易误报。

在 Arduino 上捕获加速度传感器数据

Arduino 加速度传感器处理程序位于 *arduino/accelerometer_handler.cc*。它的任务是从加速度传感器捕获数据并将数据写入模型的输入缓冲区。

我们使用的模型是通过 SparkFun Edge 开发板的数据训练的。Edge 开发板的加速度传感器以 25Hz (即每秒 25 次) 的速率采集数据。为了使模型可以很好地工作，我们需要以相同速率捕获数据。事实证明，Arduino Nano 33 BLE Sense 开发板上的加速度传感器以 119Hz 的速率采集数据。这意味着，除了捕获数据之外，我们还需要对数据进行降采样 (downsample) 以使其适用于我们的模型。

降采样虽然听起来很专业，但实际上非常简单。为了降低信号的采样率，我们可以丢弃一些数据。让我们在下面的代码中看一下它是如何工作的。

首先，我们的实现包含了自己的头文件以及其他一些头文件：

```
#include "tensorflow/lite/micro/examples/magic_wand/
accelerometer_handler.h"

#include <Arduino.h>
#include <Arduino_LSM9DS1.h>

#include "tensorflow/lite/micro/examples/magic_wand/constants.h"
```

Arduino.h 头文件提供对 Arduino 平台的一些基本功能的访问。文件 *Arduino_LSM9DS1.h* 是 *Arduino_LSM9DS1* 库的一部分，我们将使它与开发板的加速度传感器进行通信。

接下来，我们设置一些变量：

```
// A buffer holding the last 200 sets of 3-channel values
float save_data[600] = {0.0};
// Most recent position in the save_data buffer
int begin_index = 0;
// True if there is not yet enough data to run inference
bool pending_initial_data = true;
// How often we should save a measurement during downsampling
int sample_every_n;
// The number of measurements since we last saved one
int sample_skip_counter = 1;
```

这些变量包括一个缓存区 *save_data* (用于保存我们采集的数据) 以及一些其他变量 (用于跟踪我们当前在缓冲区的位置以及我们是否有足够的数据来开始运行推断)。降采样过程中使用了两个非常有趣的变量 *sample_every_n* 和 *sample_skip_counter*。我们稍后会详细介绍它们。

在文件中的下一步，程序的主循环调用 *SetupAccelerometer()* 函数，以使开发板准备好捕获数据：

```
TfLiteStatus SetupAccelerometer(tflite::ErrorReporter* error_reporter) {
    // Wait until we know the serial port is ready
    while (!Serial) {
    }
```

```

// Switch on the IMU
if (!IMU.begin()) {
    error_reporter->Report("Failed to initialize IMU");
    return kTfLiteError;
}

```

由于我们将输出一条消息以指示一切准备就绪，所以首先要做的是确保设备的串口已准备好了。然后，它打开惯性测量单元（Inertial Measurement Unit, IMU）^{译注1}，即包含加速度传感器的电子组件。IMU 对象来自 Arduino_LSM9DS1 库。

下一步是开始考虑降采样。首先我们查询 IMU 库，以确定开发板的实际采样率。有了这个数字，我们将它除以目标采样率，即在 *constants.h* 中定义的 *kTargetHz*：

```

// Determine how many measurements to keep in order to
// meet kTargetHz
float sample_rate = IMU.accelerationSampleRate();
sample_every_n = static_cast<int>(roundf(sample_rate / kTargetHz));

```

我们的目标频率为 25Hz，开发板的采样率为 119Hz，因此，两数相除得到结果 4.76。这让我们知道要达到 25Hz 的目标采样率需要保留 119Hz 采样中的多少个采样：每 4.76 个采样保留一个。

由于小数样本很难保留，因此我们使用 *roundf()* 函数将其四舍五入得到最接近的整数数字 5。即要对信号进行降采样，需要在每五个测量数据中选择保留一个。这将得到 23.8Hz 的实际有效采样率，这个近似值足以使我们的模型很好地工作。我们将此值存储在 *sample_every_n* 变量中，以供后续使用。

现在，我们已经得到了降采样的参数，我们向用户显示一条消息，通知他应用程序已准备就绪，然后从 *SetupAccelerometer()* 函数返回：

```

error_reporter->Report("Magic starts!");

return kTfLiteOk;
}

```

接下来，我们定义 *ReadAccelerometer()*。这个函数的任务是捕获新的数据并将数据写入模型的输出张量。函数的开始会在成功识别手势之后清除内部缓冲区，继而为后续手势识别提供更干净的候选数据：

```

bool ReadAccelerometer(tflite::ErrorReporter* error_reporter, float* input,
                      int length, bool reset_buffer) {
    // Clear the buffer if required, e.g. after a successful prediction
    if (reset_buffer) {

```

^{译注1}：惯性测量单元是测量物体三轴姿态角（或角速率）以及加速度的装置，即前文提到的加速度传感器。

```
    memset(save_data, 0, 600 * sizeof(float));
    begin_index = 0;
    pending_initial_data = true;
}
```

接下来，我们使用 IMU 库在循环中检查是否有可用数据。如果有可用数据，就读取数据：

```
// Keep track of whether we stored any new data
bool new_data = false;
// Loop through new samples and add to buffer
while (IMU.accelerationAvailable()) {
    float x, y, z;
    // Read each sample, removing it from the device's FIFO buffer
    if (!IMU.readAcceleration(x, y, z)) {
        error_reporter->Report("Failed to read data");
        break;
}
```

Arduino Nano 33 BLE Sense 开发板上的加速度传感器配备了 FIFO 缓冲区（FIFO buffer）^{译注 2}。这是一个特殊的内存缓冲区，位于加速度传感器，它会保存最近的 32 个测量数据。因为它是加速度传感器硬件的一部分，所以即使在我们的应用程序代码运行时，FIFO 缓冲区仍会持续产生数据并保存测量结果。如果不使用 FIFO 缓冲区，我们可能会丢失很多数据，也很难准确地记录正在做的手势。

调用 `IMU.accelerationAvailable()` 即表示查询加速度传感器，以确认其 FIFO 缓冲区中是否有新数据可用。使用循环，我们将从缓冲区中读取所有数据，直到没有数据可用为止。

接下来，我们来实现非常简单的降采样算法：

```
// Throw away this sample unless it's the nth
if (sample_skip_counter != sample_every_n) {
    sample_skip_counter += 1;
    continue;
}
```

我们的方法是在每 n 个样本中保留一个，其中 n 是存储在 `sample_every_n` 中的数字。为此，我们维护一个计数器 `sample_skip_counter`，它使我们知道自上次保存样本以来已读取了多少个样本。对于我们读取的每个测量值，我们都会检查它是否为第 n 个样本。如果不是，我们将继续循环而不会写入数据，实际上就是丢弃数据。这个简单的过程将使得我们的数据被降采样。

如果读取到第 n 个数据，则保留数据。为此，我们将数据写入 `save_data` 缓冲区中的连续位置：

译注 2：FIFO，First Input First Output，即先进先出。

```
// Write samples to our buffer, converting to milli-Gs
// and flipping y and x order for compatibility with
// model (sensor orientation is different on Arduino
// Nano BLE Sense compared with SparkFun Edge)
save_data[begin_index++] = y * 1000;
save_data[begin_index++] = x * 1000;
save_data[begin_index++] = z * 1000;
```

我们的模型接受以 x 、 y 、 z 的顺序排列的加速度传感器数据。你会注意到，这里我们正在将 y 值写入 x 之前的缓冲区。这是因为我们的模型是根据在 SparkFun Edge 开发板上采集的数据进行训练的，SparkFun Edge 开发板上的加速度传感器的轴的物理指向与 Arduino 开发板上的不同。这种差异意味着 SparkFun Edge 开发板上的 x 轴与 Arduino 开发板上的 y 轴相等，反之亦然。通过在代码中交换这些轴的数据，我们可以确保向模型提供了它可以理解的数据。

循环的最后几行设置了一些状态变量：

```
// Since we took a sample, reset the skip counter
sample_skip_counter = 1;
// If we reached the end of the circle buffer, reset
if (begin_index >= 600) {
    begin_index = 0;
}
new_data = true;
}
```

我们重置了降采样计数器，确保缓冲区不会溢出，并设置了一个标志 (`new_data`) 以指示新数据已保存。

在获取新数据之后，我们还要进行更多的检查。这次，我们要确保有足够的数据来进行推断。如果没有，或者这次没有捕获到新数据，我们就将从函数中返回，不做任何事情：

```
// Skip this round if data is not ready yet
if (!new_data) {
    return false;
}

// Check if we are ready for prediction or still pending more initial data
if (pending_initial_data && begin_index >= 200) {
    pending_initial_data = false;
}

// Return if we don't have enough data
if (pending_initial_data) {
    return false;
}
```

通过在没有新数据时返回 `false`，我们可以确保调用函数知道此时不需要运行推断。

到目前为止，我们已经获得了一些新的数据。我们将适当数量的数据（包括新的样本）复制到输入张量中：

```
// Copy the requested number of bytes to the provided input tensor
for (int i = 0; i < length; ++i) {
    int ring_array_index = begin_index + i - length;
    if (ring_array_index < 0) {
        ring_array_index += 600;
    }
    input[i] = save_data[ring_array_index];
}

return true;
}
```

就是这些！我们已经填充了输入张量，并准备运行推断。在运行推断之后，结果将被传递到手势预测程序中，手势预测程序会根据它确定是否已识别到有效手势，然后将结果将传递到输出处理程序中，接下来我们将介绍输出处理程序。

在 Arduino 上响应手势

输出处理程序位于 *arduino/output_handler.cc*。它很好也很简单：它所做的只是将检测到的手势信息记录到串口，并在每次运行推断时切换开发板的 LED。

函数在第一次运行时，为输出配置 LED：

```
void HandleOutput(tflite::ErrorReporter* error_reporter, int kind) {
    // The first time this method runs, set up our LED
    static bool is_initialized = false;
    if (!is_initialized) {
        pinMode(LED_BUILTIN, OUTPUT);
        is_initialized = true;
    }
}
```

接下来，LED 会根据每次推断的结果被打开或者关闭：

```
// Toggle the LED every time an inference is performed
static int count = 0;
++count;
if (count & 1) {
    digitalWrite(LED_BUILTIN, HIGH);
} else {
    digitalWrite(LED_BUILTIN, LOW);
}
```

最后，我们根据匹配到的手势，打印出一些漂亮的 ASCII 字符画：

```
// Print some ASCII art for each gesture
if (kind == 0) {
    error_reporter->Report(
        "WING:\n\r*      *\n\r* *      "
    )
}
```

这段代码现在看起来可能有些令人困惑，但是当你将应用程序部署到开发板上时，你就能理解这些字符画的美妙了！

运行示例

要部署当前的示例，我们需要：

- Arduino Nano 33 BLE Sense 开发板。
 - micro-USB 线。
 - Arduino IDE。



与撰写本书时相比，编译过程有可能会发生变化，因此请查看 *README.md* 以获取最新说明。

本书中的项目可以作为 TensorFlow Lite Arduino 库中的示例代码获得。如果你还没有安装此库，可以打开 Arduino IDE 并从“Tools”（工具）菜单中选择“Manage Libraries”（管理库）。在出现的窗口中，搜索并安装名为 Arduino_TensorFlowLite 的库。你应该能够使用最新的版本，但是如果你遇到问题，也可以尝试本书的版本，即 1.14-ALPHA。



你还可以从 TensorFlow Lite 团队网站下载 Arduino_TensorFlowLite 库，然后通过 *.zip* 文件安装，也可以使用 TensorFlow Lite for Microcontrollers Makefile 自行生成。如果你想这样做，请参阅附录 A。

安装库后，`magic_wind` 示例将出现在“Examples”（示例）→“Arduino_TensorFlowLite”下的“File”（文件）菜单中，如图 11-5 所示。

单击“magic_wand”以加载示例。它将显示一个新的窗口，每个源文件都有一个选项卡。第一个选项卡中的 *magic_wind* 文件相当于我们的 *main_functions.cc*。我们在之前已经介绍了它的内容。



图 11-5：“Examples”菜单



6.2.2 节已经解释了 Arduino 示例的结构，因此这里不再赘述。

除了 TensorFlow 库之外，我们还需要安装并修改 Arduino_LSM9DS1 库。默认情况下，Arduino_LSM9DS1 库没有启用示例所需的 FIFO 缓冲区，因此我们必须对其代码进行一些修改。

在 Arduino IDE 中，选择“Tools”（工具）→“Manage Libraries”（管理库），然后搜索 Arduino_LSM9DS1。为确保以下的说明有效，你需要安装 1.0.0 版的驱动程序。



当你阅读本章时，驱动程序可能已修复。你可以在 *README.md* 中找到最新的部署说明。

驱动程序将被安装到 *Arduino/libraries* 目录，位于子目录 *Arduino_LSM9DS1*。

打开驱动程序源文件 *Arduino_LSM9DS1/src/LSM9DS1.cpp*，然后跳转到名为 *LSM9DS1Class::begin()* 的函数。在函数的末尾，在 *return 1* 语句之前插入以下行：

```

// Enable FIFO (see docs https://www.st.com/resource/en/datasheet/DM00103319.pdf)
// writeRegister(LSM9DS1_ADDRESS, 0x23, 0x02);
// Set continuous mode
writeRegister(LSM9DS1_ADDRESS, 0x2E, 0x00);

```

接下来，找到名为 `LSM9DS1Class::accelerationAvailable()` 的函数。你将看到以下几行：

```

if (readRegister(LSM9DS1_ADDRESS, LSM9DS1_STATUS_REG) & 0x01) {
    return 1;
}

```

注释掉这几行，然后将其替换为以下内容：

```

// Read FIFO_SRC. If any of the rightmost 8 bits have a value, there is data.
if (readRegister(LSM9DS1_ADDRESS, 0x2F) & 63) {
    return 1;
}

```

保存文件。程序就修改完成了！

要运行示例，请通过 USB 插入 Arduino 设备。在“Tools”（工具）菜单中，确保从“Board”（开发板）下拉列表中选择了正确的设备类型，如图 11-6 所示。

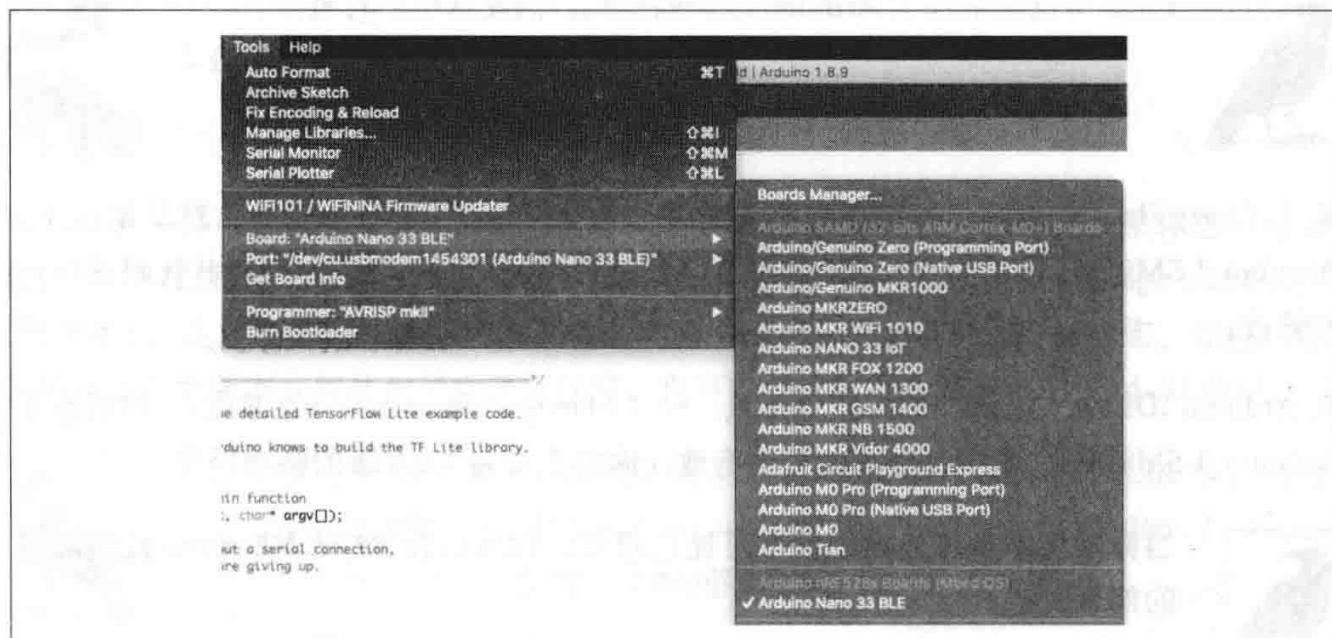


图 11-6：“Board”下拉列表

如果你的设备名称没有显示在列表中，则需要安装它的支持包。

为此，请单击“Boards Manager”（开发板管理器），然后在出现的窗口中搜索你的设备并安装相应支持包的最新版本。

接下来，确保在“Port”（端口）下拉列表中以及在“Tools”（工具）菜单中选择了设备的端口，如图 11-7 所示。

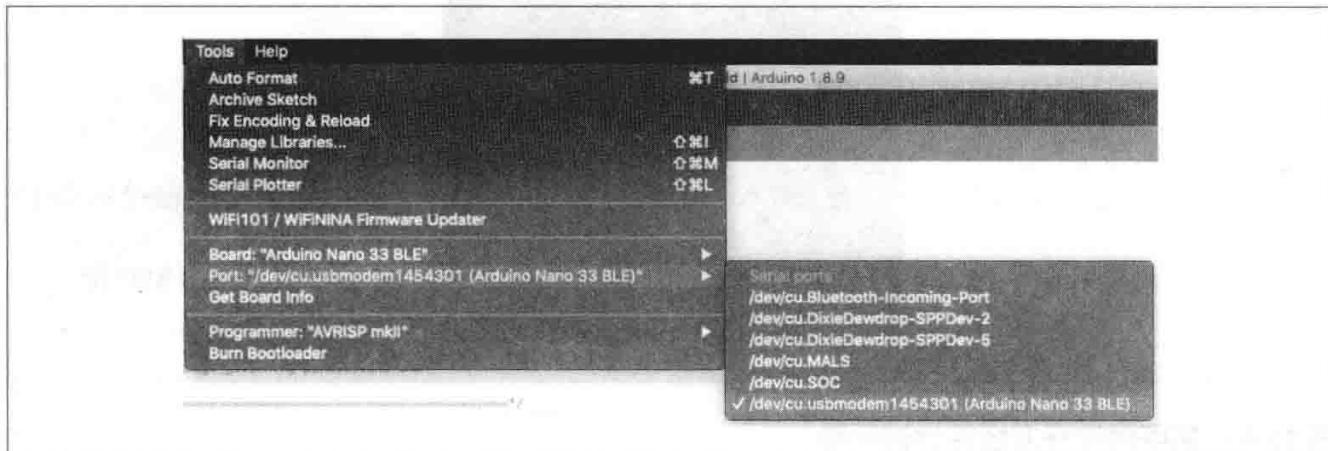


图 11-7：“Port”下拉列表

最后，单击 Arduino 窗口中的上传按钮（图 11-8 中以白色高亮的按钮），以编译代码并将其上传到 Arduino 设备。

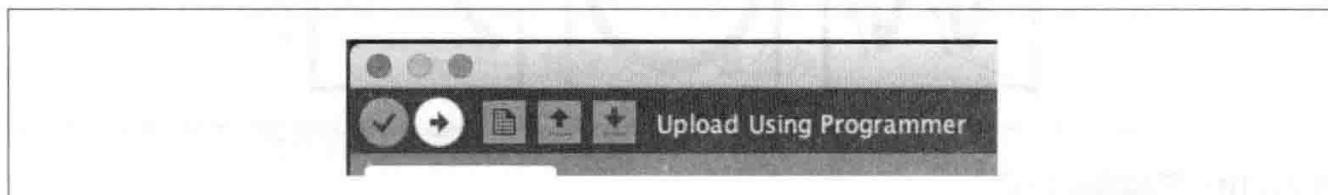


图 11-8：上传按钮

上传成功后，你应该看到 Arduino 板上的 LED 开始闪烁。

要尝试一些手势，请在“Tools”（工具）菜单中选择“Serial Monitor”（串口监视器）。你刚开始应该会看到以下输出：

Magic starts!

现在你可以尝试做出一些手势。如图 11-9 所示，用一只手拿着开发板，组件朝上，USB 适配器朝左。

图 11-10 展示了如何执行每个手势的示意图。由于我们的模型是根据开发板与魔杖相连时收集的数据进行训练的，所以你可能需要几次尝试才能检测到结果。

最简单的一个是“画翅膀”。你的手要移动得足够快，大约在一秒内完成这个动作。如果你成功了，你应该会看到以下输出，且红色的 LED 会被点亮：

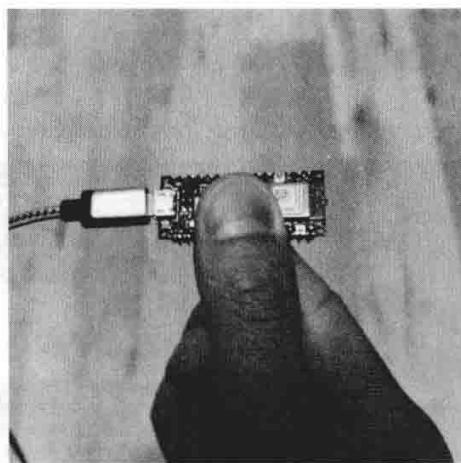


图 11-9：如何握住开发板并尝试手势

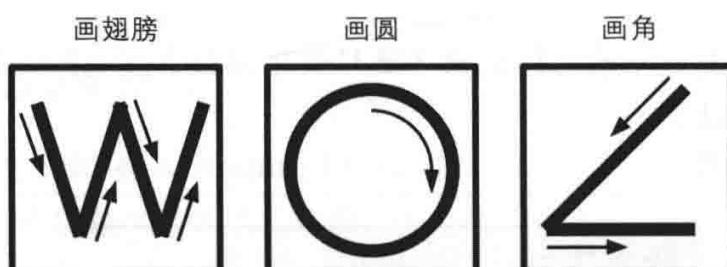


图 11-10：三种魔杖手势

WING:

```
*          *          *
*          * *          *
*          * *          *
*          * *          *
* *          * *
*          *
```

恭喜，你已使用 Arduino 施展出了第一个“魔法咒语”！



此时，你可以选择更有创造性的方法，将开发板放在魔杖的顶部，即距离你的手最远的地方。任何棍棒、直尺或其他长度约 1 英尺（约 30 厘米）的家用物品都可以。

确保设备连接牢固且方向一致，即组件朝上，USB 适配器朝左。并选择一根坚硬而不柔韧的魔杖——任何抖动都会影响加速度传感器的读数。

接下来，尝试“画圆”手势，用手（或魔杖的尖端）画一个顺时针的圆圈。同样，目标是花一秒左右的时间来执行手势。你应该看到以下内容出现，就像是魔术一样：

RING:

最后一个动作是在空中画一个三角形的角，如下面的 ASCII 字符画所示：

SLOPE:

* * * * *

像任何“魔咒”一样，你可能必须先练习一下，然后才能保证每次都能完美地执行它们。你可以在 *README.md* 中看到有关这些手势的视频演示。

如果失败了怎么办

以下是一些可能遇到的问题以及调试方法：

问题：LED 不亮。

解决方案：尝试按 reset 按钮或从 USB 断开开发板，然后重新连接。如果这些都不起作用，请尝试重新烧录开发板。

问题：LED卡在点亮或熄灭状态。

解决方案：正常情况下，LED 会在推断后立即停止闪烁，而程序会等待有足够的新数据可用才重新开始闪烁。如果 LED 停止闪烁超过几秒，则表明程序可能已经崩溃。在这种情况下，请尝试按 reset 按键。

问题：手势不起任何作用。

解决方案：首先，请确保 LED 闪烁，这表明正在进行推断。如果没有，请按 reset 按键。接下来，请确保你按照正确的方向拿着开发板。

要学习手势，请从最容易掌握的“画翅膀”开始。“画圆”要难一点，因为你需要保证画出的圆非常平滑。“角度”手势最棘手。尝试观看 *README.md* 中的视频以获取指导。

尝试修改

现在你已经部署了基本的应用程序，可以试着修改一下代码，做一些有意思的尝试。只需在 Arduino IDE 中编辑文件并保存，然后重复前面的说明，将修改后的代码部署到设备上。

以下是一些你可以尝试的事情：

- 尝试修改 *arduino/constants.cc* 中的阈值，以使手势更容易或更难执行（以更多的误报或者漏检为代价）。
- 在计算机上编写一个程序，让你可以使用手势执行某些任务。
- 扩展程序以通过蓝牙传输检测结果。ArduinoBLE 库中包含一些演示如何使用蓝牙的示例，你可以通过 Arduino IDE 下载该示例。

11.5.2 SparkFun Edge

SparkFun Edge 具有一个三轴加速度传感器、一个电池底座和蓝牙支持。由于可以无线操作，这使得它非常适合用来制作魔杖。

蓝牙

本章的实现并没有演示如何使用蓝牙，但 Ambiq SDK 提供如何使用蓝牙的示例。你可以在本节最后找到更详细的信息。

自本书出版以来，示例中也可能已经添加了蓝牙支持。请在 TensorFlow 代码仓库中查看最新版本。

在 SparkFun Edge 开发板上采集加速度数据

采集加速度传感器数据的代码位于 *sparkfun_edge/accelerometer_handler.cc*。其中很多代码是特定于设备的，我们将跳过这些细节，将重点放在重要的逻辑内容上。

采集加速度传感器数据涉及的第一步是配置硬件。`SetupAccelerometer()` 函数通过设置加速度传感器所需的各种低级参数来启动此功能：

```
TfLiteStatus SetupAccelerometer(tfLite::ErrorReporter* error_reporter) {
    // Set the clock frequency.
    am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_SYSCLK_MAX, 0);

    // Set the default cache configuration
    am_hal_cachectrl_config(&am_hal_cachectrl_defaults);
    am_hal_cachectrl_enable();
```

```
// Configure the board for low power operation.  
am_bsp_low_power_init();  
  
// Collecting data at 25Hz.  
int accInitRes = initAccelerometer();
```

你会注意到这里调用了名为 `initAccelerometer()` 的函数。这是在 SparkFun Edge BSP 的加速度传感器示例中定义的，在创建我们的项目时被选为依赖项。它执行各种任务来打开和配置开发板的加速度传感器。

在加速度传感器开始运行后，我们启用 FIFO 缓冲区。这是一个特殊的存储缓冲区，位于加速度传感器之中，可容纳最后采集的 32 个数据。通过启用它，我们能够在应用程序代码忙于推断的同时，继续收集加速度传感器的测量值。函数的其余部分设置了缓冲区并记录（如果发生了）错误：

```
// Enable the accelerometer's FIFO buffer.  
// Note: LIS2DH12 has a FIFO buffer which holds up to 32 data entries. It  
// accumulates data while the CPU is busy. Old data will be overwritten if  
// it's not fetched in time, so we need to make sure that model inference is  
// faster than 1/25Hz * 32 = 1.28s  
if (lis2dh12_fifo_set(&dev_ctx, 1)) {  
    error_reporter->Report("Failed to enable FIFO buffer.");  
}  
  
if (lis2dh12_fifo_mode_set(&dev_ctx, LIS2DH12_BYPASS_MODE)) {  
    error_reporter->Report("Failed to clear FIFO buffer.");  
    return 0;  
}  
  
if (lis2dh12_fifo_mode_set(&dev_ctx, LIS2DH12_DYNAMIC_STREAM_MODE)) {  
    error_reporter->Report("Failed to set streaming mode.");  
    return 0;  
}  
  
error_reporter->Report("Magic starts!");  
return kTfLiteOk;  
}
```

初始化完成后，我们可以调用 `ReadAccelerometer()` 函数来获取最新的数据。这将在每次推断之间运行。

首先，如果 `reset_buffer` 参数为 `true`，则 `ReadAccelerometer()` 会重置其数据缓冲区。这是在检测到有效手势之后做的，以便为其他手势提供干净的状态。作为此过程的一部分，我们调用 `am_util_delay_ms()` 函数使代码等待 10 毫秒。如果没有这 10 毫秒的延迟，代码在读取新数据时通常会挂起（截至撰写本书时，原因尚不清楚，但如果你确定有更好的解决方案，欢迎向 TensorFlow 开源项目贡献代码）：

```
bool ReadAccelerometer(tflite::ErrorReporter* error_reporter, float* input,  
                      int length, bool reset_buffer) {
```

```

// Clear the buffer if required, e.g. after a successful prediction
if (reset_buffer) {
    memset(save_data, 0, 600 * sizeof(float));
    begin_index = 0;
    pending_initial_data = true;
    // Wait 10ms after a reset to avoid hang
    am_util_delay_ms(10);
}

```

重置主缓冲区后，`ReadAccelerometer()`检查加速度传感器的 FIFO 缓冲区中是否有任何新的数据可用。如果尚无可用的数据，我们将从函数中返回：

```

// Check FIFO buffer for new samples
lis2dh12_fifo_src_reg_t status;
if (lis2dh12_fifo_status_get(&dev_ctx, &status)) {
    error_reporter->Report("Failed to get FIFO status.");
    return false;
}

int samples = status.fss;
if (status.ovrn_fifo) {
    samples++;
}

// Skip this round if data is not ready yet
if (samples == 0) {
    return false;
}

```

应用程序的主循环将继续调用 `loop()`，这意味着一旦有可用的数据，就会继续向下执行。

函数的下一部分将遍历新的数据，并将数据存储在另一个更大的缓冲区中。首先，我们设置了类型为 `axis3bit16_t` 的特殊结构体，用于保存加速度传感器数据。然后，我们调用 `lis2dh12_acceleration_raw_get()` 来填充下一个可用的测量数据。如果失败了，函数将返回 0，这时我们会显示一个错误：

```

// Load data from FIFO buffer
axis3bit16_t data_raw_acceleration;
for (int i = 0; i < samples; i++) {
    // Zero out the struct that holds raw accelerometer data
    memset(data_raw_acceleration.u8bit, 0x00, 3 * sizeof(int16_t));
    // If the return value is non-zero, sensor data was successfully read
    if (lis2dh12_acceleration_raw_get(&dev_ctx, data_raw_acceleration.u8bit)) {
        error_reporter->Report("Failed to get raw data.");
    }
}

```

如果成功采集到测量数据，我们会将其转换为模型期望的测量单位 milli-Gs，然后将数据写入 `save_data[]`，这是一个数组，我们使用它作为缓冲区来存储将用于推断的值。加速度传感器各个轴的数据都是连续存储的：

```

} else {
    // Convert each raw 16-bit value into floating point values representing
    // milli-Gs, a unit of acceleration, and store in the current position of
    // our buffer
    save_data[begin_index++] =
        lis2dh12_from_fs2_hr_to_mg(data_raw_acceleration.i16bit[0]);
    save_data[begin_index++] =
        lis2dh12_from_fs2_hr_to_mg(data_raw_acceleration.i16bit[1]);
    save_data[begin_index++] =
        lis2dh12_from_fs2_hr_to_mg(data_raw_acceleration.i16bit[2]);
    // Start from beginning, imitating loop array.
    if (begin_index >= 600) begin_index = 0;
}
}

```

`save_data[]` 数组可以存储 200 组三个轴的数值，因此当 `begin_index` 计数器的值达到 600 时，我们会将它重置为 0。

现在，我们将所有新数据合并到了 `save_data[]` 缓冲区中。接下来，我们检查是否有足够的数据来进行预测。在测试模型时，我们发现缓冲区总大小的三分之一左右是提供可靠的预测的最小数据量。因此，如果我们有至少这么多的数据，就将 `pending_initial_data` 标志设置为 `false`（默认为 `true`）：

```

// Check if we are ready for prediction or still pending more initial data
if (pending_initial_data && begin_index >= 200) {
    pending_initial_data = false;
}

```

接下来，如果仍然没有足够的数据来运行推断，就返回 `false`：

```

// Return if we don't have enough data
if (pending_initial_data) {
    return false;
}

```

到此为止，缓冲区中已经有了足够的数据可以运行推断。函数的最后一部分是将请求的数据从缓冲区复制到输入参数中，该参数是指向模型输入张量的指针：

```

// Copy the requested number of bytes to the provided input tensor
for (int i = 0; i < length; ++i) {
    int ring_array_index = begin_index + i - length;
    if (ring_array_index < 0) {
        ring_array_index += 600;
    }
    input[i] = save_data[ring_array_index];
}
return true;

```

参数 `length` 是传递给 `ReadAccelerometer()` 的参数，它决定应该复制多少数据。

由于我们的模型将 128 组三轴读数作为输入，所以 `main_functions.cc` 中调用 `ReadAccelerometer()` 使用的参数 `length` 为 384 (128×3)。

现在，我们的输入张量内充满了新的加速度数据。每次运行推断，手势预测程序会解释推断的结果，并将结果传递到输出处理程序以显示给用户。

在 SparkFun Edge 上响应手势

输出处理程序位于 `sparkfun_edge/output_handler.cc`，它非常简单。第一次运行时，我们为输出配置 LED：

```
void HandleOutput(tflite::ErrorReporter* error_reporter, int kind) {
    // The first time this method runs, set up our LEDs correctly
    static bool is_initialized = false;
    if (!is_initialized) {
        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);
        is_initialized = true;
    }
}
```

接下来，我们在每次推断时切换黄色的 LED：

```
// Toggle the yellow LED every time an inference is performed
static int count = 0;
++count;
if (count & 1) {
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_YELLOW);
} else {
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_YELLOW);
}
```

之后，我们检查是否检测到某个手势。对于每个单独的手势，我们会点亮一个 LED，清除所有其他 LED，然后通过串口输出一些精美的 ASCII 字符画。以下是处理“画翅膀”手势的代码：

```

// Set the LED color and print a symbol (red: wing, blue: ring, green: slope)
if (kind == 0) {
    error_reporter->Report(
        "WING:\n\r*          *          *\n\r*          *          *\n\r*          *          *\n\r");
    am_hal_gpio_output_set(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);
}

```

在串口监视器上，输出将如下所示：

WING:

* * * * *
* * * * *
* * * * *
* * * * *
* * * * *

每个手势使用不同的串口输出和 LED。

运行示例

现在，我们已经了解了 SparkFun Edge 代码的工作原理。接下来，让我们在硬件上运行代码。



与撰写本书时相比，编译过程有可能会发生变化，因此请查看 *README.md* 以获取最新说明。

要部署当前的示例，我们需要：

- SparkFun Edge 开发板与 Himax HM01B0 breakout 附件。
 - USB 编程器（我们推荐 SparkFun Serial Basic Breakout，它有 micro-B USB 和 USB-C 两种版本）。
 - 匹配的 USB 线。
 - Python 3 和一些依赖项。



如果你不确定是否安装了正确的 Python 版本，请查看 6.3.2 节中有关于检查 Python 版本的说明。

在终端中，克隆 TensorFlow 代码仓库，并将其切换到它的目录下：

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

接下来，我们将编译二进制文件并运行一些命令，以便使其可以被下载到设备上。为了避免一些键入操作，你可以从 *README.md* 复制和粘贴这些命令。

编译二进制文件。以下命令会下载所有需要的依赖项，然后编译 SparkFun Edge 的二进制文件：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
  TARGET=sparkfun edge magic wand bin
```

二进制文件将被创建为 `.bin` 文件，位于：

```
tensorflow/lite/micro/tools/make/gen/  
sparkfun_edge_cortex-m4/bin/magic_wand.bin
```

可以使用以下命令检查文件是否存在：

```
test -f tensorflow/lite/micro/tools/make/gen/sparkfun_edge_ \  
cortex-m4/bin/magic_wand.bin && echo "Binary was successfully created" || \  
echo "Binary is missing"
```

运行该命令时，你应该看到“`Binary was successfully created`”被打印到控制台。

如果你看“`Binary is missing`”，则说明编译过程存在问题。如果是这样，那么 `make` 命令的输出可能会提供一些有关错误的线索。

为二进制文件签名。必须先使用密钥对二进制文件签名才能将其部署到设备上。现在让我们运行一些命令来对二进制文件进行签名，以便能够将其烧录到 SparkFun Edge 上。这里使用的脚本来自 Ambiq SDK，它在运行 `Makefile` 时下载。

输入以下命令来设置一些可以用于开发的虚拟密钥：

```
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
```

接下来，运行以下命令来创建签名的二进制文件。如有需要，请用 `python` 代替 `python3`：

```
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/micro_vision.bin \  
--load-address 0xC000 \  
--magic-num 0xCB \  
-o main_nonsecure_ota \  
--version 0x0
```

它将创建 `main_nonsecure_ota.bin` 文件。现在运行此命令来创建文件的最终版本，你可以用将在下一个步骤中使用的脚本来烧录设备：

```
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
```

运行命令的目录中应该有一个名为 *main_nonsecure_wire.bin* 的文件。这是你要烧录到设备上的文件。

烧录二进制文件。 SparkFun Edge 将当前运行的程序存储在 1MB 的闪存中。如果你想让开发板运行一个新程序，你需要把程序发送到开发板，开发板会把接收到的程序存储在闪存中，并覆盖之前保存的任何程序。

正如我们在前面提到的，这个过程称为烧录。

将编程器连接到开发板上。 要将新程序下载到开发板，你将使用 SparkFun USB-C Serial Basic 串口编程器。它允许你的计算机通过 USB 与微控制器进行通信。

请执行以下操作，将编程器安装到你的开发板上：

1. 在 SparkFun Edge 的一侧找到六引脚接口。
2. 将 SparkFun USB-C Serial Basic 串口编程器插入这些引脚中，确保每个设备上被标记为 BLK 和 GRN 的引脚正确对齐。

你可以在图 11-11 中看到正确的排列。

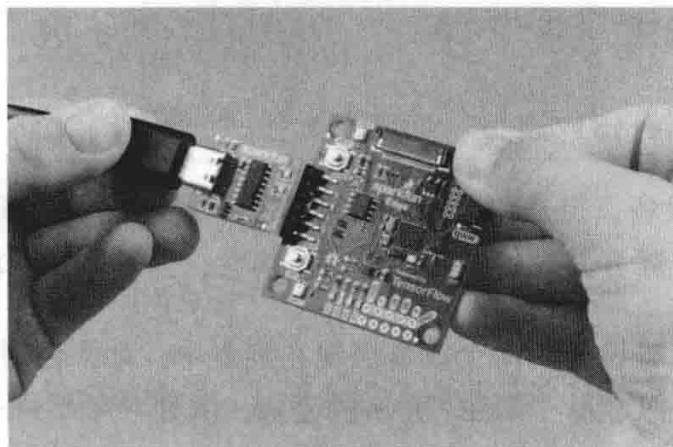


图 11-11：连接 SparkFun Edge 和 SparkFun USB-C Serial Basic（图片由 SparkFun 提供）

将编程器连接到你的计算机上。 你可以通过 USB 接口把开发板连接到你的计算机上。要给开发板编程，你需要找出设备在计算机中的名称。最好的方法是列出连接编程器之前和之后计算机上所有的设备名称，然后看看哪个设备是新增的。



已经有人报告了他们使用操作系统的默认驱动程序时遇到问题，所以我们强烈建议在继续之前安装驱动程序。

在通过 USB 连接设备之前，运行以下命令：

```
# macOS:  
ls /dev/cu*  
  
# Linux:  
ls /dev/tty*
```

运行后应该输出一个已连接设备的列表，类似于下面的输出：

```
/dev/cu.Bluetooth-Incoming-Port  
/dev/cu.MALS  
/dev/cu.SOC
```

现在，将编程器连接到你的计算机的 USB 端口，并再次运行以下命令：

```
# macOS:  
ls /dev/cu*  
  
# Linux:  
ls /dev/tty*
```

你应该在输出中看到一个额外的项，如下面的例子所示。在你的计算机上，新增的项目可能有不同的名称。此例中设备的名称是：

```
/dev/cu.Bluetooth-Incoming-Port  
/dev/cu.MALS  
/dev/cu.SOC  
/dev/cu.wchusbserial-1450
```

此名称将被用于引用设备。但是，它可能会随着编程器连接的 USB 端口的改变而改变，所以如果你从计算机上断开开发板，然后重新连接它，你可能需要再次查找它的名称。



一些用户报告说看到两个设备出现在列表中。如果你看到两个设备，正确的设备应该是以字母“wch”开头的设备，例如“/dev/wchusbserial-14410”。

确定设备名称后，将它存入 shell 变量以备后用：

```
export DEVICENAME=<your device name here>
```

在稍后运行需要设备名的命令时，可以使用这个变量。

运行脚本以烧录开发板。要烧录开发板，你需要将它置于特殊的 bootloader 状态，以使其准备好接收新的二进制文件。然后，你可以运行脚本以将二进制文件发送到开发板。

首先创建一个环境变量以指定波特率，波特率是将数据发送到设备的速度：

```
export BAUD_RATE=921600
```

现在, 请将下面的命令粘贴到你的终端中, 但是先不要按回车键! 命令中的 \${DEVICENAME} 和 \${BAUD_RATE} 将被替换为你在前面设置的值。请记住, 如有必要, 请用 python 替换 python3:

```
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
```

接下来, 将开发板重置为 bootloader 状态并烧录开发板。

在开发板上, 找到标记为 RST 和 14 的按键, 如图 11-12 所示。

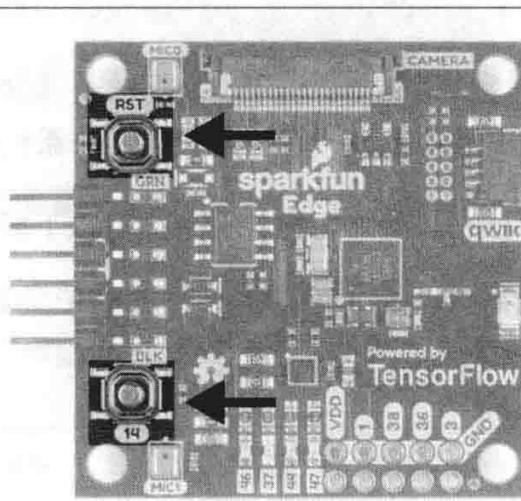


图 11-12: SparkFun Edge 上的按键

执行以下步骤:

1. 确保你的开发板已连接至编程器, 且整个设备均已通过 USB 连接到你的计算机。
2. 在开发板上, 按住按键 14, 不要松手。
3. 在保持按住按键 14 的同时, 按下按键 RST 以重置开发板。
4. 在计算机上按回车键以运行脚本。继续保持按住按键 14。

现在, 你应该在屏幕上看到如下内容:

```
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...]
```

一直按住按键 **14**，直到你看到“**Sending Data Packet of length 8180**”为止。看到此消息后可以松开按键（但如果你一直按着它也没关系）。

该程序将继续在终端上打印日志。最终，你会看到类似以下的内容：

```
[...lots more Sending Data Packet of length 8180...]
Sending Data Packet of length 8180
Sending Data Packet of length 6440
Sending Reset Command.
Done.
```

这表示烧录成功。



如果程序输出以错误结尾，请检查“**Sending Reset Command.**”是否被打印出来。如果是这样，尽管出现了错误，但烧录仍可能会成功；否则，烧录很可能会失败，需要尝试再次执行这些步骤（可以跳过设置环境变量）。

测试程序

首先按下按键 **RST**，以确保程序正在运行。当程序运行时，每运行一次推断，黄色的 LED 就会被点亮和关闭一次。

接下来，使用以下命令开始打印设备的串口输出：

```
screen ${DEVICENAME} 115200
```

刚开始你应该会看到以下输出：

```
Magic starts!
```

现在你可以尝试做出一些手势。如图 11-13 所示，用一只手拿住开发板，组件朝上，USB 适配器朝左。

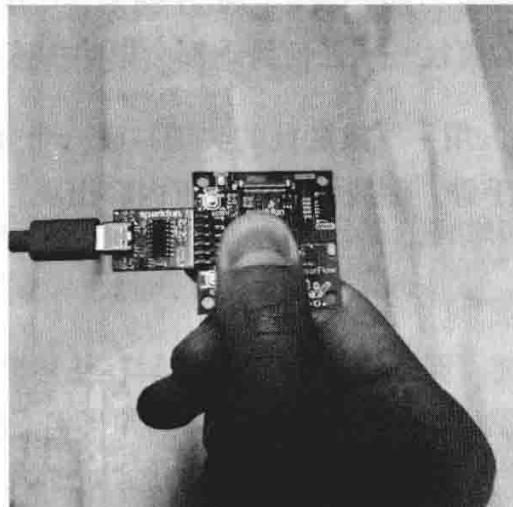


图 11-13：如何在尝试手势时拿着开发板

图 11-14 展示了如何执行每个手势的示意图。由于我们的模型是根据开发板与魔杖相连时收集的数据进行训练的，所以你可能需要几次尝试才能检测到结果。

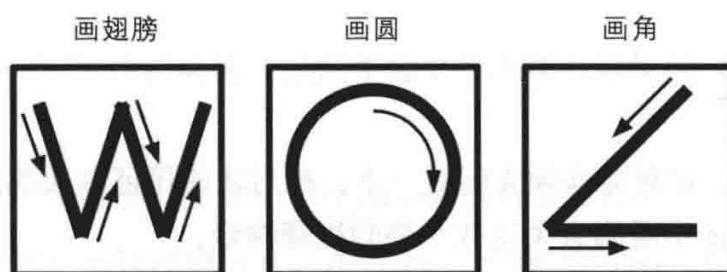


图 11-14：三种魔杖手势

最简单的手势是“画翅膀”。你的手要移动得足够快，大约在一秒内完成这个动作。如果你成功了，你应该会看到以下输出，且红色的 LED 会被点亮：

WING:

恭喜，你已使用 SparkFun Edge 施展出第一个“魔法咒语”！



此时，你可以选择更具创造力的方法，将开发板放到魔杖的顶部，即距离你的手最远的地方。任何棍棒、直尺或其他长度约一英尺（约 30 厘米）的家用物品都可以。

确保设备连接牢固且方向一致，即组件朝上，USB 适配器朝左。并且选择一根坚硬而非柔韧的魔杖——任何抖动都会影响加速度传感器的读数。

接下来，尝试“画圆”手势，用手（或魔杖的尖端）画一个顺时针的圆圈。同样，目标是花一秒左右的时间来执行手势。你应该看到以下内容出现，就像是魔术一样：

RING:

```
*  
* * *  
* * * *  
* * * * *  
* * * * * *  
* * * * * * *
```

最后一个动作是在空中画一个三角形的角，如下面的 ASCII 字符画所示：

SLOPE:

```
*  
*  
*  
*  
*  
*  
*  
* * * * * * *
```

像任何“魔咒”一样，你可能必须先练习一下，然后才能保证每次都能完美地执行它们。你可以在 *README.md* 中看到有关这些手势的视频演示。

如果失败了怎么办

以下是一些可能遇到的问题以及调试方法：

问题：烧录时，脚本在“**Sending Hello.**”的地方停顿一段时间。然后打印错误。

解决方案：运行脚本时，需要按住按键 **14**。先按住按键 **14**，然后按下按键 **RST**，在保持按住按键 **14** 的同时运行脚本。

问题：烧录后，所有 LED 都不亮。

解决方案：尝试按住按键 **RST**，或将开发板与编程器断开连接，然后重新连接。如果这些都不起作用，请尝试再次烧录开发板。

问题：LED 卡在点亮或熄灭状态。

解决方案：正常情况下，LED 会在推断后立即停止闪烁，而程序会等待有足够的新数据可用才重新开始闪烁。如果 LED 停止闪烁超过几秒，则表明程序可能已经崩溃。在这种情况下，请尝试按 reset 按键。

问题：手势不起任何作用。

解决方案：首先，请确保 LED 闪烁，这表明正在进行推断。如果没有，请按 reset 按键。接下来，请确保你按照正确的方向拿着开发板。

要学习手势，请从最容易掌握的“画翅膀”开始。“画圆”要难一点，因为你需要保证画出的圆非常平滑。“画角”手势最棘手。尝试观看 *README.md* 中的视频以获取指导。

尝试修改

现在你已经部署了基本的应用程序，可以试着修改一下代码。你可以在 *tensorflow/lite/micro/examples/magic_wind* 文件夹中找到应用程序的代码。只需编辑并保存，然后重复前面的操作，将修改后的代码部署到设备上。

以下是一些你可以尝试的事情：

- 尝试修改 *constants.cc* 中的阈值，以使手势更容易或更难执行（以更多的误报或者漏检为代价）。
- 在计算机上编写一个程序，让你可以使用手势执行某些任务。
- 扩展程序以通过蓝牙传输检测结果。在 Ambiq SDK 的 *AmbiqSuite-Rel2.0.0/boards/apollo3_evb/examples/uart_ble_bridge* 中，有一个有关如何使用蓝牙的示例。编译完魔杖应用程序后，SDK 将被下载到 *tensorflow/tensorflow/lite/micro/tools/make/downloads/AmbiqSuite-Rel2.0.0* 目录中。

11.6 小结

在本章，你看到了一个有趣的示例，它展示了嵌入式机器学习应用程序如何将模糊的传感器数据解释为更有用的形式。通过观察并发现噪声中的模式，嵌入式机器学习模型使设备能够理解周围的环境，并向我们发出提示，即使原始数据人类很难理解。

在第 12 章，我们将探讨魔杖模型的工作原理，并学习如何收集数据以及如何训练我们自己的“魔法咒语”。

第 12 章

魔杖：训练模型

在第 11 章中，我们使用了一个大小只有 20KB 的预先训练好的模型来解释原始的加速度传感器数据，并使用它来识别用户执行了一组手势中的哪一个。在本章中，我们会向你展示这个模型是如何训练的，然后讨论它是如何工作的。

我们的唤醒词模型和行人检测模型都需要使用大量的数据进行训练。这主要是由于待解决问题的复杂性。人们可以通过多种不同的方式说“yes”或者“no”，想想看，所有的口音、语调和音高变化都可以使一个人的声音与众不同。同样，一个人可以通过多种方式出现在图像中。你可能会只看到他的脸，或者整个身体，或者只有一只手，而且他还可能有各式各样的姿势。

因此，为了能够准确地对有效输入的多样性进行分类，我们需要在一组同样多样化的数据上训练模型。这就是为什么我们用于唤醒词模型和行人检测模型的训练数据集如此之大，以及为什么训练需要这么长时间。

我们的魔杖手势识别问题要简单得多。对于魔杖问题，我们没有尝试对各种自然声音或者人的外貌和姿势进行分类，而是试图了解三种特定的、人为选择的手势之间的区别。尽管不同的人执行每个手势的方式会有所不同，但我们期望用户会努力尽可能正确、统一地执行手势。

这意味着我们期望的有效输入中会有更少的变化，这使得我们不需要大量的数据，而且模型训练也要容易得多。事实上，我们将用于训练模型的数据集中，每个手势仅包含了大约 150 个示例，其总大小只有 1.5MB。想到要在如此小的数据集上训练有用的模型是令人兴奋的，因为获取足够的数据通常是机器学习项目中最困难的部分。

在 12.1 节，你将学习如何训练魔杖应用程序中使用的原始模型。在 12.2 节，我们将讨论这个模型的工作原理。在 12.3 节，你将看到如何获取自己的数据，并用这些数据来训练一个能够识别不同手势的新模型。

12.1 训练模型

为了训练我们的模型，我们使用 TensorFlow 代码仓库中的训练脚本。你可以在 *magic_wand/train* 中找到它们。

这些脚本会执行以下任务：

- 为训练准备原始数据。
- 生成合成数据^{注1}。
- 拆分数据以分别进行训练、验证和测试。
- 执行数据增强。
- 定义模型架构。
- 运行训练过程。
- 将模型转换为 TensorFlow Lite 格式。

为了使事情更简单，脚本附带了 Jupyter 笔记本，并演示了如何使用它们。你可以在 GPU 运行时 (runtime) 上的 Colaboratory (Colab) 中运行笔记本。对于我们的小型数据集，训练只需几分钟。

首先，让我们逐步完成 Colab 中的训练过程。

12.1.1 在 Colab 中训练

在 *magic_wand/train/train_magic_wand_model.ipynb* 中打开 Jupyter 笔记本，然后单击“Run in Google Colab”(在 Google Colab 中运行) 按钮，如图 12-1 所示。

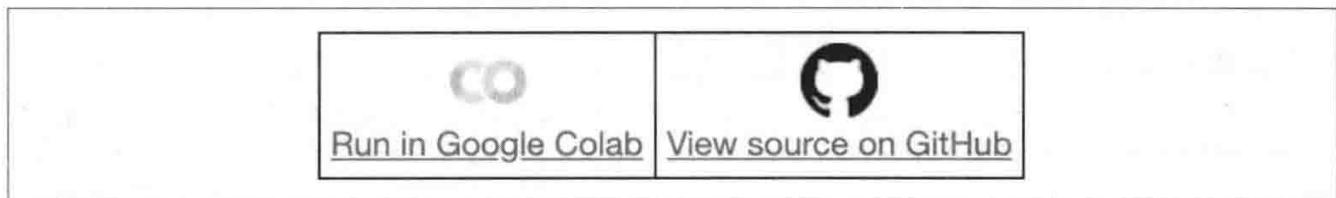


图 12-1：“Run in Google Colab”按钮



撰写本书时，GitHub 中存在一个漏洞，在显示 Jupyter 笔记本时会导致出现间歇性错误消息。如果你在尝试访问笔记本时看到“Sorry, something went wrong. Reload?”(抱歉，出了点问题。重新加载?)，请按照 4.3 节中的说明进行操作。

注 1：这是一个新的术语，我们将在后面讨论它。

笔记本会完成模型训练的整个过程。它包括以下步骤：

- 安装依赖项。
- 下载并准备数据。
- 加载 TensorBoard 以可视化训练过程。
- 训练模型。
- 生成 C 源文件。

启用 GPU 训练

训练我们的模型应该会很快，但是如果我们使用 GPU 运行将会更快。要启用 GPU 训练，请转到 Colab 的“Runtime”（运行时）菜单，然后选择“Change runtime type”（更改运行时类型），如图 12-2 所示。

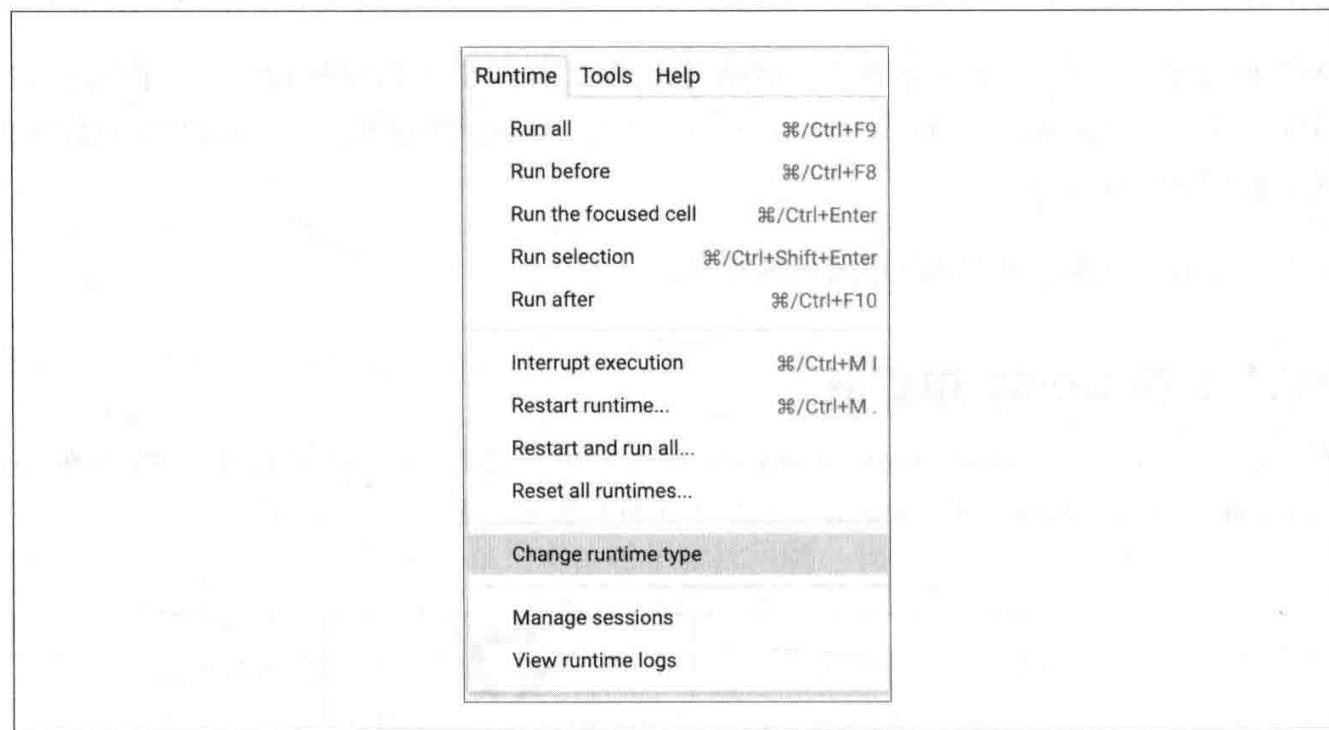


图 12-2：Colab 中的“Change runtime type”选项

这将打开“Notebook settings”（笔记本设置）对话框，如图 12-3 所示。

从“Hardware accelerator”（硬件加速器）下拉列表中选择 GPU，如图 12-4 所示，然后单击保存。

现在你就可以运行笔记本了。

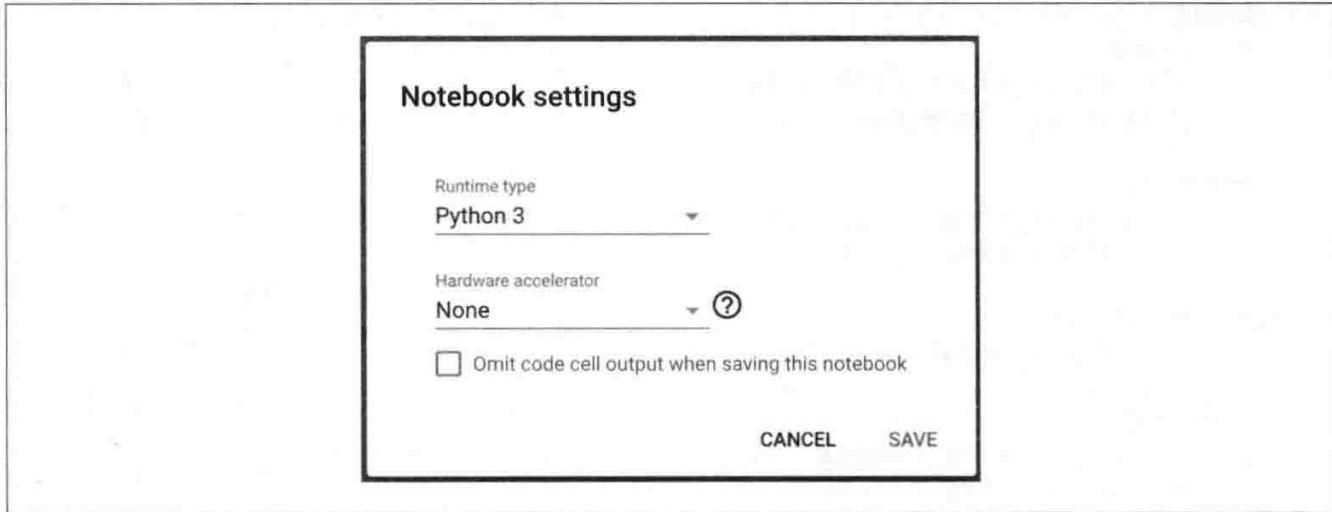


图 12-3：“Notebook settings”对话框

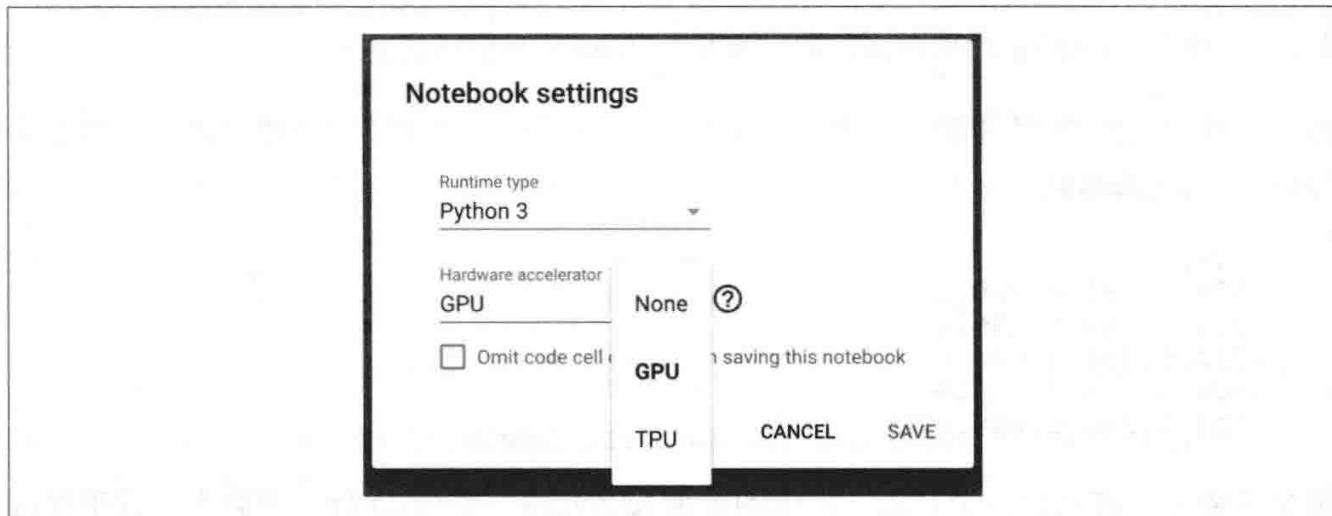


图 12-4：“Hardware acclerator”下拉列表

安装依赖项

第一步是安装所需的依赖项。在“安装依赖项”部分，运行单元格会安装正确版本的TensorFlow并获取训练脚本的副本。

准备数据

接下来，在“准备数据”部分，运行单元格会下载数据集并将其分为训练集、验证集和测试集。

第一个单元格会下载数据集并将其提取到训练脚本的目录中。数据集由四个目录组成，其中三个目录分别对应三种手势（“画翅膀”“画圆”和“画角”），还有一个“negative”目录用于表示没有明显类型的手势。每个目录都包含了原始的数据文件，这些原始数据就是执行某种手势时加速度传感器捕获到的数据：

```
data/
└── slope
    ├── output_slope_dengyl.txt
    ├── output_slope_hyw.txt
    └── ...
└── ring
    ├── output_ring_dengyl.txt
    ├── output_ring_hyw.txt
    └── ...
└── negative
    ├── output_negative_1.txt
    └── ...
└── wing
    ├── output_wing_dengyl.txt
    ├── output_wing_hyw.txt
    └── ...
```

每个手势有 10 个文件，稍后我们将逐步介绍它们。每个文件都包含一个由指定的人执行的手势，文件名的最后一部分对应它的用户 ID。例如，文件 *output_slope_dengyl.txt* 包含由 ID 为 *dengly* 的用户演示的“画角”(slope) 手势的数据。

每个文件中大约有 15 个给定手势的演示数据，每行有一个加速度传感器读数，每个手势以 -,-,- 为起始：

```
-,-,-
-766.0,132.0,709.0
-751.0,249.0,659.0
-714.0,314.0,630.0
-709.0,244.0,623.0
-707.0,230.0,659.0
```

每个手势演示都包含一个日志，记录的数据长达几秒，每秒 25 行。手势本身发生在这个窗口内的某个点，设备在其余时间都保持不动。

由于捕获测量值的方式不同，文件还包含了一些垃圾字符。我们的第二个训练单元格中运行的第一个脚本 *data_prepare.py* 将清除这些脏数据：

```
# Prepare the data
!python data_prepare.py
```

该脚本将用于从文件夹中读取原始数据文件，忽略所有垃圾字符，并将经过清理的数据写入训练脚本目录 (*data/complete_data*) 中的其他位置。由于错误、损坏等常见问题，大型数据集很容易混入一些脏数据，所以在训练机器学习模型时，清理混乱的数据源是很常见的任务。

除了清洗数据外，脚本还会生成一些合成数据 (synthetic data)。这是指通过算法生成的，而非从现实世界中捕获的数据。在这种情况下，*data_prepare.py* 中的 *generate_negative_data()* 函数会创建合成数据，这些合成数据等效于加速度传感器的运动，但不会对应任何特定的手势。这些数据将被用于训练我们的“未知”类别。

由于创建合成数据比捕获真实数据要快得多，因此这有助于增强训练过程。然而，现实世界中的变化是无法预测的，所以通常不可能通过数据合成创建整个数据集。对于我们的例子，数据合成有助于使“未知”类别更加鲁棒，但对于已知类别的手势数据来说并没有什么帮助。

在第二个单元格中运行的下一个脚本是 *data_split_person.py*:

```
# Split the data by person
!python data_split_person.py
```

该脚本将数据分为训练集、验证集和测试集。由于我们的数据是人工标记的，因此可以使用一组人员的数据进行训练，使用另一组人员的数据进行验证，最后使用一组人员的数据进行测试。数据划分如下：

```
train_names = [
    "hyw", "shiyun", "tangsy", "dengyl", "jiangyh", "xunkai", "negative3",
    "negative4", "negative5", "negative6"
]
valid_names = ["lsj", "pengxl", "negative2", "negative7"]
test_names = ["liucx", "zhangxy", "negative1", "negative8"]
```

我们使用 6 个人的数据进行训练，2 个人的数据进行验证，2 个人的数据进行测试。另外，我们还混合了与特定用户无关的负面数据。我们的总数据以大约 60%:20%:20% 的比例被分为三组，这对于机器学习来说是非常标准的。

在按人员划分数据时，我们试图确保我们的模型能够泛化到新的数据。因为模型将根据未包含在训练数据集中的人的数据进行验证和测试，所以我们的模型需要对每个手势的个体变化具有鲁棒性。

当然也可以随机划分数据，而不是按人员来划分。在这种情况下，训练集、验证集和测试集将分别包含来自每个个体的各种手势的一些样本。由此生成的模型将接受来自每个人的数据的训练，而不仅仅包含 6 个人的数据，因此它将有更多的机会接触到不同人执行手势的不同风格。

然而，由于验证集和训练集也包含来自每个人的数据，因此我们无法测试模型是否能够很好地推广到从未见过的新手势风格。以这种方式开发的模型在验证和测试期间可能会有更高的准确率，但不能保证它在处理新数据时能很好地工作。

项目构想

你可以使用脚本 *data_split.py* 替换 *data_split_person.py*，来用随机划分的方式拆分数据。

以常规方式训练模型后，请尝试修改 Colab 以使用随机划分方式，并测试哪种方式更有效。

在继续操作之前，请确保你已经运行了“准备数据”部分的两个单元格。

加载 TensorBoard

数据准备好后，我们可以运行下一个单元格以加载 TensorBoard，这将帮助我们监视训练过程：

```
# Load TensorBoard  
%load_ext tensorboard  
%tensorboard --logdir logs/scalars
```

训练日志将被写入训练脚本目录下的 *logs/scalars* 子目录，因此我们将其传递给 TensorBoard。

开始训练

TensorBoard 加载后，就可以开始训练了。运行以下单元格：

```
!python train.py --model CNN --person true
```

脚本 *train.py* 设置模型架构，使用 *data_load.py* 加载数据，然后开始训练过程。

加载数据时，*load_data.py* 还会使用 *data_augmentation.py* 中定义的代码执行数据增强。函数 *augment_data()* 将获取代表一个手势的数据，并创建它的许多新版本，每个版本都与原始版本略有不同。具体修改包括移动和扭曲数据点、添加随机噪声以及提高加速度。增强后的数据与原始数据一起用于训练模型，从而充分利用我们的小型数据集。

随着训练的进行，你会在开始运行的单元格下方看到一些输出。这里有很多东西，让我们挑选出最值得注意的部分。首先，Keras 会生成一个漂亮的表格，它展示了模型的架构：

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 128, 3, 8)	104
max_pooling2d (MaxPooling2D)	(None, 42, 1, 8)	0
dropout (Dropout)	(None, 42, 1, 8)	0
conv2d_1 (Conv2D)	(None, 42, 1, 16)	528
max_pooling2d_1 (MaxPooling2D)	(None, 14, 1, 16)	0
dropout_1 (Dropout)	(None, 14, 1, 16)	0
flatten (Flatten)	(None, 224)	0

dense (Dense)	(None, 16)	3600
dropout_2 (Dropout)	(None, 16)	0
dense_1 (Dense)	(None, 4)	68
=====		

表格展示了我们使用的所有层，以及它们的形状和参数数量——这是权重和偏差的另一个术语。你会看到我们的模型使用了 Conv2D 层，这是因为它是卷积模型。该表格没有显示模型的输入形状为 (None, 128, 3)。稍后，我们将更仔细地研究模型的架构。

输出还将向我们显示模型大小的估计值：

```
Model size: 16.796875 KB
```

这表示模型的训练参数将占用的内存量。它不包含存储模型执行图所需的额外空间，因此我们实际的模型文件会稍大一些，但它可以让我们了解正确的数量级。这绝对是一个微型模型 (tiny model)！

你最终将看到训练开始了：

```
1000/1000 [=====] - 12s 12ms/step - loss: 7.6510 -  
accuracy: 0.5207 - val_loss: 4.5836 - val_accuracy: 0.7206
```

此时，你可以看一下 TensorBoard，以了解训练过程。

评估结果

训练完成后，我们可以查看单元的输出以获取一些有用的信息。

首先，我们可以看到在最后一个轮次中，验证的准确率看起来非常不错，高达 0.9743，而且损失值也很低：

```
Epoch 50/50  
1000/1000 [=====] - 7s 7ms/step - loss: 0.0568 -  
accuracy: 0.9835 - val_loss: 0.1185 - val_accuracy: 0.9743
```

这很好，尤其是当我们使用按人划分的数据时，这意味着我们的验证数据来自一组完全不同的人。但是，我们不能仅仅依靠验证准确率来评估我们的模型。由于模型的超参数和架构是在验证集上手动调整的，因此可能会过拟合。

为了更好地了解模型的最终性能，可以通过调用 Keras 的 `model.evaluate()` 函数来根据我们的测试集评估它。输出的下一行显示了评估结果：

```
6/6 [=====] - 0s 6ms/step - loss: 0.2888 - accuracy:  
0.9323
```

尽管没有验证数据的结果那么惊人，但是我们的模型还是展示了足够好的准确率：0.9323，并且损失值仍然很低。模型将以93%的准确率预测分类，这对于我们的目的来说应该是足够的。

接下来的几行显示了由`tf.math.confusion_matrix()`函数计算的结果的混淆矩阵(confusion matrix)：

```
tf.Tensor(  
[[ 75   3   0   4]  
 [ 0  69   0  15]  
 [ 0   0  85   3]  
 [ 0   0   1 129]], shape=(4, 4), dtype=int32)
```

混淆矩阵是评估分类模型性能的有用工具。它显示了测试集中每个输入的预测类别与其实际类别的吻合程度。

混淆矩阵的每一列依次对应一个预测的标签（“画翅膀”“画圆”“画角”和“未知”）。从上到下的每一行都对应于真实的标签。从混淆矩阵中，我们可以看到绝大多数预测标签与实际标签一致。我们还可以看到发生混淆的特定位置：最重要的是，相当多的输入被错误地分类为“未知”，尤其是对于实际类别为“画圆”的输入。

混淆矩阵让我们知道模型的弱点在哪里。在这种情况下，它告诉我们：获取更多关于“画圆”手势的训练数据可能会帮助模型更好地了解“画圆”和“未知”之间的区别。

`train.py`要做的最后一件事是将模型转换为TensorFlow Lite格式，包括浮点变量和量化变量。以下输出显示了每个变体的大小：

```
Basic model is 19544 bytes  
Quantized model is 8824 bytes  
Difference is 10720 bytes
```

量化后，我们的模型从20KB缩小到8.8KB。这是一个非常小的模型，而且效果很好。

创建一个C数组

“创建C源文件”部分的下一个单元格将模型转换为C源文件。运行此单元格以查看输出：

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

我们可以将这个文件的内容复制并粘贴到我们的项目中，这样就可以在我们的应用程序

中使用新训练的模型。稍后，你将学习如何收集新数据并教应用程序了解新手势。现在，让我们继续后面的内容。

12.1.2 执行脚本的其他方式

如果你不想使用 Colab，或者要更改模型训练脚本并希望在本地进行测试，那么你可以在自己的开发计算机上轻松地运行这些脚本。你可以在 *README.md* 中找到相应的说明。

接下来，我们将逐步介绍模型是如何工作的。

12.2 模型是如何工作的

到目前为止，我们已经确定我们的模型是卷积神经网络（CNN），并且它将 128 个三轴加速度传感器读数的序列（代表大约 5 秒的时间）转换为包含 4 个概率的数组：3 个概率分别对应 3 种手势，另外一个概率代表“未知”。

CNN 很适合学习包含在相邻数值之间的重要信息。在 12.2.1 节，我们将看看我们的数据，并了解为什么 CNN 非常适合理解这种类型的数据。

12.2.1 可视化输入数据

在我们的时间序列加速度传感器数据中，相邻的加速度传感器读数为我们提供了有关设备运动的线索。例如，如果一个轴上的加速度从零迅速变为正数，然后又回到零，那么设备可能已开始朝那个方向运动了。图 12-5 给出了一个假想的例子。

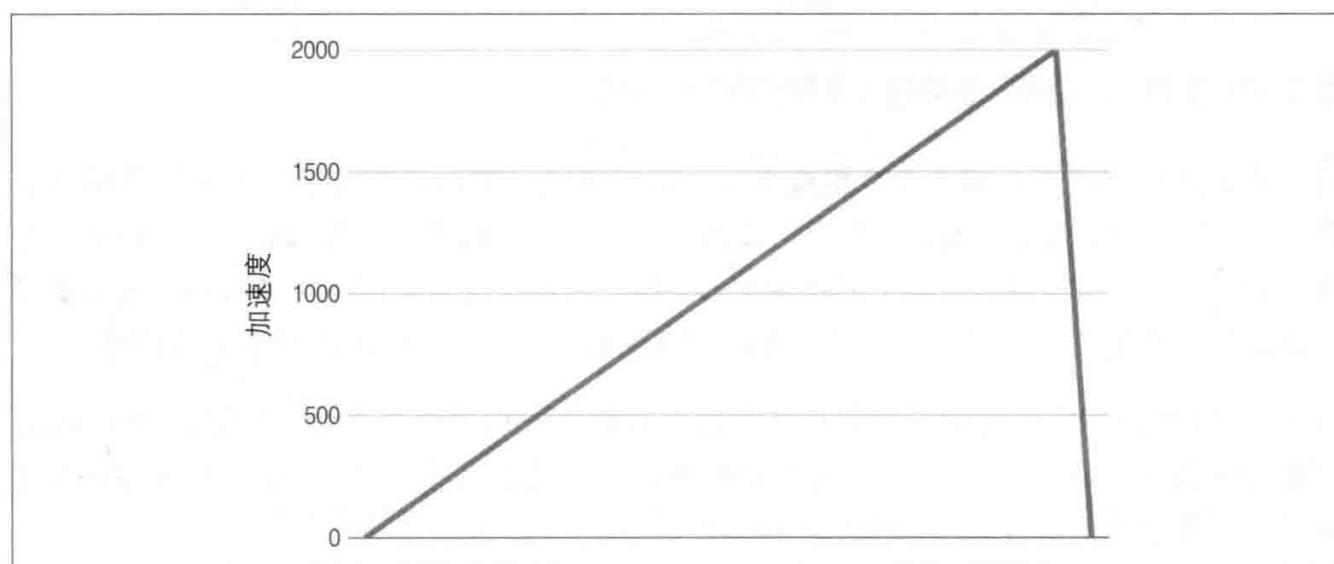


图 12-5：正在移动的设备在某个轴上的加速度值

任何一个手势都是由一系列动作组成的，一个动作接着另一个动作。例如，考虑“画翅膀”手势，如图 12-6 所示。



图 12-6：“画翅膀”手势

首先将设备向右下移动，然后向右上移动，再向右下移动，最后向右上移动。图 12-7 展示了在执行“画翅膀”手势期间采集的真实数据的样本，以 milli-Gs 为单位。

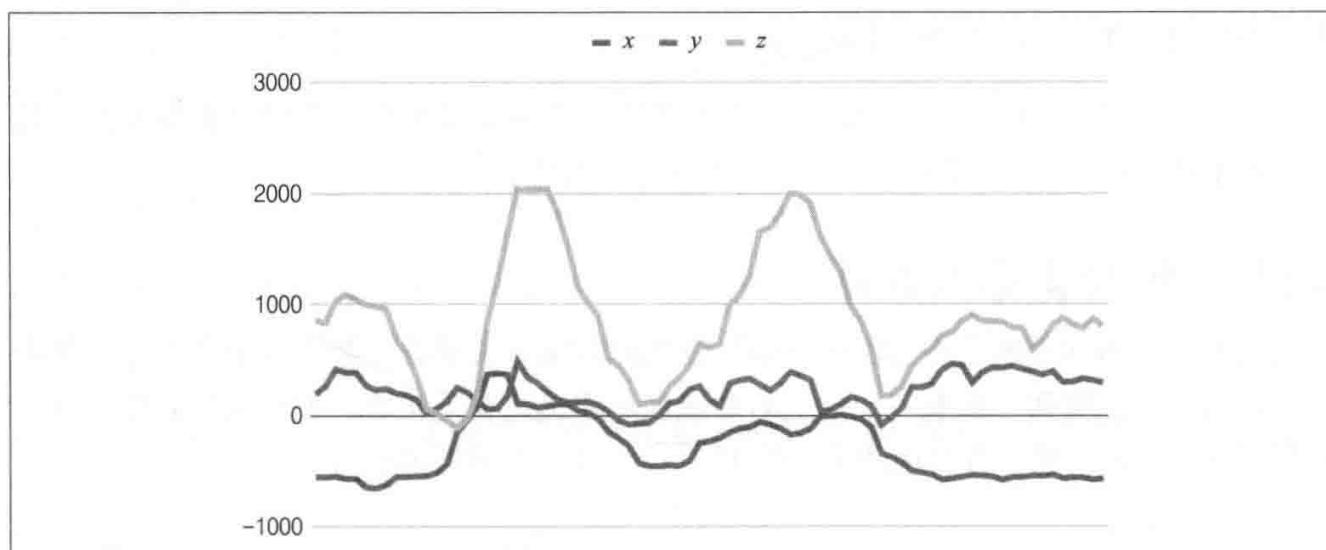


图 12-7：执行“画翅膀”手势期间采集到的加速度值

通过查看此图并将其分解为各个组成部分，我们可以了解加速度传感器正在做出哪个手势。从 z 轴上的加速度来看，很明显，设备正在按照“画翅膀”手势的形状上下移动。更微妙的是，我们可以看到 x 轴上的加速度与 z 轴上的加速度是如何相互关联的，从而显示了设备在手势宽度上的移动。同时，我们可以观察到 y 轴上的加速度基本上保持稳定。

同样，多层的 CNN 能够学习如何从各个轴上的数据中识别每个手势。例如，CNN 网络可能会学会区分上下运动，并且能理解如何适当地结合其中两个轴（比如 z 轴和 y 轴）来识别“画翅膀”手势。

为此，CNN 会学习一系列分层排列的过滤器（filter）。每个过滤器都学习在数据中发现特定类型的特征。当发现这个特征后，过滤器会把这个高级信息传递到网络的下一层。例如，网络第一层中的一个过滤器可能会学会发现一些简单的东西，例如一段时间内的加速度。当识别出这样的结构时，它会把相应的信息传递到网络的下一层。

后续的过滤层将学习如何将来自前期的简单过滤器的输出组合在一起以形成更大的结构。例如，一系列四个交替的向上和向下的加速度组合在一起可以表示“画翅膀”手势的形状“W”。

在这个过程中，有噪声的输入数据将被逐步转换为高级的符号表示。网络的后续层可以分析这些符号表示，以猜测执行了哪种手势。

在下一节中，我们将介绍实际的模型架构，并了解它如何映射到此过程。

12.2.2 理解模型架构

我们模型的架构在 `train.py` 的 `build.cnn()` 函数中定义。这个函数使用 Keras API 逐层定义模型：

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D( # input_shape=(batch, 128, 3)
        8, (4, 3),
        padding="same",
        activation="relu",
        input_shape=(seq_length, 3, 1)), # output_shape=(batch, 128, 3, 8)
    tf.keras.layers.MaxPool2D((3, 3)), # (batch, 42, 1, 8)
    tf.keras.layers.Dropout(0.1), # (batch, 42, 1, 8)
    tf.keras.layers.Conv2D(16, (4, 1), padding="same",
        activation="relu"), # (batch, 42, 1, 16)
    tf.keras.layers.MaxPool2D((3, 1), padding="same"), # (batch, 14, 1, 16)
    tf.keras.layers.Dropout(0.1), # (batch, 14, 1, 16)
    tf.keras.layers.Flatten(), # (batch, 224)
    tf.keras.layers.Dense(16, activation="relu"), # (batch, 16)
    tf.keras.layers.Dropout(0.1), # (batch, 16)
    tf.keras.layers.Dense(4, activation="softmax") # (batch, 4)
])
```

这是一个连续的模型，意味着每一层的输出都会被直接传递到下一层。让我们一步一步地浏览各层，看看发生了什么。

第一层是 `Conv2D`：

```
tf.keras.layers.Conv2D(
    8, (4, 3),
    padding="same",
    activation="relu",
    input_shape=(seq_length, 3, 1)), # output_shape=(batch, 128, 3, 8)
```

这是一个卷积层，它直接接收网络的输入，即原始加速度传感器数据序列。输入的形状由 `input_shape` 参数提供，它被设置为 `(seq_length, 3, 1)`，其中 `seq_length` 是传入的加速度传感器测量值的总数（默认为 128）。每个测量值由三个值组成，分别代表 `x` 轴、`y` 轴和 `z` 轴上的加速度，如图 12-8 所示。

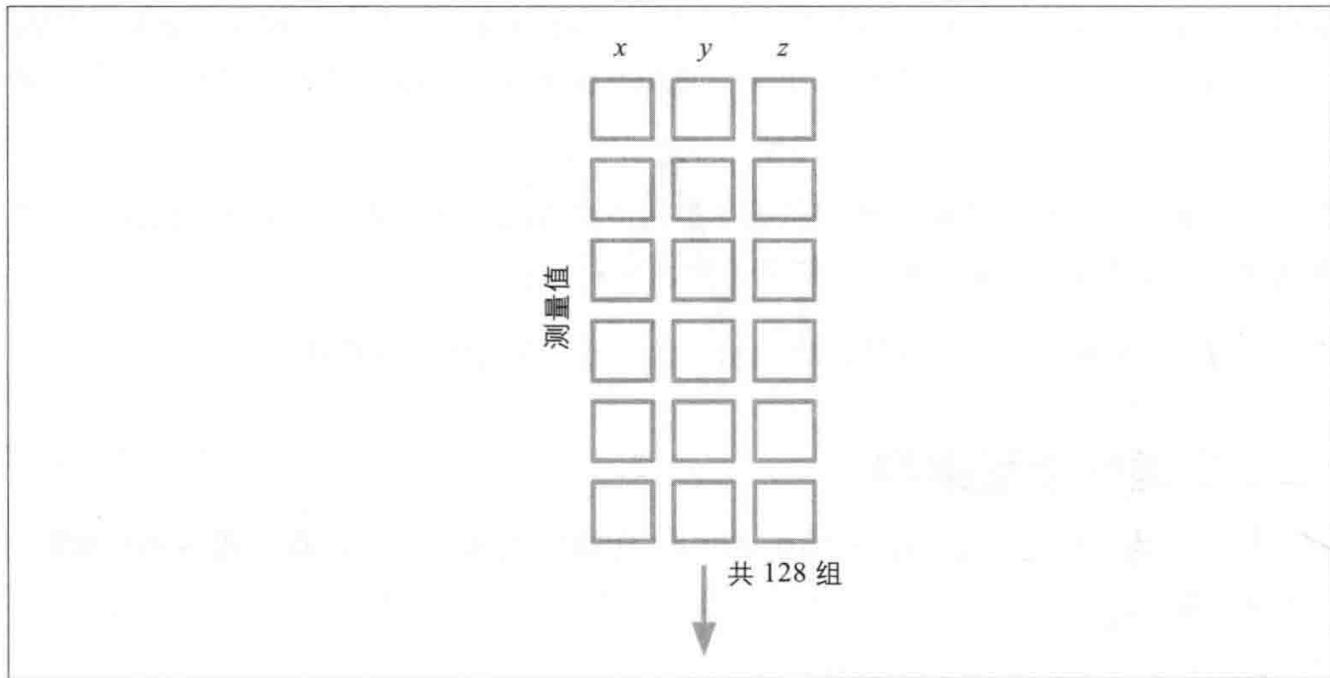


图 12-8：模型的输入

卷积层的工作是获取原始数据并提取一些可以被后续层解释的基本特征。`Conv2D()` 函数的参数将决定卷积层会提取多少个特征。这些参数的描述参见 `tf.keras.layers.Conv2D()` 文档。

第一个参数确定该图层将具有多少个过滤器。在训练过程中，每个过滤器都会学习识别原始数据中的特定特征，例如，一个过滤器可能会学习如何识别向上运动的特征。对于每个过滤器，每个图层都会输出一个特征图（feature map），以显示它学到的特征在输入中出现的位置。

我们的代码中定义的层具有 8 个过滤器，这意味着它将学习从输入数据中识别并输出 8 种不同类型的高级特征。你可以在输出形状 (`batch_size, 128, 3, 8`) 中看到这一点，该形状在其最终尺寸中具有 8 个特征通道（feature channel），每个通道对应一种特征，每个通道中的值表示输入中的对应位置出现这种特征的程度。

正如我们在第 8 章中介绍的那样，卷积层在数据中滑动一个窗口，并检测该窗口中是否存在给定的特征。`Conv2D()` 的第二个参数决定了窗口的尺寸。在我们的例子中，窗口的尺寸是 $(4, 3)$ 。这意味着过滤器正在寻找特征跨越了 4 个连续的加速度传感器测量值和所有 3 个轴。由于该窗口包括了 4 个测量值，因此每个过滤器都会分析一个短的时间快照，这意味着它可以生成代表加速度随时间变化的特征。你可以在图 12-9 中看到它是如何工作的。

`padding` 参数决定了窗口如何在数据之间移动。如果将 `padding` 设置为 `same`，那么层的输出将具有与输入相同的长度（128）和宽度（3）。由于过滤器窗口的每次移动都

会产生一个单独的输出，因此 `same` 意味着该窗口必须在每行数据上移动三次，然后下移 128 次。



图 12-9：覆盖在数据上的卷积窗口

因为窗口的宽度为 3，所以它必须从数据的左侧开始。过滤器窗口没有覆盖实际值的空白区域用零填充。要使过滤器窗口总共向下移动 128 次，过滤器还必须突出数据的顶部。你可以在图 12-10 和图 12-11 中看到它的工作原理。

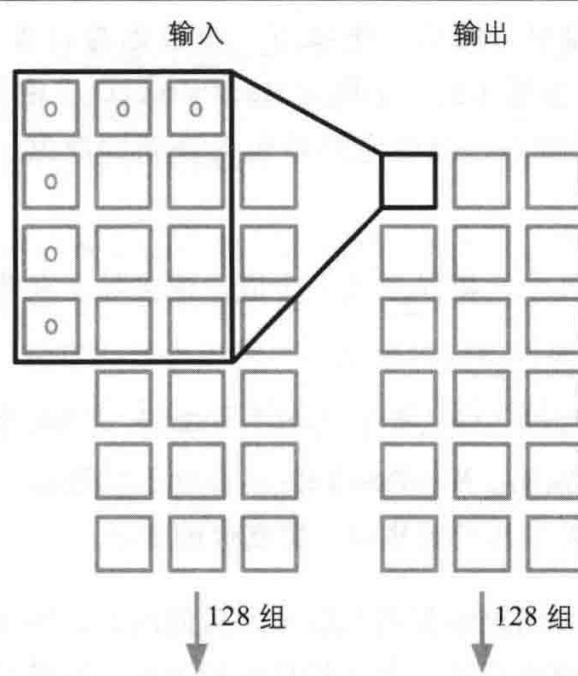


图 12-10：卷积窗口的初始位置，需要在顶部和左侧填充数据

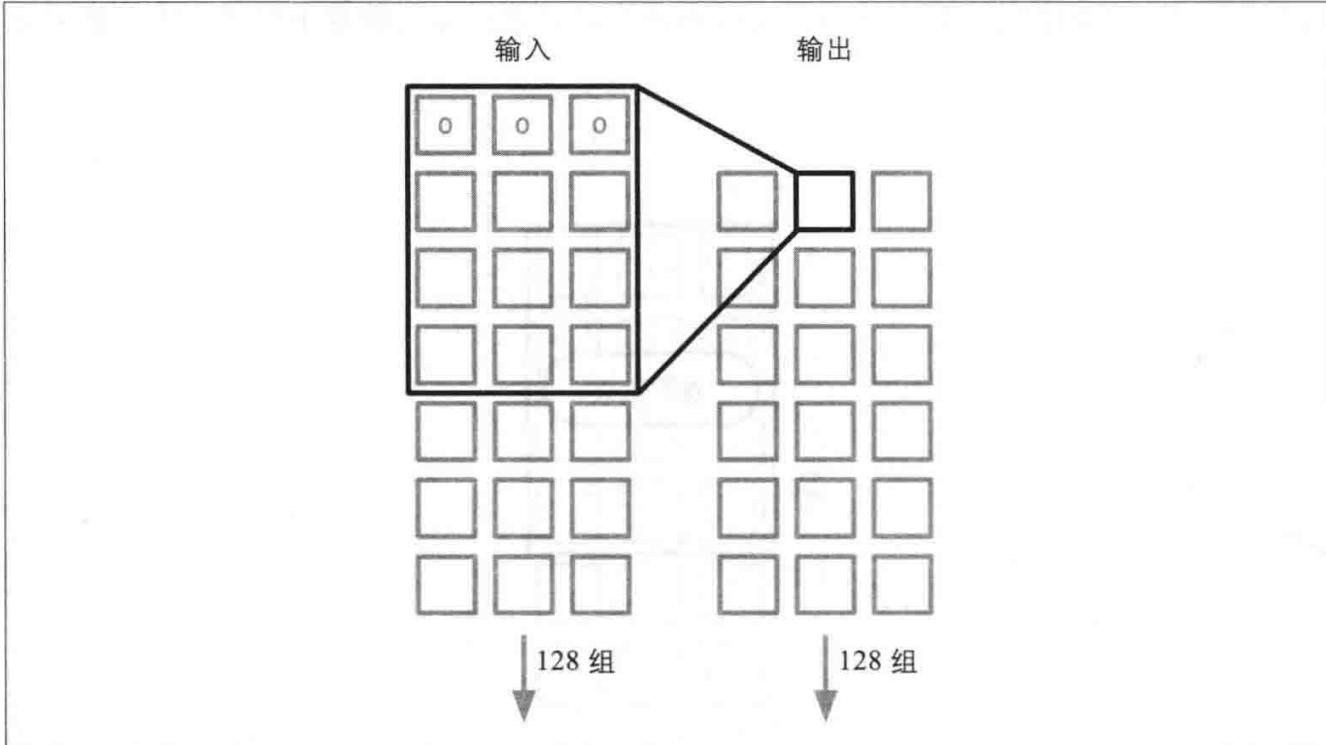


图 12-11：卷积窗口已移至第二个位置，只需在顶部填充数据

一旦卷积窗口覆盖了所有的数据，并使用每个过滤器创建了 8 个不同的特征图，输出就将被传递到下一层 MaxPool2D：

```
tf.keras.layers.MaxPool2D((3, 3)), # (batch, 42, 1, 8)
```

MaxPool2D 层接收来自上一层的输出，即 $(128, 3, 8)$ 的张量，并将其缩小为 $(42, 1, 8)$ 的张量——原始大小的三分之一。它查看输入数据的窗口，然后在窗口中选择最大值，并将该值作为输出。然后，使用下一个数据窗口重复这个过程。我们提供给 MaxPool2D() 函数的参数 $(3, 3)$ 指定 MaxPool2D 应使用 3×3 的窗口。默认情况下，窗口始终处于移动状态，以使它总是包含全新的数据。图 12-12 展示了其工作方式。

请注意，尽管图 12-12 将每个元素显示为一个值，但实际上在我们的数据里每个元素有 8 个特征通道。

但是为什么我们需要像这样缩小输入呢？当用于分类时，CNN 的目标是将大而复杂的输入张量转换成小而简单的输出。MaxPool2D 层有助于实现这一目标。它将第一个卷积层的输出简化为包含其相关信息的更集中、更高级的表示。

通过集中这些信息，我们可以剔除那些与任务无关的内容，即无法帮助我们识别输入中包含哪个手势的信息。这使得在第一个卷积层的输出中，能被最大程度体现的最重要的特征被保留了下来。有趣的是，尽管我们原始输入的测量值具有三个加速度传感器轴，

但 Conv2D 和 MaxPool2D 的组合已经将它们合并为一个值。

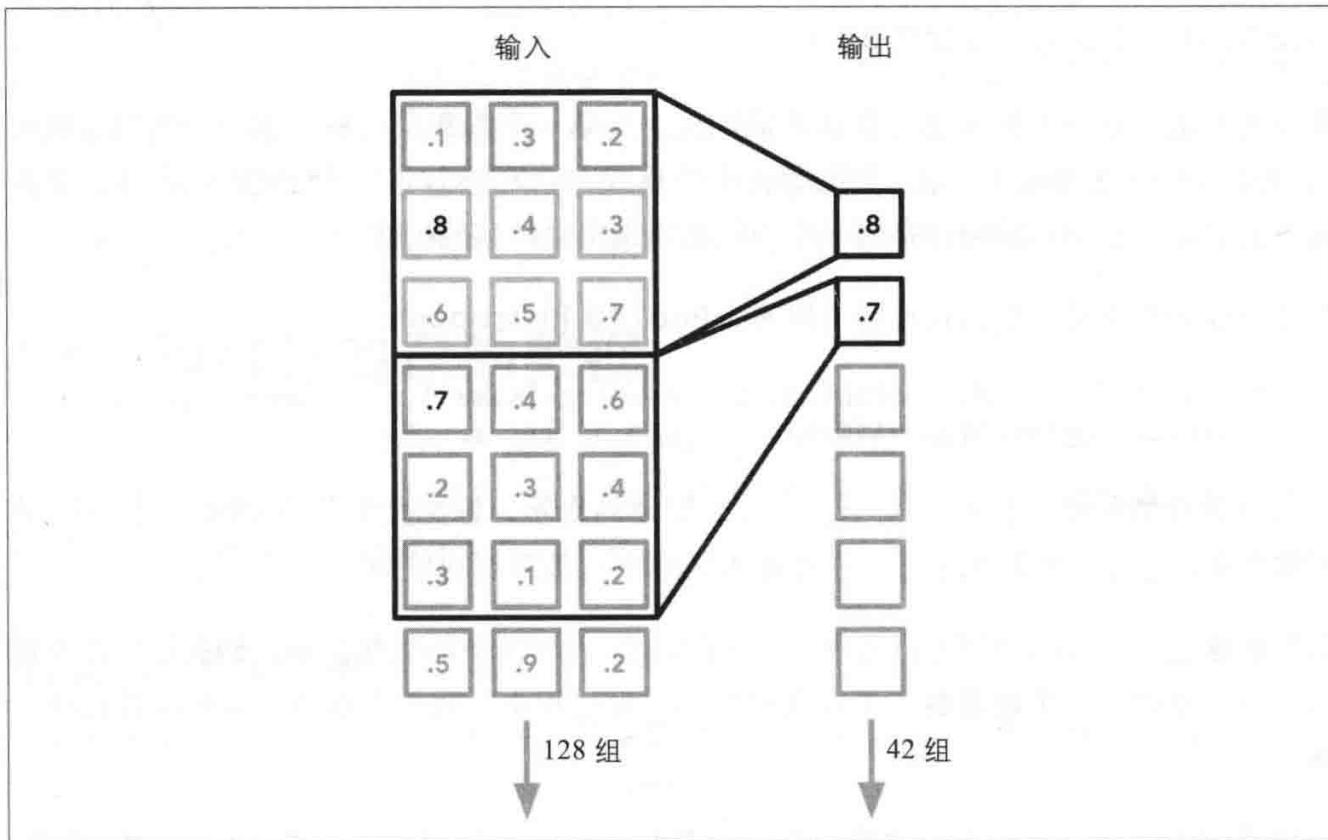


图 12-12: 最大池化 (max pooling) 的工作方式

在缩小数据之后，数据将通过下一个 Dropout 层：

```
tf.keras.layers.Dropout(0.1), # (batch, 42, 1, 8)
```

在训练过程中，Dropout 层会随机地将张量中的一些值设置为零。在我们的例子中，通过调用 `Dropout(0.1)`，我们将 10% 左右的数据设置为零，即完全消除了这些数据。这看起来似乎很奇怪，所以让我们来解释一下。

dropout 是一种正则化技术。正如书中前面提到的，正则化是改善机器学习模型的过程，可以使得模型不太可能过拟合训练数据。dropout 是一种限制过拟合的简单而有效的方法。通过随机删除某一层和下一层之间的一些数据，我们迫使神经网络学习如何应对意外的噪声和变化。在层之间添加 dropout 是一种常见且有效的实践。

Dropout 层只会在训练期间处于活动 (active) 状态。推断过程中 Dropout 层不会工作，而是直接允许所有数据通过。

在 Dropout 层之后，我们再次为 MaxPool2D 层和 Dropout 层提供数据：

```
tf.keras.layers.Conv2D(16, (4, 1), padding="same",
                      activation="relu"), # (batch, 42, 1, 16)
```

该层具有 16 个过滤器，窗口大小为 $(4, 1)$ 。这些数字属于模型超参数的一部分，并且是在模型开发过程中通过迭代选择的。设计一个有效的架构是一个反复试验的过程，这些神奇的数字是经过大量试验获得的。

你不太可能会在一开始就能选择出正确的值。与第一个卷积层一样，这一个卷积层也可以学习会识别包含有意义信息的相邻值中的模式。它的输出是对于给定输入内容的更高级别的表示。它识别的特征是第一个卷积层识别出的特征的组合。

在这个卷积层之后，我们执行另一组 MaxPool2D 和 Dropout：

```
tf.keras.layers.MaxPool2D((3, 1), padding="same"), # (batch, 14, 1, 16)
tf.keras.layers.Dropout(0.1), # (batch, 14, 1, 16)
```

这会继续将原始输入提取为更小、更易于管理的表示。输出是形状为 $(14, 1, 16)$ 的多维张量，它象征性地表示了包含在输入数据中的最重要的结构。

如果愿意，我们可以继续进行卷积和池化的过程。CNN 中的层数只是我们在模型开发期间可以调整的另一个超参数。不过在模型的开发过程中，我们发现两个卷积层就已经足够了。

到目前为止，我们一直在通过卷积层处理数据，这些卷积层只关心相邻值之间的关系，所以实际上我们并没有从全局意义上考虑过数据。但是，由于现在有了对输入中包含的主要特征的高层表示，因此可以“缩小”（zoom out）并整体地研究它们。为此，我们将数据展平并将其输入一个 Dense 层（密集层，也称为全连接层）：

```
tf.keras.layers.Flatten(), # (batch, 224)
tf.keras.layers.Dense(16, activation="relu"), # (batch, 16)
```

Flatten 层用于将多维张量转换为一维张量。在我们的例子中，形状为 $(14, 1, 16)$ 的多维张量将被压缩为形状为 (224) 的一维向量。然后它会被送入一个包含 16 个神经元的 Dense 层。这是深度学习工具箱中最基本的工具之一，它的每个输入都连接到一个神经元。通过同时考虑所有的数据，这一层可以了解输入中各种组合的含义。这个 Dense 层的输出将是一组（16 个）值，它以高度压缩的形式表示了原始输入的内容。

我们的最终任务是将这 16 个值缩小为 4 类。为此，我们首先加入更多的 dropout，然后使用最后的 Dense 层：

```
tf.keras.layers.Dropout(0.1), # (batch, 16)
tf.keras.layers.Dense(4, activation="softmax") # (batch, 4)
```

最后一层有 4 个神经元。每个神经元代表一种类型的手势。每个神经元都与上一层的所有 16 个输出相连。在训练过程中，每个神经元将学习与其所代表的手势相对应的上一层激活的组合。

这一层配置了一个 softmax 激活函数，它的输出是一组概率的组合，所有概率总和为 1。这个输出就是我们在模型的输出张量中看到的。

这种类型的模型架构——卷积层和全连接层的组合——在对基于时间序列的传感器数据（例如从加速度传感器获得的测量值）进行分类时非常有用。这种模型会学习识别代表特定输入类别的高级“指纹”特征。它体积小、运行速度快，而且不需要很长时间的训练。对于嵌入式机器学习工程师来说，这种模型架构将会是非常有用的工具。

12.3 训练你自己的数据

在本节中，我们将向你展示如何训练自己的自定义模型来识别新的手势。我们将逐步介绍如何获取加速度传感器数据，如何修改训练脚本以合并数据，如何训练新的模型，并将模型集成到嵌入式应用程序中。

12.3.1 获取数据

要获取训练数据，我们可以使用一个简单的程序在执行手势时将加速度传感器数据记录到串口。

SparkFun Edge

最快的入门方法是修改 SparkFun Edge 开发板支持软件包（Board Support Package, BSP）中的一个示例。首先，按照 SparkFun 的“将 SparkFun Edge 开发板与 Ambiq Apollo3 SDK 结合使用”指南设置 Ambiq SDK 和 SparkFun Edge BSP。

下载 SDK 和 BSP 之后，你需要调整示例代码，使它能够完成我们要求的工作。

首先，在你选择的文本编辑器中打开文件 *AmbiqSuite-Rel2.2.0/boards/SparkFun_Edge_BSP/examples/example1_edge_test/src/tf_adc/tf_adc.c*。在文件的第 61 行找到对 *am_hal_adc_samples_read()* 的调用：

```
if (AM_HAL_STATUS_SUCCESS != am_hal_adc_samples_read(g_ADCHandle,
                                                       NULL,
                                                       &ui32NumSamples,
                                                       &Sample))
```

将第二个参数更改为 *true*，以使整个函数调用如下所示：

```
if (AM_HAL_STATUS_SUCCESS != am_hal_adc_samples_read(g_ADCHandle,
                                                       true,
                                                       &ui32NumSamples,
                                                       &Sample))
```

接下来，你需要修改文件 *AmbiqSuite-Rel2.2.0/boards/SparkFun_Edge_BSP/examples/*

`example1_edge_test/src/main.cc`。在第 51 行找到 `while` 循环：

```
/*
 * Read samples in polling mode (no int)
 */
while(1)
{
    // Use Button 14 to break the loop and shut down
    uint32_t pin14Val = 1;
    am_hal_gpio_state_read( AM_BSP_GPIO_14, AM_HAL_GPIO_INPUT_READ, &pin14Val);
```

更改代码以添加以下代码中额外的行：

```
/*
 * Read samples in polling mode (no int)
 */
while(1)
{
    am_util_stdio_printf("-,-,-\r\n");
    // Use Button 14 to break the loop and shut down
    uint32_t pin14Val = 1;
    am_hal_gpio_state_read( AM_BSP_GPIO_14, AM_HAL_GPIO_INPUT_READ, &pin14Val);
```

然后，在 `while` 循环中进一步查找以下行：

```
am_util_stdio_printf("Acc [mg] %04.2f x, %04.2f y, %04.2f z,
                      Temp [deg C] %04.2f, MICO [counts / 2^14] %d\r\n",
                      acceleration_mg[0], acceleration_mg[1], acceleration_mg[2],
                      temperature_degC, (audioSample) );
```

删除原始行，并将其替换为以下内容：

```
am_util_stdio_printf("%04.2f,%04.2f,%04.2f\r\n", acceleration_mg[0],
                      acceleration_mg[1], acceleration_mg[2]);
```

程序现在将以训练脚本期望的格式输出数据。

接下来，按照 SparkFun 指南中的说明构建 `example1_edge_test` 示例应用程序，并将它烧录到设备闪存中。

记录数据

编译并烧录示例代码后，请按照以下说明捕获一些数据。

首先，打开一个新的终端窗口。然后运行以下命令，开始将终端的所有输出记录到名为 `output.txt` 的文件中：

```
script output.txt
```

接下来，在同一窗口中，使用 `screen` 连接到设备：

```
screen ${DEVICENAME} 115200
```

加速度传感器的测量值将显示在屏幕上，并以训练脚本期望的逗号分隔格式保存到 *output.txt* 文件中。

你应该在一个文件中捕获同一手势的多次执行数据。要开始采集手势的一次执行数据，请按标记为 RST 的按键。字符 -,-,- 将被写入串行端口，训练脚本使用这个输出来识别手势执行的开始。完成手势后，请按标记为 14 的按键以停止记录数据。

多次记录相同的手势后，请按 Ctrl+A 退出 `screen`，紧接着按 K 键，然后按 Y 键。退出 `screen` 后，输入以下命令以停止将数据记录到 *output.txt*：

```
exit
```

现在，你有一个 *output.txt* 文件，其中包含一个人执行某一种手势的数据。要训练一个全新的模型，你应该收集与原始数据集数量相近的数据，即收集接近 10 个人的数据且每个分别执行每种手势 15 次。

如果你不希望除了你以外的人使用你的模型，那么你可以只捕捉自己的手势的执行过程。而且，你收集的手势变化越多越好。

为了与训练脚本兼容，你应该使用以下格式重命名采集到的数据文件：

```
output_<gesture_name>_<person_name>.txt
```

例如，Daniel 尝试做出的“画角”手势的数据将被命名为：

```
output_triangle_Daniel.txt
```

训练脚本将期望通过以每种类型的手势命名的目录组织数据，例如：

```
data/
  triangle
    output_triangle_Daniel.txt
    ...
  square
    output_square_Daniel.txt
    ...
  star
    output_star_Daniel.txt
    ...
```

你还需要在名为 *negative* 的目录中提供“未知”类别的数据。在这种情况下，你可以直接重用原始数据集中的数据文件。

请注意，由于模型的架构旨在输出 4 个类别（3 个手势加“未知”）的概率，因此你也应

该提供自己的 3 个手势。如果你想训练更多或更少的手势，则需要更改训练脚本并调整模型架构。

12.3.2 修改训练脚本

要使用新的手势数据训练模型，你需要对训练脚本进行一些更改。

首先，替换以下文件中的所有手势名称：

- *data_load.py*
- *data_prepare.py*
- *data_split.py*

接下来，替换以下文件中的所有人的姓名：

- *data_prepare.py*
- *data_split_person.py*

请注意，如果你使用不同数量的人员名称（原始数据集有 10 个），并且想要在训练过程中按人员划分数据，则需要确定新的划分方式。如果你只有少数几个人的数据，那么在训练期间就不太可能按人进行划分，因此不必关心 *data_split_person.py*。

12.3.3 训练

要训练一个新的模型，请将你的数据文件目录复制到训练脚本的目录中，并按照我们在本章前面介绍的过程进行操作。

如果你只有少数几个人的数据，则应该随机划分数据，而不是按人员划分数据。为此，在准备训练时，请运行 *data_split.py* 而不是 *data_split_person.py*。

由于你正在训练新的手势，因此值得调整模型的超参数来获得最佳的准确率。例如，你可以查看是否通过训练更多或更少的轮次，或者使用不同的层数，或者使用不同数量的神经元，或者使用不同的卷积超参数来尝试获得更好的结果。你可以使用 TensorBoard 来监控训练进展。

一旦有了一个准确率可以接受的模型，你就需要对项目进行一些更改以确保它可以正常工作。

12.3.4 使用新的模型

首先，你需要将 `xxd -i` 格式的新模型的数据复制到 *magic_wand_model_data.cc* 文件

中。请确保你还更新了 `g_magic_wand_model_data_len` 的值以匹配 `xxd` 输出的数字。

接下来，你需要更新 `accelerometer_handler.cc` 的数组 `should_continuous_count` 中的值，这些值指定每个手势所需的连续预测次数，对应于执行手势所需的时间。假设原始“画翅膀”手势需要连续预测 15 次，则估计新手势相对于“画翅膀”需要花费多长时间，并更新数组中的值。你可以迭代地调整这些值，直到获得最可靠的表现。

最后，更新 `output_handler.cc` 中的代码来为新手势打印正确的名称。完成这些工作后，你就可以编译代码并烧录设备了。

12.4 小结

在本章中，我们深入研究了典型的嵌入式机器学习模型的架构。这种卷积模型是对时序数据进行分类的强大工具，你以后可能会经常遇到它。

到目前为止，希望你对嵌入式机器学习应用程序的全貌有了一定的了解，并且知道了这些应用程序代码是如何与模型一起工作以了解周围的世界的。在创建你自己的项目时，你将开始组合一个装有你熟悉的模型的工具箱，用于解决不同的问题。

学习机器学习

本书的目标是对嵌入式机器学习的可能性进行简要的介绍，而嵌入式机器学习并不是完整的机器学习本身。如果你想更深入地了解如何创建自己的模型，那么以下是一些非常棒而且很容易得到的资源，适合任何背景的读者，它们将为你学习机器学习提供一个良好的开端。

以下是一些我们最熟悉的资料，它们将会帮助你更深入地学习在本书中接触到的知识：

- François Chollet 的 *Deep Learning with Python* (Manning)。
- Aurélien Géron 的 *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Edition (O'Reilly)。
- Deeplearning.ai 的“Deep Learning Specialization”和“TensorFlow in Practice”课程。
- Udacity 的“Intro to TensorFlow for Deep Learning”课程。

接下来的内容

本书的其余章节将更深入地探讨嵌入式机器学习的工具和工作流程。你将学习如何

设计自己的 TinyML 应用程序，如何优化模型和应用程序代码以在低能耗的设备上良好地运行，如何将现有的机器学习模型移植到嵌入式设备，以及如何调试嵌入式机器学习代码。我们还将解决一些高层级的问题，例如关于部署、隐私和安全性等问题。

让我们先进一步了解 TensorFlow Lite 框架，该框架为本书中的所有示例提供了强大的支持。

TensorFlow Lite for Microcontrollers

在本章中，我们将研究本书中所有示例所使用的软件框架：TensorFlow Lite for Microcontrollers。我们会深入研究很多细节，但你不需要了解我们涵盖的所有内容，就可以在应用程序中使用它。如果你对框架背后发生的事情不感兴趣，可以跳过这一章，当遇到问题的时候，你总是可以回到这一章。为了让你更好地理解用来运行机器学习的工具，我们将在此处介绍 TensorFlow Lite for Microcontrollers 的历史和内部工作原理。

13.1 什么是 TensorFlow Lite for Microcontrollers

你可能会问的第一个问题是这个框架实际上是做什么的。要理解这一点，将（相当长的）名称进行分解并解释其各个组成部分会很有帮助。

13.1.1 TensorFlow 简介

如果你研究过机器学习，那么你可能已经听说过 TensorFlow。TensorFlow 是 Google 的开源机器学习库，其格言是“面向所有人的开源机器学习框架”（An Open Source Machine Learning Framework for Everyone）。TensorFlow 是 Google 内部开发的，于 2015 年首次发布。从那时起，围绕 TensorFlow 的大型外部社区就逐渐发展起来，来自 Google 之外的贡献者比来自 Google 内部的还多。TensorFlow 的目标平台有 Linux、Windows 以及 macOS 等桌面系统和服务器平台，它提供了许多工具、示例以及针对在云上训练和部署模型的优化。TensorFlow 是 Google 内部用来为其产品提供动力的主要机器学习库，其核心代码在内部版本和发布版本中都是一样的。

Google 以及其他一些信息来源也提供了大量的示例和教程。这些示例和教程可以告诉你如

何训练与使用各种模型，内容涵盖从语音识别到数据中心电源管理（power management）以及视频分析等。

在启动 TensorFlow 项目时，我们最大的需求是能够在桌面环境中训练并运行模型。这种需求影响了很多设计决策，例如为了更低的延迟和更多的功能而增加可执行文件的大小。这是因为在云端服务器上，RAM（内存）甚至可以将 GB 作为衡量单位，而存储空间更是以 TB 为单位，所以几百兆字节的二进制文件通常不是问题。另一个例子是，TensorFlow 在发布时的主要接口语言是 Python，这是一种在服务器上广泛使用的脚本语言。

不过，这些工程上的取舍并不适用于其他平台。在 Android 设备和 iPhone 上，哪怕是将应用程序大小仅仅增加几兆字节也可能会大大减少下载次数并降低客户满意度。你可以为这些手机平台编译 TensorFlow，但是默认情况下，这将会使应用程序至少增加 20MB，即使进行一些优化也很难缩减到 2MB 以下。

13.1.2 TensorFlow Lite 简介

为了满足移动平台对较小尺寸的要求，Google 在 2017 年启动了一个与 TensorFlow 主线相关的项目 TensorFlow Lite。该库的目标是在移动设备上高效且轻松地运行神经网络模型。为了减少框架的大小和复杂性，TensorFlow Lite 删除了在移动平台上不常用的功能。例如，它不支持训练模型，而是仅支持使用之前在云平台上训练过的模型运行推断。它还不支持 TensorFlow 主线中可用的全部数据类型（例如 `double`）。此外，TensorFlow Lite 也不支持一些使用次数比较少的算子，比如 `tf.depth_to_space`。你可以在 TensorFlow 网站上找到最新的兼容信息。

作为对这些折中的回报，TensorFlow Lite 可以只用几百字节，从而使其更适合大小受到限制的应用程序。它还为 Arm Cortex-A 系列 CPU 提供了高度优化的库，同时支持 Android Neural Network API^{译注1} 加速，以及支持通过 OpenGL 使用 GPU 等。另一个关键优点是 TensorFlow Lite 对网络的 8 位量化有很好的支持。一个模型可能有数百万个参数，仅仅是将 32 位浮点数转换为 8 位整数就能减少 75% 的大小，这是十分值得的。当然，TensorFlow Lite 也有专用代码路径允许推断在较小的数据类型上运行得更快。

13.1.3 TensorFlow Lite for Microcontrollers 简介

TensorFlow Lite 已被移动开发人员广泛使用，但是它的工程设计仍不能满足所有平台的要求。TensorFlow Lite 项目团队注意到，在嵌入式平台上构建机器学习程序可以使许多 Google 产品和外部产品受益，而现有的 TensorFlow Lite 库并不适用于这些平台。同样，

译注 1：Android Neural Networks API (NNAPI) 是一个 Android C API，专为在 Android 设备上运行计算密集型运算从而实现机器学习而设计。内置于 Android 8.1 (SDK 27) 或更高版本的 Android 系统中。

最大的限制是二进制文件的大小。对于这些环境，即使是几百 KB 也太大了，它们需要支持 20KB 或更小的二进制文件。这些平台通常也不支持很多移动开发人员认为理所当然的依赖项，比如 C 标准库，因此不能使用依赖于这些库的代码。不过，很多需求都非常相似。推断是最主要的用例，量化的网络对性能很重要，因此，在这些平台上有一个足够简单的代码库供开发人员探索和修改是当务之急。

考虑到这些需求，Google 的一个团队（包括本书的作者）于 2018 年开始尝试试验针对这些嵌入式平台的专业版 TensorFlow Lite。目的是在满足嵌入式环境苛刻要求的同时，尽可能多地重用移动项目中的代码、工具和文档。为了确保 Google 正在构建一些实用的东西，该团队专注于识别语音“唤醒词”的实际用例，类似于商业语音接口中的“Hey Google”或“Alexa”示例。针对如何解决这个端到端的具体问题，Google 努力确保我们设计的系统可以被用于真实的产品系统。

13.1.4 需求

Google 团队知道，在嵌入式环境中运行的前提会对代码的编写方式造成很多限制，因此其确定了对 TensorFlow Lite for Microcontrollers 的一些关键要求：

没有操作系统依赖项

机器学习模型本质上是一个数学黑匣子，我们将数字输入模型，然后模型将返回数字作为结果。执行这些操作不需要访问系统的绝大部分，因此可以在不调用底层操作系统的情况下编写机器学习框架。由于一些目标平台根本没有操作系统，因此在基本代码中避免任何对文件或设备的引用使得代码能够被移植到这些芯片上。

链接时没有标准的 C 或 C++ 库依赖项

这个要求比对操作系统的要求更微妙一些，但是团队的目标是将代码部署在可能只有几十 KB 的内存用来存储程序的设备上，因此二进制文件的大小非常重要。像 `sprintf()` 这样的简单函数也很容易占用约 20KB 的空间，因此该团队的目标是避免从包含 C 和 C++ 标准库实现的代码库中提取任何内容。这个问题很棘手，因为在标准库中，header-only 依赖项^{译注 2}（如 `stdint.h`，它保存数据类型的大小）和链接器时（linker-time）部分（如许多字符串函数或 `sprintf()`）之间没有良好定义的边界。在实践中，团队必须使用一些常识来理解，通常，编译时常量和宏可以被接受，但是更复杂的事情应该被避免。避免链接的一个例外是标准 C `math` 库，该库是诸如三角函数之类的函数的依赖项，我们确实需要链接它。

不需要浮点硬件

许多嵌入式平台在硬件上不支持浮点运算，因此代码必须避免在任何性能关键的地

译注 2：header-only 依赖项指编译器可以仅通过头文件得到一个库包含的所有宏、函数、类的完整定义。

方使用浮点运算。这意味着需要着重于具有 8 位整数参数的模型，并在运算中使用 8 位算术（尽管为了兼容性，在必要的时候该框架支持需要的浮点数算子）。

没有动态内存分配

许多使用微控制器的应用程序需要连续运行几个月甚至几年。如果程序的主循环使用 `malloc()`/`new` 和 `free()`/`delete` 来分配和释放内存，就很难保证堆最终不会以碎片的状态结束，从而导致内存分配失败和程序崩溃。大多数嵌入式系统的可用内存也很少，因此，对这种有限资源的预先计划比在其他平台上更为重要，而且如果没有操作系统，就甚至可能没有堆和分配的例行程序。这意味着嵌入式应用程序通常会完全避免使用动态内存分配。由于 TensorFlow Lite for Microcontrollers 就是为这些应用程序设计的，所以也需要完全避免使用动态内存分配。在实践中，框架要求调用应用程序传入一个固定大小的内存空间，让框架可以在初始化时进行临时分配（例如激活缓冲区）。如果内存空间太小，TensorFlow Lite for Microcontrollers 将立即返回错误，调用者将需要使用更大的内存空间进行重新编译。除此之外，执行推断的调用将不会进行进一步的内存分配，因此可以重复进行调用，而不会有堆碎片（heap fragmentation）或内存错误的风险。

项目团队还决定不使用嵌入式开发社区中常见的其他一些约束，因为它们会使共享代码和维护与移动端 TensorFlow Lite 的兼容性变得太困难。因此：

需要 C++ 11

用 C 语言编写嵌入式程序是很常见的，而且某些平台根本没有支持 C++ 的工具链，或者只支持比 2011 年 C++ 标准的修订版更旧的版本。TensorFlow Lite 主要是用 C++ 编写的，有一些普通的 C API，这使得从其他语言进行调用变得更加容易。它不依赖于复杂模板之类的高级特性，其风格是本着“更好的 C”的精神，使用类来帮助模块化代码。用 C 语言重写框架是一项繁重的工作，并且对于移动平台上的用户来说是一种倒退。调查了最流行的平台后，我们发现这些平台都已经有了 C++ 11 的支持，所以团队决定用取消对旧设备的支持来换取在所有 TensorFlow Lite 中更轻松地共享代码。

预期 32 位处理器

嵌入式世界中有大量不同的硬件平台可用，但是近年来的趋势是转向 32 位处理器，而不是之前普遍的 16 位或 8 位芯片。在调查了生态系统之后，Google 决定将开发重点放在较新的 32 位设备上，因为在这样的假设下，框架的移动版本和嵌入式版本的 C 代码中的 `int` 数据类型均为 32 位。已经有报告称，对一些 16 位平台的移植是成功的，但这些移植依赖于弥补这些限制的现代工具链，这不是我们的首要任务。

13.1.5 为什么解释执行模型

一个经常被问到的问题是，为什么我们选择在运行时解释执行^{译注3} 模型，而不是提前从模型生成代码。解释执行这个决定需要我们分别讨论不同方法的优势和劣势。

代码生成（code generation）涉及将模型直接转换为 C 或 C++ 代码，将所有参数作为数据数组存储在代码中，而将其架构表示为一系列将激活从一层传递到下一层的函数调用。这些代码通常被输出到一个带有少量入口点的大型源文件中。然后可以将该文件直接包含在 IDE 或者工具链中，并像其他代码一样进行编译。以下是代码生成的一些关键优势：

易于编译

用户告诉我们，这种方法的第一个好处是它可以使代码很容易地被集成到编译系统中。如果你只拥有几个 C 或 C++ 文件，而没有任何外部库依赖项，就可以轻松地将它们拖入几乎任何 IDE 中，编译项目几乎不会出现任何错误。

可修改性

当一个实现文件中有少量代码时，如果需要，可以更轻松地单步调试和更改代码。相比之下，在一个大型库中，你甚至需要先确定该库使用哪种实现。

内联数据

模型本身的数据可以作为实现源代码的一部分存储，因此不需要任何其他文件。模型也可以被直接存储为内存中的数据结构，因此不需要加载或解析的步骤。

代码大小

如果你提前知道编译的模型和平台，就可以避免包含一些永远不会被调用的代码，因此程序段的大小可以保持最小。

解释执行模型是另一种方法，它依赖于加载定义模型的数据结构。执行的代码是静态的，只有模型数据在变化，模型中的信息控制了执行哪些算子以及从何处提取参数。这更像是使用 Python 之类的解释语言运行脚本，而你可以将代码生成看作接近于 C 语言等传统编译语言。以下是代码生成与解释模型数据结构相比的一些缺点：

译注 3：解释执行主要是相对于编译执行来说的。对于编程语言，执行代码需要将代码编译为机器代码，机器才能执行。编译执行需要编译器先将代码整体编译成机器代码再执行；而解释执行需要一个解释器，将代码在运行时逐句解释成机器代码来执行，即逐句解释、逐句执行。这里的“解释执行”并不是真正意义上编程语言级别的“解释”（因为都是 C++ 源代码，本质上是编译执行），而是指不会直接将模型代码转换成 C/C++ 代码，而是采用解释器模式，只加载定义模型的数据结构，使用静态的解释器执行模型。

可升级性

如果你在本地修改了生成的代码，但又想将这些修改升级到整个框架的新版本以获得新的功能或优化，这时会发生什么？你要么需要手动将更改并入本地文件中，要么需要完全重新生成代码，然后尝试将这些代码并入本地的修改中。

多个模型

在没有大量重复源代码的情况下，很难通过一次代码生成支持多个模型。

替换模型

每个模型都表示为程序中源代码和数据数组的混合体，因此，如果不重新编译整个程序就很难更改模型。

项目团队意识到，使用所谓的项目生成（project generation）方法，可以在规避代码生成的缺点的同时享受代码生成带来的诸多益处。

13.1.6 项目生成

在 TensorFlow Lite 中，项目生成是这样的过程：只创建编译特定模型所需的源文件的副本，而不做任何更改，还可以选择设置任何特定于 IDE 的项目文件，以便轻松地编译这些文件。这种方法保留了代码生成的大多数优点，同时更具有一些关键的优点：

可升级性

所有源文件都是 TensorFlow Lite 主要代码库中原始文件的副本，它们出现在文件夹层次结构中的同一位置。因此，如果你进行本地修改，则可以轻松地将其移植回原始源文件，并且对库的升级也可以简单地使用标准合并工具进行合并。

多种模型和替换模型

底层代码是一个解释器，因此你可以拥有多个模型，而且无须重新编译就可以轻松地更换数据文件。

内联数据

如果需要，模型参数本身仍然可以作为 C 数据数组编译到程序中，而对 FlatBuffers 序列化格式的使用意味着这种表示可以直接在内存中使用，而不需要解包或解析。

外部依赖项

编译项目所需的所有头文件和源文件都将与常规 TensorFlow 代码一起被复制到文件夹中，因此无须单独下载或安装依赖项。

使用项目生成很难做到直接减小代码大小，因为解释器结构使其更难发现永远不会被调用的代码路径。在 TensorFlow Lite 中，这是通过手动使用 OpResolver 机制并只注册希

望在应用程序中使用的 kernel 实现分别处理的。

13.2 编译系统

TensorFlow Lite 最初是在 Linux 环境中开发的，因此我们的许多工具都基于传统的 UNIX 工具，例如 shell 脚本、Make 和 Python。不过，我们知道对于嵌入式开发人员而言，这并不是一种常见的组合，因此我们致力于将其他平台和编译工具链一视同仁，一并提供支持。

我们通过使用前述的“项目生成”达成这个目标。如果你从 GitHub 上获取 TensorFlow 源代码，则可以在 Linux 上使用标准 Makefile 方法为许多平台进行编译。例如，此命令行编译并测试该库的 x86 版本：

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

你可以使用以下命令来编译特定目标，例如 SparkFun Edge 平台的语音唤醒词示例：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
TARGET="sparkfun_edge" micro_speech_bin
```

如果你的项目在 Windows 计算机上运行，或者你想使用 Keil、Mbed、Arduino 或其他专用编译系统之类的 IDE，该怎么办？这就是“项目生成”的用武之地。你可以从 Linux 运行以下命令行来生成可用于 Mbed IDE 的文件夹：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
TARGET="disco_f746ng" generate_micro_speech_mbed_project
```

现在你应该能在 *tensorflow/lite/micro/tools/make/gen/disco_f746ng_x86_64/prj/micro_speech/mbed/* 中看到一组源文件以及在 Mbed 环境中需要编译的所有依赖项和项目文件。同样的方法适用于 Keil 和 Arduino，还有一个通用版本，它只输出源文件的文件夹层次结构，而不输出项目元信息 (metainformation)，尽管它确实包含一个 Visual Studio 代码文件，而该文件定义了一系列编译规则。

你可能想知道，这种 Linux 命令行方法如何帮助你使用其他平台。在每晚持续集成工作流以及发布主要版本时，我们都会自动运行这个项目生成脚本。每当运行时，脚本都会自动将结果文件放在一个公共 Web 服务器上。这意味着所有平台上的用户都应该能够找到其首选 IDE 的版本，并将该项目作为独立的文件夹下载，而不是通过 GitHub 下载。

13.2.1 专用代码

代码生成的好处之一是，可以很容易地重写库的一部分，以便在特定的平台上正常工作，甚至只需针对你知道的在用例中常见的特定参数集来优化函数。我们不想失去这

种易修改性，但我们也希望尽可能容易地将更普遍有用的更改合并回主框架的源代码中。我们还有一个额外的约束，即某些编译环境在编译过程中无法轻松传递自定义 `#define` 宏，因此我们不能依赖在编译时使用宏防护来切换到不同的实现。

为解决此问题，我们将库分为一个个小模块，每个模块都有一个实现其功能默认版本的 C++ 文件，以及一个 C++ 头文件，该头文件定义了可供其他代码调用的模块接口。然后我们采用了一个约定，如果你想编写一个模块的专用版本，就需要将新版本保存为 C++ 实现文件，该文件与原始文件的名称相同，但保存在原始文件所在目录的子文件夹中。该子文件夹的名称应为其专门针对的平台或功能的名称（参见图 13-1），并且当你为该平台或功能进行编译时，Makefile 或生成的项目将自动使用此子文件夹中的实现，而不是原始实现。这听起来似乎很复杂，所以让我们来看几个具体的例子。



图 13-1：专用音频提供程序文件的屏幕截图

语音唤醒词示例代码需要从麦克风获取音频数据，但不幸的是，我们没有跨平台的方式来捕获音频。因为至少需要在各种设备上进行编译，所以我们编写了一个默认实现，该实现无须使用麦克风，而是仅返回一个全是零值的缓冲区。以下是来自 `audio_provider.h` 的模块接口：

```
TfLiteStatus GetAudioSamples(tfLite::ErrorReporter* error_reporter,
                           int start_ms, int duration_ms,
                           int* audio_samples_size, int16_t** audio_samples);
int32_t LatestAudioTimestamp();
```

第一个函数为给定的时间段输出一个充满音频数据的缓冲区，如果出错，则返回一个错误。第二个函数返回捕获最新音频数据的时间，以便客户端可以请求正确的时间范围，并知道新数据是何时到达的。

由于默认实现不依赖于麦克风，因此 `audio_provider.cc` 中两个函数的实现非常简单：

```
namespace {
int16_t g_dummy_audio_data[kMaxAudioSampleSize];
```

```

int32_t g_latest_audio_timestamp = 0;
} // namespace

TfLiteStatus GetAudioSamples(tflite::ErrorReporter* error_reporter,
                            int start_ms, int duration_ms,
                            int* audio_samples_size, int16_t** audio_samples) {
  for (int i = 0; i < kMaxAudioSampleSize; ++i) {
    g_dummy_audio_data[i] = 0;
  }
  *audio_samples_size = kMaxAudioSampleSize;
  *audio_samples = g_dummy_audio_data;
  return kTfLiteOk;
}

int32_t LatestAudioTimestamp() {
  g_latest_audio_timestamp += 100;
  return g_latest_audio_timestamp;
}

```

每次调用函数时，时间戳都会自动递增，这样调用端的行为就好像有新数据进来一样，但是每次捕获例程都返回相同的零数组。这样做的好处是，即使你的系统还没有真的开始使用麦克风，你也可以对示例代码进行原型制作和试验。`kMaxAudioSampleSize` 在模型头文件中定义，代表函数需要的最大样本数。

在真正的设备上，代码会复杂得多，所以我们需要一个新的实现。之前，我们为 STM32F746NG Discovery 套件开发板编译了此示例，该套件内置了麦克风，并使用单独的 Mbed 库访问麦克风。代码位于 `disco_f746ng/audio_provider.cc`。由于文件太大，因此其未包含在内联代码中，但是如果查看该文件，则会看到它实现了与默认 `audio_provider.cc` 相同的两个公共函数：`GetAudioSamples()` 和 `LatestAudioTimestamp()`。这些函数的定义要复杂得多，但是从调用端的角度来看，它们的行为是相同的。复杂的实现被隐藏起来了，尽管平台有所变化，但调用端的代码可以保持不变。现在，被麦克风捕获的音频将出现在函数返回的缓冲区中，而不是每次都是全零数组。

如果查看此专用实现的完整路径 `tensorflow/lite/micro/examples/micro_speech/disco_f746ng/audio_provider.cc`，你会发现该路径与默认实现的路径 `tensorflow/lite/micro/examples/micro_speech/audio_provider.cc` 基本相同。但是专用实现位于与原始实现同一级目录下的子目录 `disco_f746ng` 中。如果回头看一下用于编译 STM32F746NG Mbed 项目的命令行，你会看到我们传入了 `TARGET=disco_f746ng` 以指定想要的平台。编译系统总是在带有目标名的子文件夹中查找 `.cc` 文件，以找到可能的专用实现。所以在这个例子中，编译系统将会使用 `disco_f746ng/audio_provider.cc` 替换父文件夹中的默认 `audio_provider.cc` 版本。当源文件被集成为 Mbed 项目副本时，子目录中的文件被复制，而父级目录中的 `.cc` 文件会被忽略。因此，最终生成的项目将使用专用代码版本。

捕获音频几乎在每个平台上都是不同的，因此我们有很多针对这个模块的不同专用实

现。甚至还有一个 macOS 版本 *osx/audio_provider.cc*, 如果你在 Mac 笔记本电脑上进行本地调试, 这将非常有用。

不过, 这种机制不仅用于提高可移植性, 它还足够灵活, 可以用于优化实现。实际上, 我们在语音唤醒词示例中使用了这种方法, 以帮助加快深度卷积运算。如果你查看 *tensorflow/lite/micro/kernels*, 将会看到 TensorFlow Lite for Microcontrollers 支持的所有算子的实现。这些默认实现被编写得简短、易于理解且支持在任何平台上运行, 但实现这些目标意味着这些实现的性能可能不是最优的。优化通常会使算法更复杂、更难理解, 因此这些参考实现预计会运行得相对缓慢。我们的想法是让开发人员首先可以用最简单的方式运行代码, 并确保他们得到正确的结果, 然后能够逐步更改代码以提高性能。这意味着可以对每一个小的更改进行测试, 以确保该更改不会破坏正确性, 从而使调试更加容易。

语音唤醒词示例中使用的模型在很大程度上依赖于深度卷积算子, 该模型在 *tensorflow/lite/micro/kernels/depthwise_conv.cc* 中有一个未优化的版本。其核心算法的实现位于 *tensorflow/lite/kernels/internal/reference/depthwiseconv_uint8.h*, 并被编写为一组简单的嵌套循环。代码如下:

```
for (int b = 0; b < batches; ++b) {
    for (int out_y = 0; out_y < output_height; ++out_y) {
        for (int out_x = 0; out_x < output_width; ++out_x) {
            for (int ic = 0; ic < input_depth; ++ic) {
                for (int m = 0; m < depth_multiplier; m++) {
                    const int oc = m + ic * depth_multiplier;
                    const int in_x_origin = (out_x * stride_width) - pad_width;
                    const int in_y_origin = (out_y * stride_height) - pad_height;
                    int32 acc = 0;
                    for (int filter_y = 0; filter_y < filter_height; ++filter_y) {
                        for (int filter_x = 0; filter_x < filter_width; ++filter_x) {
                            const int in_x =
                                in_x_origin + dilation_width_factor * filter_x;
                            const int in_y =
                                in_y_origin + dilation_height_factor * filter_y;
                            // If the location is outside the bounds of the input image,
                            // use zero as a default value.
                            if ((in_x >= 0) && (in_x < input_width) && (in_y >= 0) &&
                                (in_y < input_height)) {
                                int32 input_val =
                                    input_data[Offset(input_shape, b, in_y, in_x, ic)];
                                int32 filter_val = filter_data[Offset(
                                    filter_shape, 0, filter_y, filter_x, oc)];
                                acc += (filter_val + filter_offset) *
                                    (input_val + input_offset);
                            }
                        }
                    }
                    if (bias_data) {
                        acc += bias_data[oc];
                    }
                }
            }
        }
    }
}
```

```
    }
    acc = DepthwiseConvRound<output_rounding>(acc, output_multiplier,
                                                output_shift);
    acc += output_offset;
    acc = std::max(acc, output_activation_min);
    acc = std::min(acc, output_activation_max);
    output_data[Offset(output_shape, b, out_y, out_x, oc)] =
        static_cast<uint8>(acc);
}
}
}
}
```

只需快速查看，你可能就会看到许多提升运行速度的机会，比如预先计算每次在内部循环中计算的所有数组索引。这些更改将增加代码的复杂性，因此对于本参考实现，我们避免了这些更改。但是，语音唤醒词示例需要每秒在微控制器上运行多次，并且事实证明，这种朴素的实现是影响 SparkFun Edge Cortex-M4 处理器速度的主要瓶颈。为了使示例以可用的速度运行，我们需要添加一些优化。

为了提供优化的实现，我们在 `tensorflow/lite/micro/kernels` 中创建一个名为 `portable_optimized` 的新子文件夹，并在其中新建一个名为 `depthwise_conv.cc` 的新 C++ 源文件。这将比参考实现复杂得多，并且会利用语音模型的特定功能来实现专用优化。例如，卷积窗口的宽度为 8 比特的倍数，所以我们可以内存中使用两个 32 比特的词来加载这些值，而不是 8 个单独的字节。

你可能会注意到，我们已将子文件夹命名为 `portable_optimized`，而不是像上一个示例那样特定于平台命名。这是因为我们所做的任何更改都与特定的芯片或库无关，这些修改是一般性的优化，有望在各种处理器上加快运行速度，例如预先计算数组索引或将多个字节值作为较大的词加载。然后，我们通过将 `portable_optimized` 添加到 `ALL_TAGS` 列表，指定该实现可在 `make` 项目文件中使用。由于存在此标记，并且同名的子文件夹内有 `depthwise_conv.cc` 的实现，因此链接时将使用已优化的实现，而不是默认的参考版本。

希望这些示例可以向你展示清楚如何使用子文件夹机制来扩展和优化库代码，同时保持核心实现精简且易于理解。

13.2.2 Makefile

在易于理解这个话题上，Makefile 做得并不好。Make 编译系统已经有 40 多年的历史了，它具有许多令人困惑的特性，例如使用制表符作为有意义的语法或通过陈述性规则间接指定编译目标。我们选择使用超越 Make 的替代方案，如 Bazel 或 Cmake，因为这些编译系统足够灵活，可以实现复杂的行为，如项目生成。我们希望 TensorFlow Lite for

Microcontrollers 的大多数用户将在更现代的 IDE 中使用这些生成的项目，而不是直接与 Makefile 交互。

如果你要对核心库进行更改，则可能需要更多地了解 Makefile 中隐藏的内容，因此本节将介绍一些需要熟悉才能进行修改的约定和帮助函数。



如果你在 Linux 或 macOS 上使用 bash 终端，则应该可以通过键入通常的 `make -f tensorflow/lite/micro/tools/make/Makefile` 命令，然后按 Tab 键来查看所有可用的目标（可以编译的目标的名称）。在查找或调试目标时，这种自动补全功能非常有用。

如果你只是添加模块或算子的专用版本，则完全不需要更新 Makefile。有一个名为 `specialize()` 的自定义函数，该函数会自动获取名为 `ALL_TAGS` 的字符串列表（使用平台名称以及所有自定义标签填充）和源文件列表，并以正确的专用版本（代替原始版本）返回该源文件列表。如果你愿意的话，还可以在命令行上手动指定标记。例如，像这样：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
      TARGET="bluepill" TAGS="portable_optimized foo" test
```

这将生成一个看起来像“bluepill portable_optimized foo”的 `ALL_TAGS` 列表，并且对于每个源文件，它都将搜索子文件夹，以查找任何要用来替代的特殊版本。

如果你只是将新的 C++ 文件添加到标准文件夹中，则不需要更改 Makefile，因为其中大多数文件都由通配符规则自动拾取，例如 `MICROLITE_CC_BASE_SRCS` 的定义。

Makefile 依赖于定义要在根目录级别生成的源文件和头文件的列表，然后根据指定的平台和标记修改它们。这些修改发生在父编译项目中包含的 sub-Makefile。例如，`tensorflow/lite/micro/tools/make/targets` 文件夹中所有的 `.inc` 文件都会被自动包含在内。如果查看其中的一个文件，例如用于 Ambiq 和 SparkFun Edge 平台的 `apollo3evb_makefile.inc`，你就可以看到该文件检查了是否已经为此编译指定它的目标芯片。如果指定了，这将定义许多标志并修改源文件列表。这是一个简短的版本，其中包含一些最有趣的部分：

```
ifeq ($(TARGET),$(filter $(TARGET),apollo3evb sparkfun_edge))
  export PATH := $(MAKEFILE_DIR)/downloads/gcc_embedded/bin/:$(PATH)
  TARGET_ARCH := cortex-m4
  TARGET_TOOLCHAIN_PREFIX := arm-none-eabi-
...
  $(eval $(call add_third_party_download,$(GCC_EMBEDDED_URL), \
    $(GCC_EMBEDDED_MD5),gcc_embedded,))
  $(eval $(call add_third_party_download,$(CMSIS_URL),$(CMSIS_MD5),cmsis,))
...
  PLATFORM_FLAGS = \
    -DPART_apollo3 \
```

```

-DAM_PACKAGE_BGA \
-DAM_PART_APOLLO3 \
-DGEMMLOWP_ALLOW_SLOW_SCALAR_FALLBACK \
...
LDFLAGS += \
-mthumb -mcpu=cortex-m4 -mfpu=fpv4-sp-d16 -mfloat-abi=hard \
-nostartfiles -static \
-Wl,--gc-sections -Wl,--entry,Reset_Handler \
...
MICROLITE_LIBS := \
$(BOARD_BSP_PATH)/gcc/bin/libam_bsp.a \
$(APOLLO3_SDK)/mcu/apollo3/hal/gcc/bin/libam_hal.a \
$(GCC_ARM)/lib/gcc/arm-none-eabi/7.3.1/thumb/v7e-m/fpv4-sp/hard/crtbegin.o \
-lm
INCLUDES += \
-isystem$(MAKEFILE_DIR)/downloads/cmsis/CMSIS/Core/Include/ \
-isystem$(MAKEFILE_DIR)/downloads/cmsis/CMSIS/DSP/Include/ \
-I$(MAKEFILE_DIR)/downloads/CMSIS_ext/
...
MICROLITE_CC_SRCS += \
$(APOLLO3_SDK)/boards/apollo3_evb/examples/hello_world/gcc_patched/ \
startup_gcc.c \
$(APOLLO3_SDK)/utils/am_util_delay.c \
$(APOLLO3_SDK)/utils/am_util_faultisr.c \
$(APOLLO3_SDK)/utils/am_util_id.c \
$(APOLLO3_SDK)/utils/am_util_stdio.c

```

这是针对特定平台进行定制专用实现的地方。在此代码段中，我们向编译系统指明了要在哪里找到要使用的编译器以及要指定的架构。我们指定了一些需要下载的额外的外部库，例如 GCC 工具链和 Arm 的 CMSIS 库。我们正在为编译设置编译标志，并设置传递给链接器的参数，包括要链接的额外库和要查找的头文件的路径。我们还添加了一些额外的 C 文件，这些文件是在 Ambiq 平台上成功编译所需要的文件。

一个类似的 sub-Makefile 包含被用于编译示例。语音唤醒词示例代码在 *micro_speech/Makefile.inc* 中具有自己的 Makefile，并且定义了自己要编译的源代码文件列表，以及要下载的其他外部依赖项。

你可以使用 `generate_microlite_projects()` 函数为不同的 IDE 生成独立项目。这将获取源文件和标志的列表，然后将所需文件以及编译系统所需的任何其他项目文件复制到新文件夹。对于某些 IDE 来说，这非常简单，但是 Arduino 要求将所有 `.cc` 文件重命名为 `.cpp`，并且在复制源文件时修改其中一些包含路径。

外部库（例如用于嵌入式 Arm 处理器的 C++ 工具链）将作为 Makefile 编译过程的一部分自动下载。这是由于每个需要的库都调用了 `add_third_party_download` 规则，并传递了要提取的文件的 URL 地址，以及 MD5 校验来检查下载的文件，以确保文件正确。这些文件应该是 ZIP、GZIP、BZ2 或 TAR 文件，并且将根据文件扩展名调用合适的解压缩程序。如果编译目标需要这些文件中的任何头文件或源文件，则应将它们显

式包含在 Makefile 的文件列表中，以便可以将它们复制到任何生成的项目中，因此每个项目的源代码树都是独立的。使用头文件时很容易忘记这一点，因为设置包含路径足以使 Makefile 正常编译，而无须显式地提到每个被包含的文件，但是生成的项目将无法编译。你还应确保文件列表包含所有的许可证文件，以便外部库的副本能保留正确的属性。

13.2.3 编写测试

TensorFlow 的目标是对其所有代码进行单元测试，我们已经在第 5 章中详细介绍了其中的一些测试。测试代码通常以 *_test.cc* 文件的形式排列在与要测试的模块相同的文件夹中，并且具有与原始源文件相同的前缀。例如，深度卷积运算的实现通过 *tensorflow/lite/micro/kernels/depthwise_conv_test.cc* 进行测试。如果你要添加新的源文件并将更改提交回主线，则必须添加一个附带的单元测试来测试该源文件。这是因为我们需要支持许多不同的平台和模型，并且许多人正在我们的代码之上构建复杂的系统，因此，检查并确保核心组件的正确性是很重要的。

如果你在 *tensorflow/tensorflow/lite/experimental/micro* 的直接子文件夹中添加文件，则应该可以将其命名为 *<something>_test.cc*，该文件将被自动加载。如果你要在一个示例中测试模块，则需要添加对 Makefile 帮助函数 *microlite_test* 的显式调用，就像这样：

```
# Tests the feature provider module using the mock audio provider.  
$(eval $(call microlite_test,feature_provider_mock_test,\  
  $(FEATURE_PROVIDER_MOCK_TEST_SRCS),$(FEATURE_PROVIDER_MOCK_TEST_HDRS)))
```

这些测试本身需要在微控制器上运行，因此它们必须在动态内存分配、避免依赖操作系统和外部库方面坚持与功能代码相同的约束，这也是该框架的目标。不幸的是，这意味着像 Google Test 这种流行的单元测试系统是不可接受的。取而代之的是，我们编写了自己的最小限度的测试框架，该框架在 *micro_test.h* 头文件中定义和实现。

要使用该测试框架，请创建一个包含该头文件的 *.cc* 文件。从 *TF_LITE_MICRO_TESTS_BEGIN* 语句开始，然后定义一系列测试函数，每个函数都有一个 *TF_LITE_MICRO_TEST()* 宏。在每个测试中，你可以调用诸如 *TF_LITE_MICRO_EXPECT_EQ()* 之类的宏来声明要从被测试函数中看到的预期结果。在所有测试功能结束时，你将需要添加 *TF_LITE_MICRO_TESTS_END*。以下是一个基本示例：

```
#include "tensorflow/lite/micro/testing/micro_test.h"  
  
TF_LITE_MICRO_TESTS_BEGIN  
  
TF_LITE_MICRO_TEST(SomeTest) {
```

```
    TF_LITE_LOG_EXPECT_EQ(true, true);  
}  
  
TF_LITE_MICRO_TESTS_END
```

如果针对你的平台进行编译，则会获得一个正常的二进制文件，你应该可以运行该二进制文件。执行该文件会将类似以下内容的日志记录信息输出到 `stderr`（或等效的任何可用信息，并由你的平台上的 `ErrorReporter` 写入）：

```
-----  
Testing SomeTest  
1/1 tests passed  
~~~ALL TESTS PASSED~~~  
-----
```

测试输出被设计得易于阅读，因此你可以手动运行测试，但是字符串 `~~~ ALL TESTS PASSED ~~~` 仅在所有测试都通过的情况下才出现。这使得通过扫描输出日志并寻找这个神奇的值来与自动化测试系统集成成为可能。这就是我们能够在微控制器上运行测试的方式。只要能有一些调试日志被发送回来，主机就可以烧录二进制文件，然后监视输出日志，以确保出现预期的字符串来指示测试是否成功。

13.3 支持一个新的硬件平台

TensorFlow Lite for Microcontrollers 项目的主要目标之一是简化在许多不同设备、操作系统和体系结构上运行机器学习模型的过程。其核心代码被设计成尽可能可移植，编译系统的实现也本着易于支持新平台的原则。在本节中，我们将逐步介绍如何在新平台上运行 TensorFlow Lite for Microcontrollers。

13.3.1 打印到日志

TensorFlow Lite 唯一必需的平台依赖是要将字符串打印到可供外部检查的日志中，我们通常从台式机进行这种检查。这样我们就可以看到测试是否已经成功运行，并且可以调试正在运行的程序。因为这是一个困难的要求，所以你首先需要在你的平台上确定哪些日志记录工具可用，然后编写一个小程序来打印一些内容。

在 Linux 和其他大多数桌面操作系统中，以下是很多 C 语言培训课程开篇的经典“Hello World”示例，其代码通常看起来像这样：

```
#include <stdio.h>  
  
int main(int argc, char** argv) {  
    fprintf(stderr, "Hello World!\n");  
}
```

如果你在 Linux、macOS 或 Windows 上进行编译和链接，然后从命令行运行可执行文件，则应该看到“Hello World!”被打印到终端。如果微控制器上运行的是高级操作系统，则应该也是可以运行的。但是鉴于嵌入式系统本身没有显示器或终端，你至少需要弄清楚文本的显示位置。通常，即使编译时支持 `fprintf()`，也需要通过 USB 或其他调试连接将微控制器连接到台式机以查看日志。

从微控制器的角度来看，关于此代码有一些棘手的部分。其中一个问题 `stdio.h` 库需要链接函数，而其中一些函数非常大，这可能会增加二进制文件的大小，从而使其超出小型设备上可用的资源限制。该库还假设所有常规的 C 标准库工具可用，比如动态内存分配和字符串函数。而且对于 `stderr` 在嵌入式系统上应该输出到哪里，目前还没有一个自然而然的定义，所以这个 API 不是很清晰。

相反，大多数平台都定义了自己的调试日志记录接口。如何调用这些接口通常取决于主机和微控制器之间使用的连接类型，以及嵌入式系统上运行的硬件体系结构和操作系统（如果有的话）。例如，Arm Cortex-M 微控制器支持 semihosting，这是一种在开发过程中主机与目标系统之间进行通信的标准。如果你在主机上使用 OpenOCD 之类的连接，在微控制器上调用系统调用 `SYS_WRITE0`，将会使得寄存器 1 中的零终止字符串被输出到 OpenOCD 终端。在这里，等效于“Hello World”程序的代码如下所示：

```
void DebugLog(const char* s) {
    asm("mov r0, #0x04\n" // SYS_WRITE0
        "mov r1, %[str]\n"
        "bkpt #0xAB\n"
        :
        : [ str ] "r"(s)
        : "r0", "r1");
}

int main(int argc, char** argv) {
    DebugLog("Hello World!\n");
}
```

这里对于汇编的需求表明这是一个平台特定的解决方案，但是这确实避免了需要引入任何外部库（甚至是标准 C 库）的麻烦。

不同平台上的确切操作方法有所不同，但是一种常见的方法是使用与主机的串行 UART 连接。在 Mbed 上的操作方法如下：

```
#include <mbed.h>

// On mbed platforms, we set up a serial port and write to it for debug logging.
void DebugLog(const char* s) {
    static Serial pc(USBTX, USBRX);
    pc.printf("%s", s);
}
```

```
int main(int argc, char** argv) {
    DebugLog("Hello World!\n");
}
```

以下是 Arduino 的一个稍微复杂一些的示例：

```
#include "Arduino.h"

// The Arduino DUE uses a different object for the default serial port shown in
// the monitor than most other models, so make sure we pick the right one. See
// https://github.com/arduino/Arduino/issues/3088#issuecomment-406655244
#if defined(__SAM3X8E__)
#define DEBUG_SERIAL_OBJECT (SerialUSB)
#else
#define DEBUG_SERIAL_OBJECT (Serial)
#endif

// On Arduino platforms, we set up a serial port and write to it for debug
// logging.
void DebugLog(const char* s) {
    static bool is_initialized = false;
    if (!is_initialized) {
        DEBUG_SERIAL_OBJECT.begin(9600);
        // Wait for serial port to connect. Only needed for some models apparently?
        while (!DEBUG_SERIAL_OBJECT) {
        }
        is_initialized = true;
    }
    DEBUG_SERIAL_OBJECT.println(s);
}
int main(int argc, char** argv) {
    DebugLog("Hello World!\n");
}
```

这两个例子都创建了一个串行对象，然后期望用户通过 USB 将微控制器的串行连接安装到主机上。

移植工作关键的第一步是为你的平台创建一个最小的示例，它运行在你想要使用的 IDE 中，以某种方式将字符串打印到主机控制台。如果你能够成功使其工作，那么这部分代码会成为将被添加到 TensorFlow Lite 代码中的专用函数的基础。

13.3.2 实现 DebugLog()

如果查看文件 `tensorflow/lite/micro/debug_log.cc`，你会发现其中有一个 `DebugLog()` 函数的实现，该实现与我们展示的第一个“Hello World”示例非常相似，使用 `stdio.h` 和 `fprintf()` 将字符串输出到控制台。如果你的平台完全支持标准 C 库，并且你不介意额外的二进制文件大小，则可以使用此默认实现而忽略本节的其余部分。不过不幸的是，你更有可能需要使用其他方法。

第一步，我们将使用 DebugLog() 函数的已有测试。首先，运行以下命令：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
    generate_micro_error_reporter_test_make_project
```

当查看目录 *tensorflow/lite/micro/tools/make/gen/linux_x86_64/prj/micro_error_reporter_test/make/* 时（如果你使用的是其他主机平台，请用 *osx* 或 *windows* 替换 *linux*），你应该会看到一些文件夹，例如 *tensorflow* 和 *third_party*。这些文件夹里包含 C++ 源代码，如果将这些源文件拖到 IDE 或编译系统中并编译所有文件，则最终应该得到一个可执行文件，该可执行文件可以测试我们需要创建的错误报告功能。首次编译此代码很可能会失败，因为这部分代码仍使用 *debug_log.cc* 中的默认 DebugLog() 实现，该实现依赖于 *stdio.h* 和 C 标准库。如果想要解决此问题，请更改 *debug_log.cc* 以删除 `#include<cstdio>` 语句，并将 DebugLog() 实现替换为不执行任何操作的实现：

```
#include "tensorflow/lite/micro/debug_log.h"

extern "C" void DebugLog(const char* s) {
    // Do nothing for now.
}
```

完成更改后，请再次尝试编译这些源文件，看看能否成功。完成这些操作之后，获取生成的二进制文件并将其加载到嵌入式系统中。如果可以，即使你还看不到任何输出，也请检查程序是否顺利运行且没有崩溃。

当程序看起来可以正确编译并运行时，请查看是否可以使调试日志记录正常工作。使用上一节中用于“Hello World”程序的代码，并将其放入 *debug_log.cc* 的 DebugLog() 实现中。

实际的测试代码位于 *tensorflow/lite/micro/micro_error_reporter_test.cc*，如下所示：

```
int main(int argc, char** argv) {
    tflite::MicroErrorReporter micro_error_reporter;
    tflite::ErrorReporter* error_reporter = &micro_error_reporter;
    error_reporter->Report("Number: %d", 42);
    error_reporter->Report("Badly-formed format string %");
    error_reporter->Report("Another % badly-formed %% format string");
    error_reporter->Report("~~~%s~~~", "ALL TESTS PASSED");
}
```

这段代码并不直接调用 DebugLog()，而是首先通过 ErrorReporter 接口处理参数的变量数，但确实依赖于你刚刚编写的代码作为其底层实现。如果一切正常，你应该在调试控制台中看到以下内容：

```
Number: 42
Badly-formed format string
```

```
Another badly-formed format string
~~~ALL TESTS PASSED~~~
```

完成这些工作后，你需要将 `DebugLog()` 的实现放回主源代码树中。为此，你将使用我们前面讨论的子文件夹专用化技术。你需要确定一个短名称（不包含大写字母、空格或其他特殊字符），以用于识别你的平台。例如，对于已经支持的某些平台，我们使用 `arduino`、`sparkfun_edge` 和 `linux`。在本例中，我们将使用 `my_mcu`。首先在 `tensorflow/lite/micro/` 中创建一个名为 `my_mcu` 的新子文件夹，该代码库是你从 GitHub 检出的源代码的副本（而不是你刚刚生成或下载的源代码）。将带有实现的 `debug_log.cc` 文件复制到 `my_mcu` 文件夹，然后使用 Git 将其添加到源代码跟踪中。将生成的项目文件复制到备份位置，然后运行以下命令：

```
make -f tensorflow/lite/micro/tools/make/Makefile TARGET=my_mcu clean
make -f tensorflow/lite/micro/tools/make/Makefile \
TARGET=my_mcu generate_micro_error_reporter_test_make_project
```

如果现在查看 `tensorflow/lite/micro/tools/make/gen/my_mcu_x86_64/prj/micro_error_reporter_test/make/tensorflow/lite/micro/`，你应该会看到默认的 `debug_log.cc` 不复存在，而你的实现位于子文件夹 `my_mcu` 中。如果将这组源文件拖回 IDE 或编译系统中，那么现在应该可以看到一个能够成功编译、运行并输出到调试控制台的程序。

13.3.3 运行所有的目标

如果可以运行，那么恭喜：你已启用所有 TensorFlow 测试和可执行目标！实现调试日志记录是你需要进行的唯一特定于平台的更改，代码库中的其他内容都应以足够可移植的方式编写，不需要链接除 `math` 库之外的标准库，就可以使其在任何支持 C++ 11 的工具链上编译和运行。要编译所有目标以便在 IDE 中进行尝试，可以从终端运行以下命令：

```
make -f tensorflow/lite/micro/tools/make/Makefile generate_projects \
TARGET=my_mcu
```

这将在与生成的错误报告程序测试相似的位置创建大量文件夹，每个文件夹都会执行库的不同部分。如果要在平台上运行语音唤醒词示例，可以查看 `tensorflow/lite/micro/tools/make/gen/my_mcu_x86_64/prj/micro_speech/make/`。

现在，你已经实现了 `DebugLog()`，该程序应该可以在你的平台上运行，但是不会做任何有用的事情，因为默认的 `audio_provider.cc` 实现总是返回全零的数组。为了使其正常工作，你需要使用前面介绍的子文件夹专用化方法创建一个专门的 `audio_provider.cc` 模块，该模块返回捕获的音频。如果你不需要有效的演示，则仍然可以使用相同的示例代码查看平台上神经网络推断的延迟，或者其他一些测试。

除了对传感器和 LED 等输出设备的硬件支持外，你可能还希望利用平台的特殊功能来实现运行速度更快的神经网络。我们欢迎这种专用的优化，并且如果这些优化被证明有用，我们希望子文件夹专用化技术是将这些优化重新整合到主源代码树中的一种好方法。

13.3.4 与 Makefile 编译整合

到目前为止，我们只讨论了使用你自己的 IDE，因为使用 IDE 通常比使用 Make 系统更简单，且更为许多嵌入式程序员所熟悉。如果你希望通过我们的持续集成构建来测试你的代码，或者在特定的 IDE 之外提供代码，则需要将更改与我们的 Makefile 更完整地集成。其中一个要点是为你的平台找到一个可公开下载的工具链，以及任何 SDK 或其他依赖项的公开下载地址，这样 shell 脚本就可以自动获取所需编译的所有内容，而不必担心网站登录或注册。例如，我们使用 *tensorflow/lite/micro/tools/make/third_party_downloads.inc* 中的 URL 从 Arm 下载 GCC 嵌入式工具链的 macOS 和 Linux 版本。

然后，你需要确定传递到编译器和链接器的正确命令行标志，以及使用子文件夹专用化找不到的任何其他所需的源文件，并将这些信息编码到 *tensorflow/lite/micro/tools/make/targets* 的 sub-Makefile 中。如果你想要更进一步，可以试着了解如何在 x86 服务器上使用像 Renode 这样的工具来模拟你的微控制器，这样我们就可以在持续集成期间运行测试，而不仅仅是确认能否编译。你可以在 *tensorflow/lite/micro/testing/test_bluepill_binary.sh* 中看到我们使用 Renode 测试“Bluepill”二进制文件的脚本示例。

如果正确配置了所有编译设置，则可以运行以下命令来生成可烧录的二进制文件，请将目标 (TARGET) 设置为适合你的平台：

```
make -f tensorflow/lite/micro/tools/make/Makefile \
TARGET=bluepill micro_error_reporter_test_bin
```

如果运行测试的脚本和环境工作正常，则可以执行此操作来运行平台的所有测试：

```
make -f tensorflow/lite/micro/tools/make/Makefile TARGET=bluepill test
```

13.4 支持一个新的 IDE 或新的编译系统

TensorFlow Lite for Microcontrollers 可以为 Arduino、Mbed 和 Keil 工具链创建独立的项目，但是我们知道嵌入式工程师还使用许多其他开发环境。如果需要在新环境中运行框架，我们建议你做的第一件事就是查看是否可以将生成 Make 项目时生成的“原始”文件集导入 IDE。这种项目档案只包含特定目标所需的源文件，涉及第三方依赖项，因此在许多情况下，你可以将工具链指向根文件夹，要求其包含文件夹中所有的内容。



当你只有很少数量的文件并想将这些文件导入一个生成的项目时，将它们保存在原始源代码树的嵌套子文件夹（如 `tensorflow/lite/micro/examples/micro_speech`）中似乎有些奇怪。把项目目录层次扁平化会更有意义吗？

我们之所以选择保留深度嵌套的文件夹，是为了使将改动合并到主源代码树的过程尽可能简单，即使这样做在处理生成的项目文件时不太方便。如果从 GitHub 检出的原始代码和每个项目中的副本之间的路径总是匹配的，那么跟踪更改和更新就会容易很多。

不幸的是，这种方法不适用于所有的 IDE。例如，Arduino 库要求所有 C++ 源代码文件都具有 `.cpp` 后缀，而不是 TensorFlow 默认的 `.cc` 后缀，并且 Arduino 库无法指定包含路径，因此当我们将原始文件复制到 Arduino 时，需要改变代码中的路径。为了支持这些更复杂的转换，我们在 `Makefile` 中提供了一些规则和脚本，其中根函数 `generate_microlite_projects()` 将调用为每个 IDE 实现的专用版本，然后依赖于更多规则、Python 脚本和模板文件来创建最终输出。如果你需要为自己的 IDE 做类似的事情，则需要使用 `Makefile` 添加类似的功能，由于编译系统的使用非常复杂，因此实现起来并不容易。

13.5 在项目和代码库之间整合代码更改

代码生成系统的最大缺点之一是最终将导致源代码的多个副本分散在不同的位置，这使得处理代码更新非常棘手。为了最小化合并更改的成本，我们采用了一些会有所帮助的约定并推荐了一些流程。最常见的用例是，你对项目的本地副本中的文件进行了一些修改，并且希望更新到最新版本的 TensorFlow Lite 框架以获得额外的功能或错误修复。以下是我们建议处理该过程的方法：

1. 选择你感兴趣的框架版本，然后下载或者手动从 `Makefile` 中编译该版本中适用于你的 IDE 的项目文件的预编译存档。
2. 将这组新文件解压缩到文件夹中，并确保该文件夹和包含你正在修改的项目文件的文件夹的结构相匹配。例如，这两个文件夹的顶层都应该有 `tensorflow` 子文件夹。
3. 在两个文件夹之间运行合并工具。使用哪种工具取决于你的操作系统，不过 Meld 是一个不错的选择，它同时支持 Linux、Windows 和 macOS。合并过程的复杂度将取决于你在本地更改了多少个文件，但是我们预计大多数文件差异是框架方面的更新引起的，因此你通常应该可以直接选择“接受”(accept)。

如果只在本地更改了一两个文件，那么从旧版本复制修改后的代码并手动将其合并到新导出的项目中可能会更容易。

你还可以通过将修改后的代码检入 Git，将最新的项目文件作为新分支导入，然后使用 Git 的内置合并功能来处理集成，从而获取更高级的功能。由于我们自己还没有使用过这种方法，且对 Git 的了解不够专业，因此我们还不能提供有关这种方法的建议。

此过程与使用更传统的代码生成方法执行相同操作的最大区别在于，代码仍然被分成许多逻辑文件，这些文件的路径随着时间的推移保持不变。典型的代码生成会将所有源代码链接到一个文件中，这使得合并或跟踪更改非常困难，因为对顺序或布局的微小更改会使历史比较无法进行。

有时你可能希望进行另一个方向的更改移植，即从项目文件合并到主源代码树。这个主要的源代码树不必是 GitHub 上的官方代码仓库，也可能是你在本地维护且不分发的版本分叉（fork）。我们很乐意在主代码仓库中通过拉取请求（pull request）接受修复和升级改动，但我们知道专用的嵌入式开发并不总能做到这一点，因此我们也很乐意帮助保持分支状态良好。要注意的关键事项是，你需要尝试为开发文件保留单个“信息源”（source of truth）。尤其是如果你有多个开发人员，则很容易在项目存档中对源文件的不同本地副本进行不兼容的更改，这将使更新和调试成为一场噩梦。无论是内部共享还是公共共享，我们强烈建议你使用源代码控制系统，该系统应具有每个文件的单个副本，而不是检入多个版本。

若要将更改迁移回“信息源”仓库，则需要跟踪已修改的文件。如果你没有这些信息，可以随时返回到最初下载或生成的项目文件，然后运行 diff 来查看已更改的内容。一旦知道哪些文件已被修改或新建，只需将它们复制到 Git（或其他源代码控制系统）代码仓库中的相同路径。

这种方法唯一的例外是第三方库中的文件，因为这些文件在 TensorFlow 代码仓库中不存在。提交对第三方库文件的更改超出了本书的范围，此过程将取决于每个单独代码仓库的规则，但作为最后的手段，如果你的更改未被接受，则通常可以将项目分叉到 GitHub 并将平台的编译系统指向这个新的 URL 而不是原始 URL。假设你只更改 TensorFlow 源文件，那么现在应该有一个包含你的更改的本地修改版本的代码仓库。要验证修改已成功集成，你需要使用 Make 运行 `generate_projects()`，然后确保 IDE 和目标的项目已按预期应用了更新。完成后，运行测试以确保没有其他内容被破坏，你就可以将更改提交到你的 TensorFlow 分叉。完成这一步操作后，如果你希望公开所做的更改，则最后一步是提交拉取请求。

13.6 回馈开源

TensorFlow 在 Google 之外的贡献者数量已经超过 Google 内部的贡献者数量，而且微控制器相关的工作对协作的依赖性比大多数其他领域都要大。我们非常渴望获得社区

的帮助，最重要的帮助方法之一是通过拉取请求（尽管还有很多其他方法，例如 Stack Overflow）或创建你自己的示例项目。GitHub 上有很棒的文档，涵盖了拉取请求的基础知识。以下是关于使用 TensorFlow 的一些详细信息：

- 我们有一个由 Google 外界的项目维护人员执行的代码评审（code review）过程。这是通过 GitHub 的代码评审系统进行管理的，因此你应该期望在那里看到有关你的提交的讨论。
- 超出错误修复或优化的更改通常首先需要设计文档。有一个由外部贡献者运营的名为 SIG Micro 的小组来帮助定义我们的优先级和路线图，因此这是一个讨论新设计的好论坛。对于较小的更改，文档可以只有一两页。了解拉取请求背后的背景和动机对代码评审很有帮助。
- 维护一个公共分叉是在将试验性更改提交到主分支之前获取反馈的一个好方法，因为这样你可以放慢速度，进行各种烦琐的更改。
- 我们会针对所有的拉取请求运行公开的自动化测试，使用一些附加的 Google 内部工具，这些内部工具会检查我们依赖于 TensorFlow 的项目与 TensorFlow 项目的集成情况。这些测试的结果有时可能难以解释，更糟糕的是，这些测试有时是不稳定的，会由与你的更改无关的因素导致测试失败。我们知道这是一个不好的体验，因此我们一直在努力尝试改进此过程，但是如果你在理解为何测试失败时遇到麻烦，请在讨论主题帖中联系维护者。
- 我们的目标是 100% 的测试覆盖率，因此如果一个改动没有被现有的测试覆盖，我们会要求你提交一个新的测试。这些测试可以非常简单，我们只是想确保我们所做的一切都能被测试覆盖。
- 为了便于阅读，我们在整个 TensorFlow 代码库中始终使用 Google 的 C 和 C++ 代码格式指南，所以我们要求所有的新代码或者代码修改都采用这个格式。你可以使用 `clang-format` 加上格式参数 `google` 来自动格式化你的代码。

提前感谢你对 TensorFlow 所做的任何贡献，并感谢你对提交更改所涉及的工作的耐心。这并不总是容易的，但是你会给世界上的许多开发者带来改变！

13.7 支持新的硬件加速器

TensorFlow Lite for Microcontrollers 的目标之一是成为一个参考软件平台，以帮助硬件开发人员加快设计进度。我们观察到，让新的芯片在典型机器学习中发挥作用的很多工作都是诸如从训练环境中编写导出程序，特别是在诸如量化和实现典型机器学习模型所需的“长尾”（long tail）算子等复杂细节方面。运行这些任务花费的时间很少，因此它

们不是进行硬件优化的理想选择。

为了解决这些问题，我们希望硬件开发人员采取的第一步是获取能在其平台上运行 TensorFlow Lite for Microcontrollers 的未优化的参考代码，并产生正确的结果。这将证明除了硬件优化之外的一切都在工作，因此这可以成为剩余工作的重点。一个可能的挑战是该芯片是不支持通用 C++ 编译的加速器，因为这款芯片可能仅具有专用功能而非传统 CPU 的所有功能。对于嵌入式用例，我们发现几乎总是需要有一些通用计算可用，即使它很慢（就像一个小的微控制器），因为很多用户的图操作除了作为任意 C++ 实现外是不能被简洁表达的。我们还做出了一项设计决策，即 TensorFlow Lite for Microcontrollers 解释器将不支持子图（subgraph）的异步执行，因为这会使代码复杂化，并且在嵌入式领域中似乎并不常见（不同于移动世界中流行的 Android Neural Network API）。

这意味着 TensorFlow Lite for Microcontrollers 支持的体系架构更像是与传统处理器中基于锁步（lockstep）执行的同步协处理器（synchronous coprocessor），其中加速器（accelerator）加速了计算密集型算子，否则这些算子需要花费很长时间，但也会将具有更灵活要求的较小算子推迟到 CPU 上执行。实践的结果是，我们建议首先在 kernel 级别用对专用硬件的调用来替换算子实现。这意味着，结果和输入都应该在 CPU 可寻址的正常内存中（因为你无法保证后续算子将在哪个处理器上运行），你将需要等待加速器执行完成才能继续操作，或者使用特定于平台的代码切换到 Micro 框架之外的线程。不过，这些限制至少应能使你快速建立原型，并有望提供进行增量更改的能力，同时始终能够测试每个小的修改的正确性。

13.8 理解文件格式

TensorFlow Lite 用来存储其模型的格式有很多优点，但不幸的是，简单不是其中之一。但是，请不要被这种复杂性所困扰，在了解了一些基本知识之后，使用这种格式实际上非常简单。

正如我们在第 3 章中提到的那样，神经网络模型是带有输入和输出的算子计算图。一个算子的一些输入可能是包含学习到的值（称为权重）的大数组，其输入可能来自先前算子的结果，或者是应用程序层输入的输入值数组。这些输入可能是图像像素、音频样本数据或加速度传感器时间序列数据。在单次运行模型结束时，最终算子将在其输出中保留值的数组，通常表示对不同类别的分类预测结果。

模型通常是在台式机上训练的，所以我们需要一种将它们转移到其他设备（如手机或微控制器）上的方法。在 TensorFlow 世界中，我们使用转换程序执行此操作，该转换程序可以从 Python 获取经过训练的模型并将其导出为 TensorFlow Lite 文件。在该导出阶段

可能会遇到很多问题，因为依赖于桌面环境提供的一些功能（例如能够执行 Python 代码段或使用高级操作）创建一个模型很容易，但是更简单的平台并不支持这些功能。还需要将训练中可变的所有值（如权重）转换为常量，删除仅用于梯度反向传播的算子，并执行优化，如融合相邻算子或将昂贵的算子（如批处理标准化）折叠为性能较高的形式。更为棘手的是，主线 TensorFlow 中有 800 多个算子，并且其数量一直在增加。这意味着为一小部分模型编写自己的转换程序相当简单，但是为用户可以在 TensorFlow 中创建的更广泛的网络提供可靠的处理方式却十分困难。仅仅及时了解新的算子就是一项全职工作。

你从转换过程中获得的 TensorFlow Lite 文件不会受到大多数此类问题的影响。我们试图生成一个训练过的模型的更简单和更稳定的表示，该模型具有清晰的输入和输出、冻结在权重中的变量以及已应用的常见图优化，如融合（fusing）。这意味着，即使你不打算使用 TensorFlow Lite for Microcontrollers，我们也建议你使用 TensorFlow Lite 文件格式来使用 TensorFlow 模型进行推断，而不是在 Python 层编写自己的转换程序。

FlatBuffers

我们使用 FlatBuffers 作为序列化（serialization）库。FlatBuffers 是为性能至关重要的应用而设计的，因此非常适合嵌入式系统。它的一个很好的特性是运行时内存表示与序列化形式完全相同，因此模型可以直接嵌入闪存并立即访问，无须进行任何解析或复制。这意味着，生成的用于读取属性的代码类可能有点难以理解，因为存在两层间接寻址，但重要的数据（如权重）被直接存储为像原始 C 数组一样的小端模式（little-endian）^{译注 4} 数据块。而且 FlatBuffers 浪费的空间特别少，所以其使用的空间大小是可以承受的。

FlatBuffers 使用一个定义了我们要序列化的数据结构的 schema 以及一个编译器，将该 schema 转换为本地 C++（或 C、Python、Java 等）代码，用于读取和写入信息。对于 TensorFlow Lite，该 schema 位于 `tensorflow/lite/schema/schema.fbs`，我们将生成的 C++ 访问程序代码缓存在 `tensorflow/lite/schema/schema_genic.h` 中。可以在每次重新编译时生成该 C++ 代码，而不是将其存储在代码管理工具中，但是这将需要在所有我们需要支持的平台上包含 `flatc` 编译器以及其余的工具链，所以我们选择为了方便移植而牺牲自动生成的便利性。

如果你想了解字节级别的格式，建议你查看 FlatBuffers C++ 项目的内部页面或 C 库的文档。不过，我们希望大多数需求都能通过各种高级语言接口得到满足，而你不需要接触如此细粒度的工作。为了向你介绍格式背后的概念，我们将逐步介绍 schema 和

译注 4：“小端模式”指数据的低位保存在内存的低地址中，而数据的高位保存在内存的高地址中。对应的另一种模式为“大端模式”，即数据的低位保存在内存的高地址中，而数据的高位保存在内存的低地址中。

`MicroInterpreter` 中读取模型的代码。希望具体的示例将有助于理解这些内容。

为了介绍 schema，我们需要先滚动到 schema 文件的最末端。在这里，我们看到一行代码将 `root_type` 声明为 `Model`：

```
root_type Model;
```

FlatBuffers 需要一个容器对象，该对象充当文件中保存的其他数据结构树的根。这行语句告诉我们，该格式的根是一个 `Model`。为了弄清楚其含义，我们向上滚动几行到 `Model` 的定义：

```
table Model {
```

这告诉我们 `Model` 是被 FlatBuffers 称为 `table` 的东西。你可以将其视为 Python 中的 `Dict` 或 C 与 C++ 中的 `struct`（尽管比它更灵活）。`table` 定义了对象可以具有的属性以及它们的名称和类型。FlatBuffers 中还有一个不太灵活的类型，称为 `struct`，该类型对对象数组的内存使用效率更高，但我们目前在 TensorFlow Lite 中不使用此类型。

通过查看 `micro_speech` 示例的 `main()` 函数，你可以了解如何在实践中使用此功能：

```
// 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_tiny_conv_micro_features_model_data);
```

`g_tiny_conv_micro_features_model_data` 变量是指向包含序列化 TensorFlow Lite 模型的内存区域的指针，对 `::tflite::GetModel()` 的调用实际上只是强制转换，以获取由该基础内存表示的 C++ 对象。该操作不需要任何内存分配或数据结构的遍历，因此是一个非常快速和高效的调用。要了解如何使用它，请查看我们对数据结构执行的下一个操作：

```
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;
}
```

如果查看 schema 中 `Model` 定义的开头，可以看到此代码所引用的 `version`（版本）属性的定义：

```
// Version of the schema.
version:uint;
```

这告诉我们 `version` 属性是一个 32 位无符号整数，因此为 `model->version()` 生

成的 C++ 代码将返回该类型的值。这里我们只是进行一些错误检查，以确保版本是我们可以理解的，但 FlatBuffers 为 schema 中定义的所有属性都生成了类似的访问器函数。

要了解文件格式中更复杂的部分，可以按照 `MicroInterpreter` 类的实现进行理解，因为该类会加载模型并准备执行。构造函数中传入了一个指向内存中的模型的指针，例如上个例子中的 `g_tiny_conv_micro_features_model_data`。其第一个访问的属性是 `buffers`（缓冲区）：

```
const flatbuffers::Vector<flatbuffers::Offset<Buffer>>* buffers =
    model->buffers();
```

你可能会在类型（type）定义中看到名称 `Vector`，并且担心我们试图在没有动态内存管理的嵌入式环境中使用类似于标准模板库（Standard Template Library, STL）类型的对象，这是一个坏主意。令人高兴的是，FlatBuffers 的 `Vector` 类只是底层内存的只读包装器（read-only wrapper），因此，就像使用根 `Model` 对象一样，创建它不需要解析或内存分配。

要了解有关此 `buffers` 数组代表的更多信息，值得看一下 schema 定义：

```
// Table of raw data buffers (used for constant tensors). Referenced by tensors
// by index. The generous alignment accommodates mmap-friendly data structures.
table Buffer {
    data:[ubyte] (force_align: 16);
}
```

每个缓冲区被定义为无符号 8 比特值的原始数组，第一个值在内存中以 16 字节对齐。这是用于计算图中保存的所有权重数组（和任何其他常数值）的容器类型。张量的类型和形状是分开保存的，这个数组只保存数组中备份数据的原始字节。算子在此顶级向量中按索引引用这些常量缓冲区。

我们访问的下一个属性是子图列表：

```
auto* subgraphs = model->subgraphs();
if (subgraphs->size() != 1) {
    error_reporter->Report("Only 1 subgraph is currently supported.\n");
    initialization_status_ = kTfLiteError;
    return;
}
subgraph_ = (*subgraphs)[0];
```

子图是一组算子、算子之间的连接，以及它们使用的缓冲区、输入和输出。有一些先进的模型在未来可能需要多个子图——例如用于支持控制流（control flow），但是我们现在想要在微控制器上支持的所有网络只有一个子图，因此可以通过确保当前模型满足该要求简化后续代码。为了进一步了解子图中的内容，我们可以回顾一下 schema：

```

// The root type, defining a subgraph, which typically represents an entire
// model.
table SubGraph {
    // A list of all tensors used in this subgraph.
    tensors:[Tensor];

    // Indices of the tensors that are inputs into this subgraph. Note this is
    // the list of non-static tensors that feed into the subgraph for inference.
    inputs:[int];

    // Indices of the tensors that are outputs out of this subgraph. Note this is
    // the list of output tensors that are considered the product of the
    // subgraph's inference.
    outputs:[int];

    // All operators, in execution order.
    operators:[Operator];

    // Name of this subgraph (used for debugging).
    name:string;
}

```

每个子图的第一个属性是张量列表，MicroInterpreter 代码按如下方式对其进行访问：

```
tensors_ = subgraph_->tensors();
```

如前所述，Buffer 对象仅保存权重的原始值，而没有关于其类型或形状的任何元数据。张量是为常量缓冲区存储这些额外信息的地方。对于输入、输出或激活层等临时数组，张量也为它们保存相同的信息。你可以在 schema 文件顶部附近的定义中看到此元数据：

```

table Tensor {
    // The tensor shape. The meaning of each entry is operator-specific but
    // builtin ops use: [batch size, height, width, number of channels] (That's
    // Tensorflow's NHWC).
    shape:[int];
    type:TensorType;
    // An index that refers to the buffers table at the root of the model. Or,
    // if there is no data buffer associated (i.e. intermediate results), then
    // this is 0 (which refers to an always existent empty buffer).
    //
    // The data_buffer itself is an opaque container, with the assumption that the
    // target device is little-endian. In addition, all builtin operators assume
    // the memory is ordered such that if `shape` is [4, 3, 2], then index
    // [i, j, k] maps to data_buffer[i*3*2 + j*2 + k].
    buffer:uint;
    name:string; // For debugging and importing back into tensorflow.
    quantization:QuantizationParameters; // Optional.
    is_variable:bool = false;
}

```

`shape`（形状）是一个简单的整数列表，表示张量的维度。而 `type`（类型）是一个枚举，映射到 TensorFlow Lite 支持的可能的数据类型。`buffer`（缓冲区）属性指示了根

级别的列表中的哪个缓冲区具有代表该张量的实际值。缓冲区中的值可能是从文件中读取的，也可能是 0（如果这些值是动态计算的（例如，对于激活层））。`name`（名称）只是为了给张量提供一个人类可读的标签，这有助于调试。`quantization`（量化）属性定义了如何将低精度值映射为实数。`is_variable` 成员的存在是为了支持未来的训练以及其他高级应用，但是微控制器单元（MCU）不需要使用此属性^{译注5}。

回到 `MicroInterpreter` 的代码，我们从子图中拉取的第二个主要属性是一组算子：

```
operators_ = subgraph_->operators();
```

此列表包含模型的图结构。要了解其编码方式，我们可以回到 `Operator` 的 schema 定义：

```
// An operator takes tensors as inputs and outputs. The type of operation being
// performed is determined by an index into the list of valid OperatorCodes,
// while the specifics of each operations is configured using builtin_options
// or custom_options.
table Operator {
    // Index into the operator_codes array. Using an integer here avoids
    // complicate map lookups.
    opcode_index:uint;

    // Optional input and output tensors are indicated by -1.
    inputs:[int];
    outputs:[int];

    builtin_options:BuiltinOptions;
    custom_options:[ubyte];
    custom_options_format:CustomOptionsFormat;

    // A list of booleans indicating the input tensors which are being mutated by
    // this operator.(e.g. used by RNN and LSTM).
    // For example, if the "inputs" array refers to 5 tensors and the second and
    // fifth are mutable variables, then this list will contain
    // [false, true, false, false, true].
    //
    // If the list is empty, no variable is mutated in this operator.
    // The list either has the same length as `inputs`, or is empty.
    mutating_variable_inputs:[bool];
}
```

`opcode_index` 成员是 `Model` 内部的根级 `operator_codes` 向量的索引。由于一种特殊的算子（如 Conv2D）可能在一个图中多次出现，并且某些算子需要一个字符串来定义它们，所以为了节省序列化的大小，我们将所有算子定义保留在一个顶级数组中，并从子图中间接引用它们。

`inputs`（输入）和 `outputs`（输出）数组定义了计算图中算子及其邻居之间的连接。

译注 5：最新版本 TensorFlow Lite for Microcontrollers 已经添加对 `is_variable` 的支持。

这些是引用父级子图中的张量数组的整数列表，也可以是引用从模型中读取的常量缓冲区、由应用程序馈送到网络的输入、运行其他算子的结果或在计算完成后由应用程序读取的输出目标缓冲区的整数列表。

关于子图中包含的算子列表，需要知道的一件重要事情是它们始终是按拓扑序 (topological order) 排列的，因此，如果你按照从头到尾的顺序执行数组中的算子，那么当某个算子被执行时，其输入所依赖的先前的算子将已被计算。这使得编写解释器变得更加简单，因为执行循环不需要任何图操作，而只需按照列出的顺序执行算子即可。这意味着以不同的顺序运行相同的子图（例如，在训练中使用反向传播）并不简单，但是 TensorFlow Lite 的重点是推断，因此这是一个值得的妥协。

算子通常还需要参数，例如 Conv2D 卷积核所需的过滤器的形状 (shape) 和步幅 (stride)。不幸的是，这些参数的表示非常复杂。由于历史原因，TensorFlow Lite 支持两种不同的算子系列。首先是内置算子 (builtin operation)，它是移动应用中最常见的算子。你可以在 schema 中看到一个列表。截至 2019 年 11 月，TensorFlow Lite 只有 122 种内置算子，但 TensorFlow 支持 800 多种算子，那么其余部分该怎么办？自定义算子 (custom operation) 是由字符串名称定义的，而不是像内置算子的固定枚举那样，因此可以更容易地添加它，无须接触 schema。

对于内置算子，参数结构在 schema 中列出。以下是 Conv2D 的示例：

```
table Conv2DOptions {
    padding:Padding;
    stride_w:int;
    stride_h:int;
    fused_activation_function:ActivationFunctionType;
    dilation_w_factor:int = 1;
    dilation_h_factor:int = 1;
}
```

希望上面列出的大多数成员你都有些熟悉。它们的访问方式与其他 FlatBuffers 对象相同：通过每个 `Operator` 对象的 `buildin_options` 共用体，并根据算子代码选择适当的类型（尽管这样做的代码基于大量 `switch` 语句）。

如果算子代码显示其为自定义算子，则我们无法提前知道参数列表的结构，因此无法生成代码对象。相反，我们将参数信息打包到 FlexBuffer 中。这是 FlatBuffers 库提供的一种格式，用于在你事先不知道结构时对任意数据进行编码，这意味着实现算子的代码需要访问指定了类型的结果数据，并且使用比内置算子的语法更混乱的语法。以下是一个物体检测代码的示例：

```
const flexbuffers::Map& m = flexbuffers::GetRoot(buffer_t, length).AsMap();
op_data->max_detections = m["max_detections"].AsInt32();
```

本例中引用的缓冲区指针最终来自 `Operator` 表的 `custom_options` 成员，展示了如何从该属性访问参数数据。

`Operator` 的最后一个成员是 `mutating_variable_inputs`。这是一个试验特性，有助于管理长短期记忆网络（Long Short-Term Memory, LSTM）和其他可能希望将其输入视为变量的算子，该特性与大多数 MCU 应用程序不相关。

这些是 TensorFlow Lite 序列化格式的关键部分。我们没有介绍其他一些成员（例如 `Model` 中的 `metadata_buffer`），但是这些成员是用于可选的非必要功能的，因此通常可以忽略。希望该概述足以使你能够开始阅读、编写和调试自己的模型文件。

13.9 将 TensorFlow Lite 移动平台算子移植到 Micro

在针对移动设备的主线 TensorFlow Lite 版本中，有 100 多个“内置”算子。TensorFlow Lite for Microcontrollers 重用了其中的大部分代码，但是由于这些算子的默认实现会引入多线程、动态内存分配或嵌入式系统上不可用的其他功能等依赖关系，因此算子实现（也称为 `kernel`）需要一些工作以使得其在 Micro 上可用。

最终，我们希望统一算子实现的两个分支，但是这需要在整个框架中进行一些设计和 API 更改，因此短期内不可能完成。大多数算子应该有了 Micro 的实现，但是如果你发现了一个在移动端 TensorFlow Lite 上可用但在嵌入式版本中不可用的实现，那么本节将引导你完成转换过程。在你确定了要移植的算子后，后续过程将分为几个阶段。

13.9.1 分离引用代码

列出的所有算子都应该有了引用代码，但是函数很可能在 `reference_ops.h` 中。这是一个大约 5000 行长的单块头文件。因为该头文件涵盖了太多算子，所以它引入了很多在嵌入式平台上不可用的依赖项。要开始移植过程，首先需要将正在进行移植的算子所需引用的函数提取到单独的头文件中。你可以在 `conv.h` 以及 `pooling.h` 等较小的头文件中看到一些示例。引用函数本身应该具有与其实现的算子匹配的名称，并且对于不同的数据类型通常会有多个实现，有时也会使用模板。

一旦文件从较大的头文件中被分离出来，就需要从 `reference_ops.h` 中包含该文件，以便该头文件的所有现有用户仍然可以看到被你移出去的函数（尽管我们的 Micro 代码将只单独包含分离出来的头文件）。你还需要将头文件添加到 `kernels/internal/BUILD:reference_base` 和 `kernels/internal/BUILD:legacy_reference_base` 编译规则中。完成这些更改后，你应该能够运行测试套件并看到所有现有的移动测试通过：

```
bazel test tensorflow/lite/kernels:all
```

此时是一个很好的时机，你可以创建一个初始拉取请求来让别人审核。虽然你还没有将任何内容移植到 `micro` 分支，但是你已经为接下来的更改准备了现有代码，因此在执行以下步骤前，值得尝试先让别人审阅并提交此工作。

13.9.2 为算子创建一个 Micro 拷贝

每个 Micro 算子实现都是 `tensorflow/lite/kernels/` 中保存的移动版本的修改副本。例如，Micro 版的 `conv.cc` 基于移动版的 `conv.cc`。它们之间的不同之处很多。首先，在嵌入式环境中，动态内存分配更为棘手，因此，为推断期间使用的用来缓存计算值的 `OpData` 结构的创建被移动到单独的函数中，以便在 `Invoke()` 期间调用它，而不是从 `Prepare()` 返回。虽然每次 `Invoke()` 调用都需要做更多的工作，但是减少内存开销对于微控制器通常是有意义的。其次，`Prepare()` 中的大多数参数检查代码通常被删除。这部分代码最好用 `#if defined(DEBUG)` 来包含，而不是将其完全删除，但是删除会使代码大小保持最小。所有对外部框架的引用（`Eigen`、`gemmlowp`、`cpu_backend_support`）都应从 `include` 语句和代码中删除。在 `Eval()` 函数中，除了调用 `reference_ops::` 命名空间中函数之外的路径都需被删除。

修改后的算子实现结果应保存在 `tensorflow/lite/micro/kernels/` 文件夹中，文件的名称应与移动版本相同（通常是算子名称的小写版本）。

13.9.3 将测试移植到 Micro 框架

我们无法在嵌入式平台上运行完整的 Google Test 框架，因此需要使用 Micro Test 库。GTest 的用户应该对其很熟悉，但是它避免了任何需要动态内存分配或 C++ 全局初始化的结构。本书其他地方还有更多的文档。

你需要在嵌入式环境中运行与移动平台上相同的测试，因此你需要以 `tensorflow/lite/kernels/<你的算子名称>_test.cc` 中的版本为起点。例如，请查看 `tensorflow/lite/kernels/conv_test.cc` 和移植后的版本 `tensorflow/lite/micro/kernels/conv_test.cc`。它们之间最大的不同是：

- 移动平台的代码依赖于 C++ STL 类，比如 `std::map` 和 `std::vector`，这些实现需要使用动态内存分配。
- 移动平台代码还使用帮助程序类，并以涉及内存分配的方式传入数据对象。
- Micro 版本的实现在栈（stack）中分配所有的数据，使用看起来很像 `std::vector` 的 `std::initializer_list` 来向下传递数据，该数据类型不需要使用动态内存分配。
- 运行测试的调用表示为函数调用，而不是对象分配，因为这有助于重用许多代码而不会遇到内存分配问题。

- 可以使用大多数标准错误检查宏，但需要前缀 `TF_LITE_MICRO_`。例如，`EXPECT_EQ` 变成了 `TF_LITE_MICRO_EXPECT_EQ`。

所有测试都必须保存在一个文件中，并由一对 `TF_LITE_MICRO_TESTS_BEGIN`/`TF_LITE_MICRO_TESTS_END` 作为起止行。实际上，这会创建一个 `main()` 函数，以便测试可以作为独立的二进制文件运行。

我们还尝试确保测试仅依赖于 `kernel` 代码和 API，而不引入诸如解释器之类的其他类。测试应该使用 `GetRegistration()` 返回的 C API 直接调用 `kernel` 实现。这是因为我们要确保可以完全独立地使用 `kernel`，而不需要框架的其余部分，因此测试代码也应避免这些依赖。

13.9.4 创建一个 Bazel 测试

现在你已经创建了算子实现和测试文件，你需要检查它们能否正常工作。你需要使用 Bazel 开源编译系统来执行此算子。将 `tfLite_micro_cc_test` 规则添加到 `BUILD` 文件中，然后尝试编译并运行此命令行（将 `conv` 替换为你的算子名称）：

```
bazel test tensorflow/lite/micro/kernels:conv_test --test_output=streamed
```

毫无疑问，会发生编译错误和测试失败，因此希望你花些时间迭代修复这些错误。

13.9.5 将你的算子添加到 AllOpsResolver

由于二进制大小的因素，应用程序可以选择仅引入某些算子实现，但是有一个算子解析器可以包含所有可用的算子，从而使入门变得容易。你应该在 `all_ops_resolver.cc` 的构造函数中添加一个调用来注册你的算子实现，并确保实现和头文件也包含在 `BUILD` 规则中。

13.9.6 创建一个 Makefile 测试

到目前为止，你所做的一切都在 TensorFlow Lite 的 `micro` 分支中，但是你一直在 x86 上编译和测试。这是最简单的开发方式，其首要任务是为所有算子创建可移植的、未优化的实现，因此我们建议你在此领域中尽可能多地投入。现在你应该有一个能在台式机的 Linux 上运行、完全正常工作且经过测试的算子实现，所以是时候开始在嵌入式设备上编译和测试了。

Google 开源项目的标准编译系统是 Bazel，但不幸的是，使用它进行交叉编译（cross-compilation）和支持嵌入式工具链并不容易，因此我们不得不求助于古老的 Make 进行部署。Makefile 的内部结构是非常复杂的，但是我们希望，基于你的算子实现文件和测试

的名称及位置，Makefile 会将其自动拾取。唯一的手动步骤应该是将你创建的引用头文件添加到 `MICROLITE_CC_HDRS` 文件列表中。

要在这种环境下测试你的算子，请 `cd` 到该文件夹，然后运行以下命令（使用你自己的算子名称而不是 `conv`）：

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

希望能够成功编译并通过测试。如果没有成功，请执行常规的调试过程以找出问题所在。

这仍然在本地的 Intel x86 台式机上运行，尽管它使用与嵌入式目标相同的编译机制。你现在可以尝试将代码编译并烧录到诸如 SparkFun Edge 的真正的微控制器上（只需在 Makefile 行中传入 `TARGET=sparkfun_edge` 即可），但是为了使这个过程更轻松，我们还提供了 Cortex-M3 设备的软件仿真。你应该可以通过执行以下命令来运行测试：

```
make -f tensorflow/lite/micro/tools/make/Makefile TARGET=bluepill test_conv_test
```

这可能有点不稳定，因为有时仿真器执行所需的时间太长，导致进程超时，但是希望再次尝试可以解决该问题。如果你做到了这一点，我们鼓励你尽可能将更改贡献回开源版本。开放源代码的整个过程可能有点复杂，但是 TensorFlow 社区指南是一个很好的起点。

13.10 小结

读完这一章后，你可能会觉得自己被大量的信息淹没。我们为你提供了许多有关 TensorFlow Lite for Microcontrollers 如何工作的信息。如果你没有完全理解，甚至大部分都不理解，请不要担心，我们只是想给你足够的背景知识，这样你如果真的需要深入研究，就知道从哪里开始寻找相关资源了。这些代码都是开源的，是指导框架如何运行的最终指南，但是我们希望本章能够帮助你浏览框架的架构，并理解一些设计决策背后的考量。

在了解了如何运行一些预编译的示例并深入研究了库的工作原理之后，你可能想知道如何将学到的知识应用于自己的应用程序。本书的剩余部分将集中介绍在自己的产品中部署自定义机器学习所需的技能，包括优化、调试和移植模型，以及隐私和安全性。

设计你自己的 TinyML 应用程序

到目前为止，我们已经针对音频、图像和手势识别等重要领域探索了现有的参考应用程序。如果你想要解决的问题与其中一个示例相似，那么你应该能够自己训练模型并将其部署到设备上。但是如果你的问题并不能清晰地对应某一个示例，要如何修改我们的示例或者创建新的应用？在本章以及后续的章节中，我们将介绍如何针对一个没有简单起点的问题创建嵌入式机器学习的解决方案。学习这些示例的经验应该已经为你创建自己的系统奠定了良好的基础，但是你还需要了解有关设计、训练和部署新模型的更多知识。由于我们平台的限制非常严格，所以我们还会花费一些时间来讨论如何进行正确的优化，以满足你的存储和计算预算，同时保证一定的准确率。毫无疑问，你将花费大量时间来尝试解释程序为什么不能正常工作，因此我们还将介绍一些调试技术。最后，我们将探讨如何为用户的隐私和安全建立保护措施。

14.1 设计过程

训练模型可能会花费几天甚至几周的时间，而搭建一个新的嵌入式硬件平台也可能非常耗时——因此，任何嵌入式机器学习项目的最大风险之一就是在开始正常工作之前就用光了时间。减轻这种风险的最有效方法是计划、调研和试验，在这个过程中尽可能早地解决许多悬而未决的问题。每次更改训练数据或者调整模型架构也经常会涉及将近一周的编码和再训练，而更改部署硬件会使整个软件堆栈产生连锁反应，还会涉及大量重写之前的代码。你在项目之初能做的就是尽可能减少后续开发过程中需要进行更改的次数，这会为你节省大量的时间和成本。本章我们将介绍并推荐一些技术，用以帮助你在开始编写最终应用程序之前考虑一些重要问题。

14.2 你需要微控制器还是更大的设备

你真正需要回答的第一个问题是：至少对于初始原型而言，你是否真的需要嵌入式系统

的优势，还是可以适当地放宽对电池寿命、成本和尺寸的要求。与在嵌入式世界中进行开发相比，在具有完整现代操作系统（如 Linux）的平台上编程要容易得多，也快得多。你可以以不到 25 美元的价格获得完整的桌面系统（如树莓派（Raspberry Pi））以及许多外围设备（如摄像头和各种传感器）。如果你需要运行计算繁重的神经网络，NVIDIA 的 Jetson 系列主板起价为 99 美元，并以较小的尺寸提供强大的软件堆栈。这些设备的最大缺点是，它们会消耗几瓦的能源，如果使用电池供电，那么其寿命最多只有几小时或者几天，具体取决于储能设备的物理大小。只要延迟没有硬性限制，你甚至可以利用许多强大的云服务器来运行神经网络的运算，让客户端设备来处理接口和网络通信。

我们坚信能够在任何地方部署嵌入式机器学习系统，但是如果你只是想要确认某个想法是否完全可行，我们强烈建议你先尝试使用一个更简单、更易于试验的设备进行原型（prototype）开发。在嵌入式系统上开发是一件非常麻烦的事情，因此，在深入开发之前，越早弄清楚应用程序的实际需求，成功的概率也就越大。

举一个实际的例子，假设你想构建一个设备来监视绵羊的健康状况，最终产品将需要在没有良好网络连接的环境中运行几个星期甚至几个月，因此它必须是嵌入式系统。但是，开始的时候你也许并不想使用这样棘手的编程设备，因为你还不知道很多重要的细节，例如要运行什么样的模型、需要什么传感器以及需要什么操作来收集数据。你也还没有任何可以用于训练的数据。为了引导你的工作，你可能想要找一个友好的农夫，带一群绵羊到附近的地方放牧。你可以建立一个树莓派平台，每天晚上自己从被监视的绵羊身上取下它并充电，建立一个覆盖放牧范围的室外 WiFi 网络，以便设备可以轻松地与网络进行通信。显然，你不能指望真正的客户以后会需要这么多麻烦的操作，但是这能够帮助你回答许多相关的问题，还能帮助你了解你要搭建的内容，而且这会使你可以更快地试验新的模型、新的传感器和其他各种因素。

微控制器之所以有用，是因为它们可以通过其他硬件无法实现的方式进行扩展。它们价格便宜、体积小、几乎不消耗任何能源，但是这些优势只有在你真正需要扩大规模时才会发挥作用。如果可以的话，请尽可能推迟扩展，直到绝对必要时才使用微控制器，这样你才能确信自己正在扩展正确的事情。

14.3 了解可行性

知道深度学习能够解决什么问题并不是一件容易的事情。我们发现非常有用的一条经验法则是，神经网络模型非常适合处理人们“眨眼之间”就能解决的任务。我们似乎可以凭直觉瞬间识别出物体、声音、单词和朋友，而这些都是神经网络可以执行的任务。类似地，DeepMind 的 Go-solving 算法依赖于卷积神经网络，它能够查看棋盘并返回每个玩家所处位置的强弱估计。然后，使用这些基础知识为系统建立长期计划。

这是一个很有用的区别，因为它在不同类型的“智能”(intelligence)之间画了一条界线。神经网络无法自动规划或者完成更高级的任务，比如定理求解(theorem solving)。它更擅长吸收大量嘈杂且令人困惑的数据，并从中识别出模式。例如，对于如何指导牧羊犬牧羊，神经网络不太可能会是一个好的方案；但是对于通过从温度、脉搏、加速器等传感器中收集绵羊的数据，神经网络很可能可以从数据中学习并预测绵羊的健康状况。对于我们无意识地执行、不需要明确思考的问题，深度学习通常可以很好地解决。当然，这并不意味着神经网络不能对那些更抽象的问题提供帮助，只是对于更复杂的问题，神经网络通常只是一个更大的系统中的一个部分，使用“本能的”(instinctual)预测作为输入。

14.4 站在巨人的肩膀上

在学术领域，“回顾文献”广为人知，即阅读与你感兴趣的问题相关的科研论文和其他出版物。即使你不是研究人员，在处理深度学习问题时，阅读文献也会是一个很有用的过程。这是因为，很多将神经网络模型应用于各种挑战的尝试会为你提供非常有用的帮助。并且如果你能从他人的工作中获得一些相关的启迪，就可以节省大量的时间。理解科研论文可能很有挑战性，但收集信息最大的用处在于发现是否有人针对类似的问题训练过模型，以及你可以使用哪些现有数据集，因为收集数据通常是机器学习过程中最困难的部分之一。

例如，如果你对机械轴承的预测性维修(predictive maintenance)问题感兴趣，你可以在arxiv.org上搜索“deep learning predictive maintenance bearings”。arxiv.org是最受欢迎的机器学习研究论文网站。撰写本书时排名最高的搜索结果是Shen Zhang等人在2019年发表的调查报告“Machine Learning and Deep Learning Algorithms for Bearing Fault Diagnostics: A Comprehensive Review”。你将从中了解到有一个标准公共数据集Case Western Reserve University bearing dataset包含被标记的轴承传感器数据。使用现有的数据集非常有用，因为它能帮助你在收集自己的数据之前试验各种方法。这篇文章还很好地概述了用于此问题的各种模型架构，并讨论了它们的收益、成本以及能达到的总体效果。

14.5 找一些相似的模型训练

在对模型架构和使用的训练数据有了些想法之后，有必要花一些时间在训练环境中进行试验，看看在没有资源限制的情况下可以达到什么样的结果。本书着重于TensorFlow，因此我们建议你找到一个示例TensorFlow教程或者脚本（取决于你的经验水平），先按原样运行，然后进行修改以使它适应你的问题。如果可以，请查看本书中的训练示例以获得启发，因为它们还包括部署到嵌入式平台所需的所有步骤。

考虑哪种模型可能有效的一个好方法是观察传感器数据的特征，并尝试将数据与示例中类似的内容进行匹配。例如，如果你有来自机械轴承的单通道振动数据，那么它很可能是一个相对高频的时间序列，这与来自麦克风的音频数据有很多共通之处。首先，你可以尝试将所有车轮轴承数据转换为 `.wav` 格式，然后将它们输入语音训练过程，而不是使用标准的“语音命令”数据集作为输入。然后，你可能想进一步定制训练流程，但一开始你至少要有一个可以使用的模型，并能够将它用作进一步试验的基准。类似的过程也适用于将手势识别的示例调整为任何基于加速度传感器数据分类的问题，或将行人检测模型应用于其他的机器视觉问题。如果在本书中没有找到能明显对应的示例，那么搜索如何使用 Keras 对你感兴趣问题创建模型可能是一个不错的方法。

特征生成

许多纯机器学习教程都无法很好地涵盖的主题之一就是特征生成。特征是我们输入神经网络的值，即作为输入传入的数字数组。通常，对于现代机器视觉领域，这些是直接来自内存中图像数据的 RGB 像素矩阵，但是对于许多其他类型的传感器而言并非如此。例如，语音识别示例接收 16KHz 脉冲编码的调制数据（以每秒 16 000 次的频率捕获的样本），但是这些数据并没有直接被输入神经网络，而是被转换为频谱图——单通道的 2D 矩阵，随时间在一定范围内变化的频率幅度。人们普遍希望摆脱这种预处理，因为它需要大量的试验和工程实现，但是对于许多问题，尤其是在资源有限的情况下，仍然有必要多次尝试以获取最优的特征。不幸的是，针对特定问题的最优特征生成方法通常没有很好的文档记录，因此，如果找不到合适的范例，你可能需要向领域专家寻求建议。

14.6 查看数据

大部分机器学习方向的研究都着重于设计新的模型架构，鲜有研究关注训练数据集。这是因为，在学术界通常会有一个预先生成的训练数据集，以便能将你的模型与其他模型进行比较评分。但是在学术研究之外，通常没有用于解决问题的现有数据集，我们关心的是交付给用户的最终体验，而不是固定数据集上的评分，因此优先级会非常不同。

本书作者之一写了一篇博客文章，对此进行了更详细的介绍，但总体结论是：与模型架构相比，你应该期望花费更多的时间收集、探索、标记和改进数据，这样投资的时间回报会更高。

我们发现有一些常用的技术在数据处理阶段非常有用。一个听起来非常明显但仍然经常被忘记的技术是：查看你的数据！如果有图像数据，请将它们下载到本地计算机中，并按标签排列，然后浏览这些图像。如果你使用的是音频文件，请执行相同操作并听取其

中一些音频的内容。你很快就会发现各种意想不到的奇怪数据和错误，从标有“美洲捷豹”（jaguar cat）标签的捷豹汽车（jaguar car），到声音太微弱甚至被裁剪掉一部分单词的音频。就算你只有数字数据，通过逗号分隔值（Comma-Separated Values, CSV）在文本文件中查找数字也可能会很有帮助。过去，我们发现了一些问题，例如许多值达到了传感器的极限最大值或最小值，或者灵敏度太低以至于大多数数据都在一个很小的数值范围内。你可以通过数据分析获得更高级的信息，并且你会发现 TensorBoard 之类的工具对于数据集的聚类和其他可视化非常有用。

另一个需要注意的问题是训练集的不平衡。如果你的目标是分类，那么训练输入中不同类别出现的频率将会影响最终的预测概率。一个很容易掉入的陷阱是，认为你的网络结果代表真实的概率，例如，“yes”的得分为 0.5，就简单地认为网络预测口头单词“yes”的可能性为 50%。但实际上这种关系要复杂得多，考虑到训练数据中每种类别的比例会影响输出类别的概率，需要使用应用程序实际输入分布中每个类别的先验概率（prior probability）才能得到真实的概率。再举一个例子，想象一下在 10 种不同物种的数据集上训练鸟类图像分类器。如果你将训练后的模型部署到南极洲，那么一定会对鹦鹉的分类结果表示深深的怀疑；如果你正在观看来自亚马逊的视频，那么识别出企鹅也同样令人惊讶。在训练过程中吸收这种领域知识可能是非常困难的，因为通常你希望每个类别的样本数量大致相等，所以网络会对每种类别都“一视同仁”。取而代之的是，通常会在执行模型推断后进行校验，根据先验知识对结果进行加权。比如在南极的例子中，你可能会用很高的阈值来预测鹦鹉，但是用相对低得多的阈值来预测企鹅。

14.7 绿野仙踪

我们最喜欢的机器学习设计技术之一根本不涉及任何技术。工程学中最困难的问题是确定需求，因为很容易将大量时间和资源浪费在实际不可解的问题上，尤其是因为开发机器学习模型的过程会花费很多时间。为了避免这些无解需求，我们强烈建议你使用绿野仙踪（Wizard of Oz）方法^{译注1}。在这种情况下，你将创建最终要构建的系统的模型，但是并不是让软件来做决策，而是让人成为“幕后的人”。这样，你就可以在经历一个耗时的开发周期之前验证你的假设，以确保在将规范纳入设计之前已经对它进行很好的测试。

在实践中这是如何工作的？想象一下你正在设计一种传感器，它可以检测会议室中是否有人，如果会议室中没有人，就会将灯光调暗。使用绿野仙踪方法，你不需要一开始就构建和部署任何基于行人检测模型的无线微控制器，而是先搭建一个原型，将实况视频传送到坐在附近房间的人，并通过一个开关控制灯光，指示灯光应该在何时调暗。你

译注 1：绿野仙踪原型也被称为“沃兹原型”，通常在敏捷软件开发中被用于改善软件的业务规则实现方式。

会很快发现可用性问题，例如，如果摄像头没有覆盖整个房间，那么当有人在场时，灯会一直关闭，或者当有人进入房间时，打开灯的延迟不可接受等。你可以将这种方法应用于几乎所有的问题，这可以为产品的假设提供宝贵的验证，而无须将时间和精力花费在基于错误假设的机器学习模型上。更好的是，你可以设计这个过程，以便从中获取和生成带有标签的训练数据，因为在这个过程中你可以获得输入，以及根据输入做出的决策。

14.8 先可以在桌面系统中运行

绿野仙踪方法是使原型尽快运行的一种方法，但是即使你已经开始训练模型，也应该考虑如何尽快进行试验和迭代。导出模型并使该模型在嵌入式平台上足够快地运行可能需要很长时间，因此，一条捷径就是先将具体环境中的数据传输到附近的台式机或者云服务器进行处理。这可能会花费很多精力，而且最终不会被部署为真正的解决方案，但是只要你可以确保延迟不影响整体体验，那么它会是一种特别有用的方法，能帮助你获得在特定的上下文中运行机器学习解决方案的反馈。

另一个好处是，你可以记录传感器的每一次数据流，然后将它们反复用于模型的非正式评估。如果一个模型在过去犯过严重的错误，并且这种错误在正常情况下很难出现，那么这些记录就会非常有用。比如你的照片分类器将婴儿标记为狗，那么就算模型总体准确率达到 95%，你也要极力避免这种情况，因为这会使用户感到非常不舒服。

在台式机上运行模型有很多选项。最简单的开始方法是使用类似于树莓派的平台来收集示例数据（这种平台通常有很好的传感器支持），然后将数据批量复制到台式机（如果需要，也可以复制到云实例上）。然后，你可以在 Python 中使用标准的 TensorFlow，以离线方式训练和评估潜在的模型，不需要任何用户交互。当你拥有一个看起来前景不错的模型时，就可以采取一些渐进的步骤，例如将 TensorFlow 模型转换为 TensorFlow Lite，但可以继续根据 PC 上的批处理数据对模型进行评估。完成此操作后，你可以尝试将桌面 TensorFlow Lite 应用程序放在简单的 Web API 后面，然后在某个设备中调用它，以了解它在实际环境中工作得怎么样。

优化延迟

嵌入式系统没有太强的计算能力，这意味着神经网络所需的密集计算在嵌入式平台上花费的时间会比在大多数其他平台上花费的时间更长。由于嵌入式系统通常需要实时处理传感器数据流，因此运行得太慢可能会导致很多问题。假设你正在尝试观察一些可能只会在瞬间发生的事情（例如，在摄像头视野中出现的鸟）。如果处理时间过长，那么很可能会因为传感器采样速度太慢而错过捕捉事件。有时，使用重叠窗口重复观察传感器数据可以提高预测质量，例如唤醒词检测示例在音频数据上运行一秒的窗口以发现唤醒词，但是会将窗口向前移动 100 毫秒或者小于 100 毫秒，并取平均结果。在这些情况下，减少延迟时间可以提高整体的准确率。加快模型执行速度还可能使设备以较低的 CPU 频率运行，或者在推断之间进入睡眠状态，从而减少总体功耗。

由于延迟是优化的一个重要领域，因此本章将重点介绍一些可用于减少模型运行时间的技术。

15.1 首先确保你要优化的部分很重要

你的神经网络代码可能只占整个系统延迟中的一小部分，以至于加速神经网络的运行并不会对产品的性能产生很明显的影响。确定是否是这种情况的最简单方法是在应用程序代码中注释掉对 `tflite::MicroInterpreter::Invoke()` 的调用。这是包含所有推断计算的函数，它将阻塞直到网络运行出结果为止，因此，通过注释它，你可以观察到它对总体延迟的影响。在理想的情况下，你可以使用计时器日志语句或者剖析器（profiler）计算这个变化，但是正如前面提到的，仅仅观测 LED 闪烁的频率差异也足以使你对速度的提升有一个大致的概念。如果运行网络推断与不运行网络推断之间的差异很小，那么优化深度学习部分的代码就不会有太大的收益，在这种情况下，你应该首先关注应用程序的其他部分。

15.2 更换硬件

如果试验表明你确实需要加快神经网络代码的运行速度，那么首先要问的问题是你能否使用功能更强大的硬件设备。对于许多嵌入式产品而言，这是不可能的，因为通常在很早的时候或者由于外部因素就已经确定要使用哪个硬件平台，但是从软件角度来看，这是最容易改变的因素，因此它值得认真考虑。如果你可以选择的话，最大的限制条件通常是功耗、速度和成本。如果可以，请通过更换你使用的芯片来权衡功耗或者成本，以提升运行速度。你甚至可能会在调研过程中很幸运地发现一个更新的平台，它可以为你提供更快的速度又不损失其他任何方面！



在训练神经网络时，通常在每一个训练步中一次发送大量的训练样本。这使得能够进行大量的计算优化，这些优化在一次只提交一个样本时是无法实现的。例如，将一百张图像和标签作为一个训练单元进行训练。这种训练数据的集合就称为批次。

在嵌入式系统中，通常一次同时处理一组传感器读数，因此我们不想在触发推断之前等待收集大批次的数据。这种“单批次”（single batch）的数据处理方式意味着无法从训练过程中一些有意义的优化中受益，因此，针对云计算优化的硬件体系结构并不总能等效地转换为我们的用例。

15.3 改进模型

在更换硬件平台后，对神经网络延迟最容易产生重大影响的因素就是网络架构。如果你可以创建一个足够精确但包含较少计算量的新模型，就可以在没有任何代码改动的情况下加快推断速度。通常可以用准确率的降低来换取更快的速度，因此，如果你一开始就能够获得尽可能精确的模型，那么你将有更大的余地来考虑这些权衡。这意味着花费时间来改善和扩展训练数据在整个开发过程中都非常有帮助，即使对于诸如延迟优化等看上去没有直接联系的任务而言也是如此。

在优化程序代码时，通常更合理的方案是着重花时间优化基于高级算法的代码，而不是重写汇编中的内部循环。关注模型的架构就是基于这样的想法。如果可以，最好完全省去某些工作而不是提升它们的速度。在我们的案例中，不同之处在于，更换机器学习模型实际上比转换传统代码中的更换算法要容易得多，因为每种模型只是一个功能强大的黑匣子，可以接收输入数据并返回数值结果。收集了一组好的数据之后，在训练脚本中用一个模型替换另一个模型应该很容易。你甚至可以尝试从正在使用的模型中移除某个层，然后观察效果。神经网络往往会非常优雅地退化（degrade），因此你可以随意尝试许多不同的破坏性更改，并观察它们对准确率和延迟的影响。

15.3.1 估算模型延迟

大多数神经网络模型大部分时间都在运行大型矩阵乘法或非常近似于矩阵乘法的等效计算。这是因为每个输入值都需要针对每个输出值按不同的权重进行缩放，因此所涉及的工作量大约是输入值的数量乘网络中每个层的输出值的数量。这通常可以通过讨论网络运行单个推断所需的浮点操作（Floating-point OPeration, FLOP）的数量来近似得出。通常，一个乘加运算（通常在机器代码级别是一条指令）被视为两个 FLOP，即使你执行的是 8 位或更低的量化计算，不再涉及浮点数，它们还是经常被称为 FLOP。网络所需的 FLOP 数量可以手动逐层计算。例如，全连接层需要的 FLOP 数量等于输入向量的大小乘输出向量的大小。因此，如果你知道网络的尺寸，就可以计算出所涉及的计算量。通常，你还可以在讨论和比较模型架构（例如 MobileNet）的论文中找到 FLOP 估算值。

FLOP 可用作网络执行所需时间的粗略度量标准，因为在所有其他条件相同的情况下，涉及越少计算量的模型会运行得越快，网络运行的速度差异与 FLOP 的差异成正比。例如，你可以合理地预期需要包含 2 亿个 FLOP 的模型的运行时间大约是包含 1 亿个 FLOP 模型版本的运行时间的两倍。实际中，由于其他因素的影响，如软件对特定层的优化程度等会影响延迟等，这种结论不是完全正确的，但对于评估不同网络架构的延迟来说这仍是一个很好的起点。这有助于确定你的硬件平台的实际需求。如果你的芯片能够在 100 毫秒内运行包含 100 万个 FLOP 的模型，那么可以合理地猜测，这个芯片在运行包含 1000 万个 FLOP 的其他模型时会花费大约一秒的时间。

15.3.2 如何加速你的模型

模型的架构设计仍然是一个活跃的研究领域，因此在这一方面为初学者编写一个好的指南并不是一件容易的事情。最好的出发点是找到一些在设计时考虑到效率的现有模型，然后迭代地进行更改试验。许多模型具有特定的参数，我们可以更改这些参数来影响所需的计算量，例如 MobileNet 的深度通道系数或预期的输入大小。在其他情况下，你可能会查看每层所需的 FLOP，并尝试删除特别慢的 FLOP 或将其替换为更快的替代方法。例如，将深度卷积（depthwise convolution）替换为普通卷积（plain convolution）。如果可以的话，还值得查看在设备上运行时各层的实际延迟，而不是通过 FLOP 进行估算。不过，这将需要一些性能剖析技术，我们将在 15.6 节中讨论。



设计模型的架构既困难又耗时，但是最近在架构设计自动化方面取得了一些进展，例如 MnasNet，它使用遗传算法（genetic algorithm）等方法来改进网络设计。这些自动化设计仍然不能完全取代人类（例如，它们通常需要以已知的良好架构作为起点，还需要有关要使用的搜索空间的手动规则等），但是我们很可能会在这个领域看到快速的进展。

诸如 AutoML 之类的服务已经可以让用户避免许多烦琐的训练细节，希望这种趋势会继续下去，这样你将能够更轻松地权衡精确度和运行效率，选择最适合你的模型。

15.4 量化

运行神经网络需要为每个预测进行数十万甚至数百万次计算。大多数执行这种复杂计算的程序对数值精度非常敏感。否则，错误会累积，导致结果误差过大而无法使用。深度学习模型在这一点上非常不同——它能够应付中间计算期间数值精度的巨大损失，并且仍然可以产生总体上准确的最终结果。这个属性似乎是其训练过程的副产品——在深度网络的训练过程中，通常会输入大量噪声，因此模型学习了应对微小变化的鲁棒性，并专注于学习数据中的重要模式。

这意味着，在实践中，使用 32 位浮点表示进行的操作几乎比推断所需的精度更高。32 位浮点的训练要求更高，因为它需要对权重进行许多小的更改才能学习，但即使这样，16 位表示形式也被广泛使用。大多数推断应用程序仅使用 8 位存储权重和激活值，就可以产生与浮点等效项几乎无差别的结果。这对于嵌入式应用程序来说是个好消息，许多平台都对这些模型所依赖的 8 位乘法和累加指令提供了强大的支持，因为这些指令在信号处理算法中非常常见。

不过，将模型从浮点数转换为 8 位数值不容易。为了有效执行计算，8 位数值需要被线性转换为实数。这对于权重来说很容易，因为我们可以根据训练后的值知道每一层的范围，因此可以得出正确的缩放系数来执行转换。不过，这对于激活来说比较棘手，因为通过检查模型参数和架构并不能清楚地得到每层输出的实际范围。如果选择的范围太小，某些输出将被限制为最小值或最大值，但是如果选择的范围太大，输出的精确率就会小于其可能的精确率，并且有一定的风险损失整体的准确率。

量化现在仍然是一个活跃的研究主题，有很多不同的选择，TensorFlow 团队在过去的几年中尝试了多种方法。你可以在 Raghuraman Krishnamoorthi 撰写的“Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper”中看到其中一些试验的讨论，并且量化规范（quantization specification）涵盖了根据经验建议使用的方法。

我们已经将模型从 TensorFlow 训练环境转换为 TensorFlow Lite 计算图，并且集中进行了量化处理。我们曾经推荐一种支持训练时量化的训练方案，但是这种方法很难使用，并且我们发现使用某些其他技术可以在导出时产生等效的结果。最容易使用的量化方法是训练后权重量化（post-training weight quantization）。这种方法可以将权重降低到 8 位数值，而激活层仍保持浮点数。这很有用，因为它将模型文件的大小缩小了 75%，并能提供一些速度上的提升。这是最简单的方法，因为它不需要任何有关激活层范围的知

识，但它仍需要支持快速浮点的硬件，而很多嵌入式平台不能提供这样的硬件。

训练后整数量化（post-training integer quantization）意味着可以在没有任何浮点计算的情况下运行模型，这使得它成为我们在本书中介绍的用例的首选方法。使用它最具有挑战性的部分是，你需要在模型导出过程中提供一些示例输入，以便可以通过在图中运行一些典型的图像、音频或其他数据观察激活层输出的范围。正如前面讨论的，如果没有这些范围的估计，就不可能准确地量化这些层。过去，我们使用过其他方法，例如在训练过程中记录范围或在运行时的每次推断期间捕获范围，但是这些方法有一些缺陷，比如会使训练变得更加复杂，或者增加延迟损失，因此是最糟糕的方法。

如果你回顾一下第 10 章中有关导出行人检测模型的说明，就会发现我们向 `convert` 对象提供了 `representative_dataset` 函数。这是一个 Python 函数，可以产生激活范围估计过程所需的输入，对于行人检测器模型，我们从训练数据集中加载一些示例图像。但是，对于你训练的每个模型，你都需要了解这一点，因为每个应用程序的预期输入都不同。辨别输入如何在预处理中得到缩放和转换也可能很困难，因此创建函数可能会涉及一些反复试验。我们希望以后可以有方法简化这一过程。

在几乎所有平台上，运行完全量化的模型对优化延迟都有很大的帮助，但是，如果要支持新的设备，则可能需要优化计算量最大的算子，以利用硬件提供的专门指令。如果你正在使用卷积网络，那么开始使用 `Conv2D` 算子和 `kernel` 是一个不错的选择。你会注意到，许多 `kernel` 都有 `uint8` 和 `int8` 版本。`uint8` 版本是一种不再使用的旧量化方法，现在应该使用 `int8` 路径导出所有模型。

15.5 产品设计

你可能不认为产品设计也是一种优化延迟的方法，但实际上它是最值得投入时间的地方之一。关键是弄清楚你是否可以放松对网络速度或者准确率的要求。例如，你可能想使用摄像头以每秒若干帧的速度跟踪手势，但是如果你的人体姿势检测模型需要花费一秒的时间，那么也许可以使用更快的光学跟踪算法来跟踪速度较快的点，并在可用时以更准确但频率较低的神经网络结果对其进行更新。再举一个例子：你可以让微控制器将高级语音识别委托给通过网络访问的云 API，同时在本地设备上运行唤醒词检测。在更广泛的层面上，你可以将不确定性整合到用户界面中，以放宽对网络速度以及准确率的要求。为语音识别系统选择的唤醒词往往是短语，这些音节序列不太可能出现在常规语音中。如果你有一个手势识别系统，也许可以要求每个序列都以竖起大拇指结束，以确认命令是需要识别的。

我们的目标是尽可能提供最佳的整体用户体验，因此，在系统的其余部分，你能做的任何事情都是为了提升系统的整体容错性，这可以为权衡准确率和速度或其他需要改进的

属性提供更多的空间。

15.6 优化代码

传统的代码优化是获得更好性能的重要方法，不过我们在本章的后半部分才会讨论这个主题，因为你首先还是应该尝试其他优化延迟的方法。特别地，针对 TensorFlow Lite for Microcontrollers 的代码已经被编写得尽可能少，且可以在大多数系统上很好地运行二进制模型，因此可能存在一些仅适用于你的特定模型或平台的代码优化方法，你能够使用它们并从中受益。然而，这也是我们鼓励你尽可能推迟代码优化的原因之一——如果你更改硬件平台或所用的模型架构，那么其中许多优化将不再适用。所以，首先确定你的平台和架构是至关重要的。

性能剖析

任何代码优化工作的基础都是了解程序的不同部分分别需要运行多长时间。在嵌入式世界中，这可能是非常困难的，因为默认情况下你甚至可能没有一个简单的计时器，即使有，记录和返回所需的信息也可能会非常麻烦。这里我们列出了多种方法，其实现复杂度从易到难递增。

闪烁 (blinky)

几乎所有嵌入式开发板都至少有一个 LED，你可以通过程序控制它。如果你要测量的时间超过 0.5 秒，则可以尝试在要测量的代码的开始部分打开该 LED，然后在结束部分关闭 LED。你也许可以使用外部秒表来大致估算所需的时间，并手动计算 10 秒内你看到的闪烁次数。你还可以同时使用两个开发板来运行不同版本的代码，并通过比较闪烁频率来判断哪个版本的代码执行得更快。

霰弹枪剖析 (shotgun profiling)

在大致了解应用程序的正常运行需要花费多长时间后，估算特定代码运行时间的最简单方法是将其注释掉，并查看整体的执行速度。因为这种方法类似于霰弹枪调试 (shotgun debugging)，所以它也被称为霰弹枪剖析 (shotgun profiling)。使用这种方法，你可以删除大段代码，以便在几乎没有其他信息干扰时定位崩溃。对于神经网络调试而言，它可能会出奇地有效，因为模型执行代码中通常没有依赖于数据的分支，因此注释掉任何一部分内部实现并把它变成一段没有任何操作的代码都不会影响其他部分的执行速度。

调试日志 (debug logging)

在大多数情况下，你应该能够从嵌入式开发板上将一行文本输出回主机，所以这看上去

似乎是检测代码是否正确执行的理想方法。不幸的是，与开发机器进行通信的行为本身可能非常耗时。Arm Cortex-M 芯片上的串口调试输出可能需要最多高达 500 毫秒的时间，而且延迟差异很大，这对于简单的基于日志分析延迟来说几乎毫无用处。基于 UART 连接的调试日志记录的开销会小很多，但仍然不是很理想。

逻辑分析仪 (logic analyzer)

以与切换 LED 相似但具有更高精度的方式，你可以使代码打开和关闭 GPIO 引脚，然后使用外部逻辑分析仪（我们以前使用 Saleae Logic Pro 16）进行可视化和测量持续时间。这需要一些布线，而且设备本身可能很昂贵，但是它提供了一种非常灵活的方式来分析程序的延迟，且不需要任何软件支持，只需要一个或多个 GPIO 引脚。

计时器 (timer)

如果有一个计时器可以以足够的精度为你提供一致的当前时间，则可以使用它在感兴趣的代码部分的开始和结尾处记录时间，然后将持续的时间输出到日志中，这样就不会受任何通信延迟的影响。正是出于这个原因，我们考虑过在 TensorFlow Lite for Microcontrollers 中实现与平台无关的计时器接口，但是由于设置计时器可能很复杂，因此我们认为这可能会对将代码移植到不同平台增加太多负担。不幸的是，这意味着你将需要探索如何针对正在运行的芯片自己实现计时器的功能。还有一个缺点是，你需要在任何想要研究的代码片段前后调用计时器，因此这需要一定的工作量来确认和计算，并且每次要增加需要研究的代码执行时间时，都需要不断地重新编译和烧录。

剖析器 (profiler)

如果幸运的话，你将使用支持某种外部剖析工具的工具链和平台。这些应用程序通常将使用程序中的调试信息来匹配从设备上运行程序时收集到的执行统计信息。然后，它们能够可视化哪些功能甚至是哪几行代码花费了最多的时间。这是定位代码中速度瓶颈的最快方法，因为你将能够快速探索并放大重要的函数。

15.7 优化算子

在确保使用尽可能简单的模型并确定了代码的哪些部分花费最多的时间之后，你应该考虑如何提高它们的速度。神经网络的大部分执行时间都应该花在算子实现内部，因为它们可能涉及每一层数十万甚至数百万次的计算，因此你可能会发现其中一个或多个成为瓶颈。

15.7.1 寻找已优化的实现

TensorFlow Lite for Microcontrollers 中所有算子的默认实现都被编写为小单元、易于理解、可移植的实现，而不是快速的实现。因此，可以预期你应该能够使用更多的代码行

或内存，在速度上轻易地击败它们。我们使用第 13 章介绍的子文件夹专门化方法，在 *kernels/portable_optimized* 目录中提供了一组更快的实现。这些实现不应有任何平台依赖性，但是较之参考版本，它们会使用更多的内存。由于它们使用的是特定的子文件夹，因此你只需传递 `TAGS="portable_optimized"` 参数就可以编译生成使用这些文件夹而非默认文件夹的项目。

如果你使用的设备具有特定于平台的实现方式（例如通过 CMSIS-NN 之类的库），并且在指定目标时不会自动选择它们，则可以通过传递适当的标签来选择使用这些不可移植的版本。不过，你需要浏览平台的文档和 TensorFlow Lite for Microcontrollers 源代码树，才能找到是否已存在优化实现以及对应的标签是什么。

15.7.2 编写你自己的优化实现

如果你无法找到耗时最长的算子的优化实现，或者现有实现的速度不够快，那么你可能需要编写自己的算子。好消息是你应该能够缩小范围以使工作变得更加容易。你只会使用一定范围内的输入与输出大小和参数来调用算子，因此你只需要专注于使部分路径更快，而不用尝试优化所有的情况。例如，我们发现在 SparkFun Edge 开发板上的语音唤醒单词示例的第一个版本中，深度卷积参考代码花费了大量的时间，并且总体运行速度太慢，无法使用。当查看代码在做什么时，我们发现卷积过滤器的宽度始终为 8，这使得我们能够利用这个模式来编写一些优化代码。通过使用 32 位整数并行获取它们，我们可以一次加载 4 个输入值和 4 个以字节为单位的权重。

要开始优化过程，请先使用前面介绍的子文件夹专门化方法在 *kernels* 根目录中创建一个新目录。将参考 kernel 实现复制到该子文件夹中，作为优化代码的起点。为确保一切正常，请运行与该算子关联的单元测试，并确保它仍然可以通过。如果你传递的是正确的标记，就会使用新的实现：

```
make -f tensorflow/lite/micro/tools/make/Makefile test_depthwise_conv_\
      test TAGS="portable_optimized"
```

然后，我们建议为你的算子的单元测试代码添加一个新测试，该测试不检查正确性，而只是报告执行算子所花费的时间。拥有这样的基准将帮助你验证所做的更改是否正在以你期望的方式提升性能。对于每种情况，你都应该有一个基准测试，在这个基准测试中你会看到性能瓶颈，并且其具有与模型优化之前相同的大小和其他参数（权重和输入可以是随机值，因为在大多数情况下，这些数字不会影响执行延迟）。基准代码本身将需要依赖本章前面讨论的一种配置方法，理想情况下，你应该使用高精度计时器来测量持续时间。低精度计时器也可以，但至少不能使用 LED 或逻辑输出来观测延迟。如果测量过程的粒度太大，则可能需要在一个循环中多次执行该算子，然后除以迭代次数以捕获实际时间。编写完基准测试之后，请在进行任何更改之前记录延迟，并确保它与你从

剖析程序中看到的延迟大致匹配。

有了一个可用的代表性基准测试，你现在应该能够快速迭代潜在的优化。好的第一步是找到初始实现的最内层循环。这部分代码的执行频率最高，因此，改进这一部分会比改进其他部分产生更大的影响。希望你能够通过遍历代码并从字面上找到嵌套最深的 `for` 循环（或等效的循环）来找到对应的部分。当然，你也可以通过注释掉它并再次运行基准测试来验证对应这部分是否执行频率最高。如果延迟急剧下降（希望下降 50% 或更多），则说明你已找到合适的代码块。例如，从深度卷积的参考实现中获取以下代码：

```
for (int b = 0; b < batches; ++b) {
    for (int out_y = 0; out_y < output_height; ++out_y) {
        for (int out_x = 0; out_x < output_width; ++out_x) {
            for (int ic = 0; ic < input_depth; ++ic) {
                for (int m = 0; m < depth_multiplier; m++) {
                    const int oc = m + ic * depth_multiplier;
                    const int in_x_origin = (out_x * stride_width) - pad_width;
                    const int in_y_origin = (out_y * stride_height) - pad_height;
                    int32 acc = 0;
                    for (int filter_y = 0; filter_y < filter_height; ++filter_y) {
                        for (int filter_x = 0; filter_x < filter_width; ++filter_x) {
                            const int in_x =
                                in_x_origin + dilation_width_factor * filter_x;
                            const int in_y =
                                in_y_origin + dilation_height_factor * filter_y;
                            // If the location is outside the bounds of the input image,
                            // use zero as a default value.
                            if ((in_x >= 0) && (in_x < input_width) && (in_y >= 0) &&
                                (in_y < input_height)) {
                                int32 input_val =
                                    input_data[Offset(input_shape, b, in_y, in_x, ic)];
                                int32 filter_val = filter_data[Offset(
                                    filter_shape, 0, filter_y, filter_x, oc)];
                                acc += (filter_val + filter_offset) *
                                    (input_val + input_offset);
                            }
                        }
                    }
                    if (bias_data) {
                        acc += bias_data[oc];
                    }
                    acc = DepthwiseConvRound<output_rounding>(acc, output_multiplier,
                        output_shift);
                    acc += output_offset;
                    acc = std::max(acc, output_activation_min);
                    acc = std::min(acc, output_activation_max);
                    output_data[Offset(output_shape, b, out_y, out_x, oc)] =
                        static_cast<uint8>(acc);
                }
            }
        }
    }
}
```

只需检查缩进，就可以将找到循环嵌套最深部分，如下所示：

```
const int in_x =
    in_x_origin + dilation_width_factor * filter_x;
const int in_y =
    in_y_origin + dilation_height_factor * filter_y;
// If the location is outside the bounds of the input image,
// use zero as a default value.
if ((in_x >= 0) && (in_x < input_width) && (in_y >= 0) &&
    (in_y < input_height)) {
    int32 input_val =
        input_data[Offset(input_shape, b, in_y, in_x, ic)];
    int32 filter_val = filter_data[Offset(
        filter_shape, 0, filter_y, filter_x, oc)];
    acc += (filter_val + filter_offset) *
        (input_val + input_offset);
}
```

由于这段代码在所有循环的最中间，因此它的执行次数比函数中其他代码的执行次数要多得多，注释掉这段代码可以确认它花费了大部分时间。如果你有幸可以获得逐行分析的信息，那么也可以帮助你找到合适的代码块。

现在你找到了一个影响很大的区域，目标是将尽可能多的工作移到不太重要的部分。例如，中间有一个 `if` 语句，这意味着必须在每个内循环迭代上执行条件检查，但是可以将检查工作提升到代码的这一部分之外，即循环外部，从而减少检查的频率。你可能还会注意到，你的特定模型和基准可能不需要判断某些条件或运行某些计算。在语音唤醒词模型中，膨胀因子（dilation factor）始终为 1，因此可以跳过涉及它的乘法运算，从而省去更多的工作。不过，我们还是建议你在顶层进行检查，以保护这类特定于参数的优化，如果参数不是优化所需要的，则应退回纯参考实现。这样可以加快已知模型的运行速度，同时确保如果遇到的算子不符合这些条件，至少还可以正常运行。

为了确保你不会意外破坏代码的正确性，在进行更改时，也应该经常为运行的算子编写单元测试。这超出了本书的范畴，我们无法涵盖优化数值处理代码的所有方法，但是你可以查看 `portable_optimized` 文件夹中的算子实现，以了解一些有用的技术。

15.7.3 利用硬件特性

到目前为止，我们只讨论了与平台无关的可移植优化。这是因为重组代码以完全避免执行某些工作通常是产生重大影响的最简单方法。它还能简化并且缩小更多专业优化的重点。你可能会发现自己在使用带有 SIMD 指令的 Cortex-M 设备之类的平台，这通常对减少神经网络推断中占大部分时间的重复计算有很大帮助。你可能会想直接使用内联函数甚至汇编语言来重写你的内部循环，但是请不要这样做！至少先检查供应商提供的库文档，看看是否已经编写了一些合适的库来实现算法的大部分内容，因为这些库很可能

已经被高度优化过（虽然很可能在你不知道优化参数的情况下没有开启这些优化）。如果可以的话，请尽可能尝试调用现有函数，而不是编写你自己的版本，以计算一些常用的东西，例如快速傅里叶变换。

如果你已经完成这些阶段，就该尝试一下平台级别的优化。我们建议的方法是，在开始时用汇编代码中的机械等效项替换单个代码行，每次替换一行，这样你就可以在进行过程中验证正确性，而不必一开始就担心延迟问题。转换完成必要的代码之后，你可以尝试使用融合算子和其他技术来减少延迟。使用嵌入式系统的一个优点是，与有深层指令流水线（deep instruction pipeline）或高速缓存的更复杂的处理器相比，嵌入式系统的行
为往往更加单纯，因此，在没有太多意外副作用风险的情况下，在理论上了解性能瓶颈并建立潜在的汇编级优化也更具可行性。

15.7.4 加速器和协处理器

随着机器学习工作负载在嵌入式世界中变得越来越重要，我们看到越来越多的系统出现，它们提供了专用的硬件来加速或降低机器学习所需功耗。但是，目前还没有针对它们的清晰的编程模型或者标准 API，因此如何将它们与软件框架集成并不总是清晰明了。借助 TensorFlow Lite for Microcontrollers，我们希望支持与主处理器同步工作的硬件的直接集成，但异步组件超出了当前项目的范围。

同步（synchronous）是指加速硬件与主 CPU 紧密耦合，共享一个内存空间，并且算子可以非常快速地调用加速器并阻塞直到返回结果。TensorFlow Lite 上的线程层有可能在此阻塞期间将工作分配给另一个线程或进程，但这在当前大多数嵌入式平台上都不太可能实现。从程序员的角度看，这种加速器看起来更像是早期 x86 系统上存在的浮点协处理器（floating-point coprocessor），而不是替代模型，后者更像是 GPU。我们之所以专注于这类同步加速器，是因为它们对于我们针对的低功耗系统似乎最有意义，避免异步协调可以使运行时更加简单。

类协处理器加速器（coprocessor-like accelerator）需要非常接近系统架构中的 CPU 才能以如此低的延迟做出响应。相反的模型是现代 GPU 使用的模型，它有一个完全独立的系统，在总线的另一端具有自己的控制逻辑。对这类处理器进行编程涉及 CPU 对大量命令进行排队，这些命令的执行时间相对较长，并在批处理就绪后被立即发送出去，CPU 随即继续进行其他工作，而不必等待加速器完成。在此模型中，CPU 和加速器之间的任何通信延迟都无关紧要，因为发送命令的频率不高，并且不会阻塞结果。加速器可以从这种方法中受益，因为一次看到大量命令会提供很多机会来重新排列和优化所涉及的工作。然而这种方式在任务更加细粒度、需要按顺序执行时是很难应用的。它非常适合图形渲染，因为结果根本不需要返回到 CPU——渲染的显示缓冲区仅显示给用户。它适应了深度学习训练，通过发送大量训练样本确保一次完成大量工作，并尽可能多地将

其保留在显卡上，从而避免了复制回 CPU 的情况。随着嵌入式系统变得越来越复杂并承担更大的工作量，我们可能会重新审视有关框架的需求，并使用移动 TensorFlow Lite 中的 delegate 之类的接口来支持该流程，但这超出了 TensorFlow Lite 当前版本的范畴。

15.8 回馈开源

我们总是渴望看到对 TensorFlow Lite 的贡献，在你努力优化了一些框架代码之后，你可能有兴趣将你的代码共享回主线（mainline）。一个不错的起点是加入 SIG Micro 邮件列表，然后发送一封简短的电子邮件来总结你的工作，并提供指向 TensorFlow 代码仓库分支的代码以及你建议的更改。如果你能提供正在使用的基准测试以及一些讨论了优化是如何进行、适用于何种场景的内联文档就更好了。社区应该能够提供反馈，社区成员将寻找可以编译的、通用的、易于维护和测试的代码。我们迫不及待想看看你有什么发现，并感谢你考虑开源你的优化！

15.9 小结

在本章中，我们讨论了一些有关优化模型延迟的内容。最快的代码是根本不运行的代码，因此要记住，最关键的是在开始优化单个函数之前，首先在模型和算法级别减少要做的工作。你可能需要先解决延迟问题，然后才能让应用程序在实际设备上运行，并测试它是否按你希望的方式工作。在那之后，下一个重点可能是确保你的设备有足够的使用寿命——这就是下一章要讨论的内容，即如何优化功耗。

优化功耗

与桌面系统或者移动系统相比，嵌入式设备最重要的优势是它的功耗非常小。服务器 CPU 的功率可能会达到几十瓦甚至几百瓦，需要冷却系统和主电源才能运行。就连一部手机也会有几瓦的功率，每天都需要充电。而微控制器的运行功率不到一毫瓦，是手机 CPU 功率的几千分之一，因此可以依靠纽扣电池或者能源收集设备持续运行几个星期、几个月甚至几年。

如果你正在开发 TinyML 产品，那么你必须应对的最具挑战性的限制条件可能就是电池的寿命。需要人工干预来更换电池或者为电池充电通常并不是可行的方案，因此，设备的使用寿命（设备可以持续工作多长时间）将由它消耗的能源以及可以存储的能源来决定。电池容量通常会受到产品物理尺寸的限制，例如，即剥即贴型传感器（peel-and-stick sensor）不太可能容纳除纽扣电池以外的任何电子元件，即便你能够使用能源收集装置，它可以提供多少能源通常也会受到严格的限制。这意味着对于设备的使用寿命而言，你唯一可以影响或者控制的因素就是整个系统消耗的能源。在本章中，我们将讨论如何调查能源使用情况以及如何优化功耗。

16.1 开发直觉

大多数桌面系统工程师对各种操作需要花费多长时间都会有一个粗略的概念，他们知道网络请求可能比在内存中读取数据要慢，而从固态硬盘（Solid-State Drive, SSD）访问文件通常会比从旋转磁盘驱动（spinning-disk drive）中读取要快。然而，大多数工程师可能对功耗没有什么概念，毕竟大多数时候都不需要考虑一个功能会耗费多少能源。要建立思想模型并规划能源消耗效率，你需要对各种操作的功耗有一些经验性的概念。



在本章中，我们将在能源（energy）测量和功率（power）测量之间来回切换。功率是随时间变化的能源，例如，每秒使用 1 焦能源的 CPU 的功率为 1 瓦。由于我们最关心的是设备的使用寿命，因此将平均功耗作为度量标准通常是

最有帮助的，因为具有固定可使用电量的设备的平均功耗与设备最长运行时间成反比。这意味着我们可以轻松地预测：平均功率为 1 毫瓦的系统的寿命大致是功率为 2 毫瓦的系统的寿命的两倍。当然，有些时候，我们依旧关心某些一次性的、不能持续很长时间的操作的功耗。

16.1.1 典型的元件功耗

如果你想深入了解系统各个组件使用了多少能源，请参阅由 Sasu Tarkoma 等人撰写、剑桥大学出版社出版的 *Smartphone Energy Consumption*。这是一本很好的书。以下是我们从书中得出的一些数字：

- Arm Cortex-A9 CPU 功率为 500~2000 毫瓦。
- 显示器的功率大约为 400 毫瓦。
- 活跃的手机无线电功率大约为 800 毫瓦。
- 蓝牙的功率大约为 100 毫瓦。

除了智能手机之外，以下是我们观察到的一些嵌入式组件的测量结果：

- 麦克风传感器的功率大约为 300 微瓦 (μW)。
- 低功耗蓝牙的功率为 40 毫瓦。
- 320 像素 \times 320 像素的单色图像传感器（如 Himax HM01B0）以 30FPS 采集数据的功率为 1 毫瓦。
- Ambiq Cortex-M4F 微控制器以 48MHz 时钟频率工作的功率大约为 1 毫瓦。
- 加速度传感器的功率大约为 1 毫瓦。

这些数字会根据你使用的具体组件不同而有很大的不同，但是记住这些数字仍然非常有用，这样你至少可以知道不同操作的大致比例。一个提纲挈领的经验是，在嵌入式产品中，无线电可能会比其他部件消耗更多的能源。此外，传感器和处理器功耗下降的速度似乎比通信部件功耗下降的速度快得多，因此这种差距可能还会持续扩大。一旦你了解系统中可能会用到的组件，你就需要考虑可以存储或收集多少能源来为系统提供动力。以下是一些粗略的数字（感谢 James Meyers 的能源收集估算）：

- CR2032 纽扣电池可容纳 2500 焦的能源。这意味着，如果系统的平均功率为 1 毫瓦，那么依靠 CR2032 纽扣电池大约可以运行一个月。
- AA 电池具有大约 15 000 焦的能源，因此功率为 1 毫瓦的系统使用它可以有 6 个月的寿命。
- 从工业机器上收集温差，每平方厘米能得到 1~10 毫瓦的能源。

- 每平方厘米室内光可以收集 10 微瓦的能源。
- 每平方厘米室外光可以收集 10 毫瓦的能源。

正如你所看到的，目前只有工业温差或者室外照明对于自供电系统而言是可行的，但是随着处理器和传感器能源需求的下降，我们希望日后其他方案也足以支撑自供电系统运行。你可以关注 Matrix 或 e-peas 等商业供应商，以了解一些最新的能源收集设备。

希望这些基本的数字可以帮助你勾勒出有关系统寿命、设备成本和大小等各方因素的大致组合。它们至少足以用于初步的可行性检验。如果你能将这些信息内化为直觉，就可以快速地考虑更多的潜在方案，并在不同方案中做出权衡。

16.1.2 硬件选择

当你对产品可能使用的组件有了一个大致的了解，就需要查看可以购买的真实零件。如果你正在寻找一些资料充分、可供业余爱好者使用的文档，那么最好先浏览 SparkFun、Arduino 和 AdaFruit 等的官方网站。它们都为其组件提供了附带教程、驱动程序以及有关如何连接到其他部件的建议。这些网站通常也是制作原型的最佳起点，因为你很可能可以从网站中直接找到一个完整的系统，其中包含你所需要的全部内容。集成系统最大的缺点是你的选择范围会比较受限，而且可能无法针对整体功耗进行优化，此外额外的资源通常需要付费。

如果你想要更多的选择以及更低的价格，而且不需要太多额外的支持，你可以尝试使用 Digi-Key、Mouser Electronics 和阿里巴巴等电子产品供应商。这些网站的共同点是，它们会提供所有产品的数据表。其中包含有关每个部件的大量详细信息：从如何提供时钟信号到有关芯片及其引脚尺寸等机械数据的所有内容。不过，你首先要了解的可能是功耗，但这通常并不容易找到。例如，请查看 STMicroelectronics Cortex-M0 MCU 的数据表。它有接近 100 页，而且从目录中看不出应如何找到功耗说明。我们发现一个有用的技巧是在这些文档中搜索“milliamps”或“ma”（带有空格），因为它们通常是用来表示功耗的单位。在这个数据表中，搜索结果显示在第 47 页的表中，如图 16-1 所示，该表提供了设备的功耗数据。

这张表格虽然详细但是很难理解，我们通常感兴趣的是这个芯片可能使用多少瓦（或毫瓦）的能源。为此，我们需要将显示的电流乘电压，此处列出的电压为 3.6 伏（表格顶部高亮的部分）。如果这样做的话，我们可以看到典型的功耗大致是 100 毫瓦，睡眠模式下功耗可以降低至 10 毫瓦。这足以给我们一个大致的概念，即 MCU 的功耗相对较高，虽然它的 55 美分的价格可能会弥补这一缺点，但具体还是取决于你如何权衡。你应该能够对要使用的所有组件的数据表进行类似的调研工作，并根据所有这些部分的总和来汇总可能的总体功耗情况。

表 25：电源电压 (V_{DD}) 为 3.6V 时设备的典型和最大电流消耗^①

符号	参数	条件	HCLK 频率 (f_{HCLK})	启用所有外围设备		单位 毫安 (mA)
				典型值 (Typ)	工作环境 温度 (T_A) 下最大电流 ^②	
					85°C	
静态 电流 (I_{DD})	在运行 (Run) 模式下供电，代 码在闪存中执行	使用 HSI 或者 HSE 时钟，开启 PLL	48 MHz	22.0	22.8	毫安 (mA)
			48 MHz	26.8	30.2	
			24 MHz	12.2	13.2	
			24 MHz	14.1	16.2	
		使用 HSI 或者 HSE 时钟，关闭 PLL	8 MHz	4.4	5.2	
			8 MHz	4.9	5.6	
	在运行 (Run) 模式下供电，代码 在 RAM 中执行	使用 HSI 或者 HSE 时钟，开启 PLL	48 MHz	22.2	23.2	
			48 MHz	26.1	29.3	
			24 MHz	11.2	12.2	
			24 MHz	13.3	15.7	
		使用 HSI 或者 HSE 时钟，关闭 PLL	8 MHz	4.0	4.5	
			8 MHz	4.6	5.2	
	在休眠 (Sleep) 模式下供电，代 码在闪存或 RAM 中执行	使用 HSI 或者 HSE 时钟，开启 PLL	48 MHz	14.0	15.3	
			48 MHz	17.0	19.0	
			24 MHz	7.3	7.8	
			24 MHz	8.7	10.1	
		使用 HSI 或者 HSE 时钟，关闭 PLL	8 MHz	2.6	2.9	
			8 MHz	3.0	3.5	

① 灰色阴影用于区分 STM32P030xC 设备的值。

② 数据基于设备描述结果，除非另有说明，否则未经实际产品测试。

图 16-1: STMicroelectronics 的电流消耗表

16.2 测量实际功耗

一旦有了一组组件，就需要将它们组装成一个完整的系统。这个过程超出了本书的范畴，但是我们建议你尝试在这个过程中尽早完成一些工作，以便可以在实际环境中试用你的产品，并了解更多关于它的需求。即使你没有完全使用你想要的组件，或者还没有准备好所有软件，尽早获得反馈也是非常宝贵的。

拥有完整系统的另一个好处是可以实际测试能源的使用情况。虽然数据表和估算值对规划很有帮助，但是总有一些内容并不适用于具体模型，而集成测试通常会得到比你预期

更高的功耗。

有很多工具可以用来测量系统的功耗，而且知道如何使用万用表（multimeter）（用于测量各种电器性能的设备）通常会很用。但是，最可靠的方法还是把容量已知的电池放在设备中，然后看它能持续运行多长时间。毕竟这才是你真正关心的事情。虽然你的目标可能是几个月甚至几年的设备寿命，但最有可能的是在第一次尝试时，设备只能持续运行几小时或者几天。这种试验方法的优势在于，它可以捕获你关心的所有影响，包括电压降得太低时的故障——这不太可能在简单的模型计算中暴露出来。同时，这种方法也非常简单，即使是软件工程师也能够掌握它！

16.3 估算模型的功耗

要估计模型在特定设备上会使用多少能源，最简单的方法是测量运行一个推断的延迟，然后将它乘系统在该时间段内的平均功耗，来获得模型的功耗。在项目开始时，你不太可能获得有关延迟和功耗的确切数字，但可以估计一个大致的数字。如果你知道一个模型需要多少个算术运算，以及一个处理器每秒可以执行多少次运算，就可以大致估算出执行模型所需的时间。数据表通常会为你提供特定频率和电压下设备的功耗，但是请注意，它可能不会包括整个系统的公共部分，比如内存或者外围设备。值得一提的是，你有必要对这些早期的估算数字持怀疑态度，并将它用作实现目标的上限，但至少你可以对这种方法的可行性有所了解。

举例来说，如果你有一个模型，需要执行 6 千万次操作（如行人检测器），并且你有一个类似 Arm Cortex-M4 的芯片运行频率为 48MHz，并且你认为它可以使用其 DSP 扩展在每个周期中执行两个 8 位乘法 / 加法，你猜测最大延迟可能为 $48\ 000\ 000/60\ 000\ 000=0.8$ (秒)。如果你的芯片功率为 2 毫瓦，则每次运行推断的功率就是为 1.6 焦。

16.4 降低功耗

现在你已经知道了系统的大致寿命，那么你可能正在寻找延长它的方法。你可能会找到一些对你有用的硬件修改，比如关闭不需要的模块或者更换组件，但是这超出了本书的范畴。幸运的是，有一些通用的、不需要电气工程知识的技术也可以提供很多帮助。由于这些方法都是面向软件的，因此它们都假定微控制器本身的功耗占据了系统功耗的绝大部分。如果设备中的传感器或其他组件属于耗电设备，那么你需要进行更多的硬件调研。

16.4.1 间歇性轮停

几乎所有的嵌入式处理器都可以让自己进入睡眠模式。在睡眠模式下，它们不执行任何计算，而且只消耗很少的能源，但是能够在一定时间后或者在有外部信号传入时被

唤醒。这意味着降低功耗的最简单方法之一就是在推断调用之间插入睡眠，从而使处理器在更多的时间内处于低功耗模式。这在嵌入式世界中通常被称为间歇性轮停（duty cycling）。你可能会担心这会导致传感器收集的数据不连续，但是许多现代微控制器具有直接存储器访问（Direct Memory Access, DMA）功能，能够在没有主处理器参与的情况下连续采样模数转换器（Analog-to-Digital Converter, ADC），并将结果存储在内存中。

以类似的方式，你可以降低处理器执行指令的频率，使它实际运行得慢一点，从而大大降低处理器的功耗。前面显示的数据表展示了处理器所需的能源如何随着时钟频率的降低而降低。

间歇性轮停和频率降低都是以降低计算能力来交换功耗。实际上，这意味着如果可以减少软件的延迟，就可以用它来获得更低的功耗预算。即便你的延迟足以满足需求，如果你希望减少功耗，优化延迟依旧是值得考虑的事情。

16.4.2 级联设计

与传统的过程式编程相比，机器学习的一大优势在于，它可以轻松扩展或者缩减所需的计算和存储资源，当然，准确率通常也会有一定程度的降低。使用手动编码的算法很难做到这一点，因为通常没有明显的参数可以调整来影响这些属性。这意味着你可以创建所谓的级联模型（cascade of models）。可以将传感器数据先输入一个非常小的模型，它只需要很少的计算，即使不是特别准确，也可以针对特定场景调优，以便在特定条件出现时触发（虽然会产生大量的误报）。如果结果表明发生了一些系统可能会感兴趣的事情，就可以将相同的输入再次输入更加复杂的模型以产生更准确的结果。这个过程可以重复多个阶段。

这种方法之所以有用，是因为不够精确但是体积很小的模型非常适合低功耗的嵌入式设备，而且连续运行不会消耗太多的能源。当发现潜在事件时，它可以唤醒功能更强大的系统并运行更大的模型，以此类推。由于功能更强大的系统只会运行很短的时间，因此它的功耗也不会超出预算。这就是手机上一直保持打开的语音接口的工作方式。DSP 会持续地监听麦克风，通常有一个简单的模型用来监听“Alexa”“Siri”“Hey Google”或者类似的唤醒词。主 CPU 可以处于睡眠模式，但是当 DSP 认为可能已经监听到正确的唤醒短语时，它将发出信号将系统唤醒。然后，CPU 可以运行更大、更准确的模型，以确认它是否确实是正确的短语，如果是，就可以将接下来的语音发送给云中功能更强大的处理器。

这意味着即使嵌入式产品无法托管足够准确的模型，以使其自身可以独立执行，但它也可以能够实现相应目标。如果你训练的网络能够检测到大多数真阳性（true positive）样

本，而且假阳性（false positive）出现的频率足够低，就可以将其余工作转移到云中。虽然无线电设备非常耗电，但是如果你能够将它的使用限制在极少数情况下，且只使用相对较短的时间，那么它还是可能会满足你的能源预算。

16.5 小结

对于许多人（包括本书作者）来说，优化功耗都是一个陌生的过程。幸运的是，我们之前介绍的许多优化延迟的技术也同样适用于优化功耗，只是需要监视的指标不同而已。通常，最好先优化延迟，再关注功耗，因为你经常需要先验证你的产品是否能够为短期用户提供足够好的体验，即使它的寿命使其不足以在现实场景中使用。同样，在优化好延迟和功耗之后，第 17 章的主题——优化空间通常也很有意义。在实践中，你可能会在所有不同的权衡之间来回迭代以满足你的约束，但是在其他方面足够稳定之后，空间使用通常是最容易进行调整的部分。

优化模型和二进制文件大小

无论你选择什么平台，闪存和 RAM 都可能非常有限。大多数嵌入式系统的闪存中只有不到 1MB 的只读存储，许多嵌入式系统甚至只有几十 KB。内存也是如此：很少有具有超过 512KB 的静态 RAM (SRAM) 的嵌入式系统，对于一些低端设备，这个数字可能低至个位数。好消息是，TensorFlow Lite for Microcontrollers 只需要 20KB 的闪存和 4KB 的 SRAM 就可以工作，但是你还是需要仔细设计应用程序，并在工程设计上进行权衡，以保持较低的内存占用。本章将介绍一些方法，可以用于监视和控制内存以及存储需求。

17.1 了解系统限制

大多数嵌入式系统都基于这样的架构：程序和其他只读数据存储在闪存中，仅在上传新的可执行文件时才会被写入闪存。嵌入式系统通常也具有可读写的内存，一般使用 SRAM 技术。这与大型 CPU 使用的缓存技术相同，它提供了低功耗的快速访问，但是大小受到限制。更高级的微控制器可能会提供可修改的第二层内存，使用功耗更大但是可扩展的技术，如动态 RAM (DRAM)。

你需要了解潜在平台提供了什么架构，以及需要进行哪些权衡。例如，具有很多二级 DRAM 的芯片可能会因其灵活性而极具吸引力，但是启用额外的内存会超出你的功率预算，可能就不值得选用它。如果同本书关注的一样，希望以 1 毫瓦左右的功率进行操作，那么你通常无法使用除 SRAM 以外的任何二级内存，因为更大的内存会消耗更多的能源。这意味着，你需要考虑的两个关键指标是，有多少可用的闪存只读存储以及有多少可用的 SRAM。这些数字应列在任何你要查找的芯片描述中。希望你甚至不必像 16.1.2 节那样深入搜索就能找到它。

17.2 估算内存使用率

当你对可选的硬件有了一定的概念时，就需要了解软件会需要多少资源，以及可以进行

哪些权衡以控制这些需求。

17.2.1 闪存使用

通常，你可以编译完整的可执行文件，然后查看生成的可执行文件的大小，以确定要在闪存中预留多少确切的空间。这可能会造成混淆，因为链接器生成的第一个文件通常是带有调试符号和节点信息的带注释版本的可执行文件，格式类似于 ELF（我们会在 17.5.1 节对此进行详细讨论）。你要查看的文件是实际被写入设备中的文件，通常由 `objcopy` 之类的工具生成，用于测量所需闪存的大小的最简单公式是以下因素的总和：

操作系统大小

如果你使用了任何种类的实时操作系统（Real-Time Operating System, RTOS），则可执行文件中都需要预留空间来保存其代码。通常，这可以根据你使用的功能进行配置，而估算空间使用的最简单方法就是构建一个启用了所需功能的“Hello World”示例程序，然后查看镜像文件的大小，这将为你提供操作系统程序代码大小的基线。一些典型的模块会占用大量程序空间，包括 USB、WiFi、蓝牙和蜂窝无线电技术栈，因此，如果你打算使用这些模块，请确保它们已经被启用。

TensorFlow Lite for Microcontrollers 代码大小

ML 框架需要一定的空间以供程序逻辑加载和执行神经网络模型，包括运行核心算法的算子实现。在本章稍后的部分，我们将讨论如何配置框架以减小特定应用程序的大小，但是在配置之前，你可以先编译包含框架的标准单元测试（如 `micro_speech` 测试），并查看生成的镜像大小来估计 ML 代码的空间使用。

模型数据大小

如果你还没有训练模型，可以通过计算权重来估算它将需要多少闪存空间。例如，全连接层的权重等于其输入向量的大小乘其输出向量的大小。对于卷积层，计算要稍微复杂一些：你需要将过滤器框的宽度和高度乘输入通道的数量，再乘过滤器的数量，还需要加上每个图层关联的任何偏差向量所需的空间。这会使得计算越来越复杂，因此更简单的方法是直接在 TensorFlow 中创建候选模型，然后将其导出到 TensorFlow Lite 文件中。将这个文件直接映射到闪存中，它的大小会为你提供一个确切的数字，即占用了多少空间。你还可以通过查看 Keras 的 `model.summary()` 方法来查看权重的数量。



我们在第 4 章中介绍了量化，并在第 15 章中进一步讨论了它，但是在本章主要关注优化模型大小的背景下，我们值得快速地复习一下量化的概念。在训练过程中，权重通常被存储为浮点数，每个浮点数在内存中占用 4 字节。由于移动设备和嵌入式设备的空间限制，TensorFlow Lite 支持在量化过程中将

这些值压缩为单个字节。它的工作原理是：跟踪存储在浮点数数组中的最小值和最大值，然后将所有的浮点值线性转换为对应范围内等距的 256 个值中最接近的值。转换后的代码都被存储在一个字节中，并且可以以最小的准确率损失对其执行算术运算。

应用程序代码大小

你将需要代码来访问传感器数据，对其进行预处理以使数据可以被输入神经网络，并对结果做出响应。你可能还需要机器学习模块之外的其他类型的用户界面和业务逻辑。具体的代码大小可能很难估计，但是至少你应该尝试了解是否需要任何外部库（例如，用于快速傅里叶变换的库），并计算这些额外库的代码所需的空间。

17.2.2 RAM 使用

确定所需的可修改的内存大小比确定存储需求要困难得多，因为程序所使用的 RAM 大小会在程序的整个生命周期中一直变化。与估算闪存需求的过程类似，你需要查看软件的不同层级来估算总体的使用需求：

操作系统大小

大多数 RTOS（如 FreeRTOS）都会提供有关不同的配置选项需要多少 RAM 的文档，你应该能够使用这些信息来计划所需的 RAM 大小。你需要留意的是可能需要缓冲区的模块，尤其是 TCP/IP、WiFi 或者蓝牙等通信堆栈。这些都需要被添加到任何核心操作系统的需求中。

TensorFlow Lite for Microcontrollers 所需的 RAM 大小

ML 框架的核心运行时并不需要太大的内存，它的数据结构在 SRAM 中所占的空间不应超过几 KB。这些对象作为解释器使用的类的一部分被分配，因此，它们会在堆栈中还是通用内存中取决于你的应用程序代码会将解释器创建为全局对象还是局部对象。我们通常建议将它们创建为全局对象或者静态对象，因为空间不足通常会在链接器链接时引起错误；而堆栈分配的本地变量可能会导致运行时崩溃，运行时的错误通常更难以理解。

模型内存大小

当一个神经网络被执行时，网络中一层的结果将被馈送到后续的算子中，因此每一层的结果都必须被保留一段时间。这些激活层的寿命取决于它们在图中的位置，并且每个激活层所需的内存大小取决于该层写出的数组形状。这些变体意味着有必要计算并计划随时间变换的内存大小，以使所有这些临时缓冲区都能被容纳到尽可能小的内存区域中。目前，这些算子会在解释器首次加载模型时完成，因此，如果 arena 不够大，你将会在控制台中看到错误。如果你在错误消息中看到了可用内

存和所需内存之间的差值，并将 arena 增加了相应的大小，那么应该可以绕过这个错误。

应用程序内存大小

如果程序大小一样，那么应用逻辑的内存使用情况在代码完成之前也很难计算。不过，你可以对一些较大的内存使用者做出一些猜测，例如存储传入的样本数据所需的缓冲区，或者对某些库进行预处理所需的内存。

17.3 关于不同问题的模型准确率和规模的大致数字

了解各种问题的最新状态是很有用的，它能帮助你规划应用程序可能实现的目标。机器学习不是魔法，了解它的局限性将有助于你在构建产品时做出明智的权衡。在第 14 章中，我们研究了设计过程，这也是发展产品直觉的起点，但是你还需要考虑由模型被迫进入严格的资源约束下导致准确率降低的情况。为了解决这个问题，这里有一些为嵌入式系统设计的架构示例。如果其中某个示例接近于你需要做的事情，它可能会帮助你设想在模型创建的最后可以实现的目标。显然，你的实际结果将在很大程度上取决于特定的产品和环境，因此请使用这些示例作为规划的指导，而不要期望能够获得完全相同的性能。

17.3.1 语音唤醒词模型

使用前面作为代码示例介绍的具有 400 000 次算术运算的小型（18KB）模型，识别 4 种声音：沉默、未知单词，“yes” 和 “no”，模型最高可以达到 85% 的准确率（请参见 18.2.2 节）。这是训练评估指标，它呈现的是对模型输入长度为 1 秒的剪辑音频并进行一次分类的结果。实际上，你通常会在流式音频上使用这种模型，并根据随着时间的推移逐渐向前移动 1 秒的窗口反复预测结果，因此实际应用中的准确率可能还会低于这个数字。通常，你应该将这种大小的音频模型作为一个较大的级联处理中的第一层守卫，以便通过更复杂的模型消除和处理这个小模型引入的误差。

根据经验，在语音接口中，你可能还需要一个具有 300~400KB 的权重和数百万次算术运算的大型模型，这样的唤醒词检测准确率才足以使其能够应用于现实场景。不幸的是，由于仍然没有足够的开放式标记语音数据库可用，你可能还需要一个具备商业质量的数据集进行训练，但希望随着时间的推移，这些限制都会有所缓解。

17.3.2 加速度传感器预测性维修模型

工业生产中有各种各样的预测性维修问题，其中相对简单的一个问题就是检测电动机的轴承故障，这通常表现为独特的抖动，可以从加速度传感器数据中发现特定的模式。能

够找出这些模式的合理模型可能只需要几千个权重，大小不超过 10KB，并且只需要数十万次算术运算。在使用此类模型对这些事件进行分类时，你可以期望获得 95%以上的准确率，并且可以想象以此为基础扩大模型的复杂度，以处理更棘手的问题（例如，在具有许多活动部件的机器或正在运转的机器上检测故障）。当然，模型参数和运算的数量也会增加。

17.3.3 行人检测

在嵌入式平台上，计算机视觉还不是一项普遍的任务，因此我们仍在努力弄清楚哪些应用程序是有意义的。我们听到的一个常见需求是能够检测到何时有人出现在附近，以便能唤醒用户界面，或者执行其他某些功耗更大而不可能一直保持运行状态的任务。我们已经尝试在 Visual Wake Word Challenge 中正式提出此问题，结果表明，如果你使用 250KB 和具有大约 6000 万次算术运算的模型，对小尺寸（96 像素 × 96 像素）的单色图像进行二元分类，则基本可以达到 90% 的准确率。这是使用缩小的 MobileNet v2 架构（如前所述）的基准，我们希望随着更多研究人员尝试解决这个特定问题，可以看到准确率逐渐提高，但这个数字已经可以让你粗略估计需要使用多少微控制器内存，以及可以得到什么样的准确率来解决视觉相关问题。你可能会想知道，这么小的模型能否处理流行的 ImageNet-1000 类问题——这很难确切地估计，因为 1000 个类别的最终全连接层可以轻松占用几百甚至几千字节（参数的数量是嵌入输入乘类别的数量），但是对于大约 500KB 的总大小，你可以期望最高大约为 50% 的准确率。

17.4 模型选择

在优化模型和二进制大小方面，我们强烈建议你从现有的模型开始。正如我们在第 14 章中讨论的那样，机器学习中最容易有成效的投资领域是数据收集和改进，而不是调整模型的架构。因此，从一个已知的模型开始，可以让你尽早专注于改进数据。嵌入式平台上的机器学习软件仍处于早期阶段，因此使用现有的模型，可能会有更大的概率在你关心的设备上得到支持并优化其算子。我们希望本书随附的代码示例将成为许多不同应用程序的良好起点——我们选择这些示例以尽可能覆盖多种不同类型的传感器输入，但是如果它们不适用于你的情况，也许可以在线搜索一些替代模型。如果找不到合适的优化架构，那么可以考虑在 TensorFlow 的训练环境中从头开始创建你自己的架构，但是正如第 13 章和第 19 章所讨论的那样，将模型成功地移植到微控制器上可能会涉及很多复杂的流程。

17.5 减小可执行文件的大小

你的模型可能是微控制器应用程序中最大的只读存储器使用者之一，尽管如此，你还

是必须考虑编译后的代码需要占用多少空间。这种对代码大小的限制是我们针对嵌入式平台不能只使用未经修改的 TensorFlow Lite 的原因：它将占用数百 KB 的闪存。TensorFlow Lite for Microcontrollers 的最小编译大小为 20KB，但这可能需要你进行一些更改以避免编译应用程序不需要的代码部分。

17.5.1 测量代码大小

在开始优化代码大小之前，你首先需要知道代码的大小。在嵌入式平台上，这可能会有些棘手，因为编译过程的输出通常是一个文件，其中包含调试信息和其他不会被传输到嵌入式设备上的信息，因此它们不应该被计入总大小之中。在 Arm 和其他现代工具链上，通常最终编译的结果被称为可执行和链接格式（Executable and Linking Format, ELF）文件，无论它是否带有 `.elf` 后缀。如果你使用的是 Linux 或 macOS 开发机器，则可以运行 `file` 命令来查看工具链的输出，它会告诉你哪些文件是 ELF 文件。

更适合查看的文件是通常被称为 `bin` 的文件：实际上它是已上传到嵌入式设备闪存中的代码的二进制快照。通常，这恰好是将要使用的只读闪存的大小，因此你可以通过它了解实际的闪存使用情况。你可以通过在主机上使用 `ls -l` 或 `dir` 这样的命令行，或者甚至在 GUI 文件查看器中直接找出对应文件来查看它的大小。并非所有工具链都会自动向你显示 `bin` 文件，并且文件可能没有任何后缀。但这是你下载并通过 Mbed 的 USB 将其拖到设备上的文件，并且是使用 `gcc` 工具链通过运行 `arm-none-eabi-objcopy app.elf app.bin -O binary` 命令创建的。查看 `.o` 中间体甚至编译过程生成的 `.a` 库都是没有帮助的，因为它们包含很多元数据，这些元数据最终不会进入代码空间，并且很多未使用的代码可能最终会被裁剪掉。

因为我们希望你将模型作为 C 数据数组编译到可执行文件中（因为你不能依赖于嵌入式平台支持文件系统来加载文件），所以在包含模型的任何程序中看到的二进制大小都将包含模型数据。要了解你的实际代码占用了多少空间，你需要从二进制文件大小中减去模型的大小。通常应在包含 C 数据数组的文件中定义模型大小（例如，在 `tiny_conv_micro_features_model_data.cc` 的末尾），因此你可以从二进制文件大小中减去这个大小以了解代码实际上占用了多少空间。

17.5.2 Tensorflow Lite for Microcontrollers 占用多少空间

当你知道整个应用程序代码占用的空间大小之后，就可能会想看看 TensorFlow Lite 占用了多少空间。测试这个问题的最简单方法就是注释掉框架的所有调用（包括创建 `OpResolver` 和解释器之类的对象的语句），然后查看二进制文件变小了多少。你应该期望二进制文件至少减少 20~30KB，因此，如果你没有看到这个数量级的减小，那么请再次检查是否已经注释掉了所有的引用。这应该是可行的，因为链接器会删除任

何你没有调用的代码，并将其从内存占用中删除。这种方法也可以扩展到任何其他代码块——只要你确保没有任何实际引用，这可以帮助你更好地了解空间的使用分布情况。

17.5.3 OpResolver

TensorFlow Lite 支持 100 多种算子，但是你不太可能在一个模型中用到所有这些算子。每个算子的单独实现可能只占用几 KB，但是总占用空间很快就会增加很多。幸运的是，有一个内置的机制可以消除不需要的算子的代码占用。

TensorFlow Lite 在加载模型时，会使用 `OpResolver` 接口搜索每个被包含的算子的实现。这是你传递给解释器以加载模型的类，并且它包含在给定算子定义的情况下查找指向算子实现的函数指针的逻辑。这样做的意义在于你可以控制实际链接到哪些实现。对于大多数示例代码，你将看到我们正在创建并传递 `AllOpsResolver` 类的实例。正如我们在第 5 章中讨论的那样，它实现了 `OpResolver` 接口，顾名思义，它对 TensorFlow Lite for Microcontrollers 支持的每个算子都有一个实现。这对于入门而言非常方便，因为这意味着你可以加载任何支持的模型，而不必担心这些模型需要哪些算子。

但是，当你开始关心代码大小时，你会需要重新审视这个类。与其在应用程序的主循环中传入 `AllOpsResolver` 的实例，不如将 `all_ops_resolver.cc` 和 `all_ops_resolver.h` 文件复制到你的应用程序中，并将它们重命名为 `my_app_resolver.cc` 和 `my_app_resolver.h`，并将类重命名为 `MyAppResolver`。在你的类的构造函数中，删除所有未被你的模型使用的算子的 `AddBuiltin()` 调用。幸运的是，我们知道一种简单而且自动的方法来创建模型使用的算子列表：Netron 模型查看器是一个很好的工具，它可以协助你完成这个过程。

确保使用 `MyAppResolver` 替换传递到解释器中的 `AllOpsResolver` 实例。现在，一旦编译应用程序，你应该会看到最终文件的大小明显减小。进行此更改的原因是，大多数链接程序会自动尝试删除没有被调用的代码（或无效代码）。通过删除 `AllOpsResolver` 中的引用，你可以使链接器确定，它可以删除所有未被列出的算子的实现。

如果你只使用少量的算子，则无须像我们之前提到的那样将 `AllOpsResolver` 注册封装到新的类中。相反，你可以创建 `MicroMutableOpResolver` 类的实例，然后直接添加所需注册的算子。`MicroMutableOpResolver` 实现了 `OpResolver` 接口，但它提供了其他方法，可以允许你将算子添加到列表中，这就是为什么将其命名为 `Mutable`（可变的）。这是用于实现 `AllOpsResolver` 的类，也是你自己的任何解析器类的良好基础，但是直接调用它可能会更简单。我们在某些示例中使用了这种方法，你可以从 `micro_speech` 示例的以下代码段中看到它是如何工作的：

```
static tflite::MicroMutableOpResolver micro_mutable_op_resolver;
micro_mutable_op_resolver.AddBuiltin(
    tflite::BuiltinOperator_DEPTHWISE_CONV_2D,
    tflite::ops::micro::Register_DEPTHWISE_CONV_2D());
micro_mutable_op_resolver.AddBuiltin(
    tflite::BuiltinOperator_FULLY_CONNECTED,
    tflite::ops::micro::Register_FULLY_CONNECTED());
micro_mutable_op_resolver.AddBuiltin(tflite::BuiltinOperator_SOFTMAX,
    tflite::ops::micro::Register_SOFTMAX());
```

你可能会注意到，我们将解析器（resolver）对象声明为静态对象。这是因为解析器可能在任何时候调用它，因此它的生命周期至少要与我们为解析器创建的对象的生命周期一样长。

17.5.4 了解单个函数的大小

如果你使用的是 GCC 工具链，则可以使用 nm 之类的工具来获取有关对象（.o）中间文件中函数和对象大小的信息。以下是编译二进制文件然后检查已编译的 *audio_provider.cc* 对象文件中项目大小的示例：

```
nm -S tensorflow/lite/micro/tools/make/gen/ \
sparkfun_edge_cortex-m4/obj/tensorflow/lite/micro/ \
examples/micro_speech/sparkfun_edge/audio_provider.o
```

你应该看到类似如下所示的结果：

```
00000140 t $d
00000258 t $d
00000088 t $d
00000008 t $d
00000000 b $d
00000000 r $d
00000000 r $d
00000000 t $t
00000000 t $t
00000000 t $t
00000000 t $t
00000001 00000178 T am_adc_isr
U am_hal_adc_configure
U am_hal_adc_configure_dma
U am_hal_adc_configure_slot
```

```

U am_hal_adc_enable
U am_hal_adc_initialize
U am_hal_adc_interrupt_clear
U am_hal_adc_interrupt_enable
U am_hal_adc_interrupt_status
U am_hal_adc_power_control
U am_hal_adc_sw_trigger
U am_hal_burst_mode_enable
U am_hal_burst_mode_initialize
U am_hal_cachectrl_config
U am_hal_cachectrl_defaults
U am_hal_cachectrl_enable
U am_hal_clkgen_control
U am_hal_ctimer_adc_trigger_enable
U am_hal_ctimer_config_single
U am_hal_ctimer_int_enable
U am_hal_ctimer_period_set
U am_hal_ctimer_start
U am_hal_gpio_pinconfig
U am_hal_interrupt_master_enable
U g_AM_HAL_GPIO_OUTPUT_12
00000001 0000009c T _Z15GetAudioSamplesPN6tflite13ErrorReporterEiiPiPPs
00000001 000002c4 T _Z18InitAudioRecordingPN6tflite13ErrorReporterE
00000001 0000000c T _Z20LatestAudioTimestampv
00000000 00000001 b _ZN12_GLOBAL__N_115g_adc_dma_errorE
00000000 00000400 b _ZN12_GLOBAL__N_121g_audio_output_bufferE
00000000 00007d00 b _ZN12_GLOBAL__N_122g_audio_capture_bufferE
00000000 00000001 b _ZN12_GLOBAL__N_122g_is_audio_initializedE
00000000 00002000 b _ZN12_GLOBAL__N_122g_ui32ADCSSampleBufferOE
00000000 00002000 b _ZN12_GLOBAL__N_122g_ui32ADCSSampleBuffer1E
00000000 00000004 b _ZN12_GLOBAL__N_123g_dma_destination_indexE
00000000 00000004 b _ZN12_GLOBAL__N_124g_adc_dma_error_reporterE
00000000 00000004 b _ZN12_GLOBAL__N_124g_latest_audio_timestampE
00000000 00000008 b _ZN12_GLOBAL__N_124g_total_samples_capturedE
00000000 00000004 b _ZN12_GLOBAL__N_128g_audio_capture_buffer_startE
00000000 00000004 b _ZN12_GLOBAL__N_1L12g_adc_handleE
U _ZN6tflite13ErrorReporter6ReportEPKcz

```

这些符号中许多都是内部细节或者无关紧要的信息，但最后几个符号可以识别为我们在 *audio_provider.cc* 中定义的函数，它们的名称已经被篡改以匹配 C++ 链接器的约定。第二列显示的是它的大小（以十六进制表示）。你可以在此处看到 `InitAudioRecording()` 函数的大小为 `0x2c4` 或 708 字节，这在小型微控制器上可能非常重要，因此，如果空间紧张，则有必要研究这个函数的大小来自何处。

我们发现最好的方法是反汇编混淆后的代码。幸运的是，`objdump` 工具允许我们通过使用 `-S` 标志来执行此操作——但是与 `nm` 不同，你不能使用 Linux 或 macOS 桌面上安装的标准版本。相反，你需要使用工具链附带的工具。如果你使用 TensorFlow Lite for Microcontrollers Makefile 编译，则通常会自动下载该文件。它会位于 `tensorflow/lite/micro/tools/make/downloads/gcc_embedded/bin` 之类的地方。以下是一个运行命令，以查看有关 *audio_provider.cc* 中函数的更多信息：

```
tensorflow/lite/micro/tools/make/downloads/gcc_embedded/bin/ \
arm-none-eabi-objdump -S tensorflow/lite/micro/tools/make/gen/ \
sparkfun_edge_cortex-m4/obj/tensorflow/lite/micro/examples/ \
micro_speech/sparkfun_edge/audio_provider.o
```

我们不会展示所有的输出，因为它太长了；取而代之的是，我们提供了一个精简的版本，只展示我们感兴趣的函数：

```
...
Disassembly of section .text._Z18InitAudioRecordingPN6tflite13ErrorReporterE:
00000000 <_Z18InitAudioRecordingPN6tflite13ErrorReporterE>:
TfLiteStatus InitAudioRecording(tflite::ErrorReporter* error_reporter) {
    0: b570      push {r4, r5, r6, lr}
    // Set the clock frequency.
    if (AM_HAL_STATUS_SUCCESS != am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_SYSCLK_MAX, 0)) {
        2: 2100      movs r1, #0
    TfLiteStatus InitAudioRecording(tflite::ErrorReporter* error_reporter) {
        4: b088      sub sp, #32
        6: 4604      mov r4, r0
        am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_SYSCLK_MAX, 0)) {
        8: 4608      mov r0, r1
        a: f7ff fffe bl 0 <am_hal_clkgen_control>
        if (AM_HAL_STATUS_SUCCESS != am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_SYSCLK_MAX, 0)) {
            e: 2800      cmp r0, #0
            10: f040 80e1 bne.w      1d6 <_Z18InitAudioRecordingPN6tflite13ErrorRe-
porterE+0x1d6>
                return kTfLiteError;
            }

        // Set the default cache configuration and enable it.
        if (AM_HAL_STATUS_SUCCESS != am_hal_cachectrl_config(&am_hal_cachectrl_defaults)) {
            14: 4890      ldr r0, [pc, #576] ; (244 <am_hal_cachectrl_config+0x244>)
            16: f7ff fffe bl 0 <am_hal_cachectrl_config>
            if (AM_HAL_STATUS_SUCCESS != am_hal_cachectrl_config(&am_hal_cachectrl_defaults)) {
                1a: 2800      cmp r0, #0
                1c: f040 80d4 bne.w      1c8 <_Z18InitAudioRecordingPN6tflite13ErrorRe-
porterE+0x1c8>
                    error_reporter->Report("Error - configuring the system cache failed.");
                    return kTfLiteError;
            }
            if (AM_HAL_STATUS_SUCCESS != am_hal_cachectrl_enable()) {
                20: f7ff fffe bl 0 <am_hal_cachectrl_enable>
                24: 2800      cmp r0, #0
                26: f040 80dd bne.w      1e4 <_Z18InitAudioRecordingPN6tflite13Error\

                    ReporterE+0x1e4>
            ...
        }
```

你不需要了解汇编做了什么，但是希望你可以通过查看函数 C++ 源代码的大小的增长（反汇编行最左边的数字；例如，`InitAudioRecording()` 末尾，十六进制的 10 对应的数字）来了解空间的去向。如果你查看整个函数，就会发现所有硬件初始化代码都已

被内联到 `InitAudioRecording()` 的实现中，这也解释了这个函数为什么会这么大。

17.5.5 框架常量

我们在代码中某些地方使用硬编码的数组大小来避免动态分配内存。如果 RAM 空间变得非常紧张，那么值得尝试一下这样做，看看是否可以减小为应用程序分配的内存（或者，对于非常复杂的用例，甚至可能需要增加内存）。这些数组之一是 `TFLITE_REGISTRATIONS_MAX`，它控制可以注册多少个不同的算子，默认值为 128，这对于大多数应用程序来说可能太多了——特别是考虑到它创建了一个包含 128 个 `TfLiteRegistration` 结构体的数组，每个结构体至少需要 32 字节，也就是需要 4KB 的 RAM。你还可以在 `MicroInterpreter` 中查看类似 `kStackDataAllocatorSize` 这样的“易错点”，或尝试缩小传递到解释器构造函数中的 arena 的大小。

17.6 真正的微型模型

本章中的许多建议都与嵌入式系统有关，这些嵌入式系统可以在框架代码上使用仅占用 20KB 空间的代码来运行机器学习，并且试图以少于 10KB 的 RAM 运行。如果你的设备具有非常严格的资源限制（例如，只有几千字节的 RAM 或闪存），那么你可能并不能使用相同的方法。对于那些极端环境，你需要自己编写代码，并且非常仔细地手动调整所有内容以减少空间大小。

我们希望 TensorFlow Lite for Microcontrollers 在这些情况下仍然有用。我们建议你仍然在 TensorFlow 中训练模型，尽管它很小，然后创建并导出 TensorFlow Lite 模型文件。这可能是提取权重的一个很好的起点，并且你可以使用现有的框架代码来验证自定义版本的结果。你正在使用的算子的参考实现也应该是你自己的算子代码的良好起点；即使它们不是针对延迟的最佳选择，也应该是可移植的、易于理解的并且对内存相对友好的。

17.7 小结

在本章中，我们研究了一些用来减少嵌入式机器学习项目所需的存储空间的技术。空间可能会是你需要克服的最困难的限制条件之一，但是当你的应用程序足够小、足够快且不需要消耗过多能源时，你就有了一个清晰的路径来交付产品。剩下的就是清除所有不可避免的、会导致设备以意外的方式运行的错误。调试可能是一个令人沮丧的过程——我们听说它被描述为“谋杀之谜”（murder mystery），你同时扮演了侦探、受害者和谋杀者的角色——但这是学习交付产品必不可少的技能。第 18 章将介绍一些基本技术，可以帮助你了解机器学习系统中正在发生的事情。

调试

当你将机器学习集成到产品中时，无论是嵌入式产品还是以其他方式，你一定会遇到一些令人困惑的错误，而且通常很早就会遇到。在本章中，我们将讨论一些方法，用来理解错误出现时产品发生了什么事情。

18.1 训练与部署之间准确率的损失

当你将一个机器学习模型从 TensorFlow 这样的创作环境中提取出来，然后将它部署到应用程序中时，可能会出现很多问题。即使你能够在不报告任何错误的情况下编译模型并且使其良好运行，你可能仍然无法获得预期的准确率。这可能会非常令人沮丧，因为神经网络的推断步骤看起来像一个黑匣子，无法看到内部发生了什么，也无法知道是什么原因导致了问题。

18.1.1 预处理差异

在机器学习研究中，一个没有引起太多关注的领域是如何将训练样本转换为神经网络可以操作的形式。如果你要对图像进行对象分类，那么你必须将这些图像转换为张量，这些张量是由数字组成的多维数组。你可能会认为这很简单，因为图像本身就被存储为二维数组，通常具有红色、绿色和蓝色三个通道。即使在这种情况下，你仍然需要进行一些更改。分类模型期望它们的输入具有特定的宽度和高度，例如 224 像素宽乘 224 像素高，而摄像头或其他输入源不太可能直接产生正确尺寸的图像。这意味着你需要重新缩放捕获的数据以匹配模型的要求。你必须为训练过程做类似的事情，因为数据可能是磁盘上任意大小的图像集。

一个经常出现的微妙问题是，用于部署的缩放方法与用于训练模型的缩放方法不匹配。例如，早期版本的 Inception 使用双线性缩放（bilinear scaling）来缩小图像，这会使具有图像处理背景的人感到困惑，因为这种缩放方式会降低图像的视觉质量，通常被避免

使用。结果，许多开发人员在应用程序中使用这些模型进行推断，同时使用了更正确的区域采样（area sampling）方法，但是事实证明，这实际上会降低结果的准确率！直觉告诉我们，训练好的模型已经学会了寻找通过双线性缩放产生的伪像，而放弃使用双线性缩放会导致第一名的错误率增加几个百分点。

图像预处理也不止重新缩放这一步。还有一个问题是将通常范围为 0~255 编码的图像值转换为训练期间使用的浮点数。由于多种原因，这些图像值经常会被线性缩放到较小范围：-1.0~1.0 或者 0.0~1.0。如果要输入浮点数，则需要在应用程序中进行相同的数值缩放。如果直接输入 8 位值，那么无须在运行时执行此操作——原始 8 位值可以不经转换就被使用，但仍然需要通过 `--mean_values` 和 `--std_values` 标志将它们传递到 `toco` 等导出工具。对于 -1.0~1.0 的范围，请使用 `--mean_values=128 --std_values=128`。

令人困惑的是，通常很难知道模型代码中输入图像值的正确比例是多少，因为这是一个细节，通常被深埋在所用 API 的实现中。许多已发布的 Google 模型使用的 Slim 框架默认值为 -1.0~1.0，这是一个很好的尝试范围，但是如果没有任何好的文档，你可能最终不得不通过调试 Python 实现的训练逻辑才能弄清楚正确的比例是多少。

更糟糕的是，即使大小的调整或者数值的缩放比例有一点点错误，你最终还是可能得到大部分正确的结果，但是准确率会降低。这意味着你的应用程序在通过偶然验证后似乎可以正常工作，但最终的整体体验却不如预期。实际上，相比其他领域的数据，如音频或者加速度传感器数据等，图像预处理的相关挑战要简单得多。对于其他领域，可能存在更复杂的特征生成流水线，以将原始数据转换为神经网络可以处理的数字数组。如果查看 `micro_speech` 示例的预处理代码，你就会发现必须实现信号处理的许多阶段，才能从音频样本中得到可以输入模型的频谱图，这段代码与训练版本中使用的预处理代码之间的任何差异都可能会降低最终的准确率。

18.1.2 调试预处理

考虑到这些输入数据的转换步骤很容易出错，而且你可能很难发现它出了问题——甚至就算发现了问题，也可能很难找到原因。你该怎么做？我们发现有几种方法可以帮助你。

如果可能的话，最好还是拥有一些可以在台式机上运行的代码版本，即使没有连接任何外部设备。在 Linux、macOS 或者 Windows 环境中，你将拥有更好的调试工具，并且可以轻松地在训练工具和应用程序之间传输测试数据。对于 TensorFlow Lite for Microcontrollers 中的示例代码，我们将应用程序的不同部分划分为模块，并为 Linux 和 macOS 目标启用了 Makefile 编译，因此我们可以分别运行推断和预处理。

调试预处理问题最重要的方法是比较训练环境和你在应用程序中看到的结果。这样做最困难的部分是在训练期间从你关心的节点提取并控制输入。详细介绍如何执行此操作超出了本书的范畴，但是你需要确定与核心神经网络阶段相对应的算子（在文件解码、预处理之后，第一个接收预处理结果的算子）的名称。接收预处理结果的第一个算子对应于 `toco` 的 `--input_arrays` 参数。如果你可以识别这些算子，就能在 Python 的每个算子后面插入一个 `summarize` 设置为 `-1` 的 `tf.print` 操作。如果你运行训练循环，则可以在调试控制台的每个阶段看到张量内容的打印输出。

然后，你应该能够获取这些张量内容并将它们转换为 C 数据数组编译到你的程序中。`micro_speech` 代码中有一些示例，例如某人说“yes”的 1 秒音频样本，以及对该输入进行预处理后的预期结果。获得这些参考值之后，你应该能够将它们作为输入数据输入预处理流水线每个阶段（预处理、神经网络推断）的模块，并确保输出与你期望的输出匹配。如果时间不多，你可以使用一次性代码来完成此操作，但投入额外的精力将它们转换为单元测试是值得的，因为单元测试可以确保随着时间的推移，代码的变化仍可以不断被验证，以符合你对预处理和模型推断的预期。

18.1.3 在设备上评估

在训练结束时，使用一组测试输入对神经网络进行评估，并将预测结果与预期结果进行比较，以表征模型的整体准确率。这是训练过程中很常见的工作，但是很少会对已经部署在设备上的代码进行相同的评估。通常最大的障碍就是将构成典型测试数据集的数千个输入样本转移到资源有限的嵌入式系统上。然而，可惜的是，这是确保设备上已部署模型的准确率与训练环境中模型准确率一致的唯一方法，因为部署过程可能会引入各种微妙的错误，而且很难发现。我们没有为 `micro_speech` 演示实施完整的测试集评估，但是至少有一个端到端的测试（end-to-end test）可以确保我们为两个不同的输入获得正确的标签。

18.2 数值差异

神经网络是在大量的数字上执行的一系列复杂数学运算。原始训练通常在大规模浮点数组上完成，但是我们尝试将浮点数转换为嵌入式应用程序中精度较低的整数表示形式。算子本身也可以通过许多不同的方式实现，这取决于平台和优化的权衡。所有这些因素都意味着即使使用相同的输入，也无法期望相同的网络在不同设备上产生完全一样的结果。也就是说，你必须确定可以容忍的差异范围，以及如果差异太大应如何追溯各种差异的来源。

18.2.1 这样的数值差异是真是的问题吗

有时我们开玩笑说，唯一真正重要的指标就是应用商店的评分。我们的目标应该是打造

用户满意的产品，因此所有其他的指标都只是为了从各个角度反映用户满意度。由于使用环境与训练环境始终存在一定的数值差异，因此第一个挑战是了解它们是否真的会损害产品的体验。如果你从网络中获得的值是无意义的，那么这很明显会影响产品，但是如果它们仅与期望值相差几个百分点，就值得将具有实际用例的结果网络作为完整应用程序的一部分进行尝试。准确率损失可能不是大问题，也许还有其他更重要的问题应该被优先处理。

18.2.2 建立度量指标

当你确定你遇到了真正的问题时，用数据量化你 的问题会很有帮助。选择数字量度可能很诱人，例如输出分数向量与预期结果之间的百分比差异。但是这可能无法很好地反映用户体验。例如，如果你要进行图像分类，并且所有分数都比预期低 5%，但结果的相对顺序保持不变，那么对于许多应用程序而言，最终结果可能还不错。

相反，我们建议设计一个能够反映产品需求的度量指标。在图片分类的情况下，你可以在一组测试图片中选择所谓的 top-one 得分，因为这将显示模型选择正确标签的概率。top-one 指标表示模型选择标注数据（ground truth）作为最高得分预测的概率。top-five 与此类似，但是涵盖了标注数据在 5 个最高得分预测中的出现概率。然后，你可以使用 top-one 指标来跟踪进度，更重要的是，了解你所做的更改何时已经足够好。

你还应该注意收集一组标准的输入，以反映实际输入神经网络处理的内容，因为正如我们前面所讨论的，预处理中的很多方法都可能会引入错误。

18.2.3 与基线比较

TensorFlow Lite for Microcontrollers 旨在为其所有功能提供参考实现，而我们这样做的原因之一是，可以将其结果与优化后的代码进行比较，以调试潜在的差异。一旦有了一些标准输入，你就应该尝试在框架的桌面版本中运行它们，且不启用任何优化，以便调用参考算子的实现。如果你想要找到这种独立测试的起点，请查看 `micro_speech_test.cc`。如果你通过已建立的指标运行结果，那么你应该会看到预期的分数。如果不是这样，就可能预示着转换过程出现了一些错误，或者在工作流程的早期出现了其他问题，因此你需要重新调试以了解问题出在哪里。

如果使用参考代码确实得到了良好的结果，则应尝试在启用所有优化的目标平台上编译并运行相同的测试。当然，它可能没有那么简单，因为嵌入式设备通常没有存储所有输入数据的内存，而且如果只有调试日志记录连接，那么输出结果可能会很棘手。即使你必须将测试分成多次运行，这仍旧值得一试。得到结果后，可以通过度量指标来了解实际问题出在哪里。

18.2.4 交换实现

鉴于参考实现在嵌入式设备上的运行可能很慢，甚至难以运行，许多平台在默认情况下都会启用优化功能。禁用这些优化的方法有很多，但是我们发现最简单的方法通常是指到当前正在使用的所有 kernel 实现（通常在 `tensorflow/lite/micro/kernels` 的父目录中），然后用参考版本覆盖它们（请首先确保你拥有要被替换的文件的备份）。作为第一步，先替换所有的优化实现，然后重新在设备上运行测试，以确保你确实获得了比预期更好的分数。

完成大规模的批量替换之后，请尝试仅覆盖一半的优化 kernel，然后看看指标会受到什么程度的影响。在大多数情况下，你可以使用二分查找（binary search）方法来确定哪种优化的 kernel 实现方法导致准确率下降得最多。一旦你将范围缩小到特定的优化 kernel，就应该能够从其中一个错误运行中保留输入值和来自参考实现的预期值，来创建一个可重复利用的小用例。最简单的方法就是调试一次测试运行期间的 kernel 日志记录。

现在你有了一个可重现的用例，你应该能够通过它创建一个单元测试。你可以从参考一个标准的 kernel 测试开始，然后创建一个新的独立测试或将其添加到该 kernel 测试的现有文件中。这就为你提供了一个工具，可以使用它与负责优化实现的团队沟通你发现的问题，因为这能够帮你证明他们代码的结果和参考版本的结果有差异，而且影响了你的应用程序。也可以将同样的测试添加到主代码库中，以确保其他优化的实现不会导致类似的问题。这也会是帮助你自己调试实现的绝佳工具，因为你可以单独试验代码并快速迭代。

18.3 神秘的崩溃与挂起

嵌入式系统上最难修复的情况之一就是：程序没有运行，但是没有明显的日志记录或错误输出来说明出了什么问题。解决问题的最简单方法就是附加一个调试器（如 GDB），然后查看堆栈跟踪是否已挂起，或者单步执行代码以查找错误发生的地方。但是设置调试器并不总是那么容易，或者使用一个调试器后问题的根源可能仍然不清楚，因此你可以尝试一些其他方法。

18.3.1 桌面调试

像 Linux、macOS 和 Windows 之类的完整操作系统都具有广泛的调试工具和错误报告机制，因此，即使你必须使用某些特定于硬件的功能，也要尽可能使这些特定于硬件的部分变为伪实现，从而使程序可以被移植到这些平台之一。这就是 TensorFlow Lite for Microcontrollers 的设计方式，这意味着我们可以首先尝试在 Linux 机器上重现任何出错的内容。如果在桌面环境中出现了相同的错误，通常无须烧录设备，使用标准工具就可以轻松且快捷地跟踪和调试问题，从而加快迭代速度。即使维护完整的台式机版本的应

用程序太困难了，至少也要看看是否可以在台式机上编译部分模块，并创建单元测试和集成测试。然后，你可以尝试向这些模型提供与所遇到问题的情况类似的输入，并尝试在台式机上调试类似的错误是否可以重现。

18.3.2 日志跟踪

TensorFlow Lite for Microcontrollers 所需的唯一特定于平台的功能就是 `DebugLog()` 的实现。我们之所以有此要求，是因为它是了解开发过程中发生的事情的必不可少的工具，尽管编译和部署并不需要它。在理想的情况下，任何崩溃或程序错误都应触发日志输出——例如，我们对 STM32 设备的裸机支持具有执行此操作的错误处理程序——但这并不总是可行的。

不过，你应该始终能够自己将日志语句插入代码中。这些代码并不需要有意义，只需声明已到达代码中的什么位置即可。你甚至可以定义一个自动跟踪宏，如下所示：

```
#define TRACE DebugLog(__FILE__ ":" __LINE__)
```

然后在你的代码中像这样使用它：

```
int main(int argc, char**argv) {
    TRACE;
    InitSomething();
    TRACE;
    while (true) {
        TRACE;
        DoSomething();
        TRACE;
    }
}
```

你应该能够在调试控制台中看到输出，显示代码执行到了哪里。通常最好从最高层次的代码开始，然后查看日志记录的停止位置。这将帮助你大致了解崩溃或挂起发生的位置，然后可以添加更多 `TRACE` 语句来缩小问题发生的确切范围。

18.3.3 霹弹枪调试

有时候，跟踪并不能为你提供足够的信息以解决问题，或者问题可能只出现在你无法访问日志的环境中，例如产品环境。在这种情况下，我们建议你使用称为“霹弹枪调试”的方法。这类似于我们在第 15 章中介绍的“霹弹枪剖析”，注释掉部分代码并查看错误是否仍然发生。如果你从应用程序的顶层开始，然后逐步向下，通常可以执行类似二分查找的操作，以找出引起问题的代码行。例如，你可以在主循环中这样开始：

```
int main(int argc, char**argv) {
    InitSomething();
```

```
while (true) {
    // DoSomething();
}
}
```

如果在注释掉 `DoSomething()` 的情况下这个函数运行成功了，则说明问题出现在 `DoSomething()` 之中。然后，你可以取消注释它，并在其主体内的递归中执行相同操作，以将精力集中在那些行为异常的代码上。

18.3.4 内存损坏

由内存中的值被意外覆盖引起的错误是最令人痛苦的。嵌入式系统没有与台式机或移动 CPU 相同的硬件来防御这种情况，因此调试起来尤为困难。甚至跟踪或注释掉代码也可能产生令人困惑的结果，因为覆盖可能在使用损坏^{译注1} 的值的代码运行之前很久就发生了，因此崩溃的路径可能很长。它们甚至可能取决于传感器的输入或硬件时序，使得问题断断续续、极其难以复现。

根据我们的经验，造成此问题的第一大因素是程序栈（program stack）溢出。程序栈是存储局部变量的地方，而 TensorFlow Lite for Microcontrollers 使用大量程序栈来存储较大的对象。因此，与许多其他嵌入式应用程序相比，TensorFlow Lite for Microcontrollers 需要更多的空间。不幸的是，你很难确定确切的空间大小。通常最大的消耗者是需要传递给 `SimpleTensorAllocator` 的内存 arena，在以下示例中，`SimpleTensorAllocator` 被分配为本地数组：

```
// Create an area of memory to use for input, output, and intermediate arrays.
// The size of this will depend on the model you're using, and may need to be
// determined by experimentation.
const int tensor_arena_size = 10 * 1024;
uint8_t tensor_arena[tensor_arena_size];
tflite::SimpleTensorAllocator tensor_allocator(tensor_arena,
                                              tensor_arena_size);
```

如果你使用相同的方法，就需要确保栈的大小和 arena 的大小大致相同，再加上几 KB 以容纳运行时使用的其他变量。如果你的 arena 位于其他地方（可能是一个全局变量），那么你只需要几 KB 的栈。所需的确切内存量取决于架构、编译器和运行的模型。因此，不幸的是，要提前提供确切的值并不容易。如果你看到神秘的崩溃，那么值得尝试增加这个值，看看这样做是否有用。

如果仍然遇到问题，则应首先尝试确定要覆盖哪些变量或内存区域。希望这可以使用前面介绍的日志记录或代码消除的方法来实现，将问题范围缩小到似乎被损坏的值的读取上。一旦知道什么变量或数组条目被破坏，就可以编写对应的 TRACE 宏变体，使它输

译注 1：并非指物理意义上的损坏，而是内存内容污染。

出该内存位置的值、文件和行，并调试它。你可能需要使用一些特殊的技巧，例如将内存地址存储在全局变量中，以便从本地地址中访问更深的栈帧（stack frame）。然后，就像跟踪正常崩溃一样，可以在运行程序时跟踪该位置的内容，并尝试确定哪个代码负责覆盖它。

18.4 小结

当在训练环境中运行良好但在真实设备上无法正常工作时，寻找解决方案可能是一个漫长而令人沮丧的过程。在本章中，我们为你提供了一些工具和方法，当你发现自己陷入困境时，可以尝试这些方法。不幸的是，调试没有捷径可循，但是通过使用这些方法有条不紊地解决问题，我们相信你一定可以找出并解决嵌入式机器学习程序中的任何错误。

当在产品中使用一种模型后，你可能会开始考虑如何调整它，甚至创建一个全新的模型来解决不同的问题。第 19 章将讨论如何将自己的模型从 TensorFlow 训练环境迁移到 TensorFlow Lite 推断引擎中。

将模型从 TensorFlow 移植到 TensorFlow Lite

如果你已经走到这一步，那么你应该能够理解，我们建议在任何可能的情况下将现有模型重用于新的任务。从零开始训练一个全新的模型可能会花费大量的时间并进行大量的试验，通常，即使是专家也不可能在不尝试多种不同的原型之前就预测出最佳方法。创建新的模型架构的完整指南已超出本书的范畴，我们建议你在第 21 章中查找有关该主题的更多信息。但是，有些问题对于在资源受限的设备上的机器学习来说是通用的，比如处理一组受限的算子或预处理需求，因此，本章将提供针对这些问题的建议。

19.1 了解需要什么算子

由于本书作者在 Google 团队工作，所以本书专注于在 TensorFlow 中创建模型，但是即使是在一个框架内，也存在许多不同的创建模型的方法。如果查看语音命令训练脚本，你就会发现它正在直接使用核心 TensorFlow 算子作为构建块来构建模型，并需要手动运行训练循环。如今，这已经是一种非常老式的工作方式（这个脚本最初编写于 2017 年），而 TensorFlow 2.0 的现代示例很可能都在使用 Keras 这种更高级的 API 来处理很多细节。

然而，使用高级 API 的缺点是，模型底层使用的基础算子不再明显。取而代之的是，算子将作为层的一部分被创建，这些层在单个调用中表示模型计算图中更大的一个块。这可能会是潜在的问题，因为了解模型正在使用哪些 TensorFlow 算子对于理解这个模型能否在 TensorFlow Lite 中运行以及它的资源需求至关重要。幸运的是，只要你可以使用 `tf.keras.backend.get_session()` 检索基础的 `Session` 对象，你就可以获得 Keras 访问的基础底层算子。如果你直接在 TensorFlow 中进行编码，则可能已经在变量中包含了 `Session`，因此以下代码仍然可以使用：

```
for op in sess.graph.get_operations():
    print(op.type)
```

如果你已将会话赋值给变量 `sess`，就会打印出模型中所有算子的类型。你还可以访问它的其他属性，如 `name` 等，以获取更多信息。了解存在哪些 TensorFlow 算子会对向 TensorFlow Lite 的转换过程有很大帮助；否则，你看到的任何错误将更加难以理解。

19.2 查看 Tensorflow Lite 中支持的算子

TensorFlow Lite 仅支持 TensorFlow 的部分算子，并且有一些限制。你可以在算子兼容性指南（the ops compatibility guide）中查看最新支持算子的列表。这意味着，如果你正在计划尝试一个新的模型，那么你应该从一开始就确保你不会依赖于那些不被支持的功能或者算子。尤其是 LSTM、GRU 和其他递归神经网络目前都尚不支持。目前，完整的移动版 TensorFlow Lite 与微控制器分支之间也存在一定的差异。因为新的算子会不断被添加，所以了解 TensorFlow Lite for Microcontrollers 当前支持哪些算子的最简单方法是查看 `all_ops_resolver.cc`。

将 TensorFlow 训练会话中显示的算子与 TensorFlow Lite 支持的算子进行比较可能会令人困惑，因为在导出过程中会执行多个转换步骤。例如，这些方法将作为变量存储的权重被转换为常量，并且可能会有一定的优化将浮点运算量化为整数等效项。还有一些算子仅作为训练循环的一部分存在，例如与反向传播相关的算子，这些算子最终会被完全剔除。弄清楚你可能会遇到的问题的最好方法是在创建预期模型后，在训练之前就试着导出它，这样你可以在花费大量时间进行训练之前就调整它的架构。

19.3 将预处理和后处理移至应用程序代码

深度学习模型通常分为三个阶段。通常有一个预处理步骤，它可能很简单，例如从磁盘加载图像和标签并解码 JPEG，也可能会比较复杂，例如语音示例中将音频数据转换成频谱图。然后有一个核心神经网络，它接收数值数组并以类似的形式输出结果。最后，你需要在后处理步骤中理解这些值。对于许多分类问题，这就像将向量中的分数与相应的标签匹配一样简单，但是如果你查看类似 MobileSSD 这样的模型，其网络输出的是大量的重叠边界框，它们可能需要经历一个被称为“非最大值抑制”（non-max suppression）的复杂过程才能将输出转换为有用的结果。

核心神经网络模型通常是计算密集的，并且通常由数量相对较少的运算（例如卷积和激活）组成。而预处理阶段和后处理阶段经常需要执行更多算子和控制流，即使它们的计算量要低得多。这意味着将非核心步骤实现为应用程序中的常规代码通常比将它们放入 TensorFlow Lite 模型中更有意义。例如，机器视觉模型的神经网络部分将接收特定大小

的图像，例如 224 像素高乘 224 像素宽。在训练环境中，我们通常使用 `DecodeJpeg` 算子和 `ResizeImages` 算子将结果转换为需要的大小。但是，在设备上运行时，几乎可以肯定是从固定大小的源中获取输入图像，而且无须进行解压缩，因此编写自定义代码来创建神经网络输入比依赖于库中的通用算子更有意义。我们可能还会处理异步捕获，并且可能可以通过将算子多线程化获得额外的好处。对于语音命令，我们对流输入做了很多工作来缓存 FFT 的中间结果，以便尽可能地重复利用运算结果。

并非每个模型在训练过程中都有明显的后处理阶段，但是在设备上运行时，当我们通常希望利用时间的一致性来改善显示给用户的结果。在唤醒词检测代码中，虽然模型只是一个分类器，但是程序还是通过每秒运行多次然后求平均值来提高结果的准确率。这种代码也最好在应用程序级别实现，因为将其表示为 TensorFlow Lite 算子很困难而且没有什么显著的收益。你可以在 `detection_postprocess.cc` 中看到，通过 TensorFloat Lite 算子实现是可能的，但是在导出过程中，它会涉及大量有关 TensorFlow 计算图的基础工作，因为通常在 TensorFlow 中将其表示为小型算子并不是一种在设备上实现它的有效方式。

所有这些都意味着你应该尝试排除模型计算图中的非核心部分，这将需要做一些工作来确定哪些部分是核心的，哪些不是。我们发现 Netron 是探索 TensorFlow Lite 计算图的很好的工具，它可以帮助我们理解当前的算子，并判断它们是神经网络核心的一部分，还是仅仅是预处理或者后处理的步骤。一旦了解了内部情况，你就应该能够隔离核心网络，导出必要的算子，并将其余算子实现为应用程序代码。

19.4 按需自己实现算子

如果你确实发现某些你绝对需要的 TensorFlow 算子是 TensorFlow Lite 不支持的，那么可以将它们作为自定义算子保存为 TensorFlow Lite 文件格式，然后在框架中自己实现它们。完整的实现过程超出了本书的范畴，以下是一些关键步骤：

- 在启用 `allow_custom_ops` 的情况下运行 `toco`，以便将不被支持的算子作为自定义算子存储在序列化的模型文件中。
- 编写实现这个算子的 kernel，并通过你的应用程序中使用的算子解析器的 `AddCustom()` 注册它。
- 调用 `Init()` 方法时，解压缩以 FlexBuffer 格式存储的参数。

19.5 优化算子

即使你在新模型中使用了 TensorFlow Lite 支持的算子，你也可能正在以尚未被优化的方式使用它们。TensorFlow Lite 团队的优先级是由特定的用例驱动的，因此，如果你正

在运行一个新的模型，则可能会遇到尚未被优化的代码路径。我们在第 15 章中已经介绍了这一点，但是正如我们建议的，你应该尽快检查导出兼容性——甚至在训练模型之前——以确保在制定开发计划之前获得所需的性能，因为你可能会需要一些时间预算来处理算子延迟。

19.6 小结

成功地训练一个新的神经网络以完成某项任务已经是一个挑战了，但是弄清楚如何创建一个可以产生良好结果的网络并在嵌入式硬件上有效地运行它更加困难！本章讨论了你将面临的一些挑战，并提出了一些建议方法来帮助你克服这些挑战，但这是一个庞大且不断发展的研究领域，因此我们建议你浏览第 21 章推荐的一些资源，你可能会在其中发现新的灵感来优化你正在使用的网络架构。特别是对于嵌入式机器学习领域，关注 arXiv^{译注 1} 的最新研究论文是非常有用的。

在克服了所有这些挑战之后，你应该拥有了一款小型、快速、高能效的产品，可以将其部署到现实世界中了。但是在产品发布之前，有必要考虑一下潜在的、可能对用户造成的有害影响，因此，第 20 章将讨论关于隐私和安全性的问题。

译注 1：arXiv 是一个收集物理学、数学、计算机科学与生物学论文开放预印出版物的网站。大多数深度学习研究人员在完成他们的论文后会立刻上传到 arXiv，这样无须等待可能会持续几个月的会议接收结果，就可以宣称某项研究想法的所有权。arXiv 已经成为了解机器学习最新进展的重要途径。

隐私、安全和部署

在阅读完本书前面的章节之后，你应该已经能够构建一个依赖机器学习的嵌入式应用程序。不过，你仍然需要克服许多挑战，才能将项目转化为可以被成功部署到世界各地的产品。保护用户的隐私和安全是两个关键的挑战。本章将介绍一些我们发现的有助于克服这些挑战的方法。

20.1 隐私

设备上的机器学习依赖于传感器输入。其中一些传感器，例如麦克风和摄像头，会带来明显的隐私问题。但其他一些传感器，例如加速度传感器，虽然隐私问题不明显，但也可能会被滥用，例如，在可穿戴产品中根据步态识别用户。作为工程师，我们所有人都有责任保护用户免于遭受产品可能造成的伤害，因此在设计的各个阶段考虑隐私问题是至关重要的。处理敏感的用户数据也会涉及法律问题，但这些超出了本书的范畴，具体应咨询律师。如果你是一个大型组织的成员，你可能会有隐私专家和专业流程帮助你获取领域知识。即使你没有这些资源，也应该在项目开始时花一些时间进行自己的隐私审查，并定期重新审查，直至项目启动。关于什么是“隐私审查”（privacy review），目前尚未达成广泛共识，但是我们讨论了一些最佳实践，其中大部分围绕如何构建强大的隐私设计文档（Privacy Design Document, PDD）。

20.1.1 隐私设计文档

隐私工程（privacy engineering）领域仍然是一个非常新的领域，很难找到有关如何解决产品隐私问题的文档。许多大公司解决其应用程序中的隐私问题的方式就是创建隐私设计文档。在隐私设计文档里，你可以涵盖产品重要的隐私内容。你的文档应包含在随后小节中将提出的所有相关主题的信息。

数据收集

PDD 的第一部分应涵盖你将会收集哪些数据、如何收集以及为什么要收集这些数据。你

应该使用尽可能具体、简单的语言，例如“收集温度和湿度”而不是“获取环境大气信息”。在进行本部分的工作时，你也有机会去思考实际在收集哪些数据，并确保这是你产品所需的最低限度。如果你只是在收集较大的声音以唤醒更复杂的设备，你是否真的需要使用麦克风以 16KHz 的频率采样音频？还是可以使用更粗糙的传感器？使用更粗糙的传感器还可以确保即使存在安全漏洞，你的设备也无法完备地录制语音。在编写这部分的时候，一个简单的系统图可能很有用，它显示了信息如何在产品的不同组件（包括任何云 API）之间流动。数据收集部分的总体目标是为非技术性受众（无论是律师、高管还是董事会成员）提供一个简单的有关收集的内容的全面概述。一种思考方式是，想象一个“无情的”记者在报纸首页上会怎样描述你收集的数据。确保你已尽一切努力将用户遭受他人恶意侵害的风险降到最低。具体来说，请仔细考虑以下场景：一个有虐待倾向的前合伙人可以使用这项技术来做些什么？请尽可能地发挥想象力，以确保你能为用户提供更多的保护。

数据使用

收集数据后，如何处理这些数据？例如，许多初创公司倾向于利用用户数据来训练其机器学习模型。但是从隐私的角度来看，这是一个非常令人担忧的过程，因为它需要长时间存储和处理用户数据甚至潜在敏感信息，而这只能以非常间接的方式为用户提供好处。我们强烈建议将获取训练数据作为一个完全独立的项目，使用付费提供商提供的、在获得明确同意的情况下收集的数据，而不是将收集数据作为使用产品的副作用。

基于终端设备的机器学习的一个好处是，你能够在本地处理敏感数据而且只共享聚合的 (aggregated) 结果。例如，你可能有一个行人计数设备，它每秒捕获一次图像，但是向外传输的唯一数据是看到的人和车辆的数量。如果可以，请尝试对硬件进行工程设计，以确保这些保证永远不会被违背。如果你只使用 224 像素 × 224 像素的图像作为分类算法的输入，那么请使用分辨率较低的摄像头传感器，这样以来就无法在物理上识别人脸或车牌。如果你只打算传输几个值（如行人计数）作为摘要，请使用低比特率的无线技术，以保证即使你的设备被黑客入侵也无法传输源视频。我们希望在将来，专用性硬件 (special-purpose hardware) 将有助于强制执行这些保证；但是即使在现在，你仍然可以在系统设计级别上做很多事情，以避免过度设计并增加滥用难度。

数据分享和存储

谁有权访问你收集的数据？哪些系统可以确保只有这些人才能访问数据？数据会在设备上或者在云端保存多长时间？如果它被保留了一段时间，删除它的策略是什么？你可能会认为，存储除去明显用户 ID（例如电子邮件地址或名称）的信息是安全的，但是身份信息可以从许多来源获得，例如 IP 地址、可识别的语音甚至是步态，因此应假定你收集到的任何传感器数据都是个人身份信息（Personally Identifiable Information, PII）。最好

的策略就是像对待放射性废物一样对待 PII。如果可能的话，请尽量避免收集 PII，在需要时请妥善保管，并在使用完之后尽快处理。

当你在考虑谁有数据访问权限的时候，请不要忘记你的所有许可系统都可能迫于政府的压力被重写，尤其是在专制国家这可能会给你的用户造成严重伤害。这也是有必要将传输和存储的内容限制在最小范围内的另一个重要原因——避免这种责任并限制用户的暴露范围。

同意书

用户是否知晓了你的产品会收集什么数据？他们是否同意你使用这些数据的方式？你可能认为这是一个简单的法律问题，通过点击终端用户许可协议（end-user license agreement）就可以解决，但是我们强烈建议你将其视为一个营销挑战来更广泛地思考。想必你自己相信，你的产品所带来的好处值得用户权衡它可能会收集更多数据的弊端，但是你如何才能将这一点清晰地传达给潜在客户，以便他们和你一样相信这一点并做出明智的选择？如果无法清晰地表达这一点，那么这表明你需要重新思考你的设计，或者减少隐私问题带来的影响，或者增加使用产品的收益。

20.1.2 使用隐私设计文档

你应该将 PDD 视为一个活的文档，并随着产品的发展而不断地更新它。在与你的律师和其他业务利益相关者交流产品细节时，详细的产品信息显然是有助于交流的，这一点在许多其他情况下也同样有用。例如，你应该与你的营销团队合作，以确保营销团队传递的消息是根据你的实现来进行的，还应该与任何第三方服务提供商（例如广告商）合作，以确保它们也都能遵守你的承诺。鉴于可能存在一些隐私隐患，而且这些隐患可能只在代码实现级别可见，因此团队中的所有工程师都应该可以访问 PDD 并能够添加评论。例如，你可能使用的是基于地理编码的云 API，这个 API 可能会泄露设备的 IP 地址；或者你的微控制器上可能有一个 WiFi 芯片，虽然你并没有使用它，但从理论上讲可以启用 WiFi 来传输敏感数据。

20.2 安全

确保嵌入式设备的总体安全是非常困难的。攻击者可以轻松地获得物理系统并占为己有，然后使用各种侵入性技术来提取信息。你的第一道防线就是要确保在嵌入式系统上保留尽可能少的敏感信息，这也是 PDD 如此重要的原因。如果你依赖于与云服务的安全通信，那么你应该考虑研究安全的密码处理器（secure cryptoprocessor），以确保所有密钥都会被安全保存。这些芯片也可以被用于安全启动，以确保只有你烧录的程序才能在设备上运行。

就隐私而言，你应尝试精心设计硬件，以减少任何可能的攻击机会。如果你不需要 WiFi

或蓝牙，就构建一个不具备 WiFi 和蓝牙的设备。不要在发布产品时提供诸如 SWD 之类的调试接口，并且要考虑在 Arm 平台上禁用代码读取功能。尽管这些措施并不完美，但它们将大大提高攻击的成本。

你还应该尝试依靠已建立的库和服务来实现安全性和加密。推出自己的加密技术是一个非常糟糕的主意，因为它很容易引入一些难以发现的漏洞，从而破坏系统的安全性。嵌入式系统安全性的全部挑战超出了本书的范畴，但是你应该考虑创建一个安全设计文档 (security design document)，类似于我们为隐私推荐的文档。该文档应该涵盖你认为产品可能会遭受的攻击有什么、它们的影响如何以及如何防御这些攻击。

保护模型

我们经常听到一些工程师说，他们担心自己的机器学习模型会受到肆无忌惮的竞争者的侵害。因为工程师需要进行大量工作才能创建这些模型，而这些模型往往是在附带在设备上的，并且通常采用一种通用的易于理解的格式发布。坏消息是，没有绝对的保护措施可以防止模型被复制。从这个意义上讲，模型就像任何其他软件一样：它们可以同任何普通的机器代码一样可以被盗用和检查。不过，与软件一样，这个问题并不像乍看起来那么严重。就像分解过程性程序并不能揭示真实的源代码一样，检查量化的模型也不能提供对训练算法或者数据的任何访问权限，因此攻击者并不能有效地修改模型以用于其他用途。如果竞争对手直接将模型复制到自己的设备上，那么应该很容易发现这是直接拷贝的副本，也就很容易在法律上证明竞争对手窃取了你的知识产权，就像侵权使用其他任何软件一样。

当然，让攻击者更难访问你的模型仍然是值得的。一种简单的技术是，将序列化的模型与私钥进行异或 (XOR) 运算后再存储在闪存中，然后将其复制到 RAM 中并在使用前对其进行解密。这样可以防止简单的闪存转储泄露你的模型，但是攻击者仍然可以在运行时通过访问 RAM 访问它。你可能会认为将 TensorFlow Lite FlatBuffer 转换为专有格式会有所帮助，然而我们发现这种混淆带来的益处非常有限：由于权重参数本身是大型数值数组，从逐步调试程序中还是可以明显地看出有哪些算子以及以什么顺序进行调用。



一种用于发现模型被盗用的有趣方法是在训练过程中故意引入细微的缺陷，然后在检查可疑侵权行为时寻找这些缺陷。例如，你可以训练唤醒词检测模型，使其不仅可以监听 “Hello”，还可以秘密监听 “Ahoy, Sailor！” 。另外的一个独立训练的模型不太可能会对这个奇怪短语做出响应，因此，如果它能响应的话，这就是一个强烈的信号，表明你的模型可能被复制了，即使你无法访问设备的内部工作原理。这种技术基于一种古老的思想，即在地图、目录和词典等参考作品中加入一个虚构的条目，以帮助发现侵权行为。有人在地图上标注了一座虚构的山峰 Mountweazel，因此这种技术也被称为 mountweazeling。

20.3 部署

借助现代微控制器，很容易实现无线更新，因此即使在产品交付很长时间之后，你仍旧可以随时修改设备上正在运行的代码。这为违反安全和隐私保护的行为开辟了广阔的攻击面，我们建议你认真考虑无线更新是否真的有必要。如果没有精心设计的安全启动系统和其他的保护措施，很难确保只有你才能上传新的代码，并且如果你犯了任何错误，就很可能会将设备的完全控制权交给恶意攻击者。默认情况下，我们建议你在设备制造完成后不允许进行任何形式的代码更新。这听起来可能有些苛刻，因为它会阻止修复安全漏洞的更新，但是在几乎所有情况下，消除攻击者的代码在系统上运行的可能性所带来的帮助都将远远大于它所带来的伤害。这也会大大简化网络架构，因为不再需要任何协议来“监听”更新。设备可以在仅开启传输模式的情况下有效地运行，这也大大减少了攻击面。

这确实意味着在设备发布之前，你有更多的责任进行测试以确保发布完全正确的代码，尤其是在模型的准确率方面。前面我们讨论了诸如单元测试和通过专用测试集验证整体模型准确率之类的方法，但是它们并不能解决所有问题。当你准备发布一个版本时，我们强烈建议你使用 dog-fooding^{译注1} 方法，在实际环境中且在内部人员的监督下尝试使用这些设备。与工程测试相比，这些试验更有可能揭示意想不到的行为，因为测试通常会受到创建者想象力的限制，而现实世界却比我们任何人预测的都令人称奇。好消息是，在遇到了没有被测试覆盖的错误行为之后，你可以将它们转变为测试用例，然后这些问题就可以作为正常开发过程的一部分来解决。实际上，开发这种对产品的深层需求的制度化记忆并将其编入测试之中，可能会成为你最大的竞争优势之一，而获得它们的唯一途径就是痛苦地反复试错。

从开发板到产品

将开发板上运行的应用程序转换为可交付的产品的完整过程超出了本书的范畴，但是在开发过程中仍有一些事情值得考虑。你应该研究你正在考虑使用的微控制器的批发价格——例如在 Digi-Key 之类的网站上——以确保你的最终系统符合预算。假设你在代码开发期间使用的是同一种芯片，那么将代码移至生产设备上应该非常简单，因此从编程角度来看，主要的任务是确保你的开发板与生产目标相匹配。在代码以最终形式被部署之后，调试任何新出现的问题都会变得更加困难，尤其是如果你已按照前面所述的步骤采取了一定的平台保护措施，因此尽可能地推迟这一步是有必要的。

译注1：dog-fooding 源于俚语 “Eat your own dog food”，意思是“吃你自己家的狗粮”，指邀请公司员工参与到测试中并提供反馈和建议。

20.4 小结

保护用户的隐私和安全是工程师最重要的职责之一，但是如何找到最佳的保护方法却并不清晰。在本章中，我们介绍了隐私安全保护的基本思考和设计过程，以及一些更高级的安全注意事项。有了这些，我们就完成了构建和部署嵌入式机器学习应用程序的基础，但是我们知道，这一领域的内容远远超出了本书所能涵盖的范围。作为收尾，最后一章将介绍一些可以用来继续学习和探索的资源。

了解更多

我们希望本书可以帮助你使用便宜、低功耗的设备来解决一些重要的问题。这是一个新兴的且正在快速发展的领域，因此本书中的所有介绍只是这个领域的快照。如果你想了解更多最新的知识，以下是一些我们推荐的资源。

21.1 TinyML 基金会

TinyML 峰会（TinyML Summit）是一个年度会议，汇集了来自嵌入式硬件、软件和机器学习方向的从业者，他们一起讨论这些学科之间的协作。在美国旧金山湾区（Bay Area）和得克萨斯州奥斯汀市（Austin, TX）也有每月一次的会议，预计将来还会有更多的会议地点。如果不能亲自参加会议，你可以访问 TinyML 基金会（TinyML Foundation）网站以获取相关活动的视频、幻灯片和其他资料。

21.2 SIG Micro

本书主要介绍了 TensorFlow Lite for Microcontrollers，如果你有兴趣为这个框架做出贡献，你可以加入一个特别兴趣小组（Special Interest Group, SIG），这个小组使得外部开发人员可以协作以改进框架。SIG Micro 每月举行一次公共视频会议，它有一个邮件列表和一个 Gitter 聊天室。如果你对库中的新功能有任何想法或者有新的需求，那么这里是讨论它的好地方。你会看到所有在 Google 内部和外部从事该项目的开发人员，他们会共享路线图和未来工作计划。对项目进行的任何更改通常都是从共享设计文档开始的，设计文档可以是简单的一页更改说明，包括为什么需要进行更改以及你的更改将要做些什么。我们通常以“征求修正意见书”（Request For Comment, RFC）的形式发布文档，以允许团队成员提供反馈，然后在方案达成一致后，提出包含实际代码更改的请求。

21.3 TensorFlow 网站

TensorFlow 主网站上有一个关于我们在微控制器上的项目的主页，你可以在那里查看最新的示例和文档。特别是，我们将在训练示例代码中继续向 TensorFlow 2.0 迁移，因此如果你遇到任何兼容性问题，值得去网站上看一看有没有相关的更新。

21.4 其他框架

我们主要介绍了 TensorFlow 生态系统，因为这是我们最熟悉的库，但是在其他框架上也有很多有趣的工作。我们是 Neil Tan 的忠实粉丝，他在 uTensor 上做了很多开创性的工作，通过 TensorFlow 模型生成代码进行了许多有趣的试验。微软的 Embedded Learning Library 针对 Arduino 和 micro:bit 平台，不仅支持深度神经网络，还支持多种不同的机器学习算法。

21.5 Twitter

你是否已经建立了一个嵌入式机器学习项目，而且想要向全世界分享？我们很想知道你在解决什么问题，而与我们联系的一个好方法就是在 Twitter 上分享带有 `#tinyml` 主题标签的链接。我们自己在 Twitter 上的账号分别是 @petewarden 和 @dansitu，我们将在 @tinymlbook 上发布有关本书的更新。

21.6 TinyML 的朋友们

在这一领域有很多有趣的公司，从早期的初创公司到大型公司都开展了各式各样的工作。如果你正在开发一个产品，你很可能对这些公司能提供什么感兴趣，以下是与我们合作过的一些公司，按字母顺序排列：

- Adafruit
- Ambiq Micro
- Arduino
- Arm
- Cadence/Tensilica
- CEVA/DSP Group
- Edge Impulse
- Eta Compute

- Everactive
- GreenWaves Technologies
- Himax
- MATRIX Industries
- Nordic Semiconductor
- PixArt
- Qualcomm
- SparkFun
- STMicroelectronics
- Syntiant
- Xnor.ai

21.7 小结

感谢你和我们一同探索在嵌入式设备上应用机器学习。希望我们已经激励你开始开展自己的项目，而且我们迫不及待地想看看你做了些什么，以及如何推动这个令人兴奋的新领域向前发展！

使用和生成 Arduino 库 ZIP 文件

Arduino IDE 要求以某种方式打包源文件。TensorFlow Lite for Microcontrollers Makefile 知道如何做到这一点，并且可以生成一个包含所有源代码的 .zip 文件，你可以将其作为库导入 Arduino IDE。这将允许你编译和部署应用程序。

有关生成此文件的说明将在本附录后面介绍。但是，最简单的入门方法是使用由 TensorFlow 团队每晚生成的预编译的 .zip 文件。

下载该文件后，需要将其导入。在 Arduino IDE 的“Sketch”（项目）菜单中，选择“Include Library”（加载库）→“Add .ZIP Library”（添加 .ZIP 库），如图 A-1 所示。

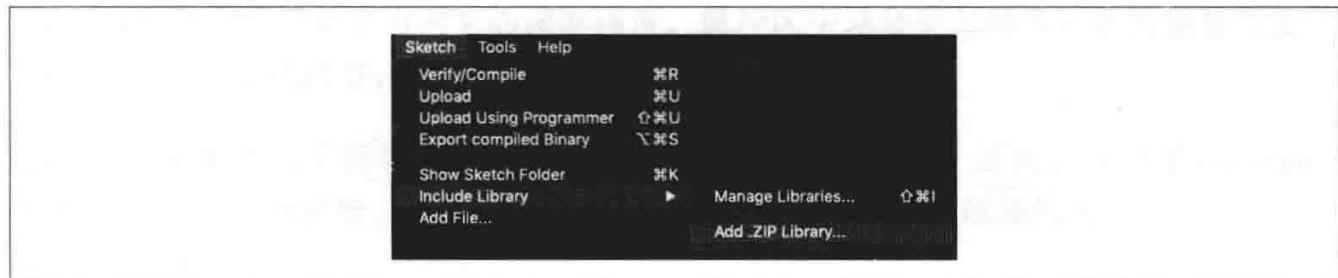


图 A-1：“Add.ZIP Library...”菜单选项

在出现的文件浏览器中，找到 .zip 文件，然后单击“Choose”（选择）以将其导入。

然而，你可能想自己生成这个库——例如，你对 TensorFlow Git 代码仓库中的代码进行了修改，并且想在 Arduino 环境中对其进行测试。

如果你需要自己生成文件，请打开终端窗口，复制 TensorFlow 代码仓库，然后跳转到其目录：

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

现在运行以下脚本来生成 .zip 文件：

```
tensorflow/lite/micro/tools/ci_build/test_arduino.sh
```

文件将被创建到这个地方：

```
tensorflow/lite/micro/tools/make/gen/arduino_x86_64/ \
prj/micro_speech/tensorflow_lite.zip
```

然后，你可以使用前面介绍的步骤将此 .zip 文件导入 Arduino IDE。如果你以前已经安装过这个库，那么你需要先删除原始版本。你可以通过从 Arduino IDE 的 *libraries* 目录中删除 *tensorflow_lite* 目录来做到这一点，你可以在 IDE 的“Preferences”（首选项）窗口的“Sketchbook location”（项目文件夹位置）中找到原始版本的路径。

在 Arduino 上捕获音频

本附录将介绍第 7 章中使用的唤醒词检测应用程序的音频捕获代码。由于它与机器学习没有直接关系，所以我们将它作为附录提供。

Arduino Nano 33 BLE Sense 有一个内置麦克风。要从麦克风接收音频数据，我们可以注册一个回调函数，当有大量新的音频数据准备就绪时，这个函数就会被调用。

每次发生这种情况时，我们都会将新的数据块写入一个用于存储数据的缓冲区。由于音频数据会占用大量内存，因此缓冲区只能容纳一定数量的数据。当缓冲区被填满时，缓冲区内的数据将会被新的数据覆盖。

每当我们的程序准备运行推断时，它就会从缓冲区读取最后一秒的数据。只要新数据的写入速度快于我们需要访问它的读取速度，缓冲区中总会有足够多的新数据被预处理，然后被输入我们的模型。

预处理和推断的每个循环都很复杂，需要一些时间才能完成。因此，我们在 Arduino 上每秒只能运行几次推断。这意味着我们的缓冲区很容易保持被填满的状态。

正如我们在第 7 章中看到的，*audio_provider.h* 实现了以下两个函数：

- `GetAudioSamples()`，它提供指向原始音频数据块的指针
- `LatestAudioTimestamp()`，它返回最新捕获的音频的时间戳

为 Arduino 实现这些函数的代码位于 *arduino/audio_provider.cc*。

在第一部分，我们引入了一些依赖项。*PDM.h* 库定义了我们将用于从麦克风获取数据的 API。*micro_model_settings.h* 包含与模型的数据要求相关的常量，这些常量将帮助我们以正确的格式读取音频：

```
#include "tensorflow/lite/micro/examples/micro_speech/
    audio_provider.h"
```

```
#include "PDM.h"
#include "tensorflow/lite/micro/examples/micro_speech/
    micro_features/micro_model_settings.h"
```

下一段代码设置了一些重要的变量：

```
namespace {
bool g_is_audio_initialized = false;
// An internal buffer able to fit 16x our sample size
constexpr int kAudioCaptureBufferSize = DEFAULT_PDM_BUFFER_SIZE * 16;
int16_t g_audio_capture_buffer[kAudioCaptureBufferSize];
// A buffer that holds our output
int16_t g_audio_output_buffer[kMaxAudioSampleSize];
// Mark as volatile so we can check in a while loop to see if
// any samples have arrived yet.
volatile int32_t g_latest_audio_timestamp = 0;
} // namespace
```

我们使用布尔值 `g_is_audio_initialized` 来跟踪麦克风是否已开始捕获音频。我们的音频捕获缓冲区由 `g_audio_capture_buffer` 定义，其大小是 `DEFAULT_PDM_BUFFER_SIZE` 的 16 倍，`DEFAULT_PDM_BUFFER_SIZE` 是 `PDM.h` 中定义的常量，表示每次调用回调时从麦克风接收的音频量。拥有一个不错的大缓冲区意味着如果程序由于某种原因变慢，那么我们不太可能耗尽数据。

除了音频捕获缓冲区之外，我们还有一个输出音频的缓冲区 `g_audio_output_buffer`，当调用 `GetAudioSamples()` 时，它将返回一个指针，指向缓冲区对象。它的长度为 `kMaxAudioSampleSize`，这是 `micro_model_settings.h` 中的一个常量，定义了可以被预处理代码立即处理的 16 位音频样本的数量。

最后，我们使用 `g_latest_audio_timestamp` 来跟踪最新音频样本所表示的时间。这和你手表上的时间并不吻合，它只是相对于音频捕获开始时间的毫秒数。这个变量被声明为 `volatile`，这意味着处理器不应尝试缓存它的值。我们将在稍后的部分说明原因。

设置完这些变量后，我们定义了每次有新的音频数据可用时都会被调用的回调函数。以下是具体代码：

```
void CaptureSamples() {
    // This is how many bytes of new data we have each time this is called
    const int number_of_samples = DEFAULT_PDM_BUFFER_SIZE;
    // Calculate what timestamp the last audio sample represents
    const int32_t time_in_ms =
        g_latest_audio_timestamp +
        (number_of_samples / (kAudioSampleFrequency / 1000));
    // Determine the index, in the history of all samples, of the last sample
    const int32_t start_sample_offset =
        g_latest_audio_timestamp * (kAudioSampleFrequency / 1000);
```

```

// Determine the index of this sample in our ring buffer
const int capture_index = start_sample_offset % kAudioCaptureBufferSize;
// Read the data to the correct place in our buffer
PDM.read(g_audio_capture_buffer + capture_index, DEFAULT_PDM_BUFFER_SIZE);
// This is how we let the outside world know that new audio data has arrived.
g_latest_audio_timestamp = time_in_ms;
}

```

这个函数有点复杂，因此我们将逐步介绍它。它的目标是确定音频捕获缓冲区中的正确索引，以便将索引对应的新数据写入其中。

首先，我们计算每次调用回调将收到多少新数据。我们用它来确定一个数字（以毫秒为单位），该数字表示缓冲区中最新音频样本的时间：

```

// This is how many bytes of new data we have each time this is called
const int number_of_samples = DEFAULT_PDM_BUFFER_SIZE;
// Calculate what timestamp the last audio sample represents
const int32_t time_in_ms =
    g_latest_audio_timestamp +
    (number_of_samples / (kAudioSampleFrequency / 1000));

```

每秒的音频采样数量为 `kAudioSampleFrequency`（这个常量在 `micro_model_settings.h` 文件中定义）。我们将 `kAudioSampleFrequency` 除以 1000，就会得到每毫秒的样本数。

接下来，我们将每个回调的样本数（`number_of_samples`）除以每毫秒的样本数，得出每个回调获得的数据的毫秒数：

```
(number_of_samples / (kAudioSampleFrequency / 1000))
```

然后将这个值添加到之前的最新音频样本的时间戳 `g_latest_audio_timestamp` 中，以得到最新音频样本的时间戳。

有了这个数字，我们就可以使用它来获取所有样本历史记录中最新样本的索引。为此，我们将之前的最新音频样本的时间戳乘每毫秒的样本数：

```

const int32_t start_sample_offset =
    g_latest_audio_timestamp * (kAudioSampleFrequency / 1000);

```

但是，我们的缓冲区没有空间来存储所有已捕获的样本。相反，它的空间是 `DEFAULT_PDM_BUFFER_SIZE` 的 16 倍。一旦有更多的数据，我们就开始用新数据覆盖缓冲区。

现在，我们有了新的样本在所有历史样本中的索引。接下来，我们需要将其转换为实际缓冲区中的样本的正确索引。为此，我们可以将历史索引除以缓冲区长度，然后得到余数。这是通过使用模运算符（%）实现的：

```

// Determine the index of this sample in our ring buffer
const int capture_index = start_sample_offset % kAudioCaptureBufferSize;

```

由于缓冲区的大小 `kAudioCaptureBufferSize` 是 `DEFAULT_PDM_BUFFER_SIZE` 的倍数，新数据将始终被整齐地放入缓冲区。模运算符将返回相对于新数据开始的缓冲区索引。

接下来，我们使用 `PDM.read()` 方法将最新的音频读取到音频捕获缓冲区：

```
// Read the data to the correct place in our buffer  
PDM.read(g_audio_capture_buffer + capture_index, DEFAULT_PDM_BUFFER_SIZE);
```

第一个参数接收一个指向应该写入数据的内存位置的指针。变量 `g_audio_capture_buffer` 是指向内存中音频捕获缓冲区开始地址的指针。通过将 `capture_index` 添加到这个位置，我们可以计算出内存中的新数据要被写入的正确位置。第二个参数定义应读取的数据量，我们取最大值 `DEFAULT_PDM_BUFFER_SIZE`。

最后，我们更新 `g_latest_audio_timestamp`：

```
// This is how we let the outside world know that new audio data has arrived.  
g_latest_audio_timestamp = time_in_ms;
```

更新的时间戳将 `LatestAudioTimestamp()` 方法暴露给程序的其他部分，这可以让其他代码知道何时有新数据可用。因为 `g_latest_audio_timestamp` 被声明为 `volatile`，所以每次访问它时，都会从内存中查找它的值。这一点很重要，因为如果不是从内存中读取，那么变量会被处理器缓存。因为它的值是在回调中设置的，所以处理器并不知道要刷新缓存的值，而且任何访问它的代码都不会读到它的当前值。

你可能想知道是什么使得 `CaptureSamples()` 可以充当回调函数。如何知道何时有新音频可用？我们的代码的下一部分将处理这个问题，这是一个启动音频捕获的函数：

```
TfLiteStatus InitAudioRecording(tflite::ErrorReporter* error_reporter) {  
    // Hook up the callback that will be called with each sample  
    PDM.onReceive(CaptureSamples);  
    // Start listening for audio: MONO @ 16KHz with gain at 20  
    PDM.begin(1, kAudioSampleFrequency);  
    PDM.setGain(20);  
    // Block until we have our first audio sample  
    while (!g_latest_audio_timestamp) {  
    }  
  
    return kTfLiteOk;  
}
```

这个函数将在第一次调用 `GetAudioSamples()` 时被调用。它首先使用 PDM 库通过调用 `PDM.onReceive()` 来连接 `CaptureSamples()` 回调。接下来，使用两个参数调用 `PDM.begin()`。第一个参数表示需要录制多少个音频通道，由于我们只想要单声道音频，所以将第一个参数指定为 1。第二个参数表示我们希望每秒接收多少个样本。

接下来，使用 `PDM.setGain()` 来配置增益（gain），这个增益定义了麦克风的音频应该被放大多少倍。我们指定增益为 20，这是经过一些试验后选定的。

最后，进行循环直到 `g_latest_audio_timestamp` 的值为 `true`。因为它是从 0 开始的，所以在回调捕获音频之前，它会一直阻塞执行，直到 `g_latest_audio_timestamp` 变为非零值。

我们刚刚讨论的两个函数允许我们启动捕获音频的过程，并将捕获的音频存储在缓冲区中。下一个函数 `GetAudioSamples()` 为代码的其他部分（即功能提供程序）提供了一种获取音频数据的机制：

```
TfLiteStatus GetAudioSamples(tflite::ErrorReporter* error_reporter,
                            int start_ms, int duration_ms,
                            int* audio_samples_size, int16_t** audio_samples) {
    // Set everything up to start receiving audio
    if (!g_is_audio_initialized) {
        TfLiteStatus init_status = InitAudioRecording(error_reporter);
        if (init_status != kTfLiteOk) {
            return init_status;
        }
        g_is_audio_initialized = true;
    }
}
```

该函数由 `ErrorReporter` 调用，用于写入日志、两个变量 (`start_ms` 和 `duration_ms`) 以及两个用于传递音频数据的指针 (`audio_samples_size` 和 `audio_samples`)。函数的第一部分调用 `InitAudioRecording()`。正如我们在前面看到的，这将阻止执行直到音频的第一个样本到达函数。我们使用变量 `g_is_audio_initialized` 来确保这段代码运行且仅运行一次。

此后，我们可以假设捕获缓冲区中存储了一些音频。我们的任务是找出缓冲区中正确的音频数据的位置。为此，我们首先确定我们想要的第一个样本在所有历史样本中的索引：

```
const int start_offset = start_ms * (kAudioSampleFrequency / 1000);
```

接下来，确定要获取的样本总数：

```
const int duration_sample_count =
duration_ms * (kAudioSampleFrequency / 1000);
```

现在我们有了这些信息，就可以找出音频捕获缓冲区中要读取的位置。我们将在一个循环中读取数据：

```
for (int i = 0; i < duration_sample_count; ++i) {
    // For each sample, transform its index in the history of all samples into
    // its index in g_audio_capture_buffer
```

```
const int capture_index = (start_offset + i) % kAudioCaptureBufferSize;
// Write the sample to the output buffer
g_audio_output_buffer[i] = g_audio_capture_buffer[capture_index];
}
```

之前，我们看到了如何使用模运算符在缓冲区中找到正确的位置，这个缓冲区只有一定的空间来容纳最新的样本。在这里，我们再次使用相同的技术——如果将所有历史样本中的当前索引除以音频捕获缓冲区的大小 `kAudioCaptureBufferSize`，则其余部分将指示该数据在缓冲区中的位置。然后，我们可以使用简单的赋值操作将数据从捕获缓冲区读取到输出缓冲区。

接下来，为了从这个函数中获取数据，我们使用两个作为参数提供的指针，分别是 `audio_samples_size`（指向音频样本的数量）和 `audio_samples`（指向输出缓冲区）：

```
// Set pointers to provide access to the audio
*audio_samples_size = kMaxAudioSampleSize;
*audio_samples = g_audio_output_buffer;

return kTfLiteOk;
}
```

我们通过返回 `kTfLiteOk` 结束函数，让调用者知道该操作已成功。

然后，在最后一部分中，我们定义 `LatestAudioTimestamp()`：

```
int32_t LatestAudioTimestamp() { return g_latest_audio_timestamp; }
```

由于它总是返回最新音频的时间戳，因此代码的其他部分可以循环检查它，以确定是否有新的音频数据到达。

这就是我们的音频提供程序的全部代码！现在，我们确认了数据提供程序能稳定地提供最新的音频样本。

作者介绍

Pete Warden 是 Google TensorFlow 面向移动和嵌入式设备部分的技术主管。他曾是 Jetpac 的首席技术官（CTO）和创始人，该公司于 2014 年被 Google 收购。他此前曾在 Apple 工作。他是 TensorFlow 团队的创始成员之一，在 Twitter 上的账号是 @petewarden，他搭建的有关实践深度学习的博客是 <https://petewarden.com/>。

Daniel Situnayake 在 Google 领导 TensorFlow Lite 的开发宣传工作。他是 Tiny Farms 的联合创始人，这是美国第一家利用自动化技术以工业规模生产昆虫蛋白的公司。他的职业生涯是从在英国伯明翰城市大学（Birmingham City University）讲授自动识别和数据捕获课程开始的。

译者介绍

魏兰 是一位软件开发工程师，现就职于 Google 北京。她是机器视觉、Android 性能优化爱好者。博客：blog.csdn.net/xiaowei_cqu。

卜杰 毕业于南京邮电大学，现于 Google 北京担任软件工程师。邮箱：jiebu.pub@gmail.com。

审校者介绍

王铁震 现就职于 Google 北京，是 TensorFlow 团队的核心软件开发工程师。邮箱：wangtz@google.com。

封面介绍

本书封面上的动物是赤叉尾蜂鸟（topaza pella），一种发现于南美洲北部的蜂鸟。它们生活在热带和亚热带的森林里，在树冠的上部和中部筑巢。

雄性赤叉尾蜂鸟平均身高约 8.7 英寸^{编注1}，雌性身高较矮，只有约 5.3 英寸。雄性和雌性赤叉尾蜂鸟均重约 10 克。人们认为它们是仅次于巨型蜂鸟的第二大蜂鸟物种。雄性呈虹彩红色，喉部呈金属绿色，头部呈黑色。雌性的羽毛大多是绿色的。

和其他蜂鸟一样，赤叉尾蜂鸟主要以花树的花蜜为食。通过以水平八字形旋转翅膀，这

编注 1：1 英寸=2.54 厘米。

些蜂鸟可以主动悬停飞行，这是零前进速度下的静止飞行。这使它们可以在半空中吮吸开花植物的花蜜。由于蜜蜂和蝴蝶无法接触到管状花朵的花粉，因此许多具有这种花朵的植物都依赖蜂鸟进行授粉。

除了每年两次的繁殖季节外，赤叉尾蜂鸟通常是独居的。雌性蜂鸟会用植物纤维制成的丝筑巢，巢会随着幼鸟的成长被逐渐延展。雌性蜂鸟通常一次产两个蛋。幼鸟在孵化后大约3周会长出羽毛。雌鸟会持续照顾它们的幼鸟大约6周。

赤叉尾蜂鸟在其栖息地中很常见，但由于很少靠近地面，因此不常被人见到。O'Reilly系列图书封面上的许多动物都濒临灭绝，这些动物对我们的世界很重要。

封面插图出自 Karen Montgomery 之手，取材于 Lydekker 的 *Royal Natural History* 中的一幅黑白版画。