O'REILLY®

华章IT

(原书第2版)

机器学习实战

基于Scikit-Learn、Keras和TensorFlow

Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow

powered by jupyter

[法] Aurélien Géron 著

宋能辉 李娴 译

机械工业出版社
China Machine Press

O’Reilly精品图书系列

机器学习实战：基于Scikit-Learn、Keras和TensorFlow：原书第2版

Hands-On Machine Learning with Scikit-Learn，Keras，and TensorFlow：Concepts，Tools，and Techniques to Build Intelligent Systems，Second Edition

（法）奥雷利安·杰龙　著

宋能辉　李娴　译

ISBN：978-7-111-66597-7

本书纸版由机械工业出版社于2020年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）在中华人民共和国境内（不包括中国香港、澳门特别行政区及中国台湾地区）制作与发行。

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

目录

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’ReillyConference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

推荐序

最近几年人工智能技术的突破性进展，比如AlphaGo战胜围棋世界冠军柯洁，Waymo开始部署自动驾驶出租车，都表明深度学习极大地推动了整个机器学习的发展。现在，即使对深度学习技术几乎一无所知的工程师和程序员，也可以使用简单而有效的工具来实现从数据中学习的复杂应用程序。本书就向你展示了具体应该如何来实现各种人工智能的应用，如计算机视觉、自然语言处理等。

本书作者是一位出色的机器学习顾问和培训师，前Google资深工程师，从2013年至2016年领导YouTube的视频分类团队，不仅具有深厚的理论功底，还有最前沿的工业界实战操作经验。作者通过使用简洁的理论和细致具体的示例，运用两个Python框架（Scikit-Learn和TensorFlow/Keras），帮助你直观地了解构建智能系统的相关概念和工具。你将从本书中学到各种机器学习技术（从简单的线性回归到各种神经网络结构）。每章都附有练习题，可以帮助你应用所学的知识，你所需要的只是编程练习。

本书内容广博，覆盖了机器学习的各个领域，不仅介绍了传统的机器学习模型，包括支持向量机、决策树、随机森林和集成方法，还提供了使用Scikit-Learn进行机器学习的端到端训练示例。作者尤其对深度神经网络进行了深入的探讨，包括各种神经网络架构（如卷积神经网络、递归神经网络等）、强化学习，以及如何使用TensorFlow/Keras库来构建和训练神经网络。

本书英文版在Amazon上的评分是4.7分（满分5分），近90%的读者给予了5星好评，在国内豆瓣读书上也得到91.5%的读者的5星好评，国内外同时有这么高的好评率，足以证明本书的价值及其良好可读性。

如果你正打算学习机器学习和深度学习，正在寻求一个切入点，那么我强烈建议你把本书当作入门教材。需要使用机器学习或者深度学习算法解决实际问题的工程师可将本书当作实战手册，它可以让你了解很多深度学习的最新研究成果和实用技巧。

张明清

布朗大学计算机系博士，

纽约州立大学阿尔巴尼分校计算机科学系副教授，

计算机视觉和机器学习实验室（CVML Lab）主任，

前通用电气公司全球研发中心计算机视觉实验室首席计算机科学家

译者序

随着AlphaGo在人机大战中一举成名，关于人工智能的研究开始广受关注，人工智能科学家也一跃成为“21世纪热门的人才”。人工智能，特别是机器学习和深度神经网络的广泛应用虽然兴起不久，但是对这两个密切关联的领域的研究其实已经持续了好几十年，早已形成了系统化的知识体系。对于想要踏入机器学习和深度学习领域的初学者和工程师而言，一本理论和实践相结合的书籍是必不可少的，本书就是这样一本书。

本书分为两部分：第一部分介绍机器学习的基础知识；第二部分介绍神经网络与深度学习。附录部分的内容也非常丰富。本书兼顾理论与实战，既适合在校学生，又适合有经验的工程师。

从理论上讲，本书最大的特色就是有深度，覆盖面广，但是书中并没有太多复杂的数学公式推导，很容易看懂。这在现在很多机器学习书籍中是不多见的。

从实战来说，本书使用了当前热门的机器学习框架Scikit-Learn及深度学习框架TensorFlow和Keras，每一章都配备相应的项目示例，代码的实操性和可读性非常好。本书也是为有经验的工程师而写的，是一本实用指南。特别是附录B给出的机器学习项目清单，如果工业界想做一套机器学习的解决方案，完全可以按照这个清单去做。

读者朋友可能非常关心第2版相比第1版有何区别，作者在第2版中不仅重写了大部分章节，还增加了很多机器学习的前沿知识，代码示例采用了Keras深度学习框架。

作者将本书所有章节的详细代码都发布在GitHub上。项目地址为：https://github.com/ageron/handson-ml2。

译者现在在比利时某科研机构从事深度学习处理器、嵌入式实时人工智能、计算机视觉和深度学习异构平台上的编程框架等研究工作，虽有多年的机器学习和计算机视觉研究和开发经验，但本书中所涉及的专业术语与概念较多，部分概念及术语尚无公认的中文译法，因此我们参考了一些网络上和研究论文中常用的译法。在翻译过程中虽然力求准确地反映原著内容，但由于译者水平有限，可能有错误或者遗漏之处，恳请读者批评指正。读者可以通过电子邮件songnh@outlook.com和译者取得联系。

感谢机械工业出版社华章公司的编辑们，特别是刘锋编辑，他们为保证本书的质量做了大量的编辑和审校工作，在此深表谢意。

还要感谢Ivannie，她在我翻译本书的过程中，给了我最大的快乐。

宋能辉

前言

机器学习海啸

2006年，Geoffrey Hinton等人发表了一篇论文[1]，展示了如何训练能够以最先进的精度（>98%）识别手写数字的深度神经网络。他们将这种技术称为“深度学习”。深度神经网络是（非常）简化的大脑皮层的模型，由一堆人工神经元层组成。当时人们普遍认为，训练深度神经网络是不可能的：Yann LeCun的深度卷积神经网络自20世纪90年代以来就一直很好地用于图像识别，尽管它们并不是通用的。，并且大多数研究人员在20世纪90年代后期就放弃了这一想法。该论文重新激发了科学界的兴趣，不久之后，许多新论文证明了（在强大的计算能力和大量数据的帮助下）深度学习不仅是可能的，而且还具有令人难以置信的成就，这是其他机器学习（ML）技术无法企及的。这种热情很快扩展到了机器学习的许多其他领域。

大约十年后，机器学习征服了整个工业界：它是当今高科技产品诸多魔力的核心，可以为你的网络搜索结果排名，为智能手机的语音识别提供支持，可以推荐视频，并在围棋比赛中击败世界冠军。在不知不觉中，它将驾驶你的汽车。

你的项目中的机器学习

因此，你自然会对机器学习感到兴奋，并很乐意加入这场盛宴！

也许你想让你的自制机器人拥有自己的大脑，使它能够识别人脸，或者学会走路。

也许你的公司拥有大量数据（用户日志、财务数据、生产数据、机器传感器数据、热线统计信息、人力资源报告等），如果你知道在

哪里看，很有可能会发现一些隐藏的宝石。借助机器学习，你可以完成以下和更多任务：

- 细分客户并为每个群体找到最佳的营销策略。
- 根据类似客户的购买记录，为每个客户推荐产品。
- 检测哪些交易可能是欺诈性的。
- 预测明年的收入。

无论出于何种原因，你都决定学习机器学习并将其实现在你的项目中。好主意！

目标与方法

本书假设你对机器学习一无所知，其目标是为你提供实现能够从数据中学习的程序所需的概念、工具和直觉。

我们将介绍大量技术，从最简单和最常用的技术（例如线性回归）到一些经常赢得比赛的深度学习技术。

本书不是实现每种算法的玩具版本，而是使用可用于生产环境的Python框架：

- Scikit-Learn非常易于使用，它有效地实现了许多机器学习算法，因此成为学习机器学习的重要切入点。Scikit-Learn由David Cournapeau于2007年创建，现在由法国计算机科学和自动化研究所的一个研究小组领导。
- TensorFlow是用于分布式数值计算的更复杂的库。通过将计算分布在数百个GPU（图形处理单元）服务器上，它可以有效地训练和运

行大型神经网络。TensorFlow（TF）是由Google创建的，并支持许多大型机器学习应用程序。它于2015年11月开源，2.0版本于2019年11月发布。

• Keras是高层深度学习API，使训练和运行神经网络变得非常简单。它可以在TensorFlow、Theano或微软Cognitive Toolkit（以前称为CNTK）之上运行。TensorFlow附带了该API自己的实现，称为tf.keras，支持某些高级TensorFlow功能（例如有效加载数据的能力）。

本书主张动手实践，通过具体的示例和一点点理论就可以对机器学习有一个直观的了解。虽然你无须拿起笔记本电脑就可以阅读本书，但我强烈建议你尝试用Jupyter notebook试验在https://github.com/ageron/handson-ml2上在线获得的代码示例。

先决条件

本书假定你具有一些Python编程经验，并且熟悉Python的主要科学库，尤其是NumPy、pandas和Matplotlib。

另外，如果你关心一些比较深入的内容，那么你应该对大学水平的数学知识（如微积分、线性代数、概率和统计）有一定的了解。

如果你还不了解Python，那么http://learnpython.org/是一个不错的起点。Python.org上的官方教程也相当不错。

如果你从未使用过Jupyter，则第2章将指导你完成安装并学习基础知识。它是工具箱中的一个强大工具。

如果你不熟悉Python的科学库，Jupyter notebook里面有一些教程。还有一个关于线性代数的快速数学教程。

路线图

本书分为两部分。第一部分涵盖以下主题：

- 什么是机器学习，它试图解决什么问题，以及其系统的主要类别和基本概念

- 典型机器学习项目中的步骤

- 通过将数据与模型进行拟合来学习

- 优化成本函数

- 处理、清洁和准备数据

- 选择和工程化特征

- 选择模型并使用交叉验证调整超参数

- 机器学习的挑战，特别是欠拟合和过拟合（偏差/方差的权衡）

- 最常见的学习算法：线性和多项式回归、逻辑回归、k-近邻算法、支持向量机、决策树、随机森林和集成方法

- 降低训练数据的维度以应对“维度的诅咒”

- 其他无监督学习技术，包括聚类、密度估计和异常检测

第二部分涵盖以下主题：

- 什么是神经网络以及它们的作用

• 使用TensorFlow和Keras构建和训练神经网络

• 最重要的神经网络架构，包括用于表格数据的前馈神经网络、用于计算机视觉的卷积网络、用于序列处理的递归网络和长短期记忆（LSTM）网络、用于自然语言处理的编码器/解码器和Transformer、自动编码器和用于生成学习的生成式对抗网络（GAN）

• 训练深度神经网络的技术

• 如何使用强化学习构建可以通过反复试错学习好的策略的代理程序（例如游戏中的机器人）

• 有效地加载和预处理大量数据

• 大规模训练和部署TensorFlow模型

第一部分主要基于Scikit-Learn，而第二部分则使用TensorFlow和Keras。

 不要草率地跳入深水：尽管深度学习无疑是机器学习中最令人兴奋的领域之一，但你应该首先掌握基础知识。而且，大多数问题可以使用更简单的技术（如第一部分中讨论的随机森林和集成学习方法）来很好地解决。如果你有足够的数据、计算能力和耐心，则深度学习最适合诸如图像识别、语音识别或自然语言处理之类的复杂问题。

第2版的变化

第2版有6个主要变化：

1.涵盖其他ML主题：更多的无监督学习技术（包括聚类、异常检测、密度估计和混合模型）；训练深度网络（包括自归一化网络）的更多技术；其他计算机视觉技术（包括Xception、SENet、使用YOLO进行物体检测，以及使用R-CNN进行语义分割）；使用卷积神经网络（CNN，包括WaveNet）处理序列；使用递归神经网络（RNN）、CNN和Transformer进行自然语言处理；GAN。

2.涵盖其他库和API（Keras、Data API、用于强化学习的TF-Agents），以及使用分布式策略API、TF-Serving和Google Cloud AI Platform大规模训练和部署TF模型；还简要介绍TF Transform、TFLite、TF Addons/Seq2Seq和TensorFlow.js。

3.讨论深度学习研究的一些最新重要成果。

4.将所有TensorFlow章节迁移到TensorFlow 2，并尽可能使用TensorFlow的Keras API（tf.keras）实现。

5.更新代码示例，使用最新版本的Scikit-Learn、NumPy、pandas、Matplotlib和其他库。

6.得益于读者的大量反馈，一些章节更加明晰，并修正了一些错误。

添加了一些章节，有些章节被重写，有些则被重新排序。有关第2版更新的更多详细信息请参见https://homl.info/changes2。

其他资源

许多优秀的资源可用于学习机器学习。例如，吴恩达（Andrew Ng）在Coursera上的机器学习课程虽然很好，但它需要投入大量的时间（数月）。

还有许多有趣的关于机器学习的网站，当然包括Scikit-Learn出色的用户指南。你可能还喜欢Dataquest（它提供了非常不错的交互式教程），以及机器学习博客（例如Quora上列出的那些博客）。最后，深度学习网站上有不错的资源清单，可供你了解更多信息。

关于机器学习，还有许多其他入门书籍。特别是：

• Joel Grus的Data Science from Scratch（O'Reilly）介绍了机器学习的基础知识，并在纯Python中实现了一些主要算法（顾名思义，从头开始）。

• Stephen Marsland的Machine Learning：An Algorithmic Perspective（Chapman&Hall）是对机器学习的出色介绍，它使用Python代码示例（也从零开始，但使用NumPy），涵盖广泛的主题。

• Sebastian Raschka的Python Machine Learning（Packt Publishing）也对机器学习进行了很好的介绍，并利用了Python开源库（Pylearn 2和Theano）。

• François Chollet的Deep Learning with Python（Manning）是一本非常实用的书，以清晰、简洁的方式涵盖了广泛的主题，书中涉及Keras库的很多内容。它偏爱代码示例甚于数学理论。

• Andriy Burkov的The Hundred-Page Machine Learning Book非常简短，涵盖了一系列令人印象深刻的主题，不仅以平易近人的方式介绍这些主题，同时也没有回避数学方程式。

• Yaser S.Abu-Mostafa、Malik Magdon-Ismail和Hsuan-Tien Lin的Learning from Data（AMLBook）介绍颇为理论化的机器学习方法，它提供了深刻的见解，尤其是在偏差/方差权衡方面（见第4章）。

• Stuart Russell和Peter Norvig的Artificial Intelligence：A Modern Approach，3rd Edition（Pearson）是一本非常出色的书，涵盖了机器学习等众多主题。它有助于正确理解机器学习。

最后，加入像Kaggle.com这样的机器学习竞赛网站，将使你在一些实际的问题上获得实践技能，并获得一些顶尖机器学习专业人员的帮助和见解。

排版约定

本书中使用以下排版约定：

斜体（Italic）

表示新的术语、URL、电子邮件地址、文件名和文件扩展名。

等宽字体（Constant width）

用于程序清单，以及段落中的程序元素，例如变量名、函数名、数据库、数据类型、环境变量、语句以及关键字。

等宽粗体（Constant width bold）

表示应由用户直接输入的命令或其他文本。

等宽斜体（Constant width italic）

表示应由用户提供的值或由上下文确定的值替换的文本。

 该图示表示提示或建议。

 该图示表示一般性说明。

 该图示表示警告或注意。

代码示例

一系列Jupyter notebook里面有很多补充材料，例如代码示例和练习，可从https://github.com/ageron/handson-ml2下载。

本书中的某些代码示例省略了与机器学习无关的重复部分或细节，这样可以把重点放在代码的重要部分上，节省空间以便覆盖更多的主题。如果需要完整的代码示例，可以在Jupyter notebook中找到它们。

请注意，当代码示例显示某些输出时，这些代码示例将在Python提示符（>>>和...）下显示，就像在Python shell中一样，可以清楚地区分代码与输出。例如，如下代码定义square（）函数，然后计算并显示3的平方：

```
>>> def square(x):
...             return x ** 2
...
>>> result = square(3)
>>> result
9
```

当代码不显示任何输出内容时，不使用提示符。但是有时结果可能会显示为注释，如下所示：

```
def square(x):
    return x ** 2
```

```
result = square(3)    #  result is 9
```

示例代码

这里的代码是为了帮助你更好地理解本书的内容。通常，可以在程序或文档中使用本书中的代码，而不需要联系O’Reilly获得许可，除非需要大段地复制代码。例如，使用本书中所提供的几个代码片段来编写一个程序不需要得到我们的许可，但销售或发布O’Reilly的配套CD-ROM则需要O’Reilly出版社的许可。引用本书的示例代码来回答问题也不需要许可，将本书中的示例代码的很大一部分放到自己的产品文档中则需要获得许可。

非常欢迎读者使用本书中的代码，希望（但不强制）注明出处。注明出处的形式包含书名、作者、出版社和ISBN，例如：

Hands-On Machine Learning with Scikit-Learn，Keras，and TensorFlow，2nd Edition，作者Aurélien Géron，由O’Reilly出版，书号978-1-492-03264-9。

如果读者觉得对示例代码的使用超出了上面所给出的许可范围，欢迎通过permission@oreilly.com联系我们。

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

近40年来，O’Reilly Media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专家和革新者组成的庞大网络，他们通过图书、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的在线学习平台允许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多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问http://oreilly.com。

如何联系我们

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

要询问技术问题或对本书提出建议，请发送电子邮件至bookquestions@oreilly.com。本书配套网站https://homl.info/oreilly2上列出了勘误表、示例以及其他信息。

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。

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

我从未想象过我的第一本书会吸引如此众多的读者。我收到了读者的大量反馈，很多人提出了许多问题，有些人指出了书中的差错，大多数人给了我鼓励。我对所有读者的大力支持表示感谢。非常感谢大家！如果你在代码示例中发现错误（或只是提出问题），请毫不犹豫地在GitHub上提交问题。如果在文本中发现错误，请提交勘误。一些读者还分享了本书如何帮助他们获得了第一份工作，或者它如何帮助他们解决了正在处理的具体问题。这种反馈极大地激励了我。如果你认为本书对你有所帮助，可以与我分享你的故事，无论是私下（例如，通过LinkedIn）还是公开地（例如，通过推文或通过亚马逊评论）与我分享。

我也非常感谢那些百忙之中抽出时间审阅本书的专家。特别要感谢François Chollet审阅了所有基于Keras和TensorFlow的章节，并给了我一些深入的反馈。由于Keras是第2版的主要新增内容之一，因此请Keras的作者审阅本书是非常值得的。我强烈推荐François的书Deep Learning with Python（Manning），它具有Keras库本身的简洁性、清晰度和深度。还要特别感谢Ankur Patel，他审阅了第2版的每一章，并给了我很好的反馈，特别是第9章（涵盖了无监督学习技术）。关于该主题，他可以写一本书，请查看Hands-On UnsupervisedLearning Using Python：How to Build Applied Machine Learning Solutions from Unlabeled Data（O’Reilly）。还要感谢Olzhas Akpambetov，他审阅了本书第二部分的所有章节，测试了许多代码，并提出了许多很好的建议。我非常感谢Mark Daoust、Jon Krohn、Dominic Monn和Josh Patterson如此全面地审阅了本书的第二部分，并用他们的专业知识提供了非常有用的反馈。

在撰写本书时，我很幸运地从TensorFlow团队成员那里得到了很多帮助，尤其是Martin Wicke，他不懈地回答了我的许多问题，并将其余的问题分发给了合适的人，包括Karmel Allison、Paige Bailey、Eugene Brevdo、William Chargin、Daniel“Wolff”

Dobson、Nick Felt、Bruce Fontaine、Goldie Gadde、Sandeep Gupta、Priya Gupta、KevinHaas、Konstantinos Katsiapis、Viacheslav Kovalevskyi、Allen Lavoie、Clemens Mewald、Dan Moldovan、Sean Morgan、Tom O’Malley、Alexandre Passos、AndréSusano Pinto、Anthony Platanios、Oscar Ramirez、Anna Revinskaya、Saurabh Saxena、Ryan Sepassi、Jiri Simsa、Xiaodan Song、Christina Sorokin、Dustin Tran、Todd Wang、Pete Warden（他审阅了第1版）、Edd Wilder-James和Yuefeng Zhou，他们都为我提供了帮助。非常感谢大家以及TensorFlow团队的所有其他成员，不仅是你们的帮助，而且也感谢你们做出如此出色的库！特别感谢TFX小组的Irene Giannoumis和Robert Crowe对第13章和第19章进行的深入审阅。

也要感谢O’Reilly出色的工作人员，尤其是Nicole Taché，他给了我颇有见地的反馈，并且总是开朗、鼓舞人心和乐于助人——我无法想象能有比他更好的编辑了。还要感谢在第2版开始时提供帮助（和耐心）的Michele Cronin，以及第2版的制作编辑Kristen Brown，她见证了本书的诞生（她还协调了第1版每次重印时的修订和更新工作）。还要感谢Rachel Monaghan和Amanda Kersey分别对第1版和第2版进行了详细编辑，也要感谢Johnny O’Toole，他管理与亚马逊的公共关系并回答了我的许多问题。感谢Marie Beaugureau、Ben Lorica、Mike Loukides和Laurel Ruma信任这个项目并帮助我确定了本书的范围。感谢Matt Hacker和所有Atlas团队回答了我有关格式、AsciiDoc和LaTeX的所有技术问题，并感谢Nick Adams、Rebecca Demarest、Rachel Head、JudithMcConville、Helen Monroe、Karen Montgomery、Rachel Roumeliotis和O’Reilly所有其他为本书做出贡献的人。

我还要感谢我以前的Google同事，特别是YouTube视频分类团队，他们教会了我很多关于机器学习的知识。没有他们，我永远不可能开始第1版。特别感谢我个人的ML专家Clément Courbet、Julien

Dubois、Mathias Kende、Daniel Kitachewsky、James Pack、Alexander Pak、Anosh Raj、Vitor Sessak、Wiktor Tomczak、Ingrid von Glehn和RichWashington。感谢在的YouTube和令人惊叹的Google山景城研究团队中与我合作的所有人。也非常感谢Martin Andrews、Sam Witteveen和Jason Zaman在Soonson Kwon的大力支持下，欢迎我加入新加坡的Google Developer Experts小组，我们就深度学习和TensorFlow进行的所有精彩讨论使我深受启发。任何对深度学习感兴趣的人都应该加入他们的新加坡深度学习聚会。特别感谢Jason，他在第19章分享了他的TFLite专业知识！

我永远不会忘记审阅本书第1版的好心人，包括David Andrzejewski、Lukas Biewald、Justin Francis、Vincent Guilbeau、Eddy Hung、Karim Matrah、Grégoire Mesnil、Salim Sémaoune、Iain Smears、Michel Tessier、Ingrid von Glehn、Pete Warden，当然还有我亲爱的兄弟Sylvain。特别感谢Haesun Park，他在把本书第1版翻译成韩语时给了我很多出色的反馈，并发现了一些错误。他还把Jupyter notebook翻译成韩文，更不用说TensorFlow的文档了。我不会说韩语，但是从他的反馈意见的质量来看，他的所有翻译都很出色！Haesun还为第2版的练习题提供了一些答案。

最后，我无限感激我心爱的妻子Emmanuelle和我们三个漂亮的孩子（Alexandre、Rémi和Gabrielle），他们鼓励我努力写本书。我也感谢他们的无限好奇心：向我的妻子和孩子们解释本书中一些最困难的概念，这有助于我阐明自己的想法，并直接改善了其中的许多内容。他们无限量地给我提供饼干和咖啡！人生如此，夫复何求？

[1] Geoffrey E.Hinton et al.，“A Fast Learning Algorithm for Deep Belief Nets”，Neural Computation 18（2006）：1527-1554.

第一部分　机器学习的基础知识

第1章　机器学习概览

当大多数人听到“机器学习”，会在脑海中浮现出一个机器人：一个可靠的管家或一个可怕的终结者，这取决于你问的是谁。但是机器学习并不是未来的幻想，它已经来了。事实上，在一些特定的应用中机器学习已经存在几十年了，比如光学字符识别（Optical Character Recognition，OCR）。但是直到20世纪90年代，第一个影响了数亿人的机器学习应用才真正变成主流，它就是垃圾邮件过滤器。虽然它并不是一个有自我意识的天网系统（Skynet），但是从技术上来说是符合机器学习的（它可以很好地进行学习，以至于用户几乎不用将某个邮件标记为垃圾邮件）。后来出现了数以百计的机器学习应用，支撑了数百个现在经常使用的产品和特性（从更好的推荐系统到语音搜索）。

那机器学习从哪里开始和在哪里结束呢？机器进行学习到底是什么意思？如果我下载了一份维基百科的副本，我的计算机就真的学会了什么吗？它突然就变聪明了吗？在本章中，我们首先会澄清机器学习到底是什么，以及为什么你想使用它。

在探索机器学习新大陆之前，先观察地图来学习下这片大陆上的主要地区和最显著的地标：监督学习和无监督学习、在线学习和批量学习、基于实例学习和基于模型学习。然后我们来看一个典型的机器学习项目的工作流程，讨论你可能会遇到的难点，并介绍如何评估和微调一个机器学习系统。

本章介绍了每个数据科学家需要牢记在心的大量基础概念（和专业术语）。虽然本章是概览（唯一没有代码的一章），相对简单，但

在继续学习本书其余章节之前，要确保掌握每一个知识点。端起一杯咖啡，开始学习吧！

 如果你已经知道机器学习的所有基础概念，可以直接学习第2章。如果你不确定，可以尝试回答本章末尾列出的问题，然后再继续。

1.1 什么是机器学习

机器学习是一门通过编程让计算机从数据中进行学习的科学（和艺术）。

下面是一个稍微通用一点的定义：

机器学习是一个研究领域，让计算机无须进行明确编程就具备学习能力。

——亚瑟·萨缪尔（Arthur Samuel），1959

更工程化的概念：

一个计算机程序利用经验E来学习任务T，性能是P，如果针对任务T的性能P随着经验E不断增长，则称为机器学习。

——汤姆·米切尔（Tom Mitchell），1997

例如，垃圾邮件过滤器就是一个机器学习程序，它可以根据垃圾邮件（比如，用户标记的垃圾邮件）和普通邮件（非垃圾邮件，也称作ham）学习标记垃圾邮件。系统用来进行学习的样例称作训练集。每个训练样例称作训练实例（或样本）。在这个示例中，任务T就是标记新邮件是否是垃圾邮件，经验E是训练数据，性能P需要定义。例如，可以使用正确分类邮件的比例。这个性能指标称为准确率，通常用在分类任务中。

如果你只下载了一份维基百科的副本，虽然你的计算机有了很多数据，但不会在任何工作中变得聪明起来。因此，下载一份维基百科的副本不是机器学习。

1.2 为什么使用机器学习

思考一下，你会如何使用传统的编程技术写一个垃圾邮件过滤器（见图1-1）：

1. 你会先看一下垃圾邮件一般都是什么样子。你可能注意到一些词或短语（比如4U、credit card、free、amazing）在邮件主题中频繁出现，也许还注意到一些在邮件的发件人名字、正文和其他地方会出现的一些固定模式，等等。

2. 你会为观察到的每个模式各写一个检测算法，如果检测到了某个规律，程序就会将邮件标记为垃圾邮件。

3. 你会测试程序，重复第1步和第2步，直到足够好可以发布。

因为这个问题很困难，你的程序很可能会变成一长串复杂的规则——很难维护。

相反，基于机器学习技术的垃圾邮件过滤器会自动学习词和短语，这些词和短语是垃圾邮件的预测因素，通过与非垃圾邮件比较，检测垃圾邮件中反复出现的词语模式（见图1-2）。这个程序更短，更易维护，也更精确。

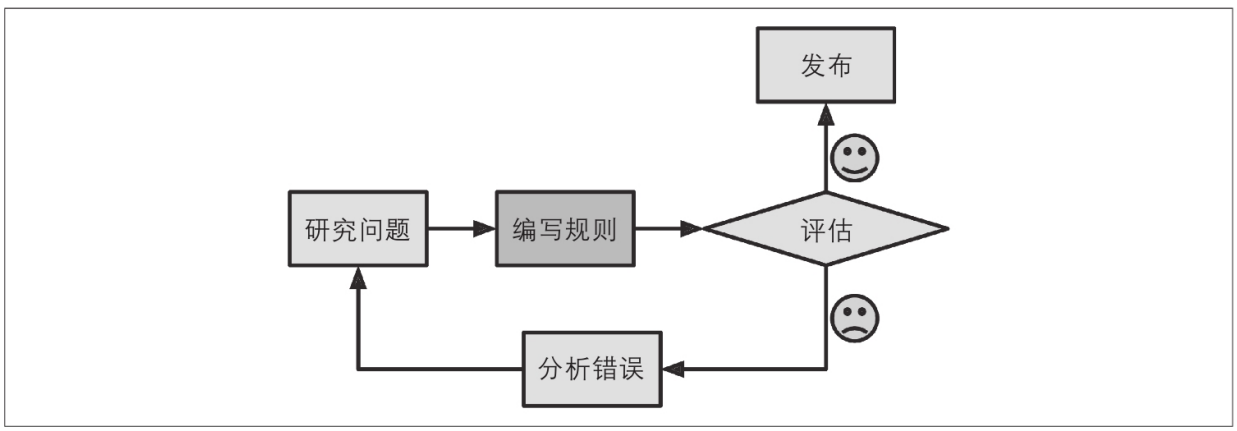


图1-1：传统方法

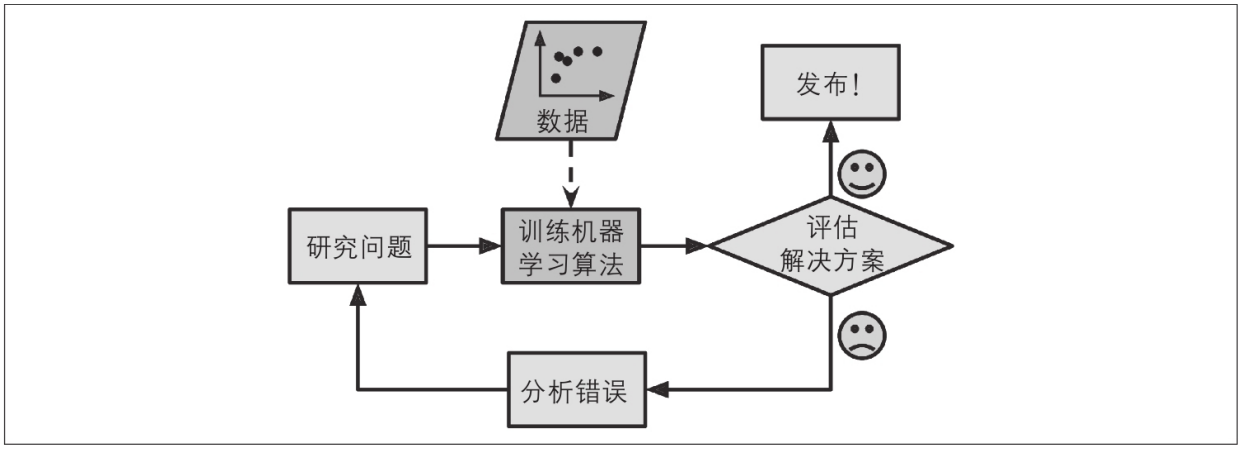


图1-2：机器学习方法

如果垃圾邮件的发送者发现所有包含“4U”的邮件都被屏蔽了，他们会转而使用“For U”。使用传统方法的垃圾邮件过滤器需要更新来标记“For U”。如果垃圾邮件的发送者持续更改，你就需要一直不停地写入新规则。

相反，基于机器学习的垃圾邮件过滤器会自动注意到“For U”在用户手动标记的垃圾邮件中频繁出现，然后就能自动标记垃圾邮件而无须人工干预了（见图1-3）。

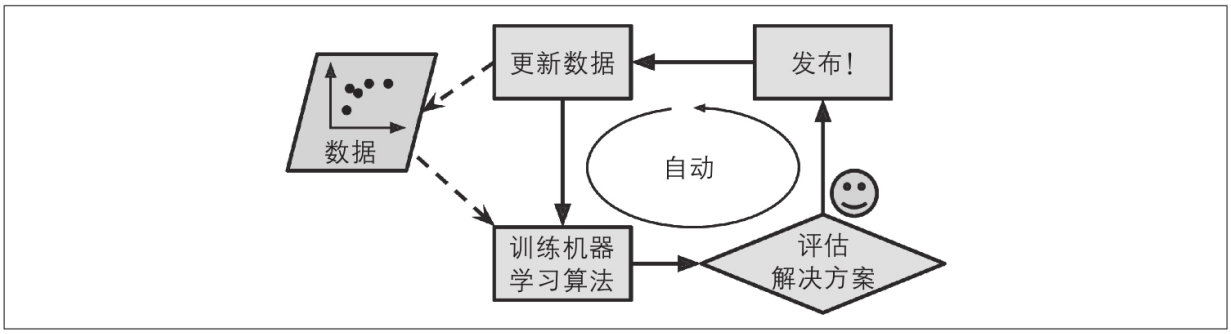


图1-3：自动适应改变

机器学习的另一个亮点是善于处理对于传统方法而言太复杂或没有已知算法的问题。例如，对于语音识别，假设你想写一个可以识别“one”和“two”的简单程序。你可能注意到“two”的起始是一个高音（“T”），因此会写一个可以测量高音强度的硬编码算法，用于区分“one”和“two”。但是很明显，这个方法不能推广到所有的语音识别（人们所处环境不同、语言不同、使用的词汇不同）。（现在）最佳的方法是根据给定的大量单词录音，写一个可以自我学习的算法。

最后，机器学习可以帮助人类进行学习（见图1-4）。机器学习算法可以检测自己学到了什么（尽管这对于某些算法很棘手）。例如，在垃圾邮件过滤器训练了足够多的垃圾邮件后，就可以用它列出垃圾邮件预测器的单词和单词组合。有时可能会发现不引人关注的关联或新趋势，这有助于更好地理解问题。使用机器学习方法挖掘大量数据来帮助发现不太明显的规律。这称作数据挖掘。

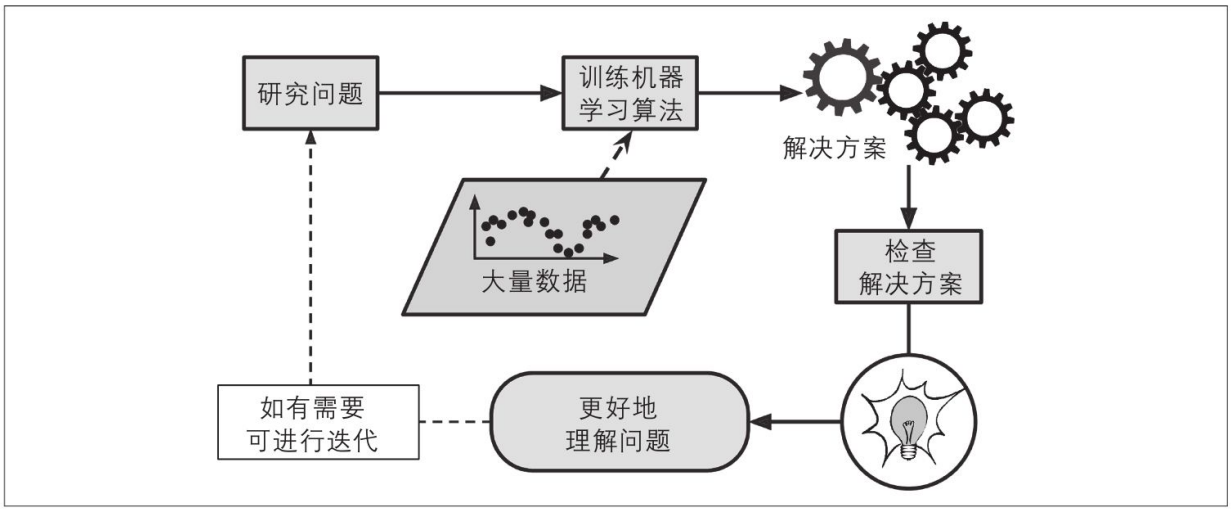


图1-4：机器学习可以帮助人类学习

总结一下，机器学习适用于：

- 有解决方案（但解决方案需要进行大量人工微调或需要遵循大量规则）的问题：机器学习算法通常可以简化代码，相比传统方法有更好的性能。

• 传统方法难以解决的复杂问题：最好的机器学习技术也许可以找到解决方案。

• 环境有波动：机器学习算法可以适应新数据。

• 洞察复杂问题和大量数据。

1.3 机器学习的应用示例

下面介绍一些机器学习的具体示例和所使用的技术。

分析生产线上的产品图像来对产品进行自动分类

这是图像分类问题，使用卷积神经网络（CNN，见第14章）的典型示例。

通过脑部扫描发现肿瘤

这是语义分割，图像中的每个像素都需要被分类（当我们想确定肿瘤的确切位置和形状时），也使用CNN。

自动分类新闻

这是自然语言处理（NLP），更具体地是文本分类，可以使用循环神经网络（RNN）、CNN或者Transformer（见第16章）。

论坛中自动标记恶评

这也是文本分类，使用相同的自然语言处理工具。

自动对长文章做总结

这是自然语言处理的一个分支，叫作文本总结，使用相同的工具。

创建一个聊天机器人或者个人助理

这涉及自然语言处理的很多分支，包括自然语言理解（NLU）和问答模块。

基于很多性能指标来预测公司下一年的收入

这是一个回归问题（如预测值），需要使用回归模型进行处理，例如线性回归或多项式回归（见第4章）、SVM回归（见第5章）、随机森林回归（见第7章）或者人工神经网络（见第10章），如果考虑过去的性能指标，可以使用RNN、CNN或者Transformer（见第15章和第16章）。

让应用对语音命令做出反应

这是语音识别，要求能处理音频采样。因为音频是很长、很复杂的序列，所以一般使用RNN、CNN或者Transformer（见第15章和第16章）进行处理。

检测信用卡欺诈

这是异常检测（见第9章）。

基于客户的购买记录来对客户进行分类，对每一类客户设计不同的市场策略

这是聚类问题（见第9章）。

用清晰而有洞察力的图表来表示复杂的高维数据集

这是数据可视化，经常涉及降维技术（见第8章）。

基于以前的购买记录给客户推荐可能感兴趣的产品

这是推荐系统，一个办法是将以前的购买记录（和客户的其他信息）输入人工神经网络（见第10章），从而输出客户最可能购买的产品。这个神经网络是在所有客户的购买记录上训练的。

为游戏建造智能机器人

这通常通过强化学习（RL，见第18章）来解决。强化学习是机器学习的一个分支，在一个给定的环境（例如游戏）中，训练代理（例如机器人）选择在一段时间内将它们的奖励最大化的行动（例如，机器人可能会在玩家每次失去一些生命值时获得奖励）。在围棋比赛中打败世界冠军的著名AlphaGo程序就是使用RL构建的。

这个列表可以一直延伸下去，但希望它能让你了解机器学习所能处理的任务的广度和复杂性，以及你在每个任务中会用到的技术类型。

1.4　机器学习系统的类型

现有的机器学习系统类型繁多，为便于理解，我们根据以下标准将它们进行大的分类：

• 是否在人类监督下训练（有监督学习、无监督学习、半监督学习和强化学习）。

• 是否可以动态地进行增量学习（在线学习和批量学习）。

• 是简单地将新的数据点和已知的数据点进行匹配，还是像科学家那样，对训练数据进行模式检测然后建立一个预测模型（基于实例的学习和基于模型的学习）。

这些标准之间互相并不排斥，你可以以你喜欢的方式将其任意组合。例如，现在最先进的垃圾邮件过滤器可能是使用深度神经网络模型对垃圾邮件和常规邮件进行训练，完成动态学习。这使其成为一个在线的、基于模型的有监督学习系统。

我们来看看这几个标准。

1.4.1　有监督学习和无监督学习

根据训练期间接受的监督数量和监督类型，可以将机器学习系统分为以下四个主要类别：有监督学习、无监督学习、半监督学习和强化学习。

有监督学习

在有监督学习中，提供给算法的包含所需解决方案的训练集称为标签（见图1-5）。

分类任务是一个典型的有监督学习任务。垃圾邮件过滤器就是一个很好的示例：通过大量的电子邮件示例及其所属的类别（垃圾邮件还是常规邮件）进行训练，然后学习如何对新邮件进行分类。

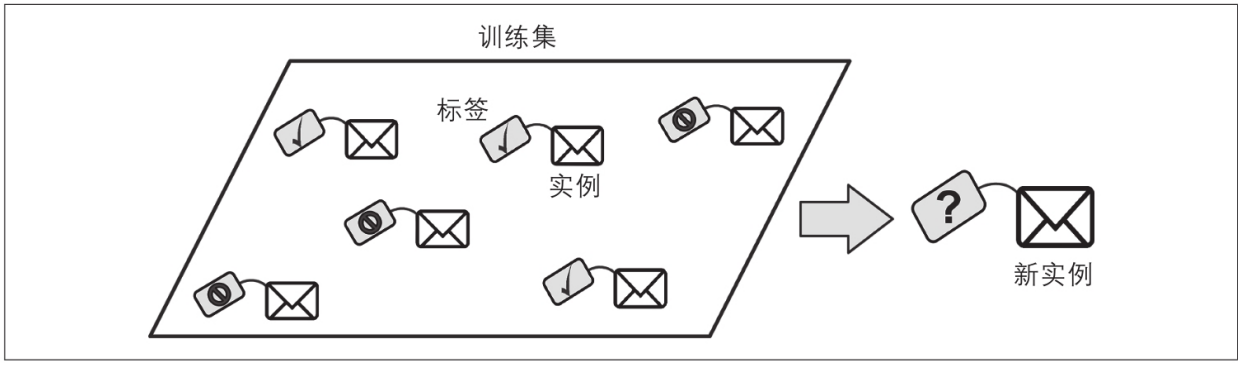


图1-5：用于垃圾邮件分类的已标记训练集（有监督学习的示例）

另一个典型的任务是通过给定一组称为预测器的特征（里程、使用年限、品牌等）来预测一个目标数值（例如汽车的价格）。这种类型的任务称为回归（见图1-6）[1]。要训练这样一个系统，需要提供大量的汽车示例，包括它们的预测器和标签（即价格）。

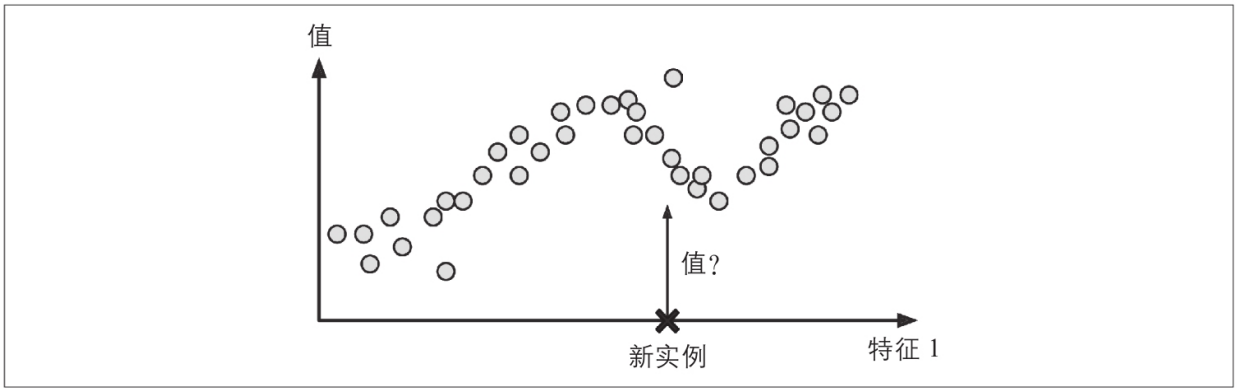


图1-6：回归问题：在给定输入特征的情况下预测值（通常有多个输入特征，有时有多个输出值）

 在机器学习里，属性是一种数据类型（例如“里程”），而特征取决于上下文，可能有多个含义，但是通常状况下，特征意味着一个属性加上其值（例如，“里程=15 000”）。尽管如此，许多人还是在使用属性和特征这两个名词时不做区分。

值得注意的是，一些回归算法也可以用于分类任务，反之亦然。例如，逻辑回归就被广泛地用于分类，因为它可以输出“属于某个给定类别的概率”的值（例如，20%的概率是垃圾邮件）。

这里是一些最重要的有监督学习算法（本书中会介绍）：

- k-近邻算法
- 线性回归
- 逻辑回归
- 支持向量机（SVM）
- 决策树和随机森林
- 神经网络[2]

无监督学习

顾名思义，无监督学习的训练数据都是未经标记的（见图1-7）。系统会在没有“老师”的情况下进行学习。

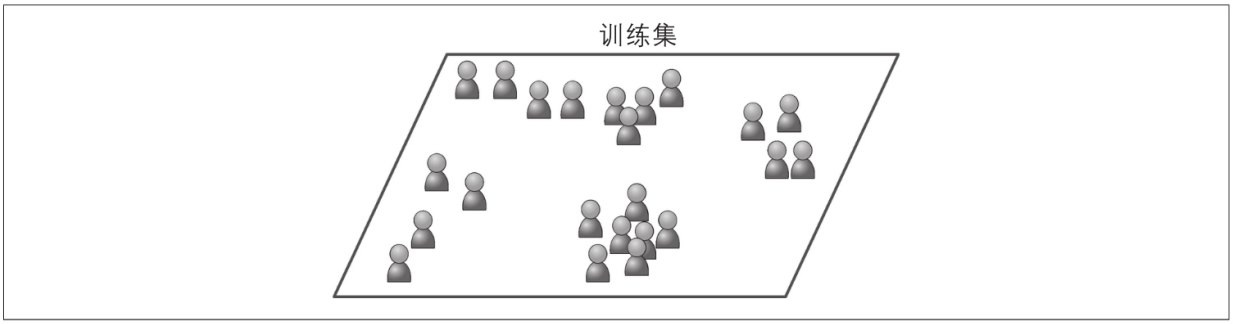


图1-7：无标签的训练集，用于无监督学习

这里有一些最重要的无监督学习算法（大部分会在第8章和第9章中介绍）：

- 聚类算法
- k-均值算法
- DBSCAN
- 分层聚类分析（HCA）
- 异常检测和新颖性检测
- 单类SVM
- 孤立森林
- 可视化和降维
- 主成分分析（PCA）
- 核主成分分析

- 局部线性嵌入（LLE）
- t-分布随机近邻嵌入（t-SNE）
- 关联规则学习
- Apriori
- Eclat

例如，假设你现在拥有大量关于自己博客访客的数据。你想通过一个聚类算法来检测相似访客的分组（见图1-8）。你不大可能告诉这个算法每个访客属于哪个分组——算法会自行寻找这种关联。例如，它可能会注意到40%的访客是喜欢漫画的男性，并且通常在夜晚阅读你的博客；20%的访客是年轻的科幻爱好者，通常在周末访问；等等。如果使用的是分层聚类算法，还可以将每组细分为更小的组。这可能有助于你针对不同的分组来发布博客内容。

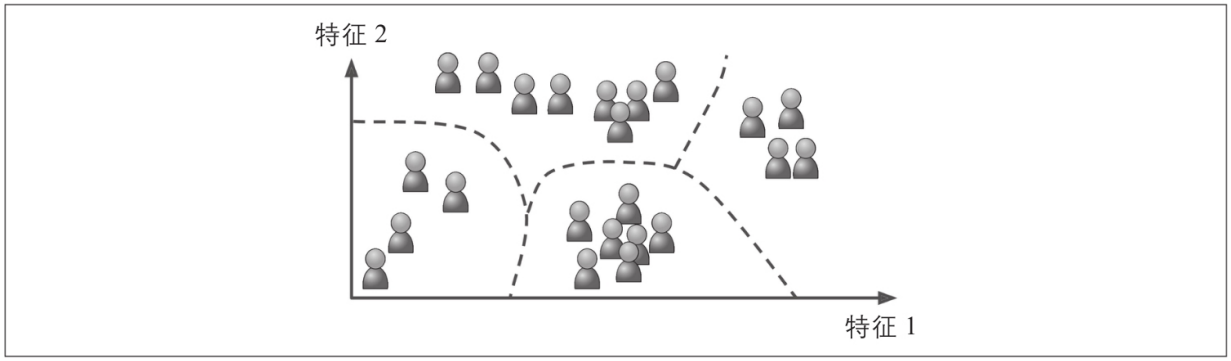


图1-8：聚类

可视化算法也是无监督学习算法的一个不错的示例：你提供大量复杂的、未标记的数据，算法轻松绘制输出2D或3D的数据表示（见图1-9）。这些算法会尽其所能地保留尽量多的结构（例如，尝试保持输入的单独集群在可视化中不会被重叠），以便于你理解这些数据是怎么组织的，甚至识别出一些未知的模式。

与之相关的一个任务是降维，降维的目的是在不丢失太多信息的前提下简化数据。方法之一是将多个相关特征合并为一个。例如，汽车里程与其使用年限存在很大的相关性，所以降维算法会将它们合并成一个代表汽车磨损的特征。这个过程叫作特征提取。

 通常比较好的做法是，先使用降维算法减少训练数据的维度，再将其提供给另一个机器学习算法（例如有监督学习算法）。这会使它运行得更快，数据占用的磁盘空间和内存都会更小，在某些情况下，执行性能也会更高。

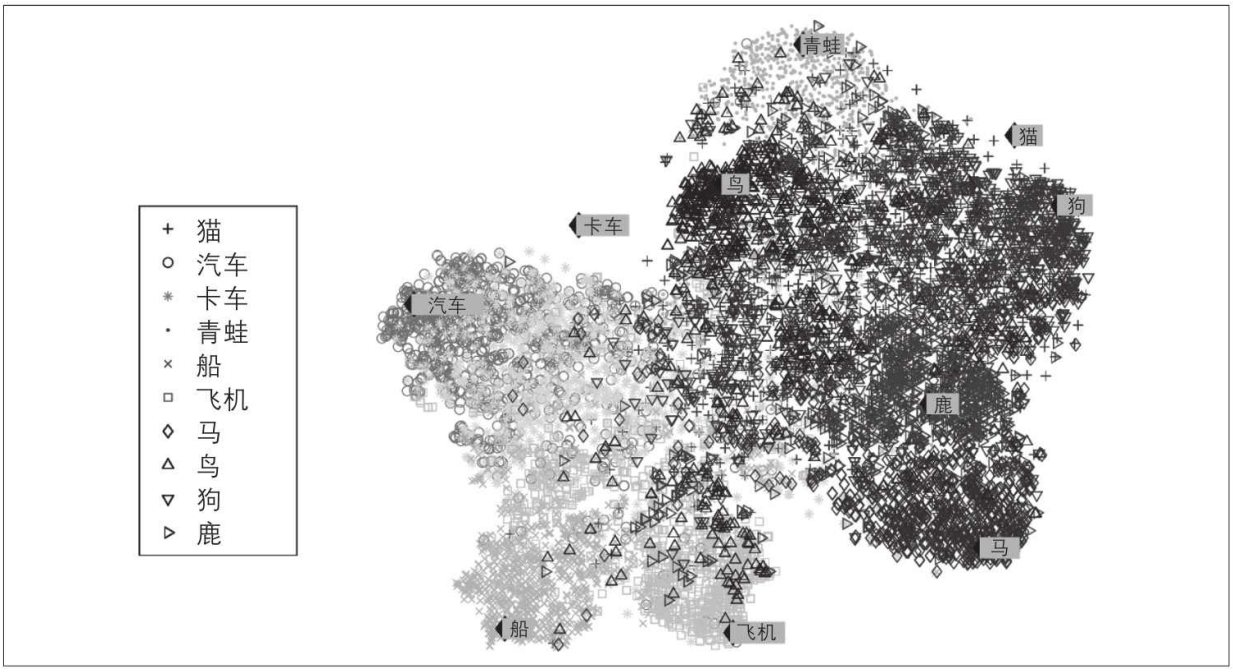


图1-9：语义聚类的t-SNE可视化示例[3]

另一个很重要的无监督任务是异常检测——例如，检测异常信用卡交易以防止欺诈，捕捉制造缺陷，或者在给另一种机器学习算法提供数据之前自动从数据集中移除异常值。系统用正常实例进行训练，然后当看到新的实例时，它就可以判断出这个新实例看上去是正常还是异常（见图1-10）。一个非常类似的任务是新颖性检测。它的目的是检测看起来与训练集中的所有实例不同的新实例。这需要一个非常“干净”的

训练集，没有你希望算法能检测到的任何实例。例如，如果你有成千上万张狗的照片，其中1%是吉娃娃犬，那么一个新颖性检测算法不应将吉娃娃犬的新图片视为新颖。另一方面，异常检测算法可能会认为这些狗非常罕见，与其他狗不同，可能会把它们归类为异常（没有对吉娃娃犬不敬的意思）。

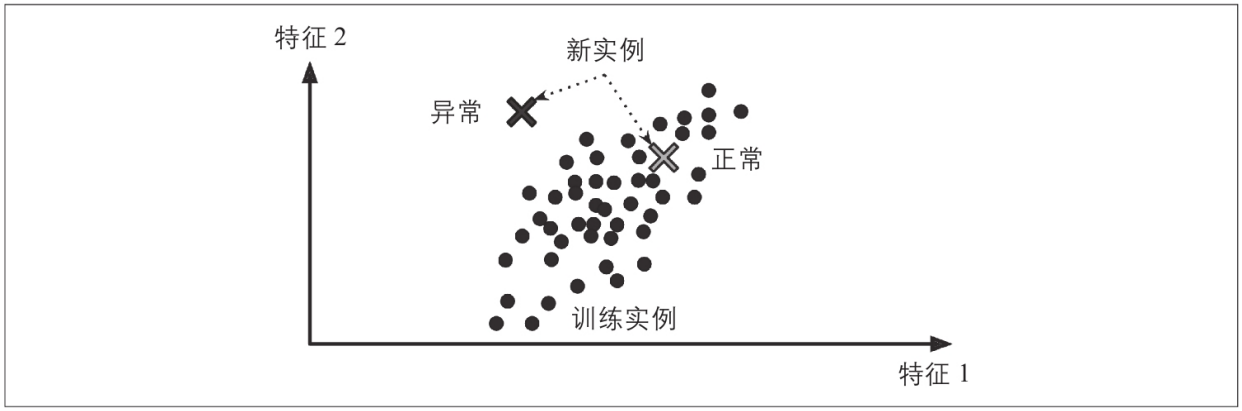


图1-10：异常检测

最后，还有一个常见的无监督任务是关联规则学习，其目的是挖掘大量数据，发现属性之间的有趣联系。例如，假设你开了一家超市，在销售日志上运行关联规则之后发现买烧烤酱和薯片的人也倾向于购买牛排。那么，你可能会将这几样商品摆放得更近一些。

半监督学习

由于通常给数据做标记是非常耗时和昂贵的，你往往会有很多未标记的数据而很少有已标记的数据。有些算法可以处理部分已标记的数据。这被称为半监督学习（见图1-11）。

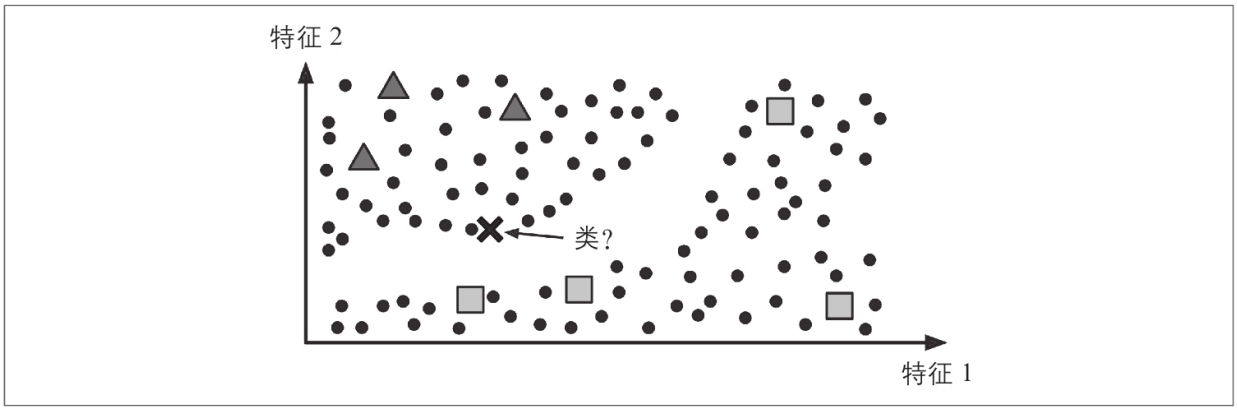


图1-11：半监督学习有两个类别（三角形和正方形）：未标记的示例（圆形）有助于将新实例（十字）分类为三角形类别而不是正方形类别，即使它更接近于标记的正方形

有些照片托管服务（例如Google相册）就是很好的示例。一旦你将所有的家庭照片上传到服务器后，它会自动识别出人物A出现在照片1、5和11中，人物B出现在照片2、5和7中。这是算法的无监督部分（聚类）。现在系统需要你做的只是告诉它这些人都是谁。给每个人一个标签之后[4]，它就可以给每张照片中的每个人命名，这对于搜索图片非常重要。

大多数半监督学习算法是无监督算法和有监督算法的结合。例如，深度信念网络（DBN）基于一种互相堆叠的无监督组件，这个组件叫作受限玻尔兹曼机（RBM）。受限玻尔兹曼机以无监督方式进行训练，然后使用有监督学习技术对整个系统进行微调。

强化学习

强化学习则是一个非常与众不同的“巨兽”。它的学习系统（在其语境中称为智能体）能够观察环境，做出选择，执行动作，并获得回报（或者是以负面回报的形式获得惩罚，见图1-12）。所以它必须自行学习什么是最好的策略，从而随着时间的推移获得最大的回报。策略代表智能体在特定情况下应该选择的动作。

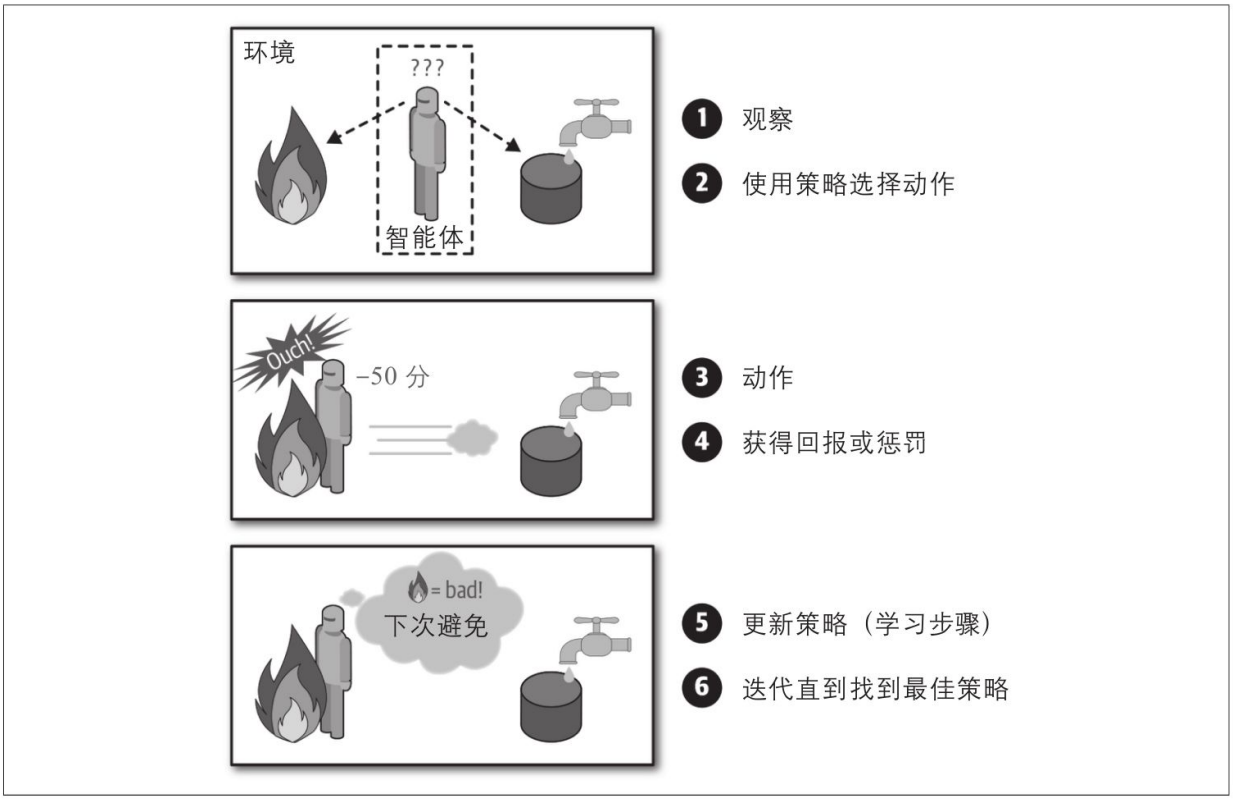


图1-12：强化学习

例如，许多机器人通过强化学习算法来学习如何行走。DeepMind的AlphaGo项目也是一个强化学习的好示例。2017年5月，AlphaGo在围棋比赛中击败世界冠军柯洁而声名鹊起。通过分析数百万场比赛，然后自己跟自己下棋，它学到了制胜策略。要注意，在跟世界冠军对弈的时候，AlphaGo处于关闭学习状态，它只是应用它所学到的策略而已。

1.4.2 批量学习和在线学习

另一个给机器学习系统分类的标准是看系统是否可以从传入的数据流中进行增量学习。

批量学习

在批量学习中，系统无法进行增量学习——即必须使用所有可用数据进行训练。这需要大量时间和计算资源，所以通常都是离线完成的。

离线学习就是先训练系统，然后将其投入生产环境，这时学习过程停止，它只是将其所学到的应用出来。

如果希望批量学习系统学习新数据（例如新型垃圾邮件），需要在完整数据集（包括新数据和旧数据）的基础上重新训练系统的新版本，然后停用旧系统，用新系统取而代之。幸运的是，整个训练、评估和启动机器学习系统的过程可以很轻易地实现自动化（如图1-3所示），所以即使是批量学习系统也能够适应变化。只是需要不断地更新数据，并根据需要频繁地训练系统的新版本。

这个解决方案比较简单，通常也都能正常工作，只是每次都使用完整数据集进行训练可能需要花上好几个小时，所以，你很有可能会选择每天甚至每周训练一次新系统。如果系统需要应对快速变化的数据（例如，预测股票价格），那么你需要一个更具响应力的解决方案。

此外，使用完整数据集训练需要耗费大量的计算资源（CPU、内存空间、磁盘空间、磁盘I/O、网络I/O等）。如果你的数据量非常大，并且每天从零开始自动执行训练系统，那最终你将为此花费大量的金钱。而假如你面对的是海量数据，甚至可能无法再应用批量学习算法。

所以如果你的资源有限（例如，一个智能手机应用程序或一个火星上的漫游器），而系统需要实现自主学习，那么像这样携带大量训练数据，占用大量资源，动辄每天耗费几小时来进行训练的方式，肯定会让你心有余而力不足。

幸运的是，在所有这些情况下，我们有一个更好的选择——能够进行增量学习的算法。

在线学习

在在线学习中，你可以循序渐进地给系统提供训练数据，逐步积累学习成果。这种提供数据的方式可以是单独的，也可以采用小批量的小

组数据来进行训练。每一步学习都很快速并且便宜，这样系统就可以根据飞速写入的最新数据进行学习（见图1-13）。

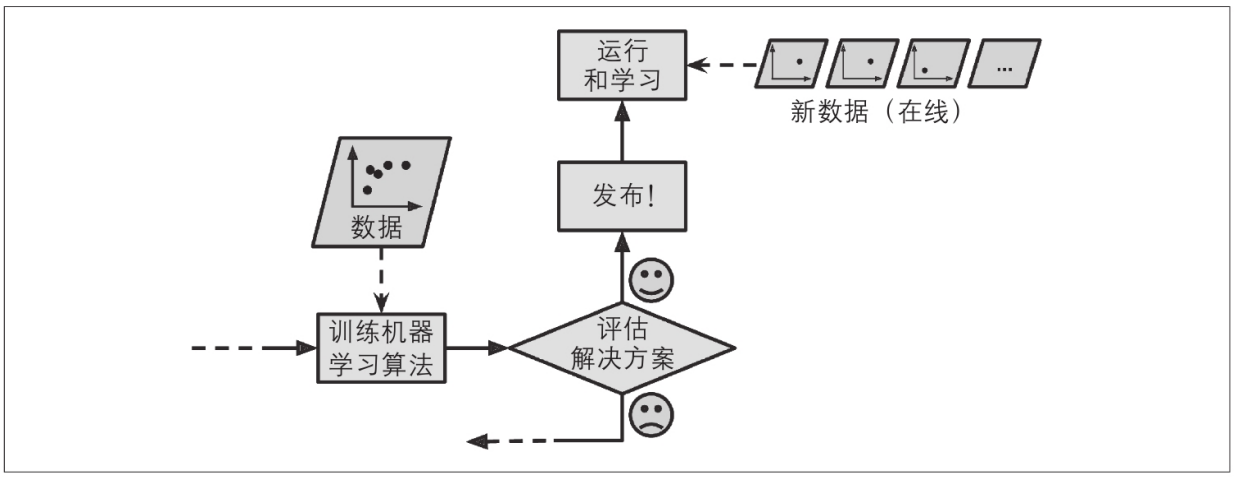


图1-13：在线学习中，模型经过训练并投入生产环境，然后随着新数据的进入而不断学习

对于这类系统——需要接收持续的数据流（例如股票价格），同时对数据流的变化做出快速或自主的反应，使用在线学习是一个非常好的方式。如果你的计算资源有限，在线学习同样也是一个很好的选择：新的数据实例一旦经过在线学习系统的学习，就不再需要，你可以将其丢弃（除非你想回滚到前一个状态，再“重新学习”数据），这可以节省大量的空间。

对于超大数据集——超出一台计算机的主存储器的数据，在线学习算法也同样适用（这称为核外学习）。算法每次只加载部分数据，并针对这部分数据进行训练，然后不断重复这个过程，直到完成所有数据的训练（见图1-14）。

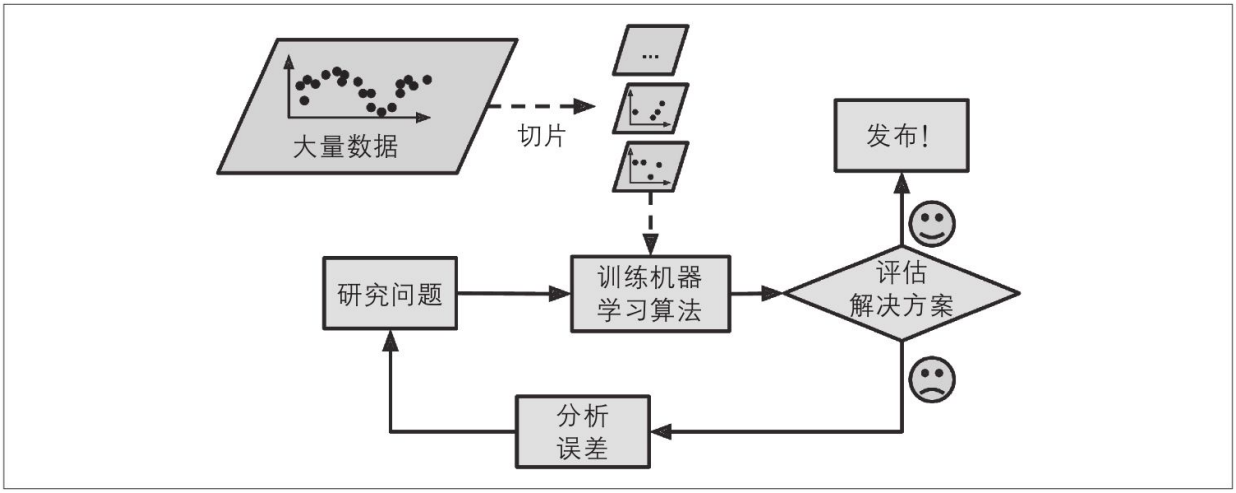


图1-14：使用在线学习来处理超大数据集

 核外学习通常是离线完成的（也就是不在实时（live）系统上），因此在线学习这个名字很容易让人产生误解。我们可以将其视为增量学习。

在线学习系统的一个重要参数是其适应不断变化的数据的速度，这就是所谓的学习率。如果设置的学习率很高，那么系统将会迅速适应新数据，但同时也会很快忘记旧数据（你肯定不希望垃圾邮件过滤器只对最新显示的邮件进行标记）。反过来，如果学习率很低，系统会有更高的惰性，也就是说，学习会更缓慢，同时也会对新数据中的噪声或者非典型数据点（离群值）的序列更不敏感。

在线学习面临的一个重大挑战是，如果给系统输入不良数据，系统的性能将会逐渐下降。现在某些实时系统的客户说不定已经注意到了这个现象。不良数据的来源可能是机器上发生故障的传感器，或者是有人对搜索引擎恶意刷屏以提高搜索结果排名等。为了降低这种风险，你需要密切监控系统，一旦检测到性能下降，就及时中断学习（可能还需要恢复到之前的工作状态）。当然，同时你还需要监控输入数据，并对异常数据做出响应（例如，使用异常检测算法）。

1.4.3 基于实例的学习与基于模型的学习

另一种对机器学习系统进行分类的方法是看它们如何泛化。大多数机器学习任务是要做出预测。这意味着系统需要通过给定的训练示例，在它此前并未见过的示例上进行预测（泛化）。在训练数据上实现良好的性能指标固然重要，但是还不够充分。真正的目的是要在新的对象实例上表现出色。

泛化的主要方法有两种：基于实例的学习和基于模型的学习。

基于实例的学习

我们最司空见惯的学习方法就是简单地死记硬背。如果以这种方式创建一个垃圾邮件过滤器，那么它可能只会标记那些与已被用户标记为垃圾邮件完全相同的邮件——这虽然不是最差的解决方案，但肯定也不是最好的。

除了完全相同的，你还可以通过编程让系统标记与已知的垃圾邮件非常相似的邮件。这里需要两封邮件之间的相似度度量。一种（基本的）相似度度量方式是计算它们之间相同的单词数目。如果一封新邮件与一封已知的垃圾邮件有许多单词相同，系统就可以将其标记为垃圾邮件。

这被称为基于实例的学习：系统用心学习这些示例，然后通过使用相似度度量来比较新实例和已经学习的实例（或它们的子集），从而泛化新实例。例如，图1-15中的新实例会归为三角形，因为大多数最相似的实例属于那一类。

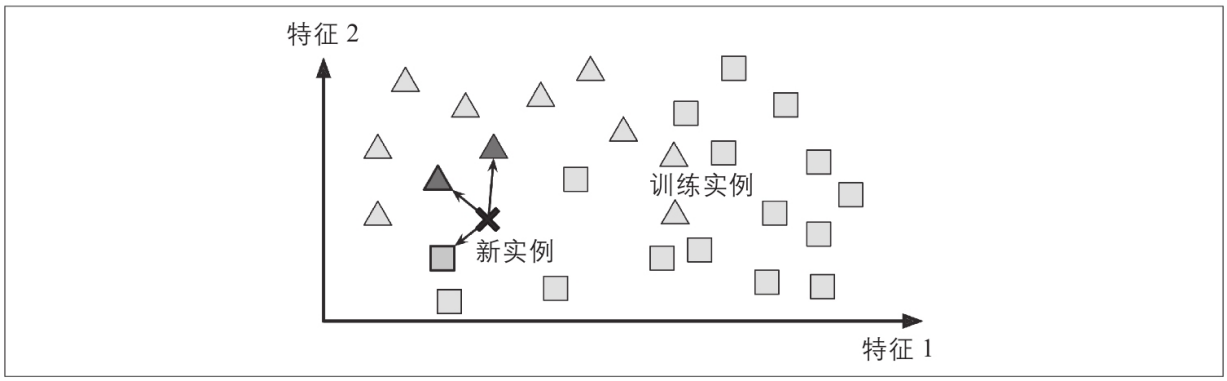


图1-15：基于实例的学习

基于模型的学习

从一组示例集中实现泛化的另一种方法是构建这些示例的模型，然后使用该模型进行预测。这称为基于模型的学习（见图1-16）。

举例来说，假设你想知道金钱是否让人感到快乐，你可以从经合组织（OECD）的网站上下载“幸福指数”的数据，再从国际货币基金组织（IMF）的网站上找到人均GDP的统计数据，将数据并入表格，按照人均GDP排序，你会得到如表1-1所示的摘要。

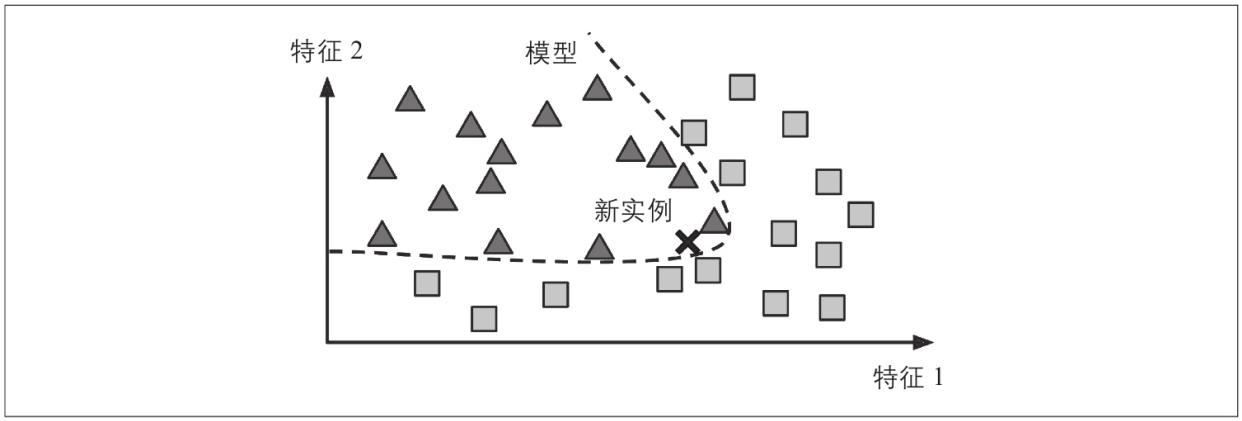


图1-16：基于模型的学习

表1-1：金钱能让人更快乐吗？

国家	人均 GDP（美元）	生活满意度
匈牙利	12 240	4.9
韩国	27 195	5.8
法国	37 675	6.5
澳大利亚	50 962	7.3
美国	55 805	7.2

让我们绘制这些国家的数据（见图1-17）。

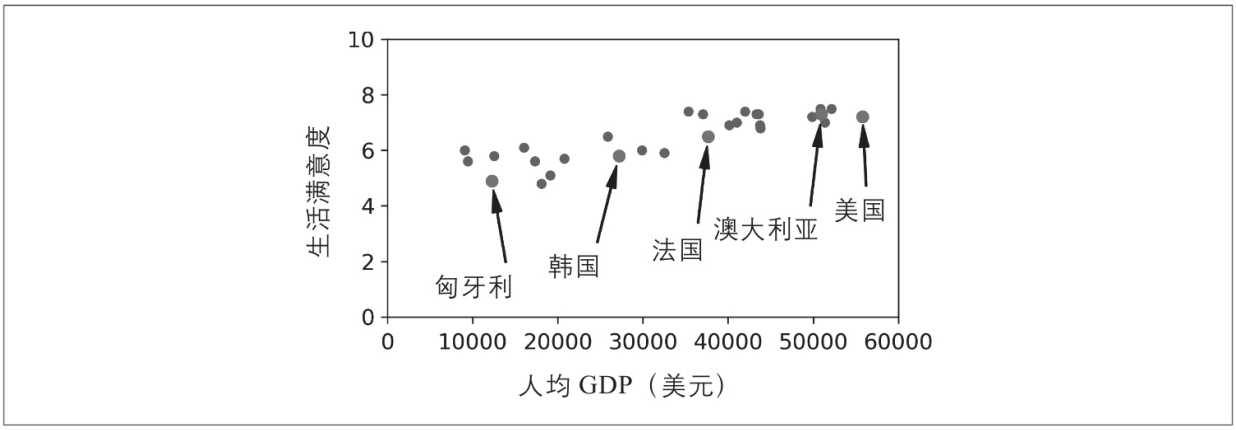


图1-17：趋势图

这里似乎有一个趋势！虽然数据包含噪声（即部分随机），但是仍然可以看出随着该国人均GDP的增加，生活满意度或多或少呈线性上升的趋势。所以你可以把生活满意度建模成一个关于人均GDP的线性函数。这个过程叫作模型选择。你为生活满意度选择了一个线性模型，该模型只有一个属性，就是人均GDP（见公式1-1）。

公式1-1：一个简单的线性模型

$$\text{生活满意度} = \theta_0 + \theta_1 \times \text{人均 GDP}$$

这个模型有两个模型参数：θ_0和θ_1[5]。通过调整这两个参数，可以用这个模型来代表任意线性函数，如图1-18所示。

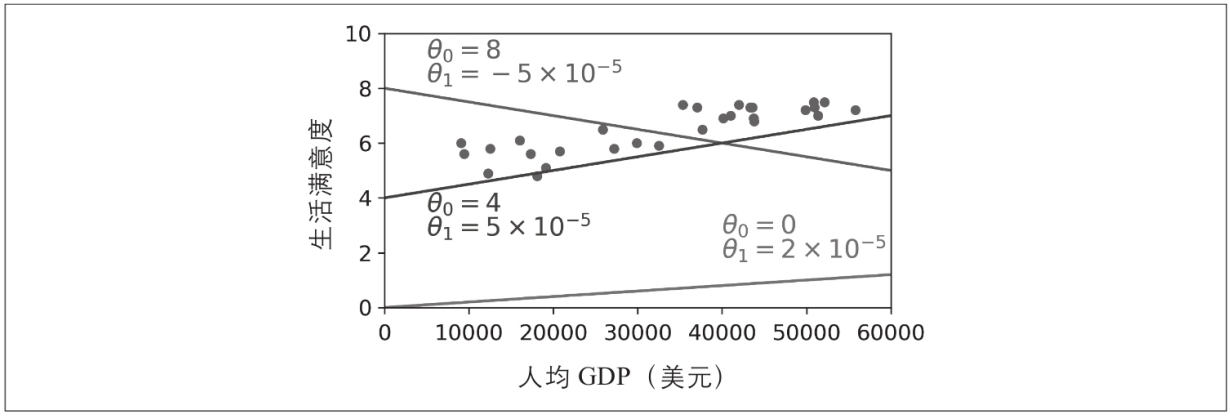


图1-18：一些可能的线性模型

在使用模型之前，需要先定义参数θ_0和θ_1的值。怎么才能知道什么值可以使模型表现最佳呢？要回答这个问题，需要先确定怎么衡量模型的性能表现。要么定义一个效用函数（或适应度函数）来衡量模型有多好，要么定义一个成本函数来衡量模型有多差。对于线性回归问题，通常的选择是使用成本函数来衡量线性模型的预测与训练实例之间的差距，目的在于尽量使这个差距最小化。

这正是线性回归算法的意义所在：通过你提供的训练样本，找出最符合提供数据的线性模型的参数，这称为训练模型。在这个案例中，算法找到的最优参数值为$\theta_0=4.85$和$\theta_1=4.91\times10^{-5}$。

 令人困惑的是，同一个词“模型”可以指模型的一种类型（例如，线性回归），到一个完全特定的模型架构（例如，有一个输入和一个输出的线性回归），或者到最后可用于预测的训练模型（例如，有一个输入和一个输出的线性回归，使用参数$\theta_0=4.85$和$\theta_1=4.91\times10^{-5}$）。模型选择包括选择模型的类型和完全指定它的架构。训练一个模型意味着运行一种寻找模型参数的算法，使其最适合训练数据（希望能对新的数据做出好的预测）。

现在，（对于线性模型而言）模型基本接近训练数据，如图1-19所示。

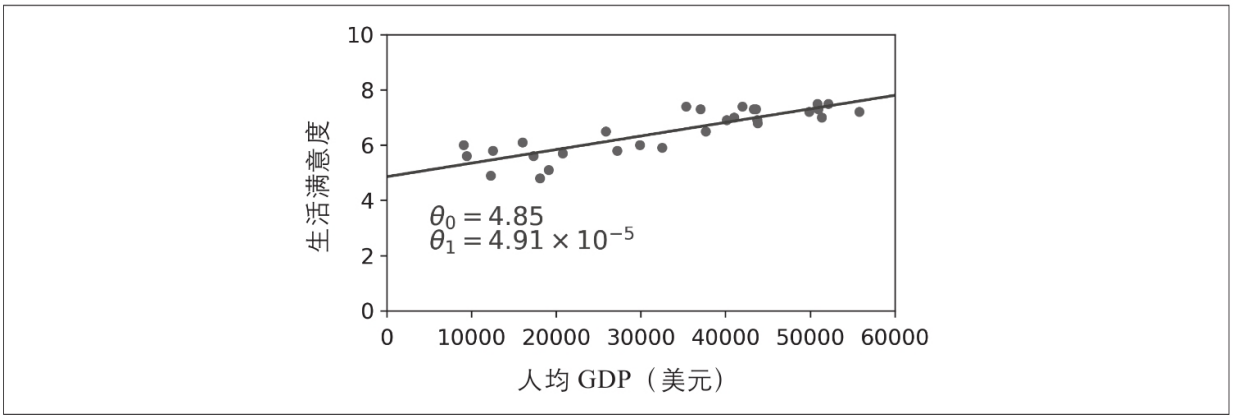


图1-19：最拟合训练数据的线性模型

现在终于可以运行模型来进行预测了。例如，你想知道塞浦路斯人有多幸福，但是经合组织的数据没有提供答案。幸好你有这个模型可以做出预测：先查查塞浦路斯的人均GDP是多少，发现是22 587美元，然后应用到模型中，发现生活满意度大约是4.85+22 587×4.91×10^{-5}=5.96。

为了激发你的兴趣，示例1-1是一段加载数据的Python代码，包括准备数据[6]，创建一个可视化的散点图，然后训练线性模型并做出预测[7]。

示例1-1：使用Scikit-Learn训练并运行一个线性模型

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.linear_model

# Load the data
oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv",thousands=',',delimiter='\t',
                             encoding='latin1', na_values="n/a")

# Prepare the data
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# Visualize the data
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()
```

```
# Select a linear model
model = sklearn.linear_model.LinearRegression()
# Train the model
model.fit(X, y)

# Make a prediction for Cyprus
X_new = [[22587]]  # Cyprus's GDP per capita
print(model.predict(X_new)) # outputs [[ 5.96242338]]
```

 如果使用基于实例的学习算法，你会发现斯洛文尼亚的人均GDP最接近塞浦路斯（20 732美元），而经合组织的数据告诉我们，斯洛文尼亚人的生活满意度是5.7，因此你很可能会预测塞浦路斯的生活满意度为5.7。如果稍微拉远一些，看看两个与之最接近的国家——葡萄牙和西班牙的生活满意度分别为5.1和6.5。取这三个数值的平均值，得到5.77，这也非常接近基于模型预测所得的值。这个简单的算法被称为k-近邻回归（在本例中，k=3）。

要将前面代码中的线性回归模型替换为k-近邻回归模型非常简单，只需要将下面这行代码：

```
import sklearn.linear_model
model = sklearn.linear_model.LinearRegression()
```

替换为：

```
import sklearn.neighbors
model = sklearn.neighbors.KNeighborsRegressor(
          n_neighbors=3)
```

如果一切顺利，你的模型将会做出很棒的预测。如果不行，则需要使用更多的属性（例如就业率、健康、空气污染等），获得更多或更高

质量的训练数据，或者选择一个更强大的模型（例如，多项式回归模型）。

简而言之：

• 研究数据。

• 选择模型。

• 使用训练数据进行训练（即前面学习算法搜索模型参数值，从而使成本函数最小化的过程）。

• 最后，应用模型对新示例进行预测（称为推断），希望模型的泛化结果不错。

以上就是一个典型的机器学习项目。在第2章中，你还将通过一个端到端的项目来体验这一切。

到目前为止，我们介绍了多个领域。你已经知道什么是真正的机器学习，它为何有用，机器学习系统最常见的类别有哪些，以及典型的项目工作流程。现在让我们看看在学习过程中可能会遇到哪些阻碍你做出准确预测的问题。

[1] 有趣的事实：这个奇怪的名字是Francis Galton在研究高个子的孩子往往比父母矮的事实时引入的一个统计术语。由于孩子比父母要矮一些，他称这种现象为回归到均值。该术语后来被他应用于分析变量之间相关性的方法。
[2] 某些神经网络架构可以是无监督的，例如自动编码器和受限玻尔兹曼机。它们也可以是半监督的，例如在深度信念网络和无监督的预训练中。
[3] 请注意，动物与车辆的隔离得很远，马与鹿的距离近却与鸟的距离远。图的使用得到了Richard Socher等人许可，“Zero-Shot Learning

Through Cross-Modal Transfer”，Proceedings of the 26th International Conference on Neural Information Processing Systems 1（2013）：935–943。

[4] 这是系统运行良好的情况。在实践中，它通常为每人创建几个集群，有时将两个看起来相似的人混合在一起，因此你可能需要为每个人提供一些标签并手动清理一些集群。

[5] 按照惯例，希腊字母 θ（theta）通常用于表示模型参数。

[6] prepare_country_stats（）函数的定义未在此处显示（如果需要所有详细信息，请参阅本章的Jupyter notebook）。这只是pandas代码，将OECD的生活满意度数据与IMF的人均GDP数据相结合。

[7] 如果你还不理解所有代码，没有关系，我们将在以下各章中介绍Scikit-Learn。

1.5 机器学习的主要挑战

简单来说，由于你的主要任务是选择一种学习算法，并对某些数据进行训练，所以最可能出现的两个问题不外乎是“坏算法”和“坏数据”，让我们先从坏数据开始。

1.5.1 训练数据的数量不足

要教一个牙牙学语的小朋友什么是苹果，你只需要指着苹果说“苹果”（可能需要重复这个过程几次）就行了，然后孩子就能够识别各种颜色和形状的苹果了，简直是天才！

机器学习还没达到这一步，大部分机器学习算法需要大量的数据才能正常工作。即使是最简单的问题，很可能也需要成千上万个示例，而对于诸如图像或语音识别等复杂问题，则可能需要数百万个示例（除非你可以重用现有模型的某些部分）。

数据的不合理有效性

在2001年发表的一篇著名论文中，微软研究员Michele Banko和Eric Brill表明，给定足够的数据，截然不同的机器学习算法（包括相当简单的算法）在自然语言歧义消除这个复杂问题上[1]，表现几乎完全一致（如图1-20所示）。

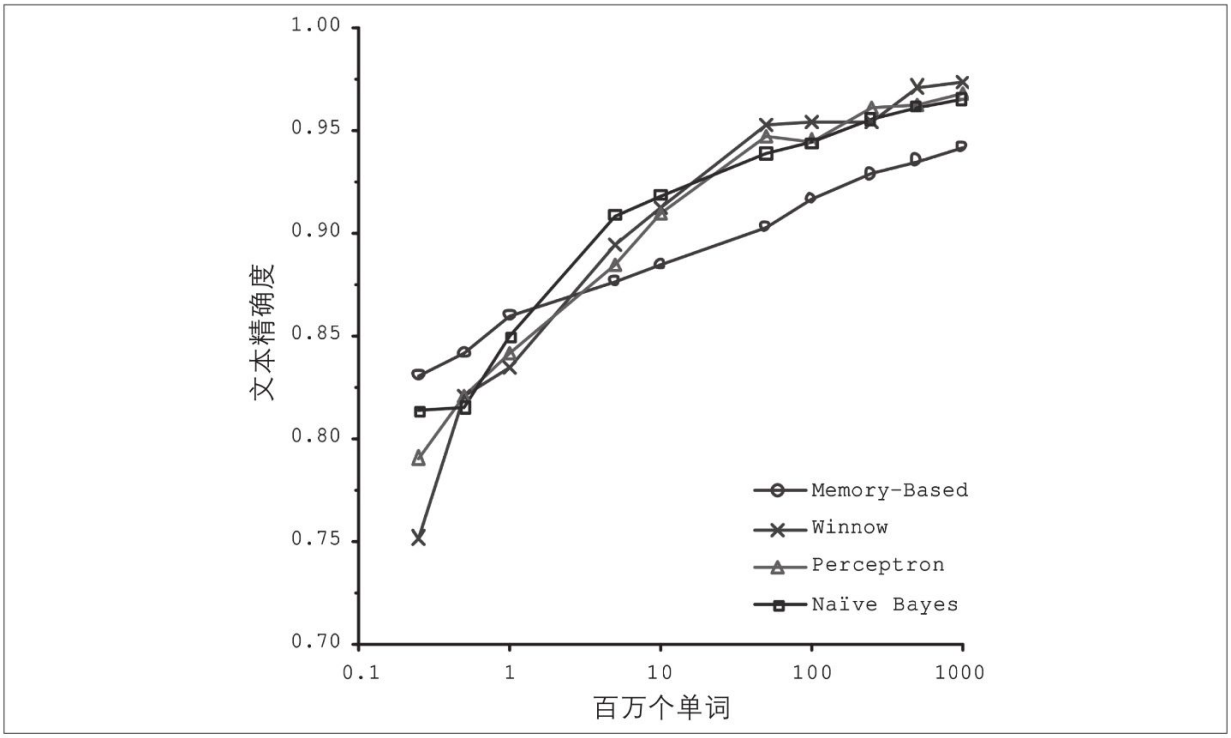


图1-20：数据与算法的重要性[2]

正如作者所说：“这些结果表明，我们可能会重新思考如何在二者之间做权衡——将钱和时间花在算法的开发上，还是花在语料库的建设上。”

对复杂问题而言，数据比算法更重要，这一想法被Peter Norvig等人进一步推广，于2009年发表论文“The Unreasonable Effectiveness of Data”[3]。不过需要指出的是，中小型数据集依然非常普遍，获得额外的训练数据并不总是一件轻而易举或物美价廉的事情，所以暂时先不要抛弃算法。

1.5.2 训练数据不具代表性

为了很好地实现泛化，至关重要的一点是对于将要泛化的新示例来说，训练数据一定要非常有代表性。无论你使用的是基于实例的学习还是基于模型的学习，都是如此。

例如，前面用来训练线性模型的国家数据集并不具备完全的代表性，有部分国家的数据缺失。图1-21显示了补充缺失国家信息之后的数据表现。

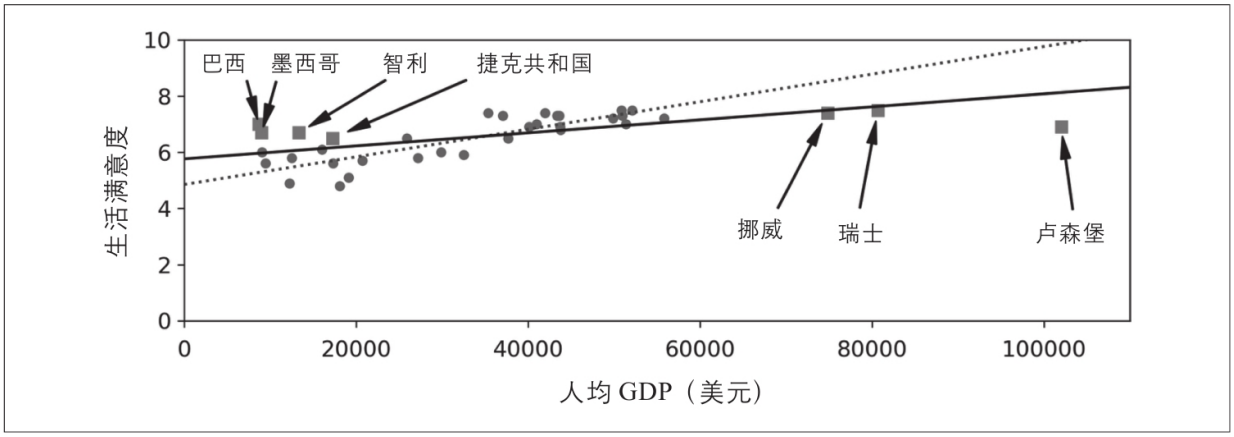


图1-21：更具代表性的训练样本

如果你用这个数据集训练线性模型，将会得到图中的实线，而虚线表示旧模型。正如你所见，添加部分缺失的国家信息不仅显著地改变了模型，也更清楚地说明这种简单的线性模型可能永远不会那么准确。看起来，某些非常富裕的国家并不比中等富裕的国家更幸福（事实上，看起来甚至是不幸福），反之，一些贫穷的国家也似乎比许多富裕的国家更加幸福。

使用不具代表性的训练集训练出来的模型不可能做出准确的预估，尤其是针对那些特别贫穷或特别富裕的国家。

针对你想要泛化的案例使用具有代表性的训练集，这一点至关重要。不过说起来容易，做起来难：如果样本集太小，将会出现采样噪声（即非代表性数据被选中）；而即便是非常大的样本数据，如果采样方式欠妥，也同样可能导致非代表性数据集，这就是所谓的采样偏差。

关于采样偏差的一个示例

最著名的采样偏差的示例发生在1936年美国总统大选期间，兰登对决罗斯福。Literary Digest当时举行了一次大范围的民意调查，向约1000万人发送邮件，并得到了240万个回复，因此做出了高度自信的预言——兰登将获得57%的选票。结果恰恰相反，罗斯福赢得了62%的选票。问题就在于Literary Digest的采样方式：

• 首先，为了获取发送民意调查的地址，Literary Digest采用了电话簿、杂志订阅名单、俱乐部会员名单等类似名簿。而所有这些名单上的人往往对富人有更大的偏好，也就更有可能支持共和党（即兰登）。

• 其次，收到民意调查邮件的人中，不到25%的人给出了回复。这再次引入了采样偏差，那些不怎么关心政治的人、不喜欢Literary Digest的人以及其他的一些关键群体直接被排除在外了。这是一种特殊类型的采样偏差，叫作无反应偏差。

再举一个示例，假设你想创建一个系统用来识别funk音乐视频。构建训练集的方法之一是直接在YouTube上搜索“funk music”，然后使用搜索结果的视频。但是，这其实基于一个假设——YouTube的搜索引擎返回的视频结果是所有能够代表funk音乐的视频。而实际的搜索结果可能会更偏向于当前流行的音乐人（如果你住在巴西，你会得到很多关于“funk carioca”的视频，这听起来跟James Brown完全不是一回事）。另一方面，你还能怎样获得一个大的训练集？

1.5.3 低质量数据

显然，如果训练集满是错误、异常值和噪声（例如，低质量的测量产生的数据），系统将更难检测到底层模式，更不太可能表现良好。所以花时间来清理训练数据是非常值得的投入。事实上，大多数数据科学家都会花费很大一部分时间来做这项工作。例如：

• 如果某些实例明显是异常情况，那么直接将其丢弃，或者尝试手动修复错误，都会大有帮助。

• 如果某些实例缺少部分特征（例如，5%的顾客没有指定年龄），你必须决定是整体忽略这些特征、忽略这部分有缺失的实例、将缺失的值补充完整（例如，填写年龄值的中位数），还是训练一个带这个特征的模型，再训练一个不带这个特征的模型。

1.5.4 无关特征

正如我们常说的：垃圾入，垃圾出。只有训练数据里包含足够多的相关特征以及较少的无关特征，系统才能够完成学习。一个成功的机器学习项目，其关键部分是提取出一组好的用来训练的特征集。这个过程叫作特征工程，包括以下几点：

• 特征选择（从现有特征中选择最有用的特征进行训练）。

• 特征提取（将现有特征进行整合，产生更有用的特征——正如前文提到的，降维算法可以提供帮助）。

• 通过收集新数据创建新特征。

现在我们已经看了不少“坏数据”的示例，再来看几个“坏算法”的示例。

1.5.5 过拟合训练数据

假设你正在国外旅游，被出租车司机敲诈，你很可能会说，那个国家的所有出租车司机都是强盗。过度概括是我们人类常做的事情，不幸的是，如果我们不小心，机器很可能也会陷入同样的陷阱。在机器学习中，这称为过拟合，也就是指模型在训练数据上表现良好，但是泛化时却不尽如人意。

图1-22显示了一个训练数据过拟合的高阶多项式生活满意度模型。虽然它在训练数据上的表现比简单的线性模型要好得多，但是你真的敢相信它的预测吗？

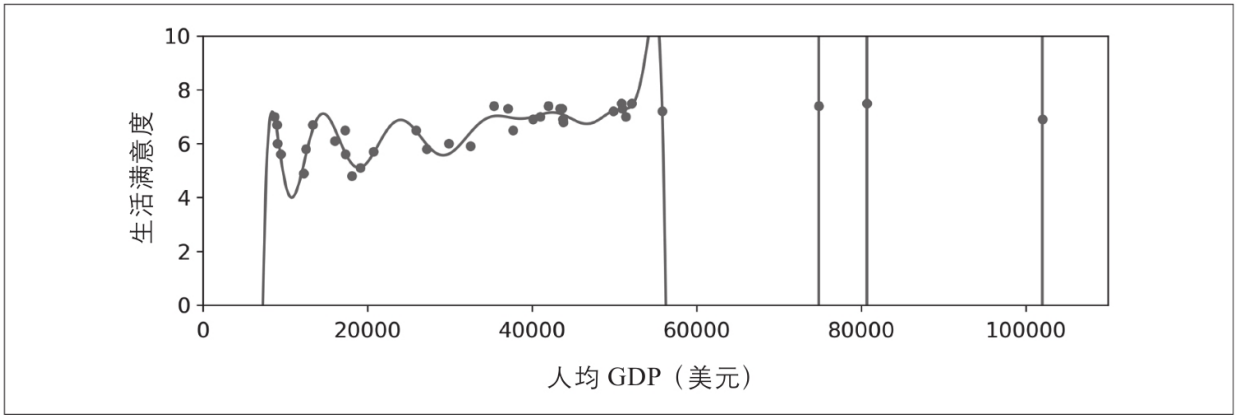


图1-22：过拟合训练数据

虽然诸如深度神经网络这类的复杂模型可以检测到数据中的微小模式，但是如果训练集本身是有噪声的，或者数据集太小（引入了采样噪声），那么很可能会导致模型检测噪声本身的模式。很显然，这些模式不能泛化至新的实例。举例来说，假设你给生活满意度模型提供了更多其他的属性，包括一些不具信息的属性（例如国家名）。在这种情况下，一个复杂模型可能会检测到这样的事实模式：训练数据中，名字中带有字母w的国家，如新西兰（New Zealand，生活满意度为7.3）、挪威（Norway，生活满意度为7.4）、瑞典（Sweden，生活满意度为7.2）和瑞士（Switzerland，生活满意度为7.5），生活满意度均大于7。当把这个w满意度规则泛化到卢旺达（Rwanda）或津巴布韦（Zim-babwe）时，你对结果有多大的自信？显然，训练数据中的这个模式仅仅是偶然产生的，但是模型无法判断这个模式是真实的还是噪声产生的结果。

 当模型相对于训练数据的数量和噪度都过于复杂时，会发生过拟合。可能的解决方案如下。

• 简化模型：可以选择较少参数的模型（例如，选择线性模型而不是高阶多项式模型）也可以减少训练数据中的属性数量，或者是约束模型。

• 收集更多的训练数据。

• 减少训练数据中的噪声（例如，修复数据错误和消除异常值）。

通过约束模型使其更简单，并降低过拟合的风险，这个过程称为正则化。例如，我们前面定义的线性模型有两个参数：θ_0和θ_1。因此，该算法在拟合训练数据时，调整模型的自由度就等于2，它可以调整线的高度（θ_0）和斜率（θ_1）。如果我们强行让$\theta_1=0$，那么算法的自由度将会降为1，并且拟合数据将变得更为艰难——它能做的全部就只是将线上移或下移来尽量接近训练实例，最后极有可能停留在平均值附近。这确实太简单了！如果我们允许算法修改θ_1，但是我们强制它只能是很小的值，那么算法的自由度将位于1和2之间，这个模型将会比自由度为2的模型稍微简单一些，同时又比自由度为1的模型略微复杂一些。你需要在完美匹配数据和保持模型简单之间找到合适的平衡点，从而确保模型能够较好地泛化。

图1-23显示了三个模型。点线表示的是在以圆圈表示的国家上训练的原始模型（没有正方形表示的国家），虚线是我们在所有国家（圆圈和方形）上训练的第二个模型，实线是用与第一个模型相同的数据训练的模型，但是有一个正则化约束。可以看到，正则化强制了模型的斜率较小：该模型与训练数据（圆圈）的拟合不如第一个模型，但它实际上更好地泛化了它没有在训练时看到的新实例（方形）。

在学习时，应用正则化的程度可以通过一个超参数来控制。超参数是学习算法（不是模型）的参数。因此，它不受算法本身的影响。超参数必须在训练之前设置好，并且在训练期间保持不变。如果将正则化超参数设置为非常大的值，会得到一个几乎平坦的模型（斜率接近零）。学习算法虽然肯定不会过拟合训练数据，但是也更加不可能找到一个好

的解决方案。调整超参数是构建机器学习系统非常重要的组成部分（将在第2章中详细举例）。

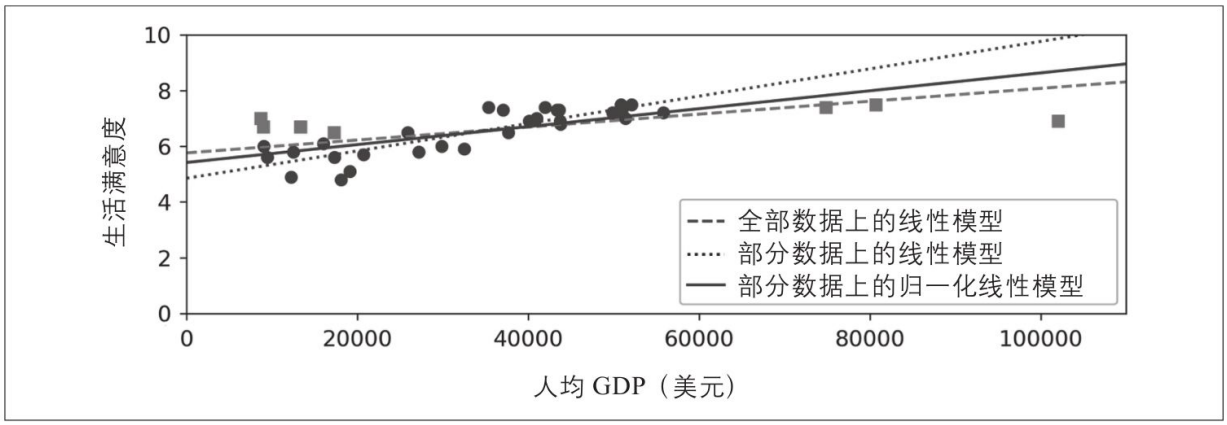


图1-23：正则化降低了过拟合的风险

1.5.6 欠拟合训练数据

你可能已经猜到了，欠拟合和过拟合正好相反。它的产生通常是因为对于底层的数据结构来说，你的模型太过简单。例如，用线性模型来描述生活满意度就属于欠拟合。现实情况远比模型复杂得多，所以即便是对于用来训练的示例，该模型产生的预测都一定是不准确的。

解决这个问题的主要方式有：

- 选择一个带有更多参数、更强大的模型。
- 给学习算法提供更好的特征集（特征工程）。
- 减少模型中的约束（例如，减少正则化超参数）。

1.5.7 退后一步

现在你已经对机器学习有了一定了解。不过讲了这么多概念，你可能有点晕，我们暂且退后一步，纵观一下全局：

• 机器学习是关于如何让机器可以更好地处理某些特定任务的理论，它从数据中学习，而无须清晰地编码规则。

• 机器学习系统有很多类型：有监督和无监督，批量的和在线的，基于实例的和基于模型的，等等。

• 在一个机器学习项目中，你从训练集中采集数据，然后将数据交给学习算法来计算。如果算法是基于模型的，它会调整一些参数来将模型适配于训练集（即对训练集本身做出很好的预测），然后算法就可以对新的场景做出合理的预测。如果算法是基于实例的，它会记住这些示例，并根据相似度度量将它们与所学的实例进行比较，从而泛化这些新实例。

• 如果训练集的数据太少或数据代表性不够，包含太多噪声或者被一些无关特征污染（垃圾进，垃圾出），那么系统将无法很好地工作。最后，你的模型既不能太简单（会导致欠拟合），也不能太复杂（会导致过拟合）。

还有最后一个要讲的重要主题是：一旦训练了一个模型，你就不能只是“希望”它可以正确地对新的场景做出泛化，你还需要评估它，必要时做出一些调整。现在我们看看怎么做到这一点。

[1] 例如，根据上下文知道是写“to”“two”还是“too”。
[2] 图经Michele Banko和Eric Brill许可转载，“Scaling to Very Very Large Corpora for Natural Language Disambiguation”，Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics（2001）：26－33.
[3] Peter Norvig et al.，“The Unreasonable Effectiveness of Data”，IEEE Intelligent Systems 24，no.2（2009）：8－12.

1.6 测试与验证

了解一个模型对于新场景的泛化能力的唯一办法就是让模型真实地去处理新场景。做法之一是将其部署在生产环境中，然后监控它的输出。这种做法不错，不过如果模型非常糟糕，你的用户就会抱怨，所以这显然不是最好的办法。

更好的选择是将数据分割成两部分：训练集和测试集。顾名思义，你可以用训练集的数据来训练模型，然后用测试集的数据来测试模型。应对新场景的误差率称为泛化误差（或者样例外误差），通过测试集来评估你的模型，就可以得到对这个误差的评估。这个估值可以告诉你模型在处理新场景时的能力如何。

如果训练误差很低（模型对于训练集来说很少出错），但是泛化误差很高，那么说明你的模型对于训练数据存在过拟合。

 通常将80%的数据用于训练，而保持20%供测试用。但是，这取决于数据集的大小。如果数据包含1000万个实例，那么保留1%意味着包含100 000个实例作为你的测试集，可能足以很好地估计泛化误差。

1.6.1 超参数调整和模型选择

评估一个模型很简单：用测试集就行了。现在假设你在两个模型（一个线性模型和一个多项式模型）之间犹豫不决，如何做出判断呢？做法是训练两个模型，然后对比它们对测试数据的泛化能力。

现在假设线性模型的泛化能力更强，但是你想要应用一些正则化来避免过拟合。问题又来了，要如何选择正则化超参数的值呢？做法

之一是使用100个不同的超参数值来训练100个不同的模型。然后假设你由此找到了最佳的超参数值，它生成的模型泛化误差最小，比如仅仅5%。然后在生产环境中运行这个模型，可是很不幸，它并没有如预期那样工作，反而产生了15%的误差。到底发生了什么？

问题出在你对测试集的泛化误差进行了多次度量，并且调整模型和超参数来得到拟合那个测试集的最佳模型。这意味着该模型对于新的数据不太可能有良好的表现。

解决此问题的常见方法称为保持验证：你只需保持训练集的一部分，以评估几种候选模型并选择最佳模型。新的保留集称为验证集，有时也称为开发集（dev set）。更具体地说，你可以在简化的训练集上（即完整训练集减去验证集）训练具有各种超参数的多个模型，并且选择在验证集上表现最佳的模型。在此保持验证之后，你在完整的训练集（包括验证集）上训练最佳模型，这就是你的最终模型。最后，你在测试集上评估这个模型以获得泛化误差的估计值。

这个解决方案通常效果很好。但是，如果验证集太小，则模型评估将不精确：你可能最终错误地选择一个次优模型。相反，如果验证集太大，则剩余训练集将比完整的训练集小得多。为什么这样不好？好吧，既然最终模型将在完整的训练集上训练，比较在更小的训练集上训练出来的候选模型不是一个好办法。就像选择最快的短跑运动员参加马拉松比赛。解决此问题的一种方法是执行使用许多小验证集重复进行的交叉验证。每个模型都在对其余数据进行训练后，在每个验证集上评估一次。通过对模型的所有评估求平均值，可以更准确地衡量模型的性能。但是有一个缺点：训练时间是验证集个数的倍数。

1.6.2　数据不匹配

在某些情况下，很容易获得大量训练数据，但是这些数据可能不能完全代表将用于生产环境的数据。

例如，假设你要创建一个移动App来拍摄花朵并自动确定其种类。你可以在网络轻松下载数以百万计的花朵图片，但它们并不能完美地代表在移动设备上使用该App拍摄的图片。也许你只有10 000张代表图片（即App实际拍摄的照片）。在这种情况下，最重要的规则是：验证集和测试集必须与在生产环境中使用的数据具有相同的代表性，因此它们应当由专用代表性图片组成：你可以将其混洗并一半放入验证集中，一半放入测试集中（确保两者不重复也不接近重复）。但是在网络图片上训练了模型之后，如果模型在验证集上的性能令人失望，那么你将不知道这是因为你的模型过拟合了训练集，还是只是由于网络图片和移动应用图片之间的不匹配。一种解决方案是将一些训练图片（网络上下载的）放到被吴恩达（Andrew Ng）称为train-dev（训练开发）集的另外一个集合中。训练模型后（在训练集而不是在train-dev集上），你可以在train-dev集上对其进行评估。如果模型表现良好，则不会过拟合训练集。如果在验证集上表现不佳，那么问题一定来自数据不匹配。你可以尝试通过预处理网络图片来使其看起来更像由移动应用拍摄的照片，然后重新训练模型。相反，如果模型在train-dev集上表现不佳，则它肯定在训练集上过拟合了，因此你应该尝试简化或规范化模型，获取更多训练数据，并清理训练数据。

没有免费的午餐定理

模型是观察的简化版。这个简化丢弃了那些不大可能泛化至新实例上的多余细节。但是，要决定丢弃哪些数据以及保留哪些数据，你必须要做出假设。例如，线性模型基于的假设就是数据基本上都是线性的，而实例与直线之间的距离都只是噪声，可以安全地忽略它们。

1996年David Wolpert在一篇著名论文中表明[1]，如果你对数据绝对没有任何假设，那么就没有理由更偏好于某个模型，这称为没有免费的午餐（No Free Lunch，NFL）定理。对某些数据集来说，最佳模型是线性模型，而对于其他数据集来说，最佳模型可能是神经网络模型。不存在一个先验模型能保证一定工作得更好（这正是定理名称的

由来）。想要知道哪个模型最好的方法就是对所有模型进行评估，但实际上这是不可能的，因此你会对数据做出一些合理的假设，然后只评估部分合理的模型。例如，对于简单的任务，你可能只会评估几个具有不同正则化水平的线性模型，而对于复杂问题，你可能会评估多个神经网络模型。

[1] David Wolpert，“The Lack of A Priori Distinctions Between Learning Algorithms”，Neural Computation 8，no.7（1996）：1341－1390.

1.7 练习题

本章中，我们提及了机器学习中最重要的一些概念。第2章将会进行更深入的探讨，也会写更多代码，但是在那之前，请先确保你已经知道如何回答下列问题：

1. 如何定义机器学习？

2. 机器学习在哪些问题上表现突出，你能给出四种类型吗？

3. 什么是被标记的训练数据集？

4. 最常见的两种监督学习任务是什么？

5. 你能举出四种常见的无监督学习任务吗？

6. 要让一个机器人在各种未知的地形中行走，你会使用什么类型的机器学习算法？

7. 要将顾客分成多个组，你会使用什么类型的算法？

8. 你会将垃圾邮件检测的问题列为监督学习还是无监督学习？

9. 什么是在线学习系统？

10. 什么是核外学习？

11. 什么类型的学习算法依赖相似度来做出预测？

12. 模型参数与学习算法的超参数之间有什么区别？

13.基于模型的学习算法搜索的是什么？它们最常使用的策略是什么？它们如何做出预测？

14.你能给出机器学习中的四个主要挑战吗？

15.如果模型在训练数据上表现很好，但是应用到新实例上的泛化结果却很糟糕，是怎么回事？能给出三种可能的解决方案吗？

16.什么是测试集，为什么要使用测试集？

17.验证集的目的是什么？

18.什么是train-dev集，什么时候需要它，怎么使用？

19.如果你用测试集来调超参数会出现什么错误？

以上练习题的答案见附录A。

第2章　端到端的机器学习项目

本章将介绍一个端到端的项目案例。假设你是一个房地产公司最近新雇用的数据科学家[1]，以下是你将会经历的主要步骤：

1. 观察大局。

2. 获得数据。

3. 从数据探索和可视化中获得洞见。

4. 机器学习算法的数据准备。

5. 选择并训练模型。

6. 微调模型。

7. 展示解决方案。

8. 启动、监控和维护系统。

[1] 项目案例纯属虚构，目的仅仅是为了说明机器学习项目的主要步骤，而不是为了了解房地产业务。

2.1 使用真实数据

学习机器学习最好使用真实数据进行实验，而不仅仅是人工数据集。我们有成千上万覆盖了各个领域的开放数据集可以选择。以下是一些可以获得数据的地方。

- 流行的开放数据存储库：

- UC Irvine Machine Learning Repository（http://archive.ics.uci.edu/ml/）

- Kaggle datasets（https://www.kaggle.com/datasets）

- Amazon’s AWS datasets（http://aws.amazon.com/fr/datasets/）

- 元门户站点（它们会列出开放的数据存储库）：

- Data Portals（http://dataportals.org/）

- OpenDataMonitor（http://opendatamonitor.eu/）

- Quandl（http://quandl.com/）

- 其他一些列出许多流行的开放数据存储库的页面：

- Wikipedia’s list of Machine Learning datasets（https://goo.gl/SJHN2k）

- Quora.com（http://goo.gl/zDR78y）

• The datasets subreddit（https://www.reddit.com/r/datasets）

本章我们从StatLib库中选择了加州住房价格的数据集（见图2-1）[1]。该数据集基于1990年加州人口普查的数据。虽然不算是最新的数据（当时你还能负担得起一个湾区的好房子），但是有很多可以学习的特质，所以我们就假定这是最新的数据吧。出于教学目的，我们还特意添加了一个分类属性，并且移除了一些特征。

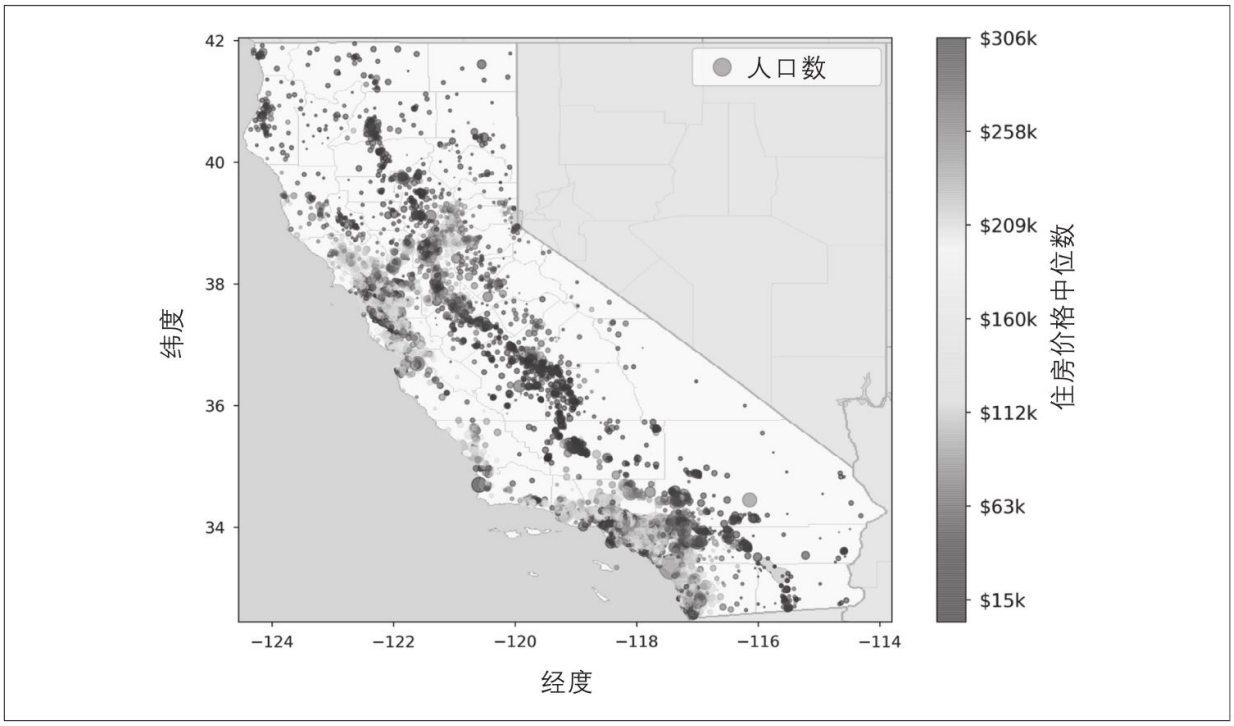


图2-1：加州住房价格

[1] 原始数据集由R.Kelley Pace和Ronald Barry提供，“Sparse Spatial Autoregressions”，Statistics&Probability Letters 33，no.3（1997）：291－297。

2.2 观察大局

欢迎来到机器学习房产公司！你要做的第一件事是使用加州人口普查的数据建立起加州的房价模型。数据中有许多指标，诸如每个街区的人口数量、收入中位数、房价中位数等。街区是美国人口普查局发布样本数据的最小地理单位（一个街区通常人口数为600到3000人）。这里，我们将其简称为“区域”。

你的模型需要从这个数据中学习，从而能够根据所有其他指标，预测任意区域的房价中位数。

 如果你是一名习惯良好的数据科学家，要做的第一件事应该是拿出机器学习项目清单。你可以从附录B中的清单项开始，它适合绝大多数机器学习项目，但还是要确保它满足你的需求。本章我们将会讨论这个清单中的部分内容，但也会跳过一部分，有些是因为不需要多做解释，有些是因为在后面的章节中会展开讨论。

2.2.1 框架问题

你问老板的第一个问题应该是业务目标是什么，因为建立模型本身可能不是最终的目标。公司期望知道如何使用这个模型，如何从中获益？这才是重要的问题，因为这将决定你怎么设定问题，选择什么算法，使用什么测量方式来评估模型的性能，以及应该花多少精力来进行调整。

老板回答说，这个模型的输出（对一个区域房价中位数的预测）将会跟其他许多信号一起被传输给另一个机器学习系统（见图2-2）[1]。而这个下游系统将被用来决策一个给定的区域是否值得投资。因为直接影响到收益，所以正确获得这个信息至关重要。

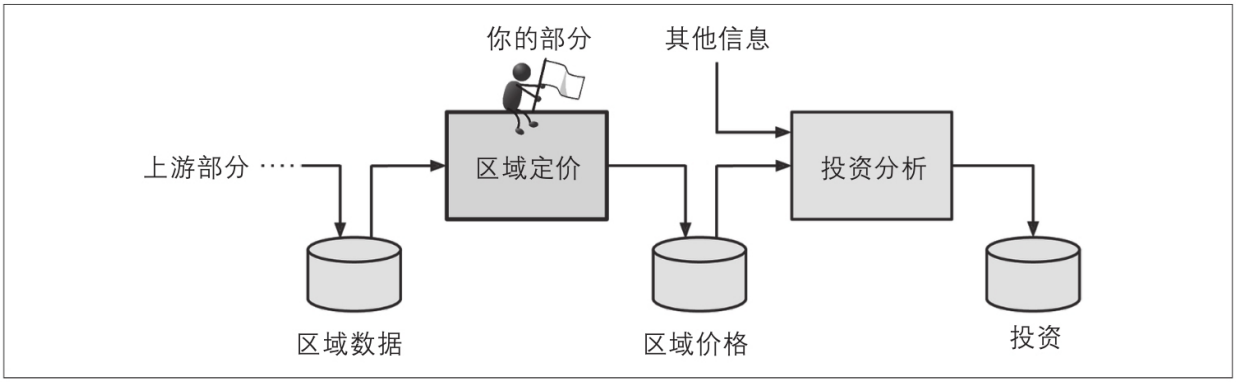


图2-2：一个针对房地产投资的机器学习流水线

流水线

一个序列的数据处理组件称为一个数据流水线。流水线在机器学习系统中非常普遍，因为需要大量的数据操作和数据转化才能应用。

组件通常是异步运行的。每个组件拉取大量的数据，然后进行处理，再将结果传输给另一个数据仓库。一段时间之后，流水线中的下一个组件会拉取前面的数据，并给出自己的输出，以此类推。每个组件都很独立：组件和组件之间的连接只有数据仓库。这使得整个系统非常简单易懂（在数据流图表的帮助下），不同团队可以专注于不同的组件。如果某个组件发生故障，它的下游组件还能使用前面的最后一个输出继续正常运行（至少一段时间），所以使得整体架构鲁棒性较强。

当然，从另一方面来说，如果没有实施适当的监控，坏掉的组件可能在一段时间内都无人发现，那么过期数据会导致整个系统的性能下降。

要向老板询问的第二个问题是当前的解决方案（如果有的话）。你可以将其当作参考，也能从中获得解决问题的洞察。老板回答说，现在是由专家团队在手动估算区域的住房价格：一个团队持续收集最新的区域信息，当他们不能得到房价中位数时，便使用复杂的规则来进行估算。

这个过程既昂贵又耗时，而且估算结果还不令人满意，在某些情况下，他们估计的房价和实际房价的偏差高达20%。这就是为什么该公司认为给定该区域的其他数据有助于训练模型来预测该区域的房价中位数。普查数据看起来像是一个可用于此目的的很好的数据集，因为它包括数千个区域的房价中位数和其他数据。

有了这些信息，你现在可以开始设计系统了。首先，你需要回答框架问题：是有监督学习、无监督学习还是强化学习？是分类任务、回归任务还是其他任务？应该使用批量学习还是在线学习技术？在继续阅读之前，请先暂停一会儿，尝试回答一下这些问题。

找到答案了吗？我们来看看：显然，这是一个典型的有监督学习任务，因为已经给出了标记的训练示例（每个实例都有预期的产出，也就是该区域的房价中位数）。并且这也是一个典型的回归任务，因为你要对某个值进行预测。更具体地说，这是一个多重回归问题，因为系统要使用多个特征进行预测（使用区域的人口、收入中位数等）。这也是一元回归问题，因为我们仅尝试预测每个区域的单个值。如果我们试图预测每个区域的多个值，那将是多元回归问题。最后，我们没有一个连续的数据流不断流进系统，所以不需要针对变化的数据做出特别调整，数据量也不是很大，不需要多个内存，所以简单的批量学习应该就能胜任。

 如果数据庞大，则可以跨多个服务器拆分批处理学习（使用MapReduce技术）或使用在线学习技术。

2.2.2　选择性能指标

下一步是选择性能指标。回归问题的典型性能指标是均方根误差（RMSE）。它给出了系统通常会在预测中产生多大误差，对于较大的误差，权重较高。公式2-1给出了计算RMSE的数学公式。

公式2-1：均方根误差（RMSE）

$$\mathrm{RMSE}(\boldsymbol{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h(\boldsymbol{x}^{(i)}) - y^{(i)}\right)^2}$$

符号表示

这个公式引入了我们在本书中将使用的几种常见的机器学习符号：

• m是要在其上测量RMSE的数据集中的实例数。

• 例如，如果你要在2000个区域的验证集上评估RMSE，则m=2000。

• $x^{(i)}$是数据集中第i个实例的所有特征值（不包括标签）的向量，而$y^{(i)}$是其标签（该实例的期望输出值）。

• 例如，如果数据集中的第一个区域位于经度-118.29°，纬度33.91°，居民1416人，收入中位数为38 372美元，房屋价值中位数为156 400美元（忽略其他特征），那么

$$\boldsymbol{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1416 \\ 38\,372 \end{pmatrix}$$

和

$$y^{(1)} = 156\,400$$

• X是一个矩阵，其中包含数据集中所有实例的所有特征值（不包括标签）。每个实例只有一行，第i行等于$x^{(i)}$的转置，记为$(x^{(i)})^T$[2]。

• 例如，如果第一个区域如上所述，则矩阵X如下所示：

$$\boldsymbol{X} = \begin{pmatrix} (\boldsymbol{x}^{(1)})^T \\ (\boldsymbol{x}^{(2)})^T \\ \vdots \\ (\boldsymbol{x}^{(1999)})^T \\ (\boldsymbol{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1416 & 38\,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

• h是系统的预测函数，也称为假设。当给系统输入一个实例的特征向量$x^{(i)}$时，它会为该实例输出一个预测值$\hat{y}^{(i)} = h(\boldsymbol{x}^{(i)})$（$\hat{y}$读为“y-帽”）。

• 例如，如果系统预测第一个区域的房价中位数为158 400美元，则$\hat{y}^{(1)} = h(\boldsymbol{x}^{(1)}) =$158 400。该区域的预测误差为$\hat{y}^{(1)} - y^{(1)} = 2000$。

• RMSE（X，h）是使用假设h在一组示例中测量的成本函数。

我们将小写斜体字体用于标量值（例如m或$y^{(i)}$）和函数名称（例如h），将小写粗斜体字体用于向量（例如$x^{(i)}$），将大写粗斜体字体用于矩阵（例如X）。

尽管RMSE通常是回归任务的首选性能指标，但在某些情况下，你可能更喜欢使用其他函数。例如，假设有许多异常区域。在这种情况下，你可以考虑使用平均绝对误差（Mean Absolute Error，MAE，也称为平均绝对偏差。参见公式2-2）：

公式2-2：平均绝对误差（MAE）

$$\mathrm{MAE}(\boldsymbol{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left| h(\boldsymbol{x}^{(i)}) - y^{(i)} \right|$$

RMSE和MAE都是测量两个向量（预测值向量和目标值向量）之间距离的方法。各种距离度量或范数是可能的：

• 计算平方和的根（RMSE）与欧几里得范数相对应：这是你熟悉的距离的概念，它也称为ℓ_2范数，记为$||\cdot||_2$（或仅记为$||\cdot||$）。

• 计算绝对值之和（MAE）对应于ℓ_1范数，记为$||\cdot||_1$。有时将其称为“曼哈顿范数”，因为如果你只能沿着正交的城市街区行动，它测量城市中两点之间的距离。

• 一般而言，包含n个元素的向量v的ℓ_k范数定义为$\|\boldsymbol{v}\|_k = (|v_0|^k + |v_1|^k + \cdots + |v_n|^k)^{1/k}$。$\ell_0$给出了向量中的非零元素数量，$\ell_\infty$给出向量中的最大绝对值。

• 范数指标越高，它越关注大值而忽略小值。这就是RMSE对异常值比MAE更敏感的原因。但是，当离群值呈指数形式稀有时（如钟形曲线），RMSE表现非常好，通常是首选。

2.2.3 检查假设

最后，列举和验证到目前为止（由你或者其他人）做出的假设，是一个非常好的习惯。这可以在初期检查出严重问题。例如，当机器学习系统输出区域价格给下游系统时，我们的假设是价格会被使用。但是，如果下游系统实际上是将价格转换成为类别（例如，廉价、中等或者昂贵），转而使用这些类别，而不是价格本身呢？在这种情形下，并不需要完全准确地预估价格，你的系统只需要得出正确的类别就够了。如果是这种情况，那么这个问题应该被设定为分类任务而不是回归任务。你肯定不会愿意在回归系统上努力了几个月之后才发现这一点。

幸运的是，跟下游系统的团队聊过之后，证实需要的确实是价格而不是类别。很好！一切就绪，绿灯亮了，现在可以开始编程了！

[1] 参考Claude Shannon的信息论，他将这种信息（信号）馈送到机器学习系统中，该信息论是他在贝尔实验室开发的，用于提高通讯质量。他的理论：你想要一个高信噪比。
[2] 回忆一下，转置运算符将列向量翻转为行向量（反之亦然）。

2.3 获取数据

现在是着手动工的时候了。不要犹豫，打开你的笔记本电脑，先过一遍Jupyter notebook里的代码示例。完整的Jupyter notebook可以通过https://github.com/ageron/handson-ml2获得。

2.3.1 创建工作区

首先，你需要安装Python。你可能早已经装好，如果还没有，那么可以在https://www.python.org/上获取[1]。接下来，你需要为机器学习的代码和数据集创建一个工作区目录。打开一个终端并输入以下命令行（在$提示符之后）：

```
$ export ML_PATH="$HOME/ml" # You can change the path if you prefer
$ mkdir -p $ML_PATH
```

此外，你还需要一些Python模块：Jupyter、NumPy、pandas、Matplotlib以及ScikitLearn。如果已经安装好所有这些模块，并运行了Jupyter，那么可以跳到2.3.2节。如果还没有完全安装所有模块（以及它们的依赖项），以下有多种办法进行安装。可以使用系统的包管理器进行安装（例如，Ubuntu的apt-get，或者MacOS上的MacPorts和Homebrew等），安装一个Scientific Python的发行版（例如Anaconda），然后使用其包管理器，又或者直接使用Python自己的包管理器pip，默认情况下，（自2.7.9版本之后）Python的二进制安装程序里应该都包含pip[2]。你可以输入以下命令行查看是否安装了pip：

```
$ python3 -m pip --version
pip 19.3.1 from [...]/lib/python3.7/site-packages/pip (python 3.7)
```

请确保安装的pip版本是最新的，要升级pip模块，请输入（实际版本可能有所不同）[3]：

```
$ python3 -m pip install --user -U pip2
Collecting pip
[...]
Successfully installed pip-19.3.13
```

创建一个隔离环境

如果你希望在一个隔离的环境里工作（强烈推荐，这样你可以在库版本不冲突的情况下处理不同的项目），可以通过运行以下pip命令来安装virtualenv[4]（如果你想在你的机器上为所有用户安装virtualenv，去掉--user并用管理员权限运行命令）：

```
$ python3 -m pip install --user -U virtualenv
Collecting virtualenv
[...]
Successfully installed virtualenv-16.7.6
```

输入以下命令创建一个隔离的Python环境：

```
$ cd $ML_PATH
$ python3 -m virtualenv my_env
Using base prefix '[...]'
New python executable in [...]/ml/my_env/bin/python3
Also creating executable in [...]/ml/my_env/bin/python
Installing setuptools, pip, wheel...done.
```

现在开始，每当想要激活这个环境时，只需要打开终端并输入：

```
$ cd $ML_PATH
$ source my_env/bin/activate # on Linux or macOS
$ .\my_env\Scripts\activate # on Windows
```

要停用这个环境，用deactivate命令。当这个环境处于激活状态时，使用pip安装的任何软件包都将被安装在这个隔离的环境中，Python只拥有这些包的访问权限（如果你还希望访问系统站点的软件包，则需要使用virtualenv的--system-sitepackages命令选项）。更多详情可以参考virtualenv的文档。

现在可以使用简单的pip命令安装必需的模块及其依赖项了（如果没有使用virtualenv，你将需要--user选项或管理员权限）：

```
$ python3 -m pip install -U jupyter matplotlib numpy pandas scipy scikit-learn
Collecting jupyter
  Downloading https://[...]/jupyter-1.0.0-py2.py3-none-any.whl
Collecting matplotlib
  [...]
```

如果你创建了virtualenv，则需要注册到Jupyter并给它一个名字：

```
$ python3 -m ipykernel install --user --name=python3
```

现在可以输入以下命令来启动Jupyter：

```
$ jupyter notebook
[...] Serving notebooks from local directory: [...]/ml
[...] The Jupyter Notebook is running at:
[...] http://localhost:8888/?token=60995e108e44ac8d8865a[...]
[...]  or http://127.0.0.1:8889/?token=60995e108e44ac8d8865a[...]
[...] Use Control-C to stop this server and shut down all kernels [...]
```

一个Jupyter服务器正在你的终端中运行，监听端口为8888。你可以打开浏览器，输入http://localhost:8888/访问该服务器（通常服务器启动时会自动运行），可以看到一个空的工作区目录（如果你遵循了上述virtualenv指令，这时仅包含env目录）。

单击New按钮，选择适当的Python版本，创建一个Python notebook（见图2-3）[5]。它会在你的工作区中创建一个名为Untitled.ipynb的新notebook文件，然后启动Jupyter Python内核来运行这个文件，最后在浏览器新标签里打开这个笔记本。先单击Untitled并键入新名称将这个笔记本重命名为“Housing”（文件将自动被重命名为Housing.ipynb）。

一个notebook内包含一个单元格列表，每个单元格可以是可执行代码或格式化的文本。现在，这个notebook只有一个空的代码单元，标记为“In[1]：”。在单元格中输入print（"Hello world!"），然后单击执行按钮（参见图2-4）或按组合键“Shift+Enter”。这时，当前单元格会被送到这个notebook的Python内核，运行并返回输出结果。结果显示在单元格下方，同时因为我们到了notebook的末尾，所以还会自动创建一个新的单元格。更多的基础知识可以从Jupyter的Help菜单“User Interface Tour”中了解。

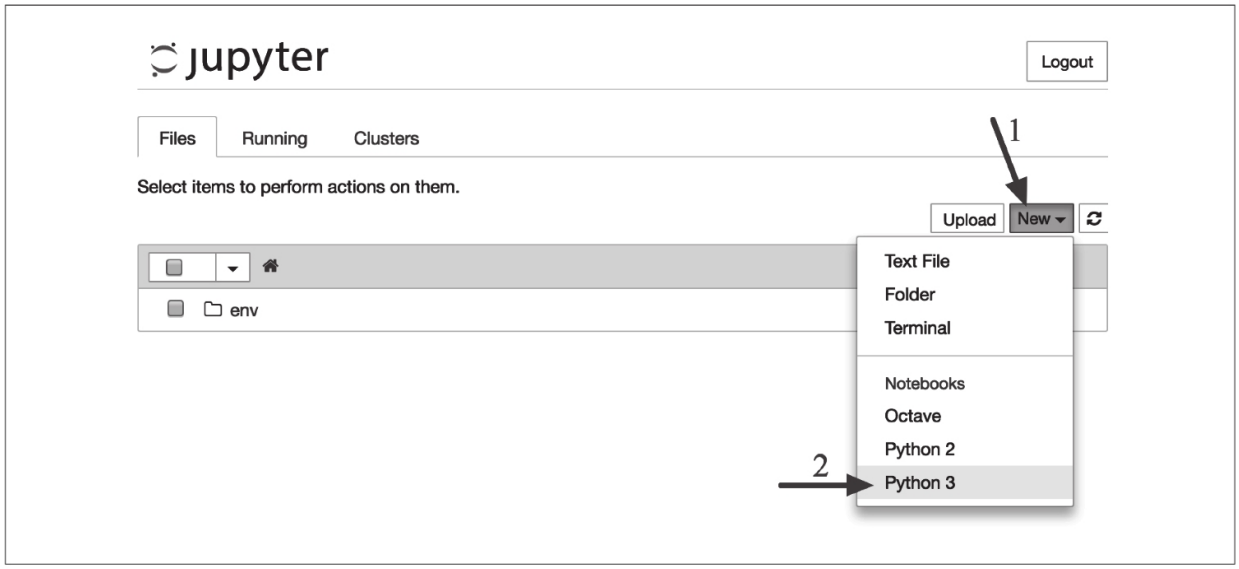


图2-3：你的Jupyter工作间

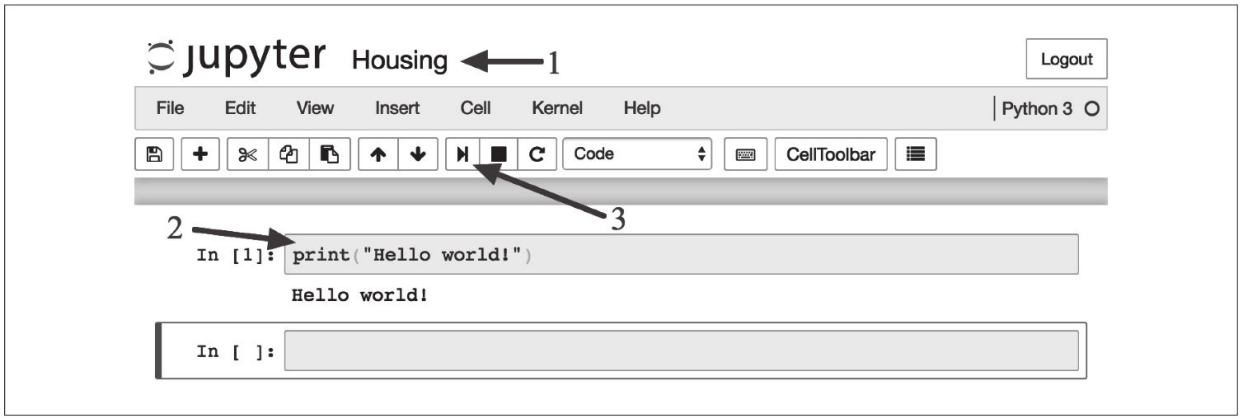


图2-4：Hello world Python notebook

2.3.2 下载数据

在典型环境中，数据存储在关系型数据库里（或其他一些常用数据存储），并分布在多个表/文档/文件中。访问前，你需要先获得证书和访问权限[6]，并熟悉数据库模式。不过在这个项目中，事情要简单得多：你只需要下载一个压缩文件housing.tgz即可，这个文件已经包含所有的数据——一个以逗号来分隔值的CSV文档housing.csv。

你可以选择使用浏览器下载压缩包，运行tar xzf housing.tgz来解压缩并提取CSV文件，但更好的选择是创建一个小函数来实现它。尤其是当数据会定期发生变化时，这个函数非常有用：你可以编写一个小脚本，在需要获取最新数据时直接运行（或者也可以设置一个定期自动运行的计划任务）。如果需要在多台机器上安装数据集，这个自动获取数据的函数也非常好用。

获取数据[7]的函数如下所示：

```
import os
import tarfile
import urllib
```

```
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    os.makedirs(housing_path, exist_ok=True)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

现在，每当你调用fetch_housing_data（），就会自动在工作区中创建一个datasets/housing目录，然后下载housing.tgz文件，并将housing.csv解压到这个目录。

现在我们来使用pandas加载数据。你也应该写一个小函数来加载数据：

```
import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

这个函数会返回一个包含所有数据的pandas DataFrame对象。

2.3.3 快速查看数据结构

我们来看看使用DataFrames的head（）方法之后的前5行是怎样的（见图2-5）。

```
In [5]: housing = load_housing_data()
        housing.head()
```

Out[5]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	populatio
0	-122.23	37.88	41.0	880.0	129.0	322.0
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0
2	-122.24	37.85	52.0	1467.0	190.0	496.0
3	-122.25	37.85	52.0	1274.0	235.0	558.0
4	-122.25	37.85	52.0	1627.0	280.0	565.0

图2-5：数据集中的前5行

每一行代表一个区域，总共有10个属性（在图2-5中可以看到前6个）：longitude、latitude、housing_median_age、total_rooms、total_bedrooms、population、households、median_income、median_house_value以及ocean_proximity。

通过info（）方法可以快速获取数据集的简单描述，特别是总行数、每个属性的类型和非空值的数量（见图2-6）。

```
In [6]: housing.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        20433 non-null float64
population            20640 non-null float64
households            20640 non-null float64
median_income         20640 non-null float64
median_house_value    20640 non-null float64
ocean_proximity       20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

图2-6：住房信息

数据集中包含20 640个实例，以机器学习的标准来看，这个数字非常小，但却是个完美的开始。注意，total_bedrooms这个属性只有20

433个非空值，这意味着有207个区域缺失这个特征。我们后面需要考虑到这一点。

所有属性的字段都是数字，除了ocean_proximity。它的类型是object，因此它可以是任何类型的Python对象，不过因为你从CSV文件中加载了该数据，所以它必然是文本属性。通过查看前5行，你可能会注意到，该列中的值是重复的，这意味着它有可能是一个分类属性。你可以使用value_counts（）方法查看有多少种分类存在，每种类别下分别有多少个区域：

```
>>> housing["ocean_proximity"].value_counts()
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

再来看看其他区域，通过describe（）方法可以显示数值属性的摘要（见图2-7）。

count、mean、min以及max行的意思很清楚。需要注意的是，这里的空值会被忽略（因此本例中，total_bedrooms的count是20 433而不是20 640）。std行显示的是标准差（用来测量数值的离散程度）[8]。25%、50%和75%行显示相应的百分位数：百分位数表示一组观测值中给定百分比的观测值都低于该值。例如，对于housing_median_age的值，25%的区域小于18，50%的区域小于29，以及75%的区域小于37。这些通常分别称为第25百分位数（或者第一四分位数）、中位数以及第75百分位数（或者第三四分位数）。

```
In [8]: housing.describe()
```

Out[8]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrc
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.0000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.385070
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.00000

图2-7：每个数值属性的摘要

另一种快速了解数据类型的方法是绘制每个数值属性的直方图。直方图用来显示给定值范围（横轴）的实例数量（纵轴）。你可以一次绘制一个属性，也可以在整个数据集上调用hist（）方法（如以下代码示例所示），绘制每个属性的直方图（见图2-8）。

```
%matplotlib inline   # only in a Jupyter notebook
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

hist（）方法依赖于Matplotlib，而Matplotlib又依赖于用户指定的图形后端才能在屏幕上完成绘制。所以在绘制之前，你需要先指定Matplotlib使用哪个后端。最简单的选择是使用Jupyter的神奇命令%matplotlib inline。它会设置Matplotlib从而使用Jupyter自己的后端，随后图形会在notebook上呈现。需要注意的是，因为Jupyter在执行每个单元格时会自动显示图形，所以在Jupyter notebook中调用show（）是可选的。

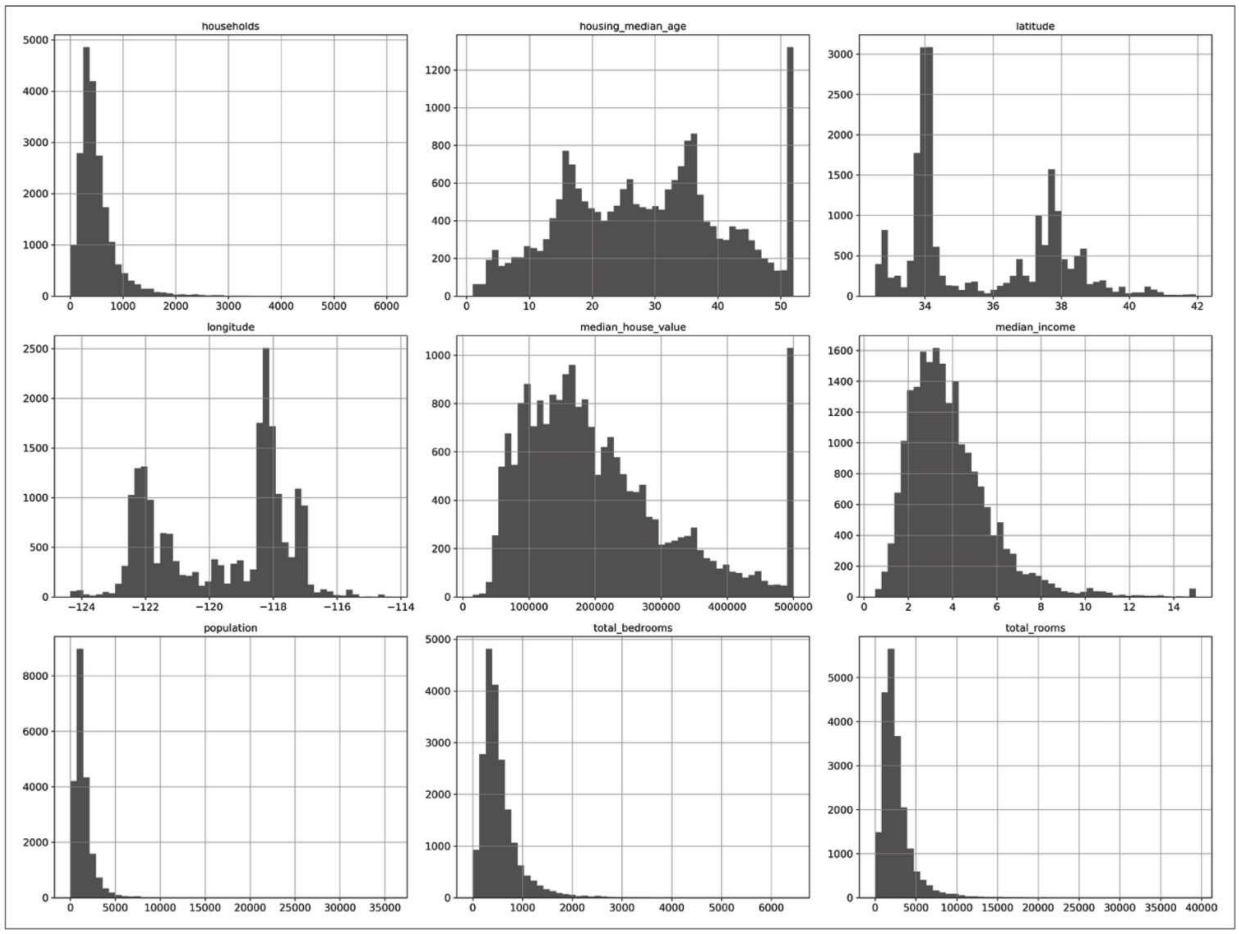


图2-8：每个数值属性的直方图

回到直方图，请注意以下几点：

1.首先，收入中位数这个属性看起来不像是用美元（USD）在衡量。经与收集数据的团队核实，你得知数据已经按比例缩小，并框出中位数的上限为15（实际为15.0001），下限为0.5（实际为0.4999）。数字后的单位为万美元，例如，15代表15万美元。在机器学习中，使用经过预处理的属性是很常见的事情，倒不一定是个问题，但是你至少需要了解数据是如何计算的。

2.房龄中位数和房价中位数也被设定了上限。而由于后者正是你的目标属性（标签），因此这可能是个大问题。你的机器学习算法很可能会学习到价格永远不会超过这个限制。你需要再次与客户团队（使用你的系统输出的团队）进行核实，查看是否存在问题。如果他们告诉你，

他们需要精确的预测值，甚至可以超过50万美元，那么，通常你有两个选择：

a. 对那些标签值被设置了上限的区域，重新收集标签值。

b. 将这些区域的数据从训练集中移除（包括从测试集中移除，因为如果预测值超过50万美元，系统不应被评估为不良）。

3. 这些属性值被缩放的程度各不相同。这将在本章稍后探讨特征缩放时再做讨论。

4. 最后，许多直方图都表现出重尾：图形在中位数右侧的延伸比左侧要远得多。这可能会导致某些机器学习算法难以检测模式。稍后我们会尝试一些转化方法，将这些属性转化为更偏向钟形的分布。

相信现在你对正在处理的数据应该有了更好的理解。

 等等！在进一步查看数据之前，你需要先创建一个测试集，然后即可将其放置一旁，不用过多理会。

2.3.4 创建测试集

在这个阶段主动搁置部分数据听起来可能有些奇怪。毕竟，你才只简单浏览了一下数据而已，在决定用什么算法之前，当然还需要了解更多的知识，对吧？没错，但是大脑是个非常神奇的模式检测系统，也就是说，它很容易过拟合：如果是你本人来浏览测试集数据，很可能会跌入某个看似有趣的测试数据模式，进而选择某个特殊的机器学习模型。然后当你再使用测试集对泛化误差率进行估算时，估计结果将会过于乐观，该系统启动后的表现将不如预期那般优秀。这称为数据窥探偏误（data snooping bias）。

理论上，创建测试集非常简单，只需要随机选择一些实例，通常是数据集的20%（如果数据集很大，比例将更小），然后将它们放在一边：

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

你可以这样使用如下函数[9]：

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

是的，这确实能行，但这并不完美。如果再运行一遍，它又会产生一个不同的数据集！这样下去，你（或者你的机器学习算法）将会看到整个完整的数据集，而这正是创建测试集时需要避免的。

解决方案之一是在第一次运行程序后即保存测试集，随后的运行只是加载它而已。另一种方法是在调用np.random.permutation（）之前设置一个随机数生成器的种子（例如，np.random.seed（42））[10]，从而让它始终生成相同的随机索引。

但是，这两种解决方案在下一次获取更新的数据时都会中断。为了即使在更新数据集之后也有一个稳定的训练测试分割，常见的解决方案是每个实例都使用一个标识符来决定是否进入测试集（假定每个实例都有一个唯一且不变的标识符）。例如，你可以计算每个实例标识符的哈

希值，如果这个哈希值小于或等于最大哈希值的20%，则将该实例放入测试集。这样可以确保测试集在多个运行里都是一致的，即便更新数据集也仍然一致。新实例的20%将被放入新的测试集，而之前训练集中的实例也不会被放入新测试集。

实现方式如下：

```
from zlib import crc32

def test_set_check(identifier, test_ratio):
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32

def split_train_test_by_id(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

不幸的是，housing数据集没有标识符列。最简单的解决方法是使用行索引作为ID：

```
housing_with_id = housing.reset_index()   # adds an `index` column
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

如果使用行索引作为唯一标识符，你需要确保在数据集的末尾添加新数据，并且不会删除任何行。如果不能保证这一点，那么你可以尝试使用某个最稳定的特征来创建唯一标识符。例如，一个区域的经纬度肯定几百万年都不会变，你可以将它们组合成如下的ID[11]

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

Scikit-Learn提供了一些函数，可以通过多种方式将数据集分成多个子集。最简单的函数是train_test_split（），它与前面定义的函数split_train_test（）几乎相同，除了几个额外特征。首先，它也有random_state参数，让你可以像之前提到过的那样设置随机生成器种子；其次，你可以把行数相同的多个数据集一次性发送给它，它会根据相同的索引将其拆分（例如，当你有一个单独的DataFrame用于标记时，这就非常有用）：

```
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

到目前为止，我们思考过了纯随机的抽样方法。如果数据集足够庞大（特别是相较于属性的数量而言），这种方式通常不错；如果不是，则有可能会导致明显的抽样偏差。如果一家调查公司想要打电话给1000个人来调研几个问题，他们不会在电话簿中纯随机挑选1000个人。他们试图确保让这1000人能够代表全体人口。例如，美国人口组成为51.3%的女性和48.7%的男性，所以，要想在美国进行一场有效的调查，样本中应该试图维持这一比例，即513名女性和487名男性。这就是分层抽样：将人口划分为均匀的子集，每个子集称为一层，然后从每层抽取正确的实例数量，以确保测试集合代表了总的人口比例。如果使用纯随机的抽样方法，将有12%的可能得到抽样偏斜的测试集——要么女性比例不到49%，要么女性比例超过54%。不论出现哪种情况都会导致调查结果出现显著偏差。

如果你咨询专家，他们会告诉你，要预测房价中位数，收入中位数是一个非常重要的属性。于是你希望确保在收入属性上，测试集能够代表整个数据集中各种不同类型的收入。由于收入中位数是一个连续的数值属性，所以你得先创建一个收入类别的属性。我们先来看一下收入中位数的直方图（见图2-8）：大多数收入中位数值聚集在1.5～6（15 000～60 000美元）左右，但也有一部分远远超过了6万美元。在数据集

中，每一层都要有足够数量的实例，这一点至关重要，不然数据不足的层，其重要程度很有可能会被错估。也就是说，你不应该将层数分得太多，每一层应该要足够大才行。下面这段代码是用pd.cut（）来创建5个收入类别属性的（用1～5来做标签），0～1.5是类别1，1.5～3是类别2，以此类推：

```
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])
```

这些收入类别如图2-9所示。

```
housing["income_cat"].hist()
```

现在，你可以根据收入类别进行分层抽样了。使用Scikit-Learn的StratifiedShuffleSplit类：

```
from sklearn.model_selection import StratifiedShuffleSplit
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

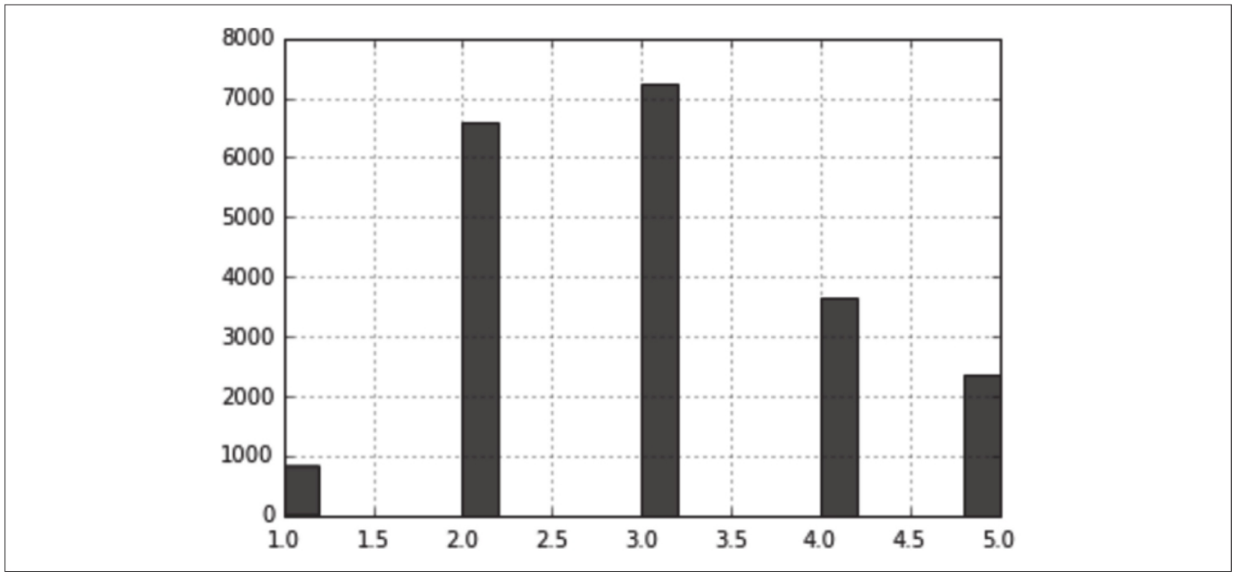


图2-9：收入类别的直方图

看看这个运行是否如我们所料。首先，可以看看测试集中收入类别比例分布：

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)
3    0.350533
2    0.318798
4    0.176357
5    0.114583
1    0.039729
Name: income_cat, dtype: float64
```

使用类似代码你还可以测量测试集中的收入类别比例分布。图2-10比较了在三种不同的数据集（完整数据集、分层抽样的测试集、纯随机抽样的测试集）中收入类别比例分布。正如你所见，分层抽样的测试集中的比例分布与完整数据集中的分布几乎一致，而纯随机抽样的测试集结果则是有偏的。

	Overall	Stratified	Random	Rand. %error	Strat. %error
1	0.039826	0.039729	0.040213	0.973236	-0.243309
2	0.318847	0.318798	0.324370	1.732260	-0.015195
3	0.350581	0.350533	0.358527	2.266446	-0.013820
4	0.176308	0.176357	0.167393	-5.056334	0.027480
5	0.114438	0.114583	0.109496	-4.318374	0.127011

图2-10：分层抽样与纯随机抽样的抽样偏差比较

现在你可以删除income_cat属性，将数据恢复原样了：

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

我们花了相当长的时间在测试集的生成上，理由很充分：这是机器学习项目中经常被忽视但是却至关重要的一部分。并且，当讨论到交叉验证时，这里谈到的许多想法也对其大有裨益。现在，是时候进入下一个阶段（数据探索）了。

[1] 建议使用最新版本的Python 3。Python 2.7+也许也可以使用，但是现在不推荐使用，所有主要的科学库都放弃了对它的支持，因此你应该尽快迁移到Python 3。

[2] 我将在Linux或macOS系统上的bash shell中使用pip显示安装步骤。你可能需要适应一下自己的系统。在Windows上，我建议安装Anaconda。

[3] 如果要为计算机上的所有用户而不是你自己升级pip，则应删除--user选项并确保你具有管理员权限（例如，在Linux或macOS上，在整个命令之前添加sudo）。

[4] 替代工具包括venv（非常类似于virtualenv且包含在标准库中）、virtualenvwrapper（在virtualenv之上提供了额外的功能）、pyenv（允许在Python版本之间轻松切换）和pipenv（由流行的

requests库同一作者撰写的出色打包工具，建立在pip和virtualenv之上）。

[5] 请注意，Jupyter可以处理Python的多个版本，甚至可以处理许多其他语言，例如R或Octave。

[6] 你可能还需要检查法律约束，例如私有数据永远不应复制到不安全数据存储区。

[7] 在实际项目中，你可以将此代码保存在Python文件中，但现在你只需在Jupyter notebook中编写即可。

[8] 标准差通常表示为σ（希腊字母sigma），它是方差的平方根，方差是均值平方差的平均值。当特征具有常见的钟形正态分布（也称为高斯分布）时，适用“68-95-99.7”规则：大约68%的值落在均值的1σ内，95%落在2σ以内，99.7%落在3σ之内。

[9] 在本书中，当代码示例包含代码和输出的混合时（如此处所示），其格式类似于Python解释器中的格式，以提高可读性：代码行以>>>开头（或...缩进块），并且输出没有前缀。

[10] 你经常会看到人们将随机种子设置为42。这个数字没有特殊的意义，除了可以作为生命、宇宙和一切最终问题的答案。

[11] 位置信息实际上是相当粗糙的，因此许多区域将具有完全相同的ID，它们最终会在同一组（测试集或训练集）中。这引入了一些不幸的抽样偏差。

2.4　从数据探索和可视化中获得洞见

到现在为止，我们还只是在快速浏览数据，从而对手头上正在处理的数据类型形成一个大致的了解。本阶段的目标是再深入一点。

首先，把测试集放在一边，你能探索的只有训练集。此外，如果训练集非常庞大，你可以抽样一个探索集，这样后面的操作更简单快捷一些。不过我们这个案例的数据集非常小，完全可以直接在整个训练集上操作。让我们先创建一个副本，这样可以随便尝试而不损害训练集：

```
housing = strat_train_set.copy()
```

2.4.1　将地理数据可视化

由于存在地理位置信息（经度和纬度），因此建立一个各区域的分布图以便于可视化数据是一个很好的想法（见图2-11）：

```
housing.plot(kind="scatter", x="longitude", y="latitude")
```

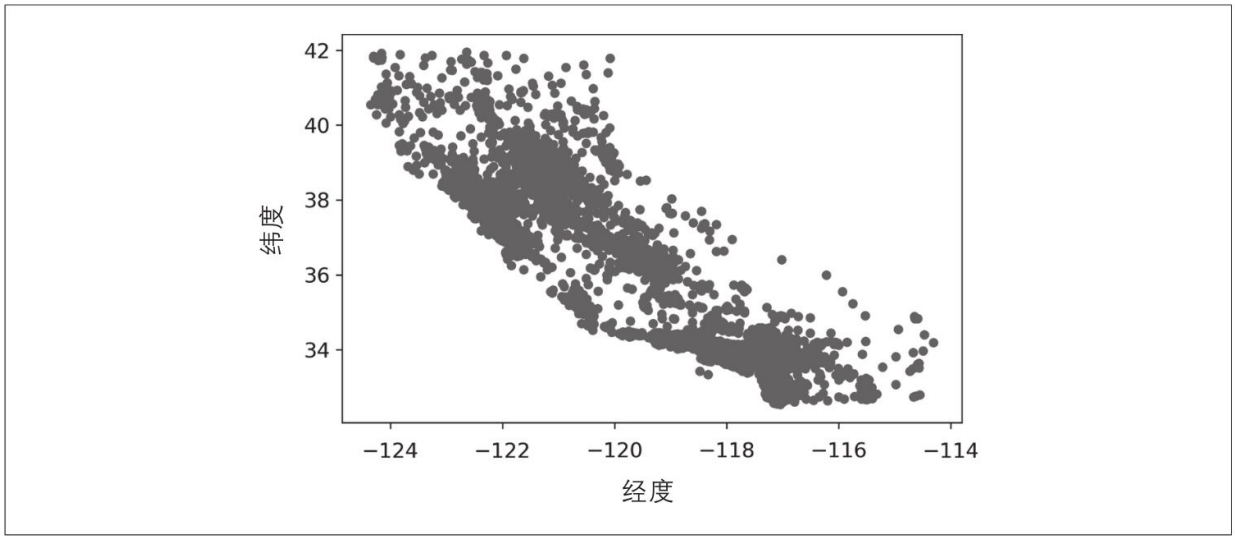


图2-11：数据的地理散点图

没错，这除了看起来跟加州一样以外，很难再看出任何其他的模式。将alpha选项设置为0.1，可以更清楚地看出高密度数据点的位置（见图2-12）：

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

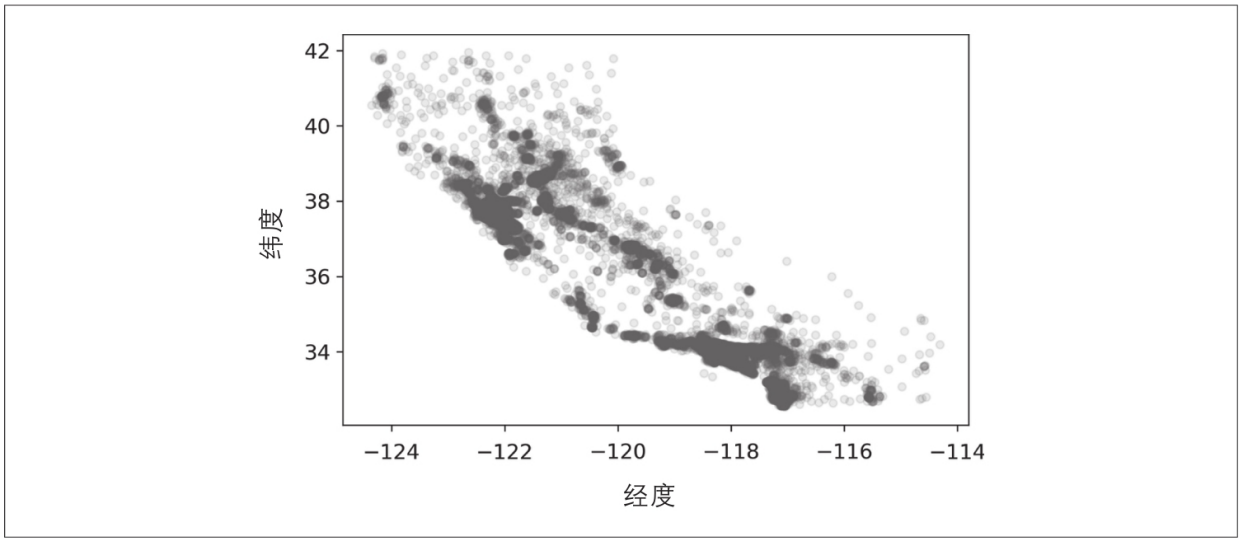


图2-12：突出高密度区域的更好的可视化

现在好多了：可以清楚地看到高密度区域，也就是湾区、洛杉矶和圣地亚哥附近，同时在中央山谷有一条相当高密度的长线，特别是萨克拉门托和弗雷斯诺附近。

我们的大脑非常善于从图片中发现模式，但是你需要玩转可视化参数才能让这些模式凸显出来。

现在，再来看看房价（见图2-13）。每个圆的半径大小代表了每个区域的人口数量（选项s），颜色代表价格（选项c）。我们使用一个名叫jet的预定义颜色表（选项cmap）来进行可视化，颜色范围从蓝（低）到红（高）[1]：

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
    s=housing["population"]/100, label="population", figsize=(10,7),
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

图2-13表明你房价与地理位置（例如靠近海）和人口密度息息相关，这一点你可能早已知晓。一个通常很有用的方法是使用聚类算法来检测主集群，然后再为各个集群中心添加一个新的衡量邻近距离的特征。海洋邻近度可能就是一个很有用的属性，不过在北加州，沿海地区的房价并不是太高，所以这个简单的规则也不是万能的。

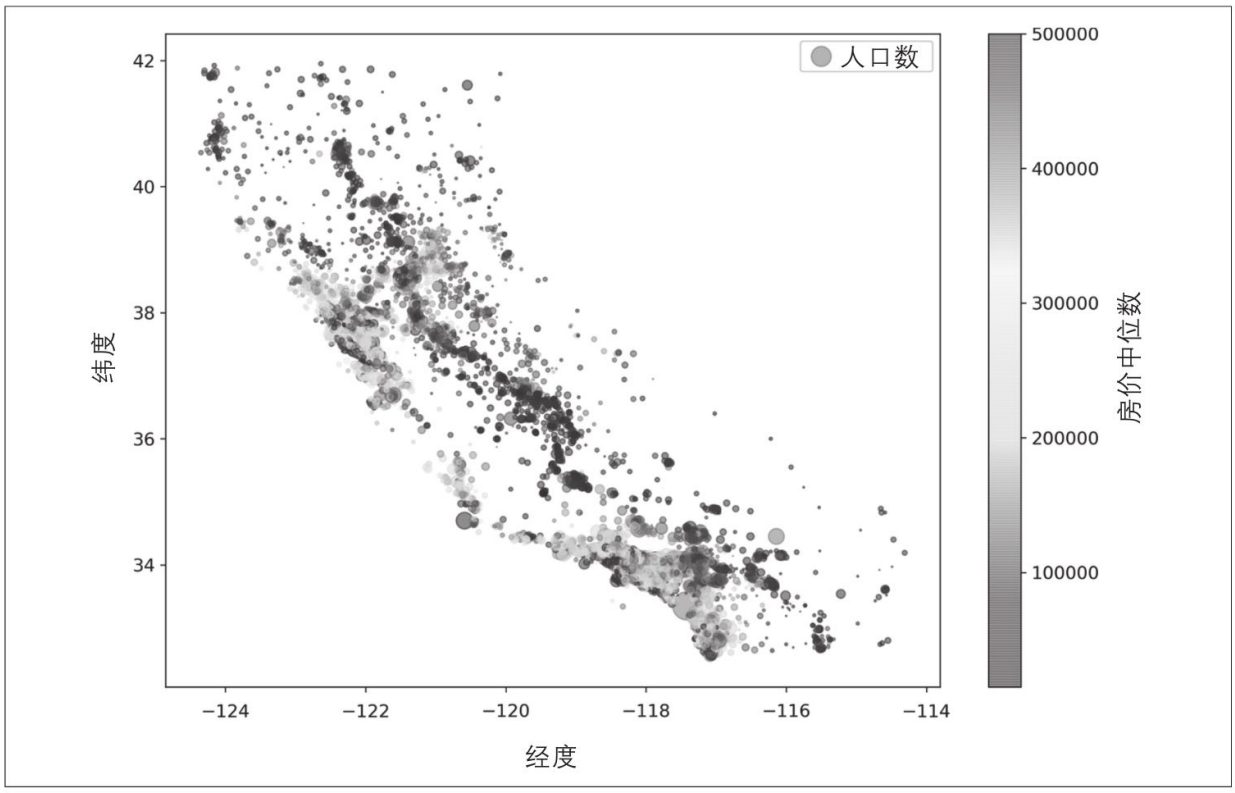


图2-13：加州房价：红色昂贵，蓝色便宜，较大的圆圈表示人口较多的地区

2.4.2 寻找相关性

由于数据集不大，你可以使用corr（）方法轻松计算出每对属性之间的标准相关系数（也称为皮尔逊r）：

```
corr_matrix = housing.corr()
```

现在看看每个属性与房价中位数的相关性分别是多少：

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age    0.114220
households            0.064702
```

```
total_bedrooms        0.047865
population           -0.026699
longitude            -0.047279
latitude             -0.142826
Name: median_house_value, dtype: float64
```

相关系数的范围从-1变化到1。越接近1，表示有越强的正相关。例如，当收入中位数上升时，房价中位数也趋于上升。当系数接近于-1时，表示有较强的负相关。我们可以看到纬度和房价中位数之间呈现出轻微的负相关（也就是说，越往北走，房价倾向于下降）。最后，系数靠近0则说明二者之间没有线性相关性。图2-14显示了横轴和纵轴之间相关性系数的多种图像。

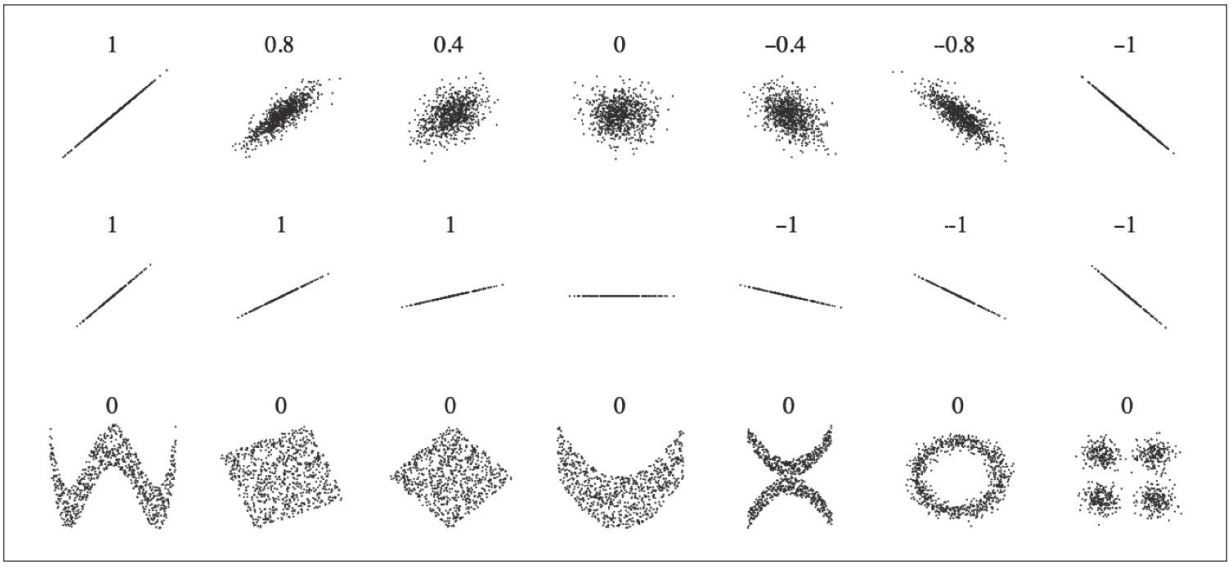


图2-14：各种数据集的标准相关系数（来源：维基百科，公共领域图像）

 相关系数仅测量线性相关性（“如果x上升，则y上升/下降”）。所以它有可能彻底遗漏非线性相关性（例如“如果x接近于0，则y会上升”）。注意图2-14最下面一排的图像，它们的相关性系数都是0，但是显然我们可以看出横轴和纵轴之间的关系并不是彼此完全独立的：这是非线性关系的示例。此外，图2-14中第二行显示了相关性为1或-1时的示例，需要注意的是这个相关性跟斜率完全无关。例如，你本人用英寸来计量的身高与你用英尺甚至是纳米来计量的身高之间的相关系数等于1。

还有一种方法可以检测属性之间的相关性，就是使用pandas的scatter_matrix函数，它会绘制出每个数值属性相对于其他数值属性的相关性。现在我们有11个数值属性，可以得到11^2=121个图像，篇幅原因无法完全展示，这里我们仅关注那些与房价中位数属性最相关的，可算作是最有潜力的属性（见图2-15）：

```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```

如果pandas绘制每个变量对自身的图像，那么主对角线（从左上到右下）将全都是直线，这样毫无意义。所以取而代之的方法是，pandas在这几个图中显示了每个属性的直方图（还有其他选项可选，详情请参考pandas文档）。

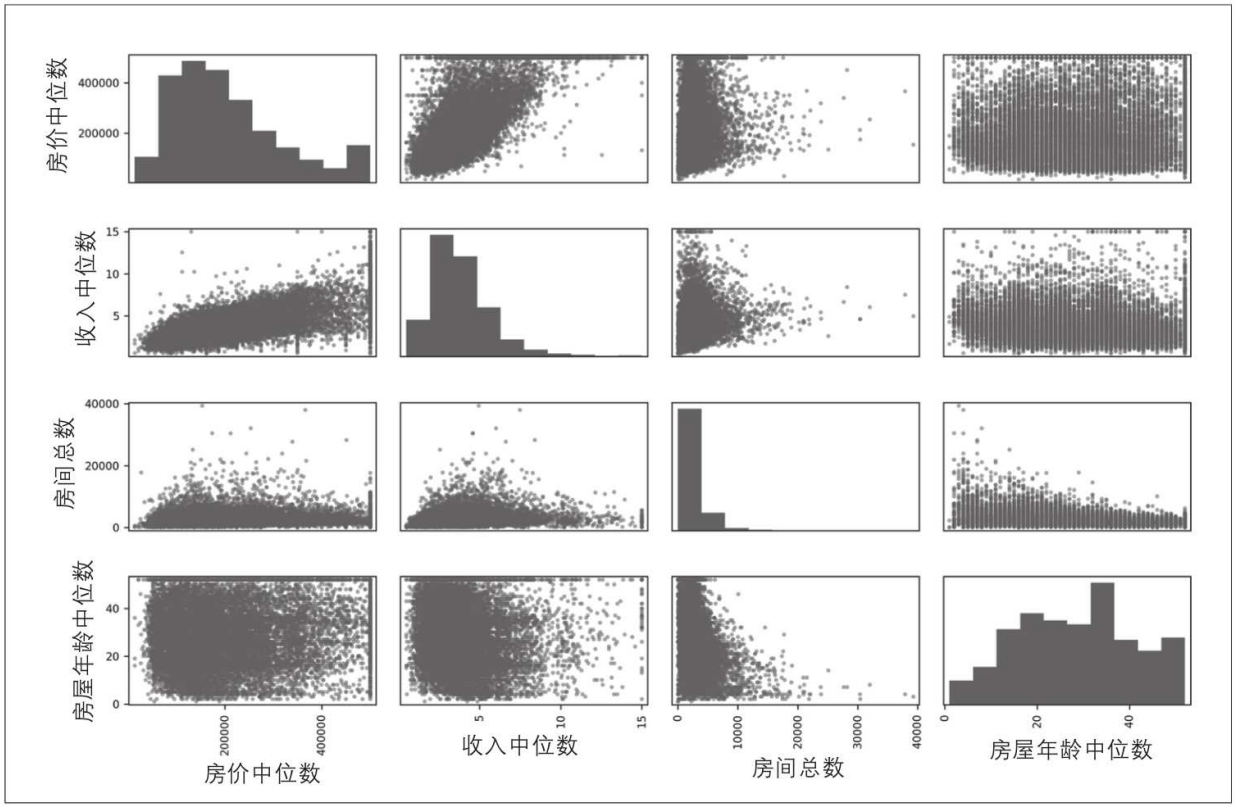


图2-15：这个散布矩阵显示每个数值属性相对于其他数值属性，并绘制每个数值属性的直方图

最有潜力能够预测房价中位数的属性是收入中位数，所以我们放大来看看其相关性的散点图（见图2-16）：

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1)
```

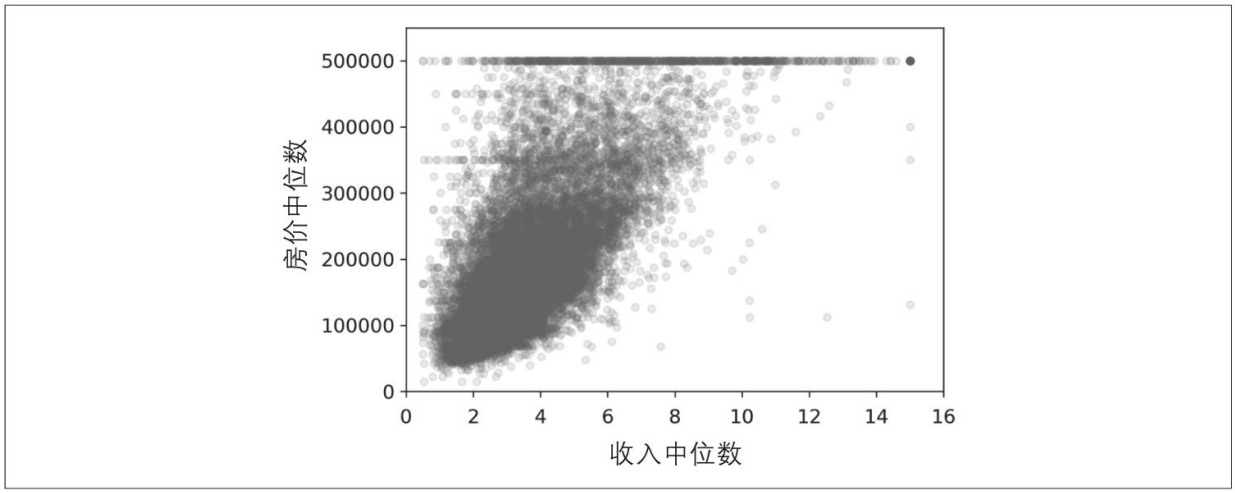


图2-16：收入中位数与房价中位数

图2-16说明了几个问题。首先，二者的相关性确实很强，你可以清楚地看到上升的趋势，并且点也不是太分散。其次，前面我们提到过50万美元的价格上限在图中是一条清晰的水平线，不过除此之外，图2-16还显示出几条不那么明显的直线：45万美元附近有一条水平线，35万美元附近也有一条，28万美元附近似乎隐约也有一条，再往下可能还有一些。为了避免你的算法学习之后重现这些怪异数据，你可能会尝试删除这些相应区域。

2.4.3 试验不同属性的组合

通过前面几节的介绍，希望你了解到了一些探索数据并从中获得洞见的方法。在准备开始给机器学习算法输入数据之前，你可能识别出了一些异常数据，需要提前清理掉。同时，说不定你还发现了不同属性之间的某些有趣联系，特别是跟目标属性相关的联系。再有，你还可能注意到某些属性的分布显示出了明显的“重尾”分布，于是你还需要对它们进行转换处理（例如，计算其对数）。当然，每个项目的历程都不一样，但是大致思路都相似。

在准备给机器学习算法输入数据之前，你要做的最后一件事应该是尝试各种属性的组合。例如，如果你不知道一个区域有多少个家庭，那么知道一个区域的“房间总数”也没什么用。你真正想要知道的是一个

家庭的房间数量。同样，单看“卧室总数”这个属性本身也没什么意义，你可能想拿它和“房间总数”来对比，或者拿来同“每个家庭的人口数”这个属性组合似乎也挺有意思。我们来试着创建这些新属性：

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]
```

再来看看相关矩阵：

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value          1.000000
median_income               0.687160
rooms_per_household         0.146285
total_rooms                 0.135097
housing_median_age          0.114110
households                  0.064506
total_bedrooms              0.047689
population_per_household   -0.021985
population                 -0.026920
longitude                  -0.047432
latitude                   -0.142724
bedrooms_per_room          -0.259984
Name: median_house_value, dtype: float64
```

怎么样？还不错吧！新属性bedrooms_per_room较之“房间总数”或“卧室总数”与房价中位数的相关性都要高得多。显然，卧室/房间比例更低的房屋往往价格更贵。同样，“每个家庭的房间数量”也比“房间总数”更具信息量——房屋越大，价格越贵。

这一轮的探索不一定要多么彻底，关键是迈开第一步，快速获得洞见，这将有助于你获得非常棒的第一个原型。这也是一个不断迭代的过程：一旦你的原型产生并且开始运行，你可以分析它的输出以获得更多洞见，然后再次回到这个探索步骤。

[1] 如果你看到的是灰度图片，请握住一支红笔，在从湾区到圣地亚哥的大部分海岸线上做记号（你可能会想到）。你也可以在萨克拉门托周围添加黄色标记。

2.5 机器学习算法的数据准备

现在，终于是时候给你的机器学习算法准备数据了。这里你应该编写函数来执行，而不是手动操作，原因如下：

• 你可以在任何数据集上轻松重现这些转换（例如，获得更新的数据集之后）。

• 你可以逐渐建立起一个转换函数的函数库，可以在以后的项目中重用。

• 你可以在实时系统中使用这些函数来转换新数据，再输入给算法。

• 你可以轻松尝试多种转换方式，查看哪种转换的组合效果最佳。

但是现在，让我们先回到一个干净的训练集（再次复制strat_train_set），然后将预测器和标签分开，因为这里我们不一定对它们使用相同的转换方式（需要注意drop（）会创建一个数据副本，但是不影响strat_train_set）：

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

2.5.1 数据清理

大部分的机器学习算法无法在缺失的特征上工作，所以我们要创建一些函数来辅助它。前面我们已经注意到total_bedrooms属性有部

分值缺失，所以我们要解决它。有以下三种选择：

1.放弃这些相应的区域。

2.放弃整个属性。

3.将缺失的值设置为某个值（0、平均数或者中位数等）。

通过DataFrame的dropna（）、drop（）和fillna（）方法，可以轻松完成这些操作：

```
housing.dropna(subset=["total_bedrooms"])    # option 1
housing.drop("total_bedrooms", axis=1)       # option 2
median = housing["total_bedrooms"].median()  # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

如果选择方法3，你需要计算出训练集的中位数值，然后用它填充训练集中的缺失值，但也别忘了保存这个计算出来的中位数值，因为后面可能需要用到。当重新评估系统时，你需要更换测试集中的缺失值；或者在系统上线时，需要使用新数据替代缺失值。Scikit-Learn提供了一个非常容易上手的类来处理缺失值：SimpleImputer。使用方法如下：首先，你需要创建一个SimpleImputer实例，指定你要用属性的中位数值替换该属性的缺失值：

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
```

由于中位数值只能在数值属性上计算，所以我们需要创建一个没有文本属性ocean_proximity的数据副本：

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

使用fit（）方法将imputer实例适配到训练数据：

```
imputer.fit(housing_num)
```

这里imputer仅仅只是计算了每个属性的中位数值，并将结果存储在其实例变量statistics_中。虽然只有total_bedrooms这个属性存在缺失值，但是我们无法确认系统启动之后新数据中是否一定不存在任何缺失值，所以稳妥起见，还是将imputer应用于所有的数值属性：

```
>>> imputer.statistics_
array([ -118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. , 3.5409])
>>> housing_num.median().values
array([ -118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. , 3.5409])
```

现在，你可以使用这个“训练有素”的imputer将缺失值替换成中位数值从而完成训练集转换：

```
X = imputer.transform(housing_num)
```

结果是一个包含转换后特征的NumPy数组。如果你想将它放回pandas DataFrame，也很简单：

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
```

Scikit-Learn的设计

Scikit-Learn的API设计得非常好。其主要的设计原则是[1]：

一致性

所有对象共享一个简单一致的界面。

估算器

能够根据数据集对某些参数进行估算的任意对象都可以称为估算器（例如，imputer就是一个估算器）。估算由fit（）方法执行，它只需要一个数据集作为参数（或者两个——对于有监督学习算法，第二个数据集包含标签）。引导估算过程的任何其他参数都视为超参数（例如，imputer’s strategy），它必须被设置为一个实例变量（一般通过构造函数参数）。

转换器

有些估算器（例如imputer）也可以转换数据集，这些称为转换器。同样，API也非常简单：由transform（）方法和作为参数的待转换数据集一起执行转换，返回的结果就是转换后的数据集。这种转换的过程通常依赖于学习的参数，比如本例中的imputer。所有的转换器都可以使用一个很方便的方法，即fit_transform（），相当于先调用fit（）然后再调用transform（）（但是fit_transform（）有时是被优化过的，所以运行得更快一些）。

预测器

最后，还有些估算器能够基于一个给定的数据集进行预测，这称为预测器。例如，第1章的LinearRegression模型就是一个预测器，它基于一个国家的人均GDP预测该国家的生活满意度。预测器的predict（）方法会接受一个新实例的数据集，然后返回一个包含相应预测的数据集。值得一提的还有一个score（）方法，可以用来衡量给定测试集的预测质量（以及在有监督学习算法里对应的标签）[2]。

检查

所有估算器的超参数都可以通过公共实例变量（例如，imputer.strategy）直接访问，并且所有估算器的学习参数也可以通过有下划线后缀的公共实例变量来访问（例如，imputer.statistics）。

防止类扩散

数据集被表示为NumPy数组或SciPy稀疏矩阵，而不是自定义的类型。超参数只是普通的Python字符串或者数字。

构成

现有的构件块尽最大可能重用。例如，任意序列的转换器最后加一个预测器就可以轻松创建一个Pipeline估算器。

合理的默认值

Scikit-Learn为大多数参数提供了合理的默认值，从而可以快速搭建起一个基本的工作系统。

2.5.2　处理文本和分类属性

到目前为止，我们只处理数值属性，但现在让我们看一下文本属性。在此数据集中，只有一个：ocean_proximity属性。我们看看前10个实例的值：

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(10)
       ocean_proximity
17606        <1H OCEAN
18632        <1H OCEAN
14650       NEAR OCEAN
3230            INLAND
3555         <1H OCEAN
19480           INLAND
8879         <1H OCEAN
13685           INLAND
4937         <1H OCEAN
4861         <1H OCEAN
```

它不是任意文本，而是有限个可能的取值，每个值代表一个类别。因此，此属性是分类属性。大多数机器学习算法更喜欢使用数字，因此让我们将这些类别从文本转到数字。为此，我们可以使用Scikit-Learn的OrdinalEncoder类[3]：

```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> ordinal_encoder = OrdinalEncoder()
>>> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
>>> housing_cat_encoded[:10]
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

你可以使用Categories_实例变量获取类别列表。这个列表包含每个类别属性的一维数组（在这种情况下，这个列表包含一个数组，因为只有一个类别属性）：

```
>>> ordinal_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

这种表征方式产生的一个问题是，机器学习算法会认为两个相近的值比两个离得较远的值更为相似一些。在某些情况下这是对的（对一些有序类别，像“坏”“平均”“好”“优秀”），但是，对ocean_proximity而言情况并非如此（例如，类别0和类别4之间就比类别0和类别1之间的相似度更高）。为了解决这个问题，常见的解决方案是给每个类别创建一个二进制的属性：当类别是“<1H OCEAN”时，一个属性为1（其他为0），当类别是“INLAND”时，另一个属性为1（其他为0），以此类推。这就是独热编码，因为只有一个属性为1（热），其他均为0（冷）。新的属性有时候称为哑（dummy）属性。Scikit-Learn提供了一个OneHotEncoder编码器，可以将整数类别值转换为独热向量。我们用它来将类别编码为独热向量[4]。

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> cat_encoder = OneHotEncoder()
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
  with 16512 stored elements in Compressed Sparse Row format>
```

注意到这里的输出是一个SciPy稀疏矩阵，而不是一个NumPy数组。当你有成千上万个类别属性时，这个函数会非常有用。因为在独热编码完成之后，我们会得到一个几千列的矩阵，并且全是0，每行仅有一个1。占用大量内存来存储0是一件非常浪费的事情，因此稀疏矩阵选择仅存储非零元素的位置。而你依旧可以像使用一个普通的二维数组那样来使用它[5]，当然如果你实在想把它转换成一个（密集的）NumPy数组，只需要调用toarray（）方法即可：

```
>>> housing_cat_1hot.toarray()
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       ...,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

你可以再次使用编码器的categories_实例变量来得到类别列表：

```
>>> cat_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

如果类别属性具有大量可能的类别（例如，国家代码、专业、物种），那么独热编码会导致大量的输入特征，这可能会减慢训练并降低性能。如果发生这种情况，你可能想要用相关的数字特征代替类别输入。例如，你可以用与海洋的距离来替换ocean_proximity特征（类似地，可以用该国家的人口和人均GDP来代替国家代码）。或者，你可以用可学习的低维向量（称为嵌入）来替换每个类别。每个类别的表征可以在训练期间学习。这是表征学习的示例（更多细节参见第13章和第17章）。

2.5.3 自定义转换器

虽然Scikit-Learn提供了许多有用的转换器，但是你仍然需要为一些诸如自定义清理操作或组合特定属性等任务编写自己的转换器。你当然希望让自己的转换器与Scikit-Learn自身的功能（比如流水线）无缝衔接，而由于Scikit-Learn依赖于鸭子类型的编译，而不是继承，所以你所需要的只是创建一个类，然后应用以下三种方法：fit（）（返回self）、transform（）、fit_transform（）。

你可以通过添加TransformerMixin作为基类，直接得到最后一种方法。同时，如果添加BaseEstimator作为基类（并在构造函数中避免*args和**kargs），你还能额外获得两种非常有用的自动调整超参数的方法（get_params（）和set_params（））。例如，我们前面讨论过的组合属性，这里有个简单的转换器类，用来添加组合后的属性：

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

在本例中，转换器有一个超参数add_bedrooms_per_room默认设置为True（提供合理的默认值通常是很有帮助的）。这个超参数可以让你轻松知晓添加这个属性是否有助于机器学习算法。更一般地，如果你对数据准备的步骤没有充分的信心，就可以添加这个超参数来进行把关。这些数据准备步骤的执行越自动化，你自动尝试的组合也就越多，从而有更大可能从中找到一个重要的组合（还节省了大量时间）。

2.5.4 特征缩放

最重要也最需要应用到数据上的转换就是特征缩放。如果输入的数值属性具有非常大的比例差异，往往会导致机器学习算法的性能表

现不佳，当然也有极少数特例。案例中的房屋数据就是这样：房间总数的范围从6～39 320，而收入中位数的范围是0～15。注意，目标值通常不需要缩放。

同比例缩放所有属性的两种常用方法是最小-最大缩放和标准化。

最小-最大缩放（又叫作归一化）很简单：将值重新缩放使其最终范围归于0～1之间。实现方法是将值减去最小值并除以最大值和最小值的差。对此，Scikit-Learn提供了一个名为MinMaxScaler的转换器。如果出于某种原因，你希望范围不是0～1，那么可以通过调整超参数feature_range进行更改。

标准化则完全不一样：首先减去平均值（所以标准化值的均值总是零），然后除以方差，从而使得结果的分布具备单位方差。不同于最小-最大缩放的是，标准化不将值绑定到特定范围，对某些算法而言，这可能是个问题（例如，神经网络期望的输入值范围通常是0～1）。但是标准化的方法受异常值的影响更小。例如，假设某个地区的平均收入为100（错误数据），最小-最大缩放会将所有其他值从0～15降到0～0.15，而标准化则不会受到很大影响。Scikit-Learn提供了一个标准化的转换器StandadScaler。

 重要的是，跟所有转换一样，缩放器仅用来拟合训练集，而不是完整的数据集（包括测试集）。只有这样，才能使用它们来转换训练集和测试集（和新数据）。

2.5.5　转换流水线

正如你所见，许多数据转换的步骤需要以正确的顺序来执行。而Scikit-Learn正好提供了Pipeline类来支持这样的转换。下面是一个数值属性的流水线示例：

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Pipeline构造函数会通过一系列名称/估算器的配对来定义步骤序列。除了最后一个是估算器之外，前面都必须是转换器（也就是说，必须有fit_transform（）方法）。至于命名可以随意，你喜欢就好（只要它们是独一无二的，不含双下划线），它们稍后在超参数调整中会有用。

当调用流水线的fit（）方法时，会在所有转换器上按照顺序依次调用fit_transform（），将一个调用的输出作为参数传递给下一个调用方法，直到传递到最终的估算器，则只会调用fit（）方法。

流水线的方法与最终的估算器的方法相同。在本例中，最后一个估算器是StandardScaler，这是一个转换器，因此流水线有一个transform（）方法，可以按顺序将所有的转换应用到数据中（这也是我们用过的fit_transform（）方法）。

到目前为止，我们分别处理了类别列和数值列。拥有一个能够处理所有列的转换器会更方便，将适当的转换应用于每个列。在0.20版中，Scikit-Learn为此引入了ColumnTransformer，好消息是它与pandas DataFrames一起使用时效果很好。让我们用它来将所有转换应用到房屋数据：

```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]
```

```
full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```

首先导入ColumnTransformer类，接下来获得数值列名称列表和类别列名称列表，然后构造一个ColumnTransformer。构造函数需要一个元组列表，其中每个元组都包含一个名字[6]、一个转换器，以及一个该转换器能够应用的列名字（或索引）的列表。在此示例中，我们指定数值列使用之前定义的num_pipeline进行转换，类别列使用OneHotEncoder进行转换。最后，我们将ColumnTransformer应用到房屋数据：它将每个转换器应用于适当的列，并沿第二个轴合并输出（转换器必须返回相同数量的行）。

请注意，OneHotEncoder返回一个稀疏矩阵，而num_pipeline返回一个密集矩阵。当稀疏矩阵和密集矩阵混合在一起时，ColumnTransformer会估算最终矩阵的密度（即单元格的非零比率），如果密度低于给定的阈值，则返回一个稀疏矩阵（通过默认值为sparse_threshold=0.3）。在此示例中，它返回一个密集矩阵。我们有一个预处理流水线，该流水线可以获取全部房屋数据并对每一列进行适当的转换。

 不使用转换器的情况下，如果你要删除列，则可以指定字符串"drop"。如果你希望这些列保持不变，也可以指定"passthrough"。默认情况下，其余列（即未列出的列）将被删除。如果你希望以不同方式处理这些列，则可以将remainder这个超参数设置为任意转换器（或"passthrough"）。

如果你使用的是Scikit-Learn 0.19或更早版本，则可以使用第三方库，例如sklearnpandas，或者可以推出自己的自定义转换器来获得

和ColumnTransformer相同的功能。另外，你可以使用FeatureUnion类，该类可以应用于不同的转换器并合并其输出。但是你不能为每个转换器指定不同的列，它们都适用于整个数据。你也可以使用针对列的自定义转换器来解决这个限制（有关示例，请参见Jupyter notebook）。

[1] 关于设计原则的更多细节，参见Lars Buitinck等人的论文，“API Design for Machine Learning Software：Experiences from the Scikit-Learn Project”，arXiv preprint arXiv：1309.0238（2013）。
[2] 一些预测器还提供了测量其预测的置信度的方法。
[3] 此类在Scikit-Learn 0.20及更高版本中可用。如果使用较早版本，请考虑升级，或使用pandas的Series.factorize（）方法。
[4] 在Scikit-Learn 0.20之前，该方法只能对整数分类值进行编码，但是从0.20开始，它还可以处理其他类型的输入，包括文本类别输入。
[5] 更多详细信息，请参见SciPy的文档。
[6] 就像流水线一样，该名称可以是任意的，只要它不包含双下划线即可。

2.6 选择和训练模型

至此，你提出了问题，获得了数据，也进行了数据探索，然后对训练集和测试集进行了抽样并编写了转换流水线，从而可以自动清理和准备机器学习算法的数据！现在是时候选择机器学习模型并展开训练了。

2.6.1 训练和评估训练集

好消息是，在经过前面的步骤之后，事情现在变得比想象中容易很多。首先，如同我们在第1章所做的，先训练一个线性回归模型：

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

现在你有一个可以工作的线性回归模型了。让我们用几个训练集的实例试试：

```
>>> some_data = housing.iloc[:5]
>>> some_labels = housing_labels.iloc[:5]
>>> some_data_prepared = full_pipeline.transform(some_data)
>>> print("Predictions:", lin_reg.predict(some_data_prepared))
Predictions: [ 210644.6045  317768.8069  210956.4333  59218.9888  189747.5584]
>>> print("Labels:", list(some_labels))
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

可以工作了，虽然预测还不是很准确（例如，第一次的预测失效接近40%！）。我们可以使用Scikit-Learn的mean_squared_error（）函数来测量整个训练集上回归模型的RMSE：

```
>>> from sklearn.metrics import mean_squared_error
>>> housing_predictions = lin_reg.predict(housing_prepared)
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse = np.sqrt(lin_mse)
>>> lin_rmse
68628.19819848922
```

好吧，这虽然比什么都没有要好，但显然也不是一个好看的成绩：大多数区域的median_housing_values分布在120 000～265 000美元之间，所以典型的预测误差达到68 628美元只能算是差强人意。这就是一个典型的模型对训练数据欠拟合的案例。这种情况发生时，通常意味着这些特征可能无法提供足够的信息来做出更好的预测，或者是模型本身不够强大。我们在第1章已经提到，想要修正欠拟合，可以通过选择更强大的模型，或为算法训练提供更好的特征，又或者减少对模型的限制等方法。我们这个模型不是一个正则化的模型，所以就排除了最后那个选项。你可以试试添加更多的特征（例如，人口数量的日志），但首先，让我们尝试一个更复杂的模型，看看它到底是怎么工作的。

我们来训练一个DecisionTreeRegressor。这是一个非常强大的模型，它能够从数据中找到复杂的非线性关系（关于决策树在第6章会有更详细的介绍）。下面的代码现在看来应该已经很熟悉了：

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

既然这个模型已经训练有素，我们可以用训练集来评估一下：

```
>>> housing_predictions = tree_reg.predict(housing_prepared)
>>> tree_mse = mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse = np.sqrt(tree_mse)
>>> tree_rmse
0.0
```

等等，什么！完全没有错误？这个模型真的可以做到绝对完美吗？当然，更有可能的是这个模型对数据严重过拟合了。我们怎么确认呢？前面提到过，在你有信心启动模型之前，都不要触碰测试集，所以这里，你需要拿训练集中的一部分用于训练，另一部分用于模型验证。

2.6.2 使用交叉验证来更好地进行评估

评估决策树模型的一种方法是使用train_test_split函数将训练集分为较小的训练集和验证集，然后根据这些较小的训练集来训练模型，并对其进行评估。这虽然有一些工作量，但是不会太难，并且非常有效。

另一个不错的选择是使用Scikit-Learn的K-折交叉验证功能。以下是执行K-折交叉验证的代码：它将训练集随机分割成10个不同的子集，每个子集称为一个折叠，然后对决策树模型进行10次训练和评估——每次挑选1个折叠进行评估，使用另外的9个折叠进行训练。产生的结果是一个包含10次评估分数的数组：

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

 Scikit-Learn的交叉验证功能更倾向于使用效用函数（越大越好）而不是成本函数（越小越好），所以计算分数的函数实际上是负的MSE（一个负值）函数，这就是为什么上面的代码在计算平方根之前会先计算出-scores。让我们看看结果：

```
>>> def display_scores(scores):
...     print("Scores:", scores)
...     print("Mean:", scores.mean())
...     print("Standard deviation:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Scores: [70194.33680785 66855.16363941 72432.58244769 70758.73896782
 71115.88230639 75585.14172901 70262.86139133 70273.6325285
 75366.87952553 71231.65726027]
Mean: 71407.68766037929
Standard deviation: 2439.4345041191004
```

这次的决策树模型好像不如之前表现得好。事实上，它看起来简直比线性回归模型还要糟糕！请注意，交叉验证不仅可以得到一个模型性能的评估值，还可以衡量该评估的精确度（即其标准差）。这里该决策树得出的评分约为71 407，上下浮动±2439。如果你只使用了一个验证集，就收不到这样的结果信息。交叉验证的代价就是要多次训练模型，因此也不是永远都行得通。

保险起见，让我们也计算一下线性回归模型的评分：

```
>>> lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
...                              scoring="neg_mean_squared_error", cv=10)
...
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Scores: [66782.73843989 66960.118071   70347.95244419 74739.57052552
 68031.13388938 71193.84183426 64969.63056405 68281.61137997
 71552.91566558 67665.10082067]
Mean: 69052.46136345083
Standard deviation: 2731.674001798348
```

没错，决策树模型确实是严重过拟合了，以至于表现得比线性回归模型还要糟糕。

我们再来试试最后一个模型：RandomForestRegressor。在第7章中，我们将会了解到随机森林的工作原理：通过对特征的随机子集进行许多个决策树的训练，然后对其预测取平均。在多个模型的基础之

上建立模型，称为集成学习，这是进一步推动机器学习算法的好方法。这里我们将跳过大部分代码，因为与其他模型基本相同：

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
18603.515021376355
>>> display_scores(forest_rmse_scores)
Scores: [49519.80364233 47461.9115823  50029.02762854 52325.28068953
 49308.39426421 53446.37892622 48634.8036574  47585.73832311
 53490.10699751 50021.5852922 ]
Mean: 50182.303100336096
Standard deviation: 2097.0810550985693
```

哇，这个就好多了：随机森林看起来很有戏。但是，请注意，训练集上的分数仍然远低于验证集，这意味着该模型仍然对训练集过拟合。过拟合的可能解决方案包括简化模型、约束模型（即使其正规化），或获得更多的训练数据。不过在深入探索随机森林之前，你应该先尝试一遍各种机器学习算法的其他模型（几种具有不同内核的支持向量机，比如神经网络模型等），但是记住，别花太多时间去调整超参数。我们的目的是筛选出几个（2～5个）有效的模型。

 每一个尝试过的模型你都应该妥善保存，以便将来可以轻松回到你想要的模型当中。记得还要同时保存超参数和训练过的参数，以及交叉验证的评分和实际预测的结果。这样你就可以轻松地对比不同模型类型的评分，以及不同模型造成的错误类型。通过Python的pickle模块或joblib库，你可以轻松保存Scikit-Learn模型，这样可以更有效地将大型NumPy数组（可以用pip安装）序列化：

```
import joblib

joblib.dump(my_model, "my_model.pkl")
# and later...
my_model_loaded = joblib.load("my_model.pkl")
```

2.7 微调模型

假设你现在有了一个有效模型的候选列表。现在你需要对它们进行微调。我们来看几个可行的方法。

2.7.1 网格搜索

一种微调的方法是手动调整超参数，直到找到一组很好的超参数值组合。这个过程非常枯燥乏味，你可能坚持不到足够的时间来探索出各种组合。

相反，你可以用Scikit-Learn的GridSearchCV来替你进行探索。你所要做的只是告诉它你要进行实验的超参数是什么，以及需要尝试的值，它将会使用交叉验证来评估超参数值的所有可能组合。例如，下面这段代码搜索RandomForestRegressor的超参数值的最佳组合：

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```

当你不知道超参数应该赋什么值时，一个简单的方法是尝试10的连续幂次方（如果你想要得到更细粒度的搜索，可以使用更小的数，参考这个示例中所示的n_estimators超参数）。

这个param_grid告诉Scikit-Learn，首先评估第一个dict中的n_estimator和max_features的所有3×4=12种超参数值组合（先不要担心这些超参数现在意味着什么，我们将在第7章中进行解释）；接着，尝试第二个dict中超参数值的所有2×3=6种组合，但这次超参数bootstrap需要设置为False而不是True（True是该超参数的默认值）。

总而言之，网格搜索将探索RandomForestRegressor超参数值的12+6=18种组合，并对每个模型进行5次训练（因为我们使用的是5-折交叉验证）。换句话说，总共会完成18×5=90次训练！这可能需要相当长的时间，但是完成后你就可以获得最佳的参数组合：

```
>>> grid_search.best_params_
{'max_features': 8, 'n_estimators': 30}
```

 因为被评估的n_estimator最大值是8和30，所以你还可以试试更高的值，评分可能还会继续改善。

你可以直接得到最好的估算器：

```
>>> grid_search.best_estimator_
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features=8, max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=30, n_jobs=None, oob_score=False, random_state=None,
           verbose=0, warm_start=False)
```

 如果GridSearchCV被初始化为refit=True（默认值），那么一旦通过交叉验证找到了最佳估算器，它将在整个训练集上重新训练。这通常是个好方法，因为提供更多的数据很可能提升其性能。

当然还有评估分数：

```
>>> cvres = grid_search.cv_results_
>>> for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
...     print(np.sqrt(-mean_score), params)
...
63669.05791727153 {'max_features': 2, 'n_estimators': 3}
55627.16171305252 {'max_features': 2, 'n_estimators': 10}
53384.57867637289 {'max_features': 2, 'n_estimators': 30}
60965.99185930139 {'max_features': 4, 'n_estimators': 3}
52740.98248528835 {'max_features': 4, 'n_estimators': 10}
50377.344409590376 {'max_features': 4, 'n_estimators': 30}
58663.84733372485 {'max_features': 6, 'n_estimators': 3}
52006.15355973719 {'max_features': 6, 'n_estimators': 10}
50146.465964159885 {'max_features': 6, 'n_estimators': 30}
57869.25504027614 {'max_features': 8, 'n_estimators': 3}
51711.09443660957 {'max_features': 8, 'n_estimators': 10}
49682.25345942335 {'max_features': 8, 'n_estimators': 30}
62895.088889905004 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54658.14484390074 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59470.399594730654 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52725.01091081235 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57490.612956065226 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51009.51445842374 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

在本例中，我们得到的最佳解决方案是将超参数max_features设置为8，将超参数n_estimators设置为30。这个组合的RMSE分数为49 682，略优于之前使用默认超参数值的分数50 182。恭喜你，你成功地将模型调整到了最佳模式！

 不要忘了，有些数据准备的步骤也可以当作超参数来处理。例如，网格搜索会自动查找是否添加你不确定的特征（比如是否使用转换器CombinedAttributesAdder的超参数add_bedrooms_per_room）。同样，还可以用它来自动寻找处理问题的最佳方法，例如处理异常值、缺失特征，以及特征选择等。

2.7.2 随机搜索

如果探索的组合数量较少（例如上一个示例），那么网格搜索是一种不错的方法。但是当超参数的搜索范围较大时，通常会优先选择使用RandomizedSearchCV。这个类用起来与GridSearchCV类大致相同，但它不会尝试所有可能的组合，而是在每次迭代中为每个超参数选择一个随机值，然后对一定数量的随机组合进行评估。这种方法有两个显著好处：

- 如果运行随机搜索1000个迭代，那么将会探索每个超参数的1000个不同的值（而不是像网格搜索方法那样每个超参数仅探索少量几个值）。

- 通过简单地设置迭代次数，可以更好地控制要分配给超参数搜索的计算预算。

2.7.3 集成方法

还有一种微调系统的方法是将表现最优的模型组合起来。组合（或“集成”）方法通常比最佳的单一模型更好（就像随机森林比其所依赖的任何单个决策树模型更好一样），特别是当单一模型会产生不同类型误差时更是如此。我们将在第7章中更详细地介绍这个主题。

2.7.4 分析最佳模型及其误差

通过检查最佳模型，你总是可以得到一些好的洞见。例如在进行准确预测时，RandomForestRegressor可以指出每个属性的相对重要程度：

```
>>> feature_importances = grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([7.33442355e-02, 6.29090705e-02, 4.11437985e-02, 1.46726854e-02,
       1.41064835e-02, 1.48742809e-02, 1.42575993e-02, 3.66158981e-01,
```

```
       5.64191792e-02, 1.08792957e-01, 5.33510773e-02, 1.03114883e-02,
       1.64780994e-01, 6.02803867e-05, 1.96041560e-03, 2.85647464e-03])
```

将这些重要性分数显示在对应的属性名称旁边：

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_encoder = full_pipeline.named_transformers_["cat"]
>>> cat_one_hot_attribs = list(cat_encoder.categories_[0])
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.3661589806181342, 'median_income'),
 (0.1647809935615905, 'INLAND'),
 (0.10879295677551573, 'pop_per_hhold'),
 (0.07334423551601242, 'longitude'),
 (0.0629090704826203, 'latitude'),
 (0.05641917918195401, 'rooms_per_hhold'),
 (0.05335107734767581, 'bedrooms_per_room'),
 (0.041143798478729635, 'housing_median_age'),
 (0.014874280890402767, 'population'),
 (0.014672685420543237, 'total_rooms'),
 (0.014257599323407807, 'households'),
 (0.014106483453584102, 'total_bedrooms'),
 (0.010311488326303787, '<1H OCEAN'),
 (0.002856474637320158, 'NEAR OCEAN'),
 (0.00196041559947807, 'NEAR BAY'),
 (6.028038672736599e-05, 'ISLAND')]
```

有了这些信息，你可以尝试删除一些不太有用的特征（例如，本例中只有一个ocean_proximity是有用的，我们可以试着删除其他所有特征）。

然后，你还应该查看一下系统产生的具体错误，尝试了解它们是怎么产生的，以及该怎么解决（通过添加额外的特征、删除没有信息的特征、清除异常值等）。

2.7.5 通过测试集评估系统

通过一段时间的训练，你终于有了一个表现足够优秀的系统。现在是用测试集评估最终模型的时候了。这个过程没有什么特别的，只需要从测试集中获取预测器和标签，运行full_pipeline来转换数据

（调用transform（）而不是fit_transform（）），然后在测试集上评估最终模型：

```
final_model = grid_search.best_estimator_

X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)    # => evaluates to 47,730.2
```

在某些情况下，泛化误差的这种点估计将不足以说服你启动生产环境：如果它仅比当前生产环境中的模型好0.1%？你可能想知道这个估计的精确度。为此，你可以使用scipy.stats.t.interval（）计算泛化误差的95%置信区间：

```
>>> from scipy import stats
>>> confidence = 0.95
>>> squared_errors = (final_predictions - y_test) ** 2
>>> np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
...                          loc=squared_errors.mean(),
...                          scale=stats.sem(squared_errors)))
...
array([45685.10470776, 49691.25001878])
```

如果之前进行过大量的超参数调整，这时的评估结果通常会略逊于你之前使用交叉验证时的表现结果（因为通过不断调整，系统在验证数据上终于表现良好，在未知数据集上可能达不到这么好的效果）。在本例中，结果虽然并非如此，但是当这种情况发生时，你一定不要继续调整超参数，不要试图再努力让测试集的结果变得好看一些，因为这些改进在泛化到新的数据集时又会变成无用功。

现在进入项目预启动阶段：你将要展示你的解决方案（强调学习了什么，什么有用，什么没有用，基于什么假设，以及系统的限制有哪些），记录所有事情，通过清晰的可视化和易于记忆的陈述方式制作漂亮的演示文稿（例如，“收入中位数是预测房价的首要指标”）。在这个加州住房的示例里，系统的最终性能并不比专家估算的效果好，通常会下降20%左右，但这仍然是一个不错的选择，因为这为专家腾出了一些时间以便他们可以投入更有趣和更有生产力的任务上。

2.8 启动、监控和维护你的系统

很好，你已获准启动生产环境！现在，你需要准备好解决方案以进行上线（例如，完善代码、编写文档和测试等）。然后你可以将模型部署到生产环境中。一种方法是保存训练好的Scikit-Learn模型（使用joblib），包括完整的预处理和预测流水线，然后在你的生产环境中加载经过训练的模型并通过调用prepare（）方法来进行预测。例如，也许该模型在网站内使用：用户将输入有关新地区的一些数据，然后单击估算价格按钮。这将发送一个查询，其中包含数据发送到Web服务器，然后将其转发到Web应用程序，最后，你的代码调用模型的predict（）方法（在服务器启动时加载模型，而不是每次使用模型时都加载模型）。或者，你可以将模型包装在Web应用程序中，使用专用的Web服务，通过REST API来进行查询[1]（参见图2-17）。这样可以轻松地将你的模型升级到新版本，而不会中断主应用程序。这样也简化了扩展，因为可以根据需要来启动任意数量的Web服务，并且平衡那些Web服务上Web应用程序的请求负载。而且，它允许你的Web应用程序使用任何语言，而不仅仅是Python。

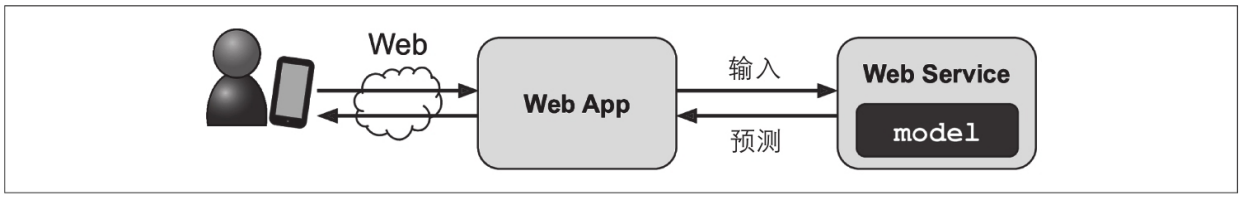


图2-17：部署为Web服务并被Web应用程序使用的模型

另一种流行的策略是将模型部署到云上，例如在Google Cloud AI Platform（以前称为Google Cloud ML Engine）上：只需使用joblib来保存模型并上传到Google CloudStorage（GCS），然后转到Google Cloud AI Platform并创建一个新的模型版本，将其指向GCS文件。这为你提供了一个简单的Web服务，该服务负责负载平衡和扩展。它接受包含输入数据（例如一个地区）的JSON请求并返回包含预测的JSON响应。

然后，你可以在你的网站（或你使用的任何生产环境）中使用此Web服务。如我们在第19章中所见，在AI平台上部署TensorFlow模型和部署Scikit-Learn模型没有太大的不同。

但是部署并不是故事的结束。你还需要编写监控代码以定期检查系统的实时性能，并在系统性能降低时触发警报。由于基础架构中的组件损坏，性能下降可能会很大，但也有可能是轻微的下降，长时间内很容易被忽视。这很常见，因为模型会随着时间的流逝而“腐烂”：的确，世界在变化，因此，如果使用去年的数据对模型进行训练，则可能不会适应今天的数据。

 即使是经过训练可以对猫和狗的图片进行分类的模型，也可能需要定期重新训练，不是因为猫和狗会一夜之间有变化，而是因为相机在不断变化，图像格式、清晰度、亮度和尺寸也会变化。而且，人们明年可能会爱上不同的品种，或者他们可能会给宠物戴上小帽子，谁知道呢？

因此，你需要监控模型的实时性能。但是怎么做呢？在某些情况下，可以从下游推断模型的性能指标。例如，如果你的模型是推荐系统的一部分，并且推荐用户可能感兴趣的产品，那么很容易监控每天出售的推荐产品的数量。如果该数字下降（与不推荐相比），那么主要的嫌疑就是模型。这可能是因为数据流水线断开，或者可能需要对新数据重新训练模型（我们稍后将进行讨论）。

但是，如果没有人工分析，确定模型的性能并不总是可行的。例如，假设你训练了一个图像分类模型（参见第3章）来检测生产线上的产品缺陷。你怎么能在数以千计的次品发货给客户之前，得到模型的性能是否下降的警告？一种解决方案是将一些模型分类的图片（尤其是对模型来说不确定的图片）送给人工评估者。根据任务，评估者可能需要是专家或者非专家，例如众包平台上的工人（例如Amazon Mechanical

Turk）。在某些应用中，他们甚至可能是用户自己，通过调查或重新设计的验证码[2]来响应。

无论哪种方式，你都需要建立一个监控系统（是否有人来评估实时模型），以及所有的相关流程，以定义发生故障怎么做以及如何为故障做准备。不幸的是，这可能需要做很多工作。实际上，工作量通常比构建和训练模型要多得多。

如果数据不断发展，则需要更新数据集并定期重新训练模型。你应该使整个过程尽可能地自动化。可以通过以下操作来使其自动化：

• 定期收集新数据并做标记（例如，使用人工评估者）。

• 编写脚本来训练模型并自动微调超参数。该脚本可以根据你的需求（例如每天或每周）自动运行。

• 编写另一个脚本，该脚本将在更新的测试集上评估新模型和旧模型，如果性能良好，则将该模型部署到生产环境中（如果性能降低，请调查原因）。

你还应该确保评估模型的输入数据质量。有时由于信号质量较差（例如故障、传感器发送随机值，或者其他的输出过时了），但是你的系统性能可能需要一段时间才能触发警报。如果你监控模型的输入，则可能会更早一点发现。例如，越来越多的输入缺少某个特征，或者其平均值或标准差偏离训练集太远，或者分类特征中出现了新类别。

最后，请保留每个模型的备份，准备好流程和工具，以便在新模型出现时快速回滚到以前的模型，以防新模型由于某种情况开始出现严重故障。进行备份还可以轻松实现将新模型与以前的模型进行比较。同样，你应该保留每个版本的数据集，以便在新数据集遭到破坏的情况下可以回滚到先前的数据集（如果事实能证明添加到其中的新数据都是离群值）。备份数据集还可以针对任何先前的数据集来评估任何模型。

 你可能需要创建测试集的几个子集，以便评估你的模型在特定部分数据的效果如何。例如，你可能希望有一个仅包含最新数据，或用于特定类型输入的测试集（例如，位于内陆的区域与靠近海洋的区域）。这将使你更深入地了解模型的长处和短处。

如你所见，机器学习涉及很多基础架构，如果你的第一个ML项目花费大量精力和时间来构建和部署到生产环境，请不要觉得奇怪。幸运的是，一旦所有基础设施都准备就绪，从构想到生产环境将会很快实现。

[1] 简而言之，REST（或RESTful）API是基于HTTP的API，遵循某些约定，如使用标准HTTP动词来读取、更新、创建或删除资源（GET、POST、PUT和DELETE）并使用JSON作为输入和输出。
[2] 验证码是一种确保用户不是机器人的测试，这些测试经常被用作标记训练数据的廉价方法。

2.9 试试看

希望通过本章内容，你能够了解一个机器学习项目大概是什么样子，同时了解一些用来训练系统的工具。如你所见，大部分工作主要用在数据准备、构建监控工具、建立人工评估的流水线以及自动定期训练模型上。机器学习的算法固然重要，但是最好还是对整个流程都熟悉一遍，掌握3个或4个合适的算法，不要将所有的时间都用来探索高级算法，而对整个流程视而不见。

如果你还没有动手尝试，现在是打开电脑的好时机，选择一个你感兴趣的数据集，尝试从A到Z的整个过程。诸如http://kaggle.com/的竞赛网站是一个不错的起点：它会给你一个数据集、一个明确的目标，以及可以一起分享经验的同伴。

2.10 练习题

使用本章的房屋数据集完下面的练习题：

1.使用不同的超参数，如kernel="linear"（具有C超参数的多种值）或kernel="rbf"（C超参数和gamma超参数的多种值），尝试一个支持向量机回归器（sklearn.svm.SVR），不用担心现在不知道这些超参数的含义。最好的SVR预测器是如何工作的？

2.尝试用RandomizedSearchCV替换GridSearchCV。

3.尝试在准备流水线中添加一个转换器，从而只选出最重要的属性。

4.尝试创建一个覆盖完整的数据准备和最终预测的流水线。

5.使用GridSearchCV自动探索一些准备选项。

以上练习题的解决方案可以在Jupyter notebook上获得，链接地址为https://github.com/ageron/handson-ml2。

第3章　分类

第1章提到，最常见的有监督学习任务包括回归任务（预测值）和分类任务（预测类）。第2章探讨了一个回归任务——预测住房价格，用到了线性回归、决策树以及随机森林等各种算法（我们将会在后续章节中进一步讲解这些算法）。本章中我们将把注意力转向分类系统。

3.1 MNIST

本章将使用MNIST数据集，这是一组由美国高中生和人口调查局员工手写的70 000个数字的图片。每张图片都用其代表的数字标记。这个数据集被广为使用，因此也被称作是机器学习领域的“Hello World”：但凡有人想到了一个新的分类算法，都会想看看在MNIST上的执行结果。因此只要是学习机器学习的人，早晚都要面对MNIST。

Scikit-Learn提供了许多助手功能来帮助你下载流行的数据集。MNIST也是其中之一。下面是获取MNIST数据集的代码：[1]

```
>>> from sklearn.datasets import fetch_openml
>>> mnist = fetch_openml('mnist_784', version=1)
>>> mnist.keys()
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details',
           'categories', 'url'])
```

Scikit-Learn加载的数据集通常具有类似的字典结构，包括：

- DESCR键，描述数据集。
- data键，包含一个数组，每个实例为一行，每个特征为一列。
- target键，包含一个带有标记的数组。

我们来看看这些数组：

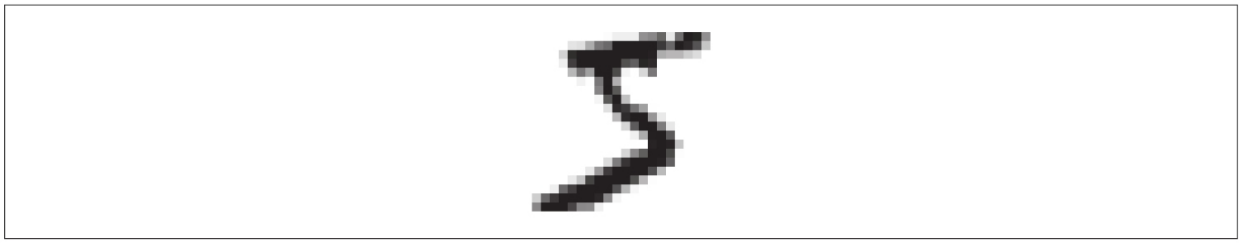
```
>>> X, y = mnist["data"], mnist["target"]
>>> X.shape
(70000, 784)
>>> y.shape
(70000,)
```

共有7万张图片，每张图片有784个特征。因为图片是28×28像素，每个特征代表了一个像素点的强度，从0（白色）到255（黑色）。先来看看数据集中的一个数字，你只需要随手抓取一个实例的特征向量，将其重新形成一个28×28数组，然后使用Matplotlib的imshow（）函数将其显示出来：

```
import matplotlib as mpl
import matplotlib.pyplot as plt

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)

plt.imshow(some_digit_image, cmap="binary")
plt.axis("off")
plt.show()
```



看起来像5，而标签告诉我们没错：

```
>>> y[0]
'5'
```

注意标签是字符，大部分机器学习算法希望是数字，让我们把y转换成整数：

```
>>> y = y.astype(np.uint8)
```

图3-1显示了更多MNIST数据集中的数字图像，让你对分类任务的复杂程度有一个初步的感受。

可是等等！在开始深入研究这些数据之前，你还是应该先创建一个测试集，并将其放在一边。事实上，MNIST数据集已经分成训练集（前6万张图片）和测试集（最后1万张图片）了：

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

图3-1：MNIST数据集中的数字

同样，我们先将训练集数据混洗，这样能保证交叉验证时所有的折叠都差不多（你肯定不希望某个折叠丢失一些数字）。此外，有些机器学习算法对训练实例的顺序敏感，如果连续输入许多相似的实例，可能导致执行性能不佳。给数据集混洗正是为了确保这种情况不会发生[2]。

[1] 默认情况下，Scikit-Learn将下载的数据集缓存在$HOME/scikit_learn_data目录下。
[2] 在某些情况下，例如，如果你正在处理时间序列数据（例如股市价格或天气状况），则混洗可能不是一个好主意。我们将在下一章中对此进行探讨。

3.2 训练二元分类器

现在先简化问题，只尝试识别一个数字，比如数字5。那么这个“数字5检测器”就是一个二元分类器的示例，它只能区分两个类别：5和非5。先为此分类任务创建目标向量：

```
y_train_5 = (y_train == 5)  # True for all 5s, False for all other digits
y_test_5 = (y_test == 5)
```

接着挑选一个分类器并开始训练。一个好的初始选择是随机梯度下降（SGD）分类器，使用Scikit-Learn的SGDClassifier类即可。这个分类器的优势是能够有效处理非常大型的数据集。这部分是因为SGD独立处理训练实例，一次一个（这也使得SGD非常适合在线学习），稍后我们将会看到。此时先创建一个SGDClassifier并在整个训练集上进行训练：

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

 SGDClassifier在训练时是完全随机的（因此得名“随机”），如果你希望得到可复现的结果，需要设置参数random_state。

现在可以用它来检测数字5的图片了：

```
>>> sgd_clf.predict([some_digit])
array([ True])
```

分类器猜这个图像代表5（True）。看起来这次它猜对了！那么，下面评估一下这个模型的性能。

3.3 性能测量

评估分类器比评估回归器要困难得多，因此本章将用很多篇幅来讨论这个主题，同时会涉及许多性能考核的方法。所以，不妨再倒一杯咖啡，做好准备学习更多的新概念和缩略词吧！

3.3.1 使用交叉验证测量准确率

正如第2章所述，交叉验证是一个评估模型的好办法。

实现交叉验证

相比于Scikit-Learn提供cross_val_score（）这一类交叉验证的函数，有时你可能希望自己能控制得多一些。在这种情况下，你可以自行实现交叉验证，操作也简单明了。下面这段代码与前面的cross_val_score（）大致相同，并打印出相同的结果：

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3, random_state=42)

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]

    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred)) # prints 0.9502, 0.96565, and 0.96495
```

每个折叠由StratifiedKFold执行分层抽样（参见第2章）产生，其所包含的各个类的比例符合整体比例。每个迭代会创建一个分类器的副

本，用训练集对这个副本进行训练，然后用测试集进行预测。最后计算正确预测的次数，输出正确预测的比率。

现在，用cross_val_score（）函数来评估SGDClassifier模型，采用K-折交叉验证法（3个折叠）。记住，K-折交叉验证的意思是将训练集分解成K个折叠（在本例中，为3折），然后每次留其中1个折叠进行预测，剩余的折叠用来训练（参见第2章）：

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.96355, 0.93795, 0.95615])
```

所有折叠交叉验证的准确率（正确预测的比率）超过93%？看起来挺神奇的，是吗？不过在你开始激动之前，我们来看一个蠢笨的分类器，它将每张图都分类成“非5”：

```
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        return self
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

能猜到这个模型的准确率吗？我们看看：

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.91125, 0.90855, 0.90915])
```

没错，准确率超过90%！这是因为只有大约10%的图片是数字5，所以如果你猜一张图不是5，90%的概率你都是正确的，简直超越了大预言

家！

这说明准确率通常无法成为分类器的首要性能指标，特别是当你处理有偏数据集时（即某些类比其他类更为频繁）。

3.3.2 混淆矩阵

评估分类器性能的更好方法是混淆矩阵，其总体思路就是统计A类别实例被分成为B类别的次数。例如，要想知道分类器将数字3和数字5混淆多少次，只需要通过混淆矩阵的第5行第3列来查看。

要计算混淆矩阵，需要先有一组预测才能将其与实际目标进行比较。当然，可以通过测试集来进行预测，但是现在先不要动它（测试集最好留到项目的最后，准备启动分类器时再使用）。作为替代，可以使用cross_val_predict（）函数：

```
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

与cross_val_score（）函数一样，cross_val_predict（）函数同样执行K-折交叉验证，但返回的不是评估分数，而是每个折叠的预测。这意味着对于每个实例都可以得到一个干净的预测（“干净”的意思是模型预测时使用的数据在其训练期间从未见过）。

现在可以使用confusion_matrix（）函数来获取混淆矩阵了。只需要给出目标类别（y_train_5）和预测类别（y_train_pred）即可：

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53057,  1522],
       [ 1325,  4096]])
```

混淆矩阵中的行表示实际类别，列表示预测类别。本例中第一行表示所有“非5”（负类）的图片中：53 057张被正确地分为“非5”类别（真负类），1522张被错误地分类成了“5”（假正类）；第二行表示所有“5”（正类）的图片中：1325张被错误地分为“非5”类别（假负类），4096张被正确地分在了“5”这一类别（真正类）。一个完美的分类器只有真正类和真负类，所以它的混淆矩阵只会在其对角线（左上到右下）上有非零值：

```
>>> y_train_perfect_predictions = y_train_5  # pretend we reached perfection
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579,     0],
       [    0,  5421]])
```

混淆矩阵能提供大量信息，但有时你可能希望指标更简洁一些。正类预测的准确率是一个有意思的指标，它也称为分类器的精度（见公式3-1）：

公式3-1：精度

$$\text{精度} = \frac{TP}{TP + FP}$$

TP是真正类的数量，FP是假正类的数量。

做一个单独的正类预测，并确保它是正确的，就可以得到完美精度（精度=1/1=100%）。但这没什么意义，因为分类器会忽略这个正类实例之外的所有内容。因此，精度通常与另一个指标一起使用，这个指标就是召回率，也称为灵敏度或者真正类率：它是分类器正确检测到的正类实例的比率（见公式3-2）：

公式3-2：召回率

$$召回率 = \frac{TP}{TP + FN}$$

FN是假负类的数量。

如果你对混淆矩阵还是感到疑惑，图3-2或许可以帮助你理解。

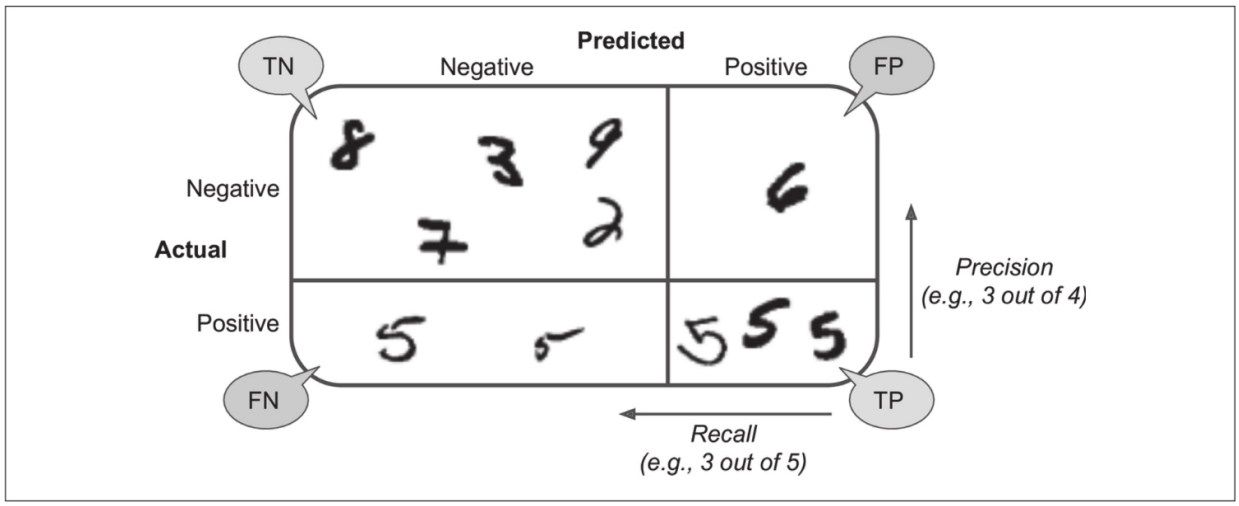


图3-2：混淆矩阵显示了真负（左上）、假正（右上）、假负（左下）和真正（右下）的示例

3.3.3 精度和召回率

Scikit-Learn提供了计算多种分类器指标的函数，包括精度和召回率：

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1522)
0.7290850836596654
>>> recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)
0.7555801512636044
```

现在再看，这个5-检测器看起来并不像它的准确率那么光鲜亮眼了。当它说一张图片是5时，只有72.9%的概率是准确的，并且也只有75.6%的数字5被它检测出来了。

因此我们可以很方便地将精度和召回率组合成一个单一的指标，称为F_1分数。当你需要一个简单的方法来比较两种分类器时，这是个非常不错的指标。F_1分数是精度和召回率的谐波平均值（见公式3-3）。正常的平均值平等对待所有的值，而谐波平均值会给予低值更高的权重。因此，只有当召回率和精度都很高时，分类器才能得到较高的F1分数。

公式3-3：F_1

$$F_1 = \frac{2}{\frac{1}{精度} + \frac{1}{召回率}} = 2 \times \frac{精度 \times 召回率}{精度 + 召回率} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

要计算F1分数，只需要调用f1_score（）即可：

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7420962043663375
```

F1分数对那些具有相近的精度和召回率的分类器更为有利。这不一定能一直符合你的期望：在某些情况下，你更关心的是精度，而另一些情况下，你可能真正关心的是召回率。例如，假设你训练一个分类器来检测儿童可以放心观看的视频，那么你可能更青睐那种拦截了很多好视频（低召回率），但是保留下来的视频都是安全（高精度）的分类器，而不是召回率虽高，但是在产品中可能会出现一些非常糟糕的视频的分类器（这种情况下，你甚至可能会添加一个人工流水线来检查分类器选出来的视频）。反过来说，如果你训练一个分类器通过图像监控来检测

小偷：你大概可以接受精度只有30%，但召回率能达到99%（当然，安保人员会收到一些错误的警报，但是几乎所有的窃贼都在劫难逃）。

遗憾的是，鱼和熊掌不可兼得，你不能同时增加精度又减少召回率，反之亦然。这称为精度/召回率权衡。

3.3.4 精度/召回率权衡

要理解这个权衡过程，我们来看看SGDClassifier如何进行分类决策。对于每个实例，它会基于决策函数计算出一个分值，如果该值大于阈值，则将该实例判为正类，否则便将其判为负类。图3-3显示了从左边最低分到右边最高分的几个数字。假设决策阈值位于中间箭头位置（两个5之间）：在阈值的右侧可以找到4个真正类（真的5）和一个假正类（实际上是6）。因此，在该阈值下，精度为80%（4/5）。但是在6个真正的5中，分类器仅检测到了4个，所以召回率为67%（4/6）。现在，如果提高阈值（将其挪动到右边箭头的位置），假正类（数字6）变成了真负类，因此精度得到提升（本例中提升到100%），但是一个真正类变成一个假负类，召回率降低至50%。反之，降低阈值则会在增加召回率的同时降低精度。

Scikit-Learn不允许直接设置阈值，但是可以访问它用于预测的决策分数。不是调用分类器的predict（）方法，而是调用decision_function（）方法，这种方法返回每个实例的分数，然后就可以根据这些分数，使用任意阈值进行预测了：

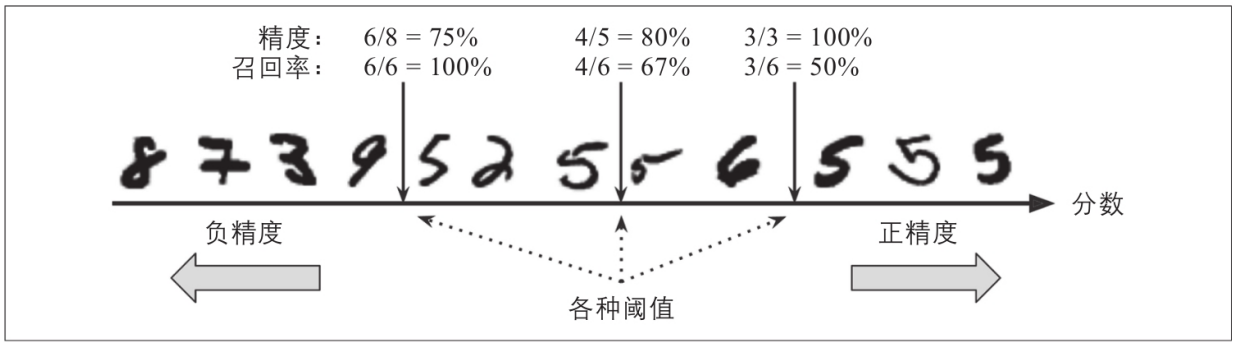


图3-3：在这个精度/召回率权衡中，图像按其分类器评分进行排名，而高于所选决策阈值的图像被认为是正的；阈值越高，召回率越低，但是（通常）精度越高

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([2412.53175101])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True])
```

SGDClassifier分类器使用的阈值是0，所以前面代码的返回结果与predict（）方法一样（也就是True）。我们来试试提升阈值：

```
>>> threshold = 8000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False])
```

这证明了提高阈值确实可以降低召回率。这张图确实是5，当阈值为0时，分类器可以检测到该图，但是当阈值提高到8000时，就错过了这张图。

那么要如何决定使用什么阈值呢？首先，使用cross_val_predict（）函数获取训练集中所有实例的分数，但是这次需要它返回的是决策分数而不是预测结果：

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

有了这些分数，可以使用precision_recall_curve（）函数来计算所有可能的阈值的精度和召回率：

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

最后，使用Matplotlib绘制精度和召回率相对于阈值的函数图（见图3-4）：

```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    [...] # highlight the threshold and add the legend, axis label, and grid

plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```

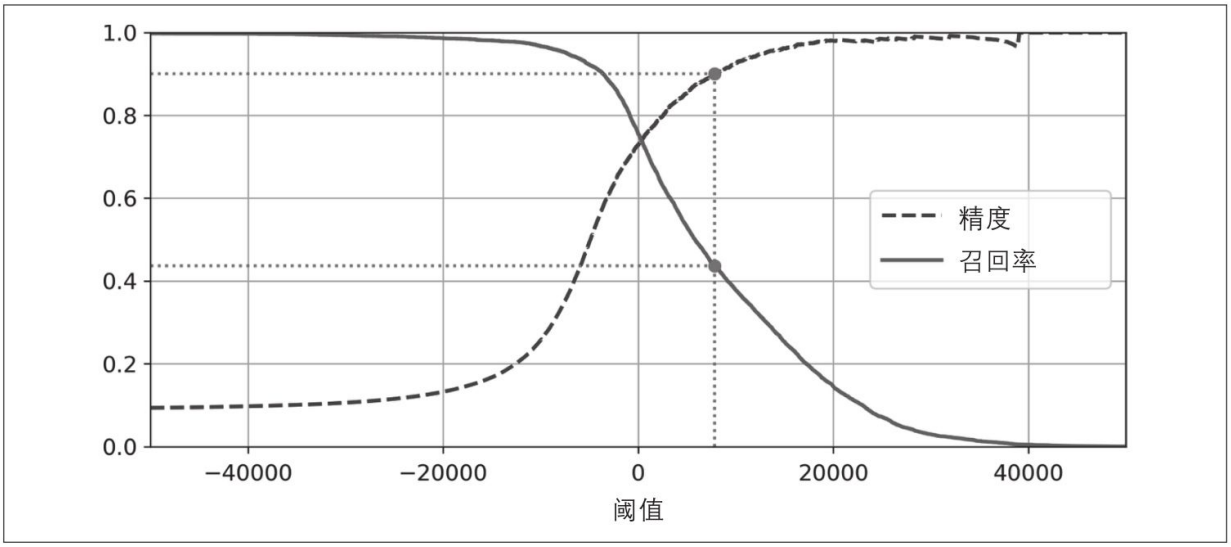


图3-4：精度和召回率与决策阈值

 你可能会感到好奇，为什么在图3-4中精度曲线比召回率曲线要崎岖一些？原因在于，当你提高阈值时，精度有时也有可能会下降（尽管总体趋势是上升的）。要理解原因，可以回头看图3-3，注意，当把阈值从中间箭头往右移动一位数时：精度从4/5（80%）下降到3/4（75%）。另一方面，当阈值上升时，召回率只会下降，这就解释了为什么召回率的曲线看起来很平滑。

另一种找到好的精度/召回率权衡的方法是直接绘制精度和召回率的函数图，如图3-5所示（突出显示与前面相同的阈值）。

从图中可以看到，从80%的召回率往右，精度开始急剧下降。你可能会尽量在这个陡降之前选择一个精度/召回率权衡——比如召回率60%。当然，如何选择取决于你的项目。

假设你决定将精度设为90%。查找图3-4并发现需要设置8000的阈值。更精确地说，你可以搜索到能提供至少90%精度的最低阈值（np.argmax（）会给你最大值的第一个索引，在这种情况下，它表示第一个True值）：

```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]
    # ~7816
```

要进行预测（现在是在训练集上），除了调用分类器的predict（）方法，也可以运行这段代码：

```
y_train_pred_90 = (y_scores >= threshold_90_precision)
```

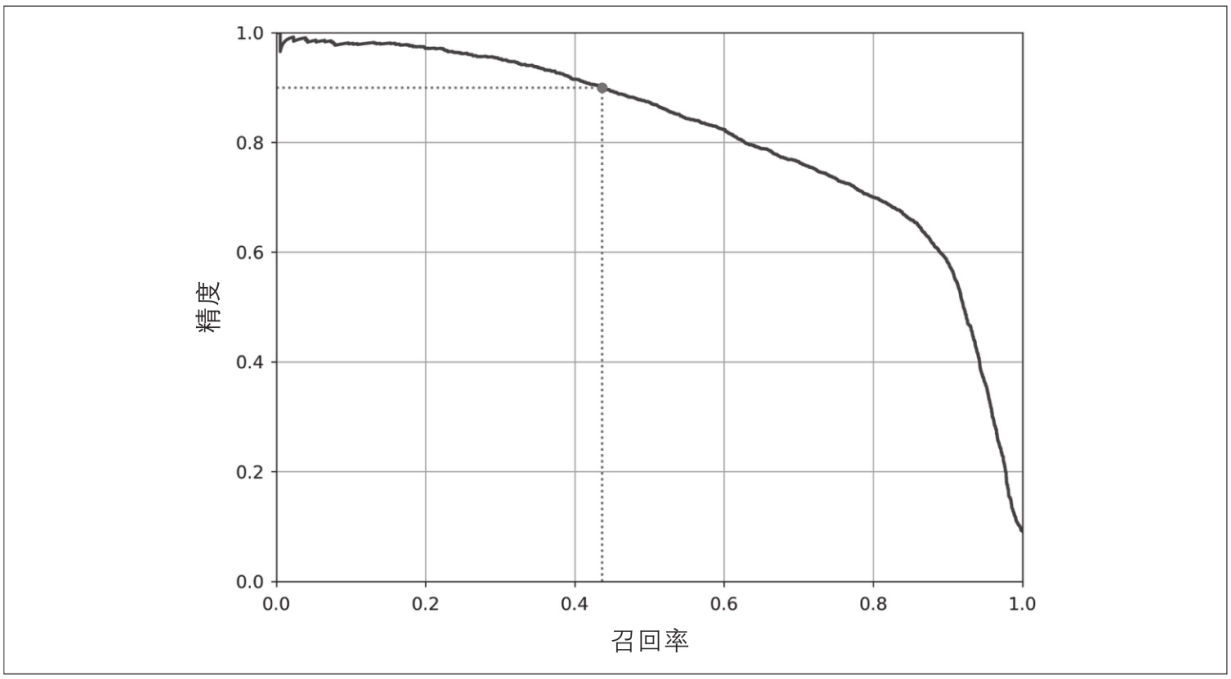


图3-5：精度与召回率

检查一下这些预测结果的精度和召回率：

```
>>> precision_score(y_train_5, y_train_pred_90)
0.9000380083618396
>>> recall_score(y_train_5, y_train_pred_90)
0.4368197749492714
```

现在你有一个90%精度的分类器了（或者足够接近）！如你所见，创建任意一个你想要的精度的分类器是相当容易的事情：只要阈值足够高即可！然而，如果召回率太低，精度再高，其实也不怎么有用！

 如果有人说：“我们需要99%的精度。”你就应该问：“召回率是多少？”

3.3.5 ROC曲线

还有一种经常与二元分类器一起使用的工具，叫作受试者工作特征曲线（简称ROC）。它与精度/召回率曲线非常相似，但绘制的不是精度和召回率，而是真正类率（召回率的另一名称）和假正类率（FPR）。FPR是被错误分为正类的负类实例比率。它等于1减去真负类率（TNR），后者是被正确分类为负类的负类实例比率，也称为特异度。因此，ROC曲线绘制的是灵敏度（召回率）和（1-特异度）的关系。

要绘制ROC曲线，首先需要使用roc_curve（）函数计算多种阈值的TPR和FPR：

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

然后，使用Matplotlib绘制FPR对TPR的曲线。下面的代码可以绘制出图3-6的曲线：

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--') # Dashed diagonal
    [...] # Add axis labels and grid

plot_roc_curve(fpr, tpr)
plt.show()
```

同样这里再次面临一个折中权衡：召回率（TPR）越高，分类器产生的假正类（FPR）就越多。虚线表示纯随机分类器的ROC曲线、一个优秀的分类器应该离这条线越远越好（向左上角）。

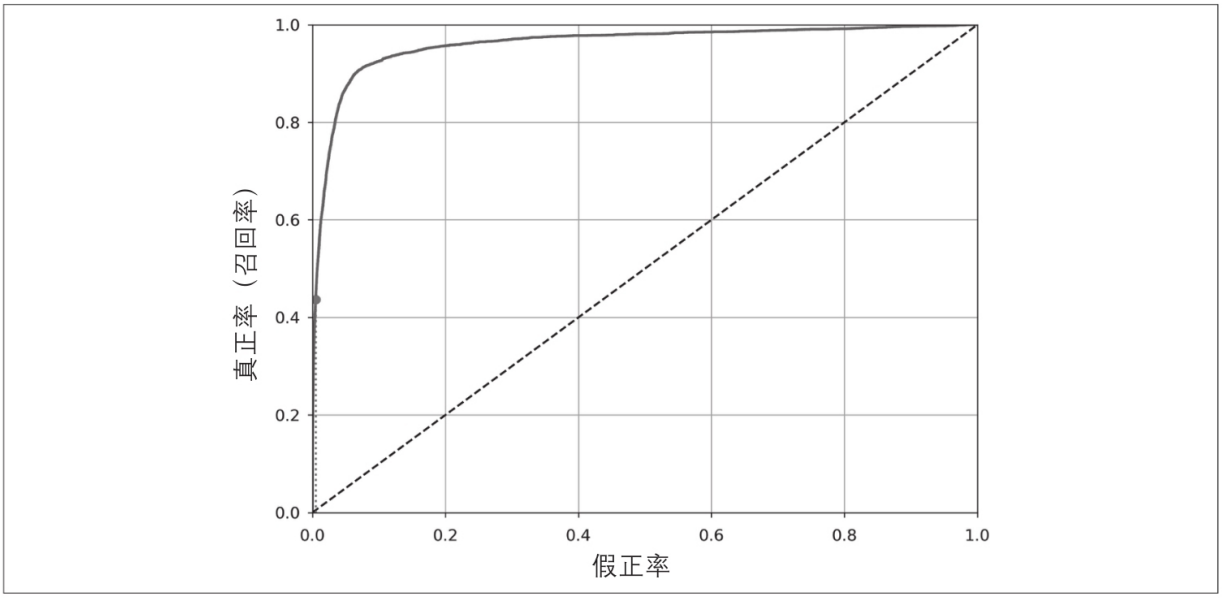


图3-6：该ROC曲线绘制了所有可能阈值的假正率与真正率的关系。粗点处突出显示了选定的比率（召回率为43.68%）

有一种比较分类器的方法是测量曲线下面积（AUC）。完美的分类器的ROC AUC等于1，而纯随机分类器的ROC AUC等于0.5。Scikit-Learn提供计算ROC AUC的函数：

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.9611778893101814
```

 由于ROC曲线与精度/召回率（PR）曲线非常相似，因此你可能会问如何决定使用哪种曲线。有一个经验法则是，当正类非常少见或者你更关注假正类而不是假负类时，应该选择PR曲线，反之则是ROC曲线。例如，看前面的ROC曲线图（以及ROC AUC分数），你可能会觉得分类器真不错。但这主要是因为跟负类（非5）相比，正类（数字5）的数量真的很少。相比之下，PR曲线清楚地说明分类器还有改进的空间（曲线还可以更接近左上角）。

现在我们来训练一个RandomForestClassifier分类器，并比较它和SGDClassifier分类器的ROC曲线和ROC AUC分数。首先，获取训练集中每个实例的分数。但是由于它的工作方式不同（参见第7章），RandomForestClassifier类没有decision_function（）方法，相反，它有dict_proba（）方法。Scikit-Learn的分类器通常都会有这两种方法中的一种（或两种都有）。dict_proba（）方法会返回一个数组，其中每行代表一个实例，每列代表一个类别，意思是某个给定实例属于某个给定类别的概率（例如，这张图片有70%的可能是数字5）：

```
from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")
```

roc_curve（）函数需要标签和分数，但是我们不提供分数，而是提供类概率。我们直接使用正类的概率作为分数值：

```
y_scores_forest = y_probas_forest[:, 1]   # score = proba of positive class
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

现在可以绘制ROC曲线了。绘制第一条ROC曲线来看看对比结果（见图3-7）：

```
plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="lower right")
plt.show()
```

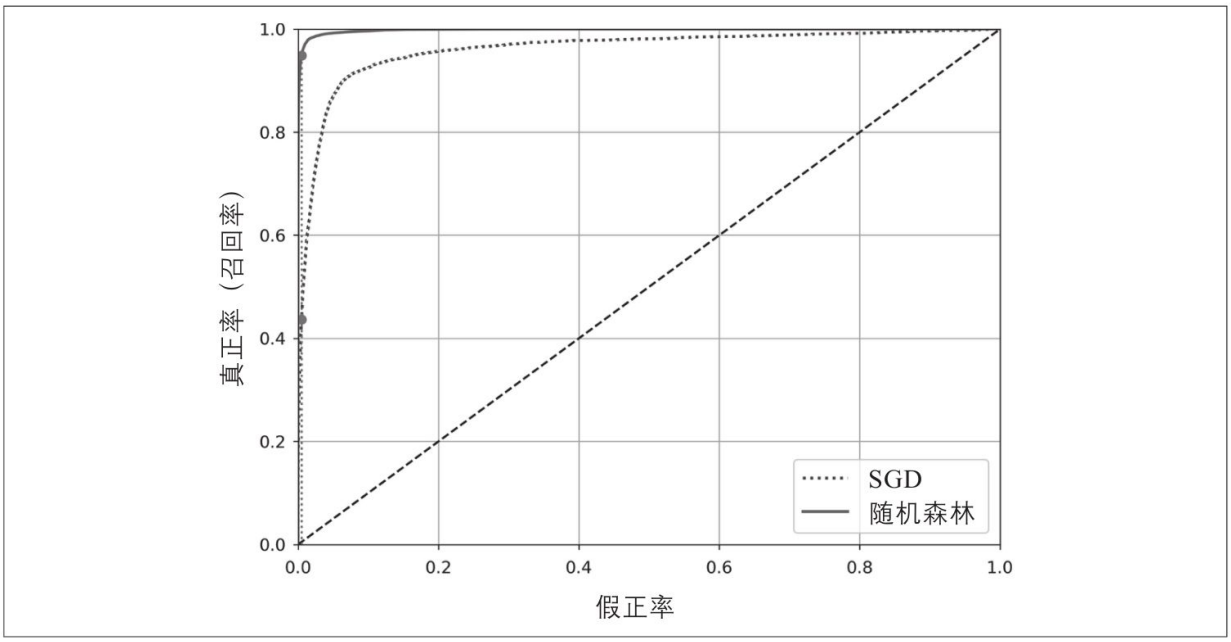


图3-7：比较ROC曲线：随机森林分类器优于SGD分类器，因为它的ROC曲线更靠近左上角，并且具有更大的AUC

如图3-7所示，RandomForestClassifier的ROC曲线看起来比SGDClassifier好很多，它离左上角更接近，因此它的ROC AUC分数也高得多：

```
>>> roc_auc_score(y_train_5, y_scores_forest)
0.9983436731328145
```

再测一测精度和召回率的分数：99.0%的精度和86.6%的召回率，也还不错！

希望现在你已经掌握了如何训练二元分类器，如何选择合适的指标利用交叉验证来对分类器进行评估，如何选择满足需求的精度/召回率权衡，以及如何使用ROC曲线和ROC AUC分数来比较多个模型。我们再来试试对数字5之外的检测。

3.4 多类分类器

二元分类器在两个类中区分，而多类分类器（也称为多项分类器）可以区分两个以上的类。

有一些算法（如随机森林分类器或朴素贝叶斯分类器）可以直接处理多个类。也有一些严格的二元分类器（如支持向量机分类器或线性分类器）。但是，有多种策略可以让你用几个二元分类器实现多类分类的目的。

要创建一个系统将数字图片分为10类（从0到9），一种方法是训练10个二元分类器，每个数字一个（0-检测器、1-检测器、2-检测器，以此类推）。然后，当你需要对一张图片进行检测分类时，获取每个分类器的决策分数，哪个分类器给分最高，就将其分为哪个类。这称为一对剩余（OvR）策略，也称为一对多（one-versus-all）。

另一种方法是为每一对数字训练一个二元分类器：一个用于区分0和1，一个区分0和2，一个区分1和2，以此类推。这称为一对一（OvO）策略。如果存在N个类别，那么这需要训练N×（N-1）/2个分类器。对于MNIST问题，这意味着要训练45个二元分类器！当需要对一张图片进行分类时，你需要运行45个分类器来对图片进行分类，最后看哪个类获胜最多。OvO的主要优点在于，每个分类器只需要用到部分训练集对其必须区分的两个类进行训练。

有些算法（例如支持向量机分类器）在数据规模扩大时表现糟糕。对于这类算法，OvO是一个优先的选择，因为在较小训练集上分别训练多个分类器比在大型数据集上训练少数分类器要快得多。但是对大多数二元分类器来说，OvR策略还是更好的选择。

Scikit-Learn可以检测到你尝试使用二元分类算法进行多类分类任务，它会根据情况自动运行OvR或者OvO。我们用sklearn.svm.SVC类来试试SVM分类器（见第5章）：

```
>>> from sklearn.svm import SVC
>>> svm_clf = SVC()
>>> svm_clf.fit(X_train, y_train) # y_train, not y_train_5
>>> svm_clf.predict([some_digit])
array([5], dtype=uint8)
```

非常容易！这段代码使用原始目标类0到9（y_train）在训练集上对SVC进行训练，而不是以“5”和“剩余”作为目标类（y_train_5），然后做出预测（在本例中预测正确）。而在内部，Scikit-Learn实际上训练了45个二元分类器，获得它们对图片的决策分数，然后选择了分数最高的类。

要想知道是不是这样，可以调用decision_function（）方法。它会返回10个分数，每个类1个，而不再是每个实例返回1个分数：

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])
>>> some_digit_scores
array([[ 2.92492871,  7.02307409,  3.93648529,  0.90117363,  5.96945908,
         9.5       ,  1.90718593,  8.02755089, -0.13202708,  4.94216947]])
```

最高分确实是对应数字5这个类别：

```
>>> np.argmax(some_digit_scores)
5
>>> svm_clf.classes_
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
>>> svm_clf.classes_[5]
5
```

 当训练分类器时，目标类的列表会存储在classes_属性中，按值的大小排序。在本例里，classes_数组中每个类的索引正好对应其类本身（例如，索引上第5个类正好是数字5这个类），但是一般来说，不会这么恰巧。

如果想要强制Scikit-Learn使用一对一或者一对剩余策略，可以使用OneVsOneClassifier或OneVsRestClassifier类。只需要创建一个实例，然后将分类器传给其构造函数（它甚至不必是二元分类器）。例如，下面这段代码使用OvR策略，基于SVC创建了一个多类分类器：

```
>>> from sklearn.multiclass import OneVsRestClassifier
>>> ovr_clf = OneVsRestClassifier(SVC())
>>> ovr_clf.fit(X_train, y_train)
>>> ovr_clf.predict([some_digit])
array([5], dtype=uint8)
>>> len(ovr_clf.estimators_)
10
```

训练SGDClassifier或者RandomForestClassifier同样简单：

```
>>> sgd_clf.fit(X_train, y_train)
>>> sgd_clf.predict([some_digit])
array([5], dtype=uint8)
```

这次Scikit-Learn不必运行OvR或者OvO了，因为SGD分类器直接就可以将实例分为多个类。调用decision_function（）可以获得分类器将每个实例分类为每个类的概率列表：让我们看一下SGD分类器分配到的每个类：

```
>>> sgd_clf.decision_function([some_digit])
array([[-15955.22628, -38080.96296, -13326.66695,   573.52692, -17680.68466,
         2412.53175, -25526.86498, -12290.15705, -7946.05205, -10631.35889]])
```

你可以看到分类器对其预测相当有信心：几乎所有分数基本上都是负的，而第5类的得分是2412.5。该模型有一点疑问是关于第3类，得分为573.5。现在，你当然要评估这个分类器。与往常一样，可以使用交叉验证。使用cross_val_score（）函数来评估SGDClassifier的准确性：

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([0.8489802 , 0.87129356, 0.86988048])
```

在所有的测试折叠上都超过了84%。如果是一个纯随机分类器，准确率大概是10%，所以这个结果不是太糟，但是依然有提升的空间。例如，将输入进行简单缩放（如第2章所述）可以将准确率提到89%以上：

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
array([0.89707059, 0.8960948 , 0.90693604])
```

3.5 误差分析

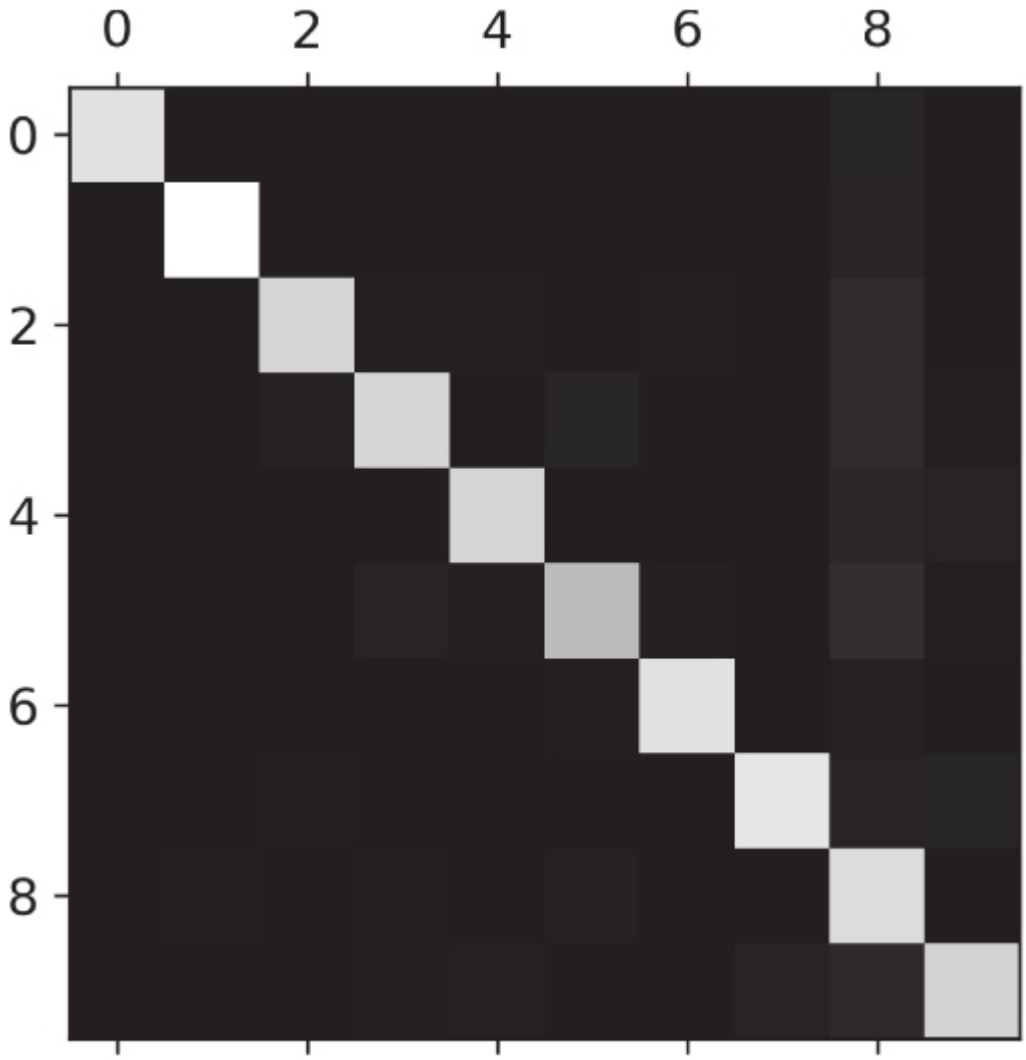
当然，如果这是一个真正的项目，你将遵循机器学习项目清单中的步骤（见附录B）：探索数据准备的选项，尝试多个模型，列出最佳模型并用GridSearchCV对其超参数进行微调，尽可能自动化，等等。正如你在之前的章节里尝试的那些。在这里，假设你已经找到了一个有潜力的模型，现在你希望找到一些方法对其进一步改进。方法之一就是分析其错误类型。

首先看看混淆矩阵。就像之前做的，使用cross_val_predict（）函数进行预测，然后调用confusion_matrix（）函数：

```
>>> y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
>>> conf_mx = confusion_matrix(y_train, y_train_pred)
>>> conf_mx
array([[5578,    0,   22,    7,    8,   45,   35,    5,  222,    1],
       [   0, 6410,   35,   26,    4,   44,    4,    8,  198,   13],
       [  28,   27, 5232,  100,   74,   27,   68,   37,  354,   11],
       [  23,   18,  115, 5254,    2,  209,   26,   38,  373,   73],
       [  11,   14,   45,   12, 5219,   11,   33,   26,  299,  172],
       [  26,   16,   31,  173,   54, 4484,   76,   14,  482,   65],
       [  31,   17,   45,    2,   42,   98, 5556,    3,  123,    1],
       [  20,   10,   53,   27,   50,   13,    3, 5696,  173,  220],
       [  17,   64,   47,   91,    3,  125,   24,   11, 5421,   48],
       [  24,   18,   29,   67,  116,   39,    1,  174,  329, 5152]])
```

数字有点多，使用Matplotlib的matshow（）函数来查看混淆矩阵的图像表示通常更加方便（见下图）：

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



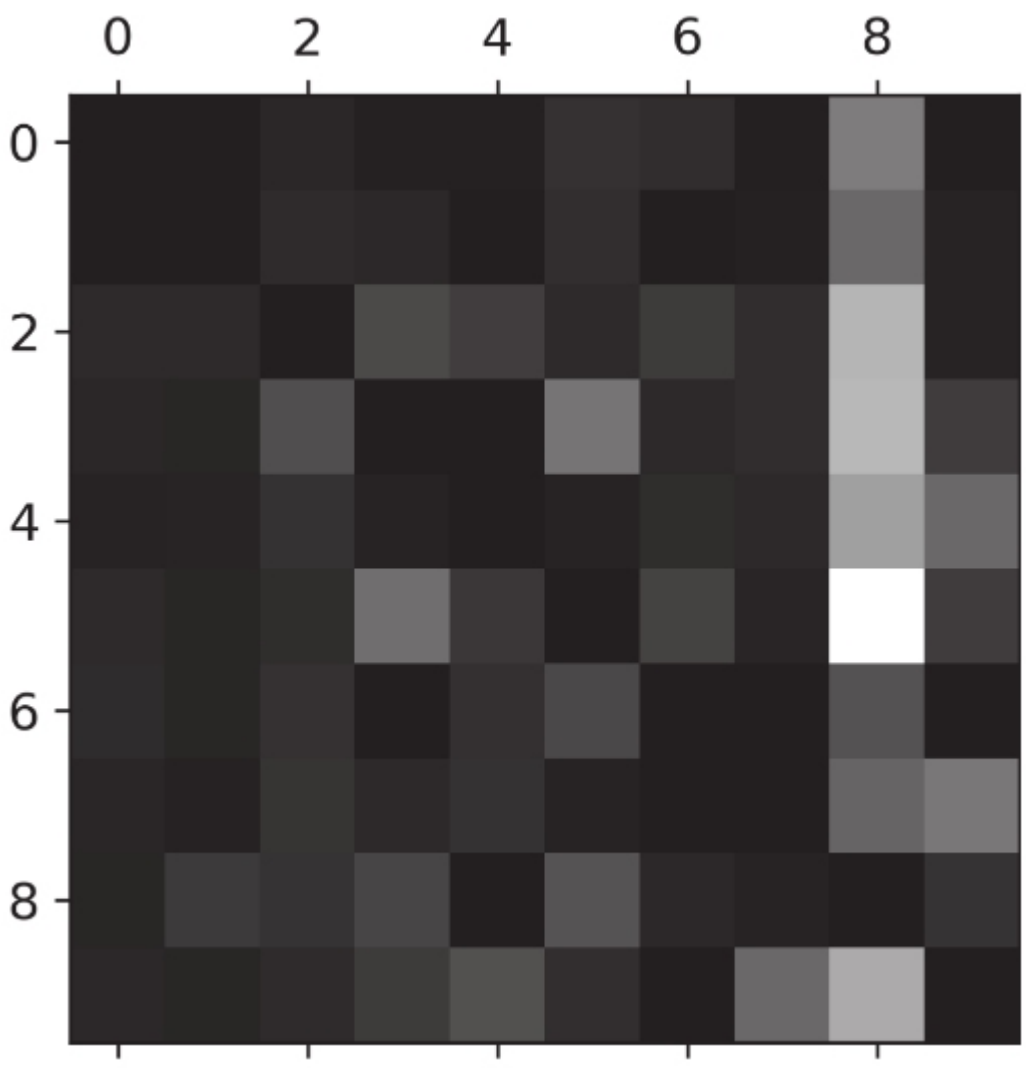
混淆矩阵看起来很不错，因为大多数图片都在主对角线上，这说明它们被正确分类。数字5看起来比其他数字稍稍暗一些，这可能意味着数据集中数字5的图片较少，也可能是分类器在数字5上的执行效果不如在其他数字上好。实际上，你可能会验证这两者都属实。

让我们把焦点放在错误上。首先，你需要将混淆矩阵中的每个值除以相应类中的图片数量，这样你比较的就是错误率而不是错误的绝对值（后者对图片数量较多的类不公平）：

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

用0填充对角线，只保留错误，重新绘制结果（见下图）：

```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```

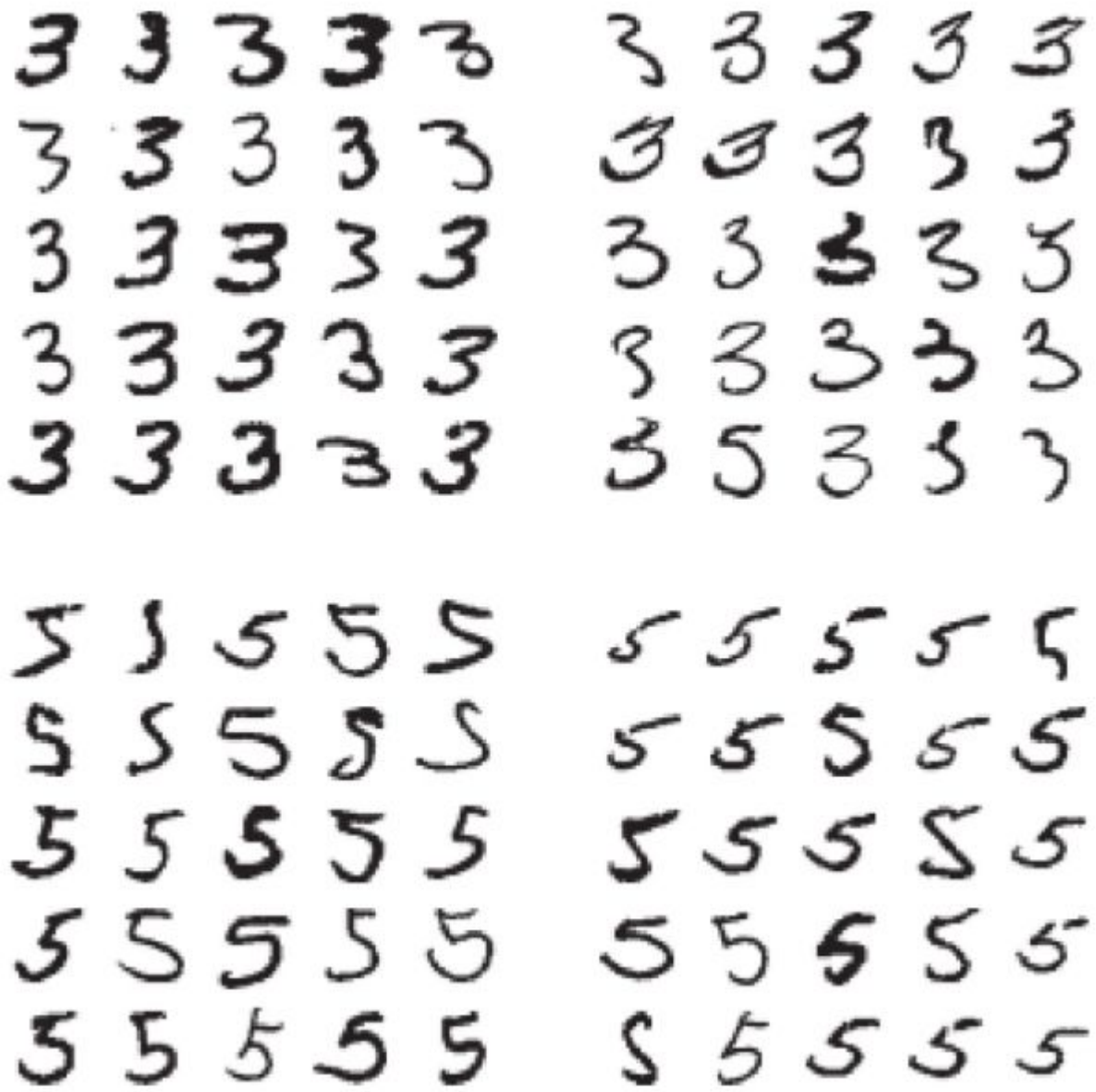


现在可以清晰地看到分类器产生的错误种类了。记住，每行代表实际类，而每列表示预测类。第8列看起来非常亮，说明有许多图片被错误地分类为数字8了。然而，第8行不那么差，告诉你实际上数字8被正确分类为数字8。注意，错误不是完全对称的，比如，数字3和数字5经常被混淆（在两个方向上）。

分析混淆矩阵通常可以帮助你深入了解如何改进分类器。通过上图来看，你的精力可以花在改进数字8的分类错误上。例如，可以试着收集更多看起来像数字8的训练数据，以便分类器能够学会将它们与真实的数字区分开来。或者，也可以开发一些新特征来改进分类器——例如，写一个算法来计算闭环的数量（例如，数字8有两个，数字6有一个，数字5没有）。再或者，还可以对图片进行预处理（例如，使用Scikit-Image、Pillow或OpenCV）让某些模式更为突出，比如闭环之类的。

分析单个的错误也可以为分类器提供洞察：它在做什么？它为什么失败？但这通常更加困难和耗时。例如，我们来看看数字3和数字5的示例（plot_digits（）函数只是使用Matplotlib的imshow（）函数，详见Jupyter notebook）：

```
cl_a, cl_b = 3, 5
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]
plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
plt.show()
```



左侧的两个5×5矩阵显示了被分类为数字3的图片，右侧的两个5×5矩阵显示了被分类为数字5的图片。分类器弄错的数字（即左下方和右上方的矩阵）里，确实有一些写得非常糟糕，即便是人类也很难做出区分（例如，第1行的数字5看起来真的很像数字3）。然而，对我们来说，大多数错误分类的图片看起来还是非常明显的错误，我们很难理解分类器为什么会弄错[1]。原因在于，我们使用的简单的SGDClassifier模型是一个线性模型。它所做的就是为每个像素分配一个各个类别的权重，当它看到新的图像时，将加权后的像素强度汇总，从而得到一个分数进行分类。而数字3和数字5只在一部分像素位上有区别，所以分类器很容易将其弄混。

数字3和数字5之间的主要区别是在于连接顶线和下方弧线的中间那段小线条的位置。如果你写的数字3将连接点略往左移，分类器就可能将其分类为数字5，反之亦然。换言之，这个分类器对图像移位和旋转非常敏感。因此，减少数字3和数字5混淆的方法之一，就是对图片进行预处理，确保它们位于中心位置并且没有旋转。这也同样有助于减少其他错误。

[1] 但是请记住，我们的大脑是一个奇妙的模式识别系统，并且在任何信息到达我们的意识之前，我们的视觉系统都会进行很多复杂的预处理，因此感觉简单的事实并不意味着事实简单。

3.6 多标签分类

到目前为止，每个实例都只会被分在一个类里。而在某些情况下，你希望分类器为每个实例输出多个类。例如，人脸识别的分类器：如果在一张照片里识别出多个人怎么办？当然，应该为识别出来的每个人都附上一个标签。假设分类器经过训练，已经可以识别出三张脸——爱丽丝、鲍勃和查理，那么当看到一张爱丽丝和查理的照片时，它应该输出[1，0，1]（意思是“是爱丽丝，不是鲍勃，是查理”）这种输出多个二元标签的分类系统称为多标签分类系统。

为了阐释清楚，这里不讨论面部识别，让我们来看一个更为简单的示例：

```
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

这段代码会创建一个y_multilabel数组，其中包含两个数字图片的目标标签：第一个表示数字是否是大数（7、8、9），第二个表示是否为奇数。下一行创建一个KNeighborsClassifier实例（它支持多标签分类，不是所有的分类器都支持），然后使用多个目标数组对它进行训练。现在用它做一个预测，注意它输出两个标签：

```
>>> knn_clf.predict([some_digit])
array([[False,  True]])
```

结果是正确的！数字5确实不大（False），为奇数（True）。

评估多标签分类器的方法很多，如何选择正确的度量指标取决于你的项目。比如方法之一是测量每个标签的F1分数（或者之前讨论过的任何其他二元分类器指标），然后简单地计算平均分数。下面这段代码计算所有标签的平均F1分数：

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel, cv=3)
>>> f1_score(y_multilabel, y_train_knn_pred, average="macro")
0.976410265560605
```

这里假设所有的标签都同等重要，但实际可能不是这样。特别地，如果训练的照片里爱丽丝比鲍勃和查理要多很多，你可能想给区分爱丽丝的分类器更高的权重。一个简单的办法是给每个标签设置一个等于其自身支持的权重（也就是具有该目标标签的实例的数量）。为此，只需要在上面的代码中设置average="weighted"即可[1]。

[1] Scikit-Learn提供了其他一些平均选项和多标签分类器指标。有关更多详细信息，请参见文档。

3.7 多输出分类

我们即将讨论的最后一种分类任务称为多输出-多类分类（或简单地称为多输出分类）。简单来说，它是多标签分类的泛化，其标签也可以是多类的（比如它可以有两个以上可能的值）。

为了说明这一点，构建一个系统去除图片中的噪声。给它输入一张有噪声的图片，它将（希望）输出一张干净的数字图片，与其他MNIST图片一样，以像素强度的一个数组作为呈现方式。请注意，这个分类器的输出是多个标签（一个像素点一个标签），每个标签可以有多个值（像素强度范围为0到225）。所以这是个多输出分类器系统的示例。

 分类和回归之间的界限有时很模糊，比如这个示例。可以说，预测像素强度更像是回归任务而不是分类。而多输出系统也不仅仅限于分类任务，可以让一个系统给每个实例输出多个标签，同时包括类标签和值标签。

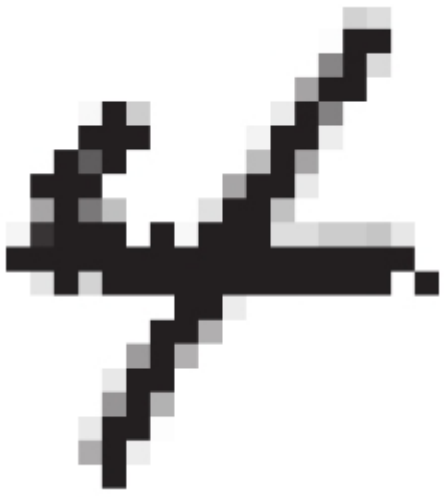
还先从创建训练集和测试集开始，使用NumPy的randint（）函数为MNIST图片的像素强度增加噪声。目标是将图片还原为原始图片：

```
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

瞥一眼测试集中的图像（没错，我们正在窥探测试数据，你现在确实应该皱眉头）：

左边是有噪声的输入图片，右边是干净的目标图片。现在通过训练分类器，清洗这张图片：

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```



看起来离目标够接近了。分类器之旅到此结束。希望现在你掌握了如何为分类任务选择好的指标，如何选择适当的精度/召回率权衡，

如何比较多个分类器，以及更为概括地说，如何为各种任务构建卓越的分类系统。

3.8 练习题

1.为MNIST数据集构建一个分类器，并在测试集上达成超过97%的准确率。提示：KNeighborsClassifier对这个任务非常有效，你只需要找到合适的超参数值即可（试试对weights和n_neighbors这两个超参数进行网格搜索）。

2.写一个可以将MNIST图片向任意方向（上、下、左、右）移动一个像素的功能[1]。然后对训练集中的每张图片，创建四个位移后的副本（每个方向一个），添加到训练集。最后，在这个扩展过的训练集上训练模型，测量其在测试集上的准确率。你应该能注意到，模型的表现甚至变得更好了！这种人工扩展训练集的技术称为数据增广或训练集扩展。

3.Kaggle上非常棒的起点：处理泰坦尼克（Titanic）数据集。

4.创建一个垃圾邮件分类器（更具挑战性的练习）：

- 从Apache SpamAssassin的公共数据集中下载垃圾邮件和非垃圾邮件。

- 解压数据集并熟悉数据格式。

- 将数据集分为训练集和测试集。

- 写一个数据准备的流水线将每封邮件转换为特征向量。你的流水线应将电子邮件转换为一个“指示出所有可能的词存在与否”的（稀疏）向量。比如，如果所有的邮件都只包含四个词“Hello”“how”“are”“you”，那么邮件“Hello you Hello Hello you”会被转换成为向量[1，0，0，1]（意思是“Hello”存在，“how”不

存在，“are”不存在，“you”存在），如果你希望算上每个词出现的次数，那么这个向量就是[3，0，0，2]。

在流水线上添加超参数来控制是否剥离电子邮件标题，是否将每封邮件转换为小写，是否删除标点符号，是否将“URLs”替换成“URL”，是否将所有小写number替换为“NUMBER”，甚至是否执行词干提取（即去掉单词后缀，有可用的Python库可以实现该操作）。

最后，多试几个分类器，看看是否能创建出一个高召回率且高精度的垃圾邮件分类器。

以上练习题的解答可以在Jupyter notebook上获得，链接地址为：https://github.com/ageron/handson-ml2。

[1] 你可以使用scipy.ndimage.interpolation模块中的shift（）函数。例如，shift（image，[2，1]，cval=0）将图像向下移动两个像素，向右移动一个像素。

第4章　训练模型

到目前为止，我们已经探讨了不同机器学习的模型，但是它们各自的训练算法在很大程度上还是一个黑匣子。回顾前几章里的部分案例，你大概感到非常惊讶，在对系统内部一无所知的情况下，居然已经实现了这么多：优化了一个回归系统，改进了一个数字图片分类器，从零开始构建了一个垃圾邮件分类器，所有这些，你都不知道它们实际是如何工作的。确实是这样，在许多情况下，你并不需要了解实施细节。

但是，很好地理解系统如何工作也是非常有帮助的。针对你的任务，它有助于快速定位到合适的模型、正确的训练算法，以及一套适当的超参数。不仅如此，后期还能让你更高效地执行错误调试和错误分析。最后还要强调一点，本章探讨的大部分主题对于理解、构建和训练神经网络（本书第二部分）是至关重要的。

本章我们将从最简单的模型之一——线性回归模型，开始介绍两种非常不同的训练模型的方法：

- 通过“闭式”方程，直接计算出最拟合训练集的模型参数（也就是使训练集上的成本函数最小化的模型参数）。

- 使用迭代优化的方法，即梯度下降（GD），逐渐调整模型参数直至训练集上的成本函数调至最低，最终趋同于第一种方法计算出来的模型参数。我们还会研究几个梯度下降的变体，包括批量梯度下降、小批量梯度下降以及随机梯度下降。等我们进入到第二部分神经网络的学习时，会频繁地使用这几个的变体。

接着我们将会进入多项式回归的讨论，这是一个更为复杂的模型，更适合非线性数据集。由于该模型的参数比线性模型更多，因此

更容易造成对训练数据过拟合，我们将使用学习曲线来分辨这种情况是否发生。然后，再介绍几种正则化技巧，降低过拟合训练数据的风险。

最后，我们将学习两种经常用于分类任务的模型：Logistic回归和Softmax回归。

 本章将会出现不少数学公式，需要用到线性代数和微积分的一些基本概念。要理解这些方程式，你需要知道什么是向量和矩阵，如何转置向量和矩阵，什么是点积、逆矩阵、偏导数。如果你不熟悉这些概念，请先通过在线补充材料中的Jupyter notebook，进行线性代数和微积分的入门学习。对于极度讨厌数学的读者，还是需要学习这一章，但是可以跳过那些数学公式，希望文字足以让你了解大多数的概念。

4.1 线性回归

在第1章中，我们学过一个简单的生活满意度的回归模型：life_satisfaction= $\theta_0 + \theta_1 \times$GDP_per_capita。

这个模型就是输入特征GDP_per_capita的线性函数，θ_0和θ_1是模型的参数。

更为概括地说，线性模型就是对输入特征加权求和，再加上一个我们称为偏置项（也称为截距项）的常数，以此进行预测，如公式4-1所示。

公式4-1：线性回归模型预测

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

在此等式中：

- $\hat{y}$是预测值。
- n是特征数量。
- x_i是第i个特征值。
- θ_j是第j个模型参数（包括偏差项θ_0和特征权重θ_1，θ_2，…，θ_n）。

可以使用向量化的形式更简洁地表示，如公式4-2所示。

公式4-2：线性回归模型预测（向量化形式）

$$\hat{y} = h_{\theta}(\boldsymbol{x}) = \boldsymbol{\theta} \cdot \boldsymbol{x}$$

在此等式中：

• θ 是模型的参数向量，其中包含偏差项 θ_0和特征权重 θ_1至 θ_n。

• x是实例的特征向量，包含从x_0到x_n，x_0始终等于1。

• θ • x是向量 θ 和x的点积，它当然等于 $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$。

• h_θ是假设函数，使用模型参数 θ 。

 在机器学习中，向量通常表示为列向量，是有单一列的二维数组。如果 θ 和x为列向量，则预测为$\hat{y} = \boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{x}$，其中 θ^T为 θ （行向量而不是列向量）的转置，且 $\theta^T x$为 θ^T和x的矩阵乘积。当然这是相同的预测，除了现在是以单一矩阵表示而不是一个标量值。在本书中，我将使用这种表示法来避免在点积和矩阵乘法之间切换。

这就是线性回归模型，我们该怎样训练线性回归模型呢？回想一下，训练模型就是设置模型参数直到模型最拟合训练集的过程。为此，我们首先需要知道怎么测量模型对训练数据的拟合程度是好还是差。在第2章中，我们了解到回归模型最常见的性能指标是均方根误差（RMSE）（见公式2-1）。因此，在训练线性回归模型时，你需要找到最小化RMSE的 θ 值。在实践中，将均方误差（MSE）最小化比最小化RMSE更为简单，二者效果相同（因为使函数最小化的值，同样也使其平方根最小）[1]。

在训练集X上，使用公式4-3计算训练集X上线性回归的MSE，h_θ为假设函数。

公式4-3：线性回归模型的MSE成本函数

$$\mathrm{MSE}=(\boldsymbol{X},h_\theta)=\frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{x}^{(i)}-y^{(i)})^2$$

这些符号大多数在第2章中提到过，唯一的区别是h换成了h_θ，以便清楚地表明模型被向量θ参数化。为了简化符号，我们将MSE（X，h_θ）直接写作MSE（θ）。

4.1.1 标准方程

为了得到使成本函数最小的θ值，有一个闭式解方法——也就是一个直接得出结果的数学方程，即标准方程（见公式4-4）。

公式4-4：标准方程

$$\hat{\boldsymbol{\theta}}=(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}$$

这个方程中：

- $\hat{\boldsymbol{\theta}}$是使成本函数最小的θ值。
- y是包含$y^{(1)}$到$y^{(m)}$的目标值向量。

我们生成一些线性数据来测试这个公式（见图4-1）：

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

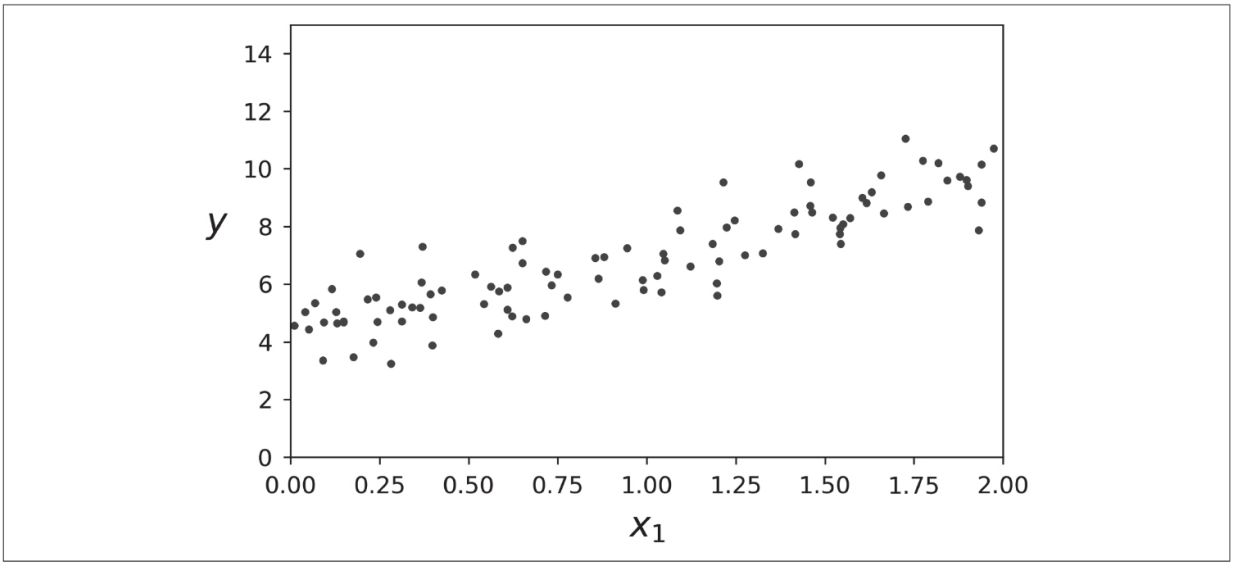


图4-1：随机生成的线性数据集

现在我们使用标准方程来计算$\hat{\theta}$。使用NumPy的线性代数模块（np.linalg）中的inv（）函数来对矩阵求逆，并用dot（）方法计算矩阵的内积：

```
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

我们实际用来生成数据的函数是$y=4+3x_1$+高斯噪声。来看看公式的结果：

```
>>> theta_best
array([[4.21509616],
       [2.77011339]])
```

我们期待的是 θ_0=4， θ_1=3得到的是 θ_0=4.215， θ_1=2.770。非常接近，噪声的存在使其不可能完全还原为原本的函数。

在可以用$\hat{\theta}$现做出预测：

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # add x0 = 1 to each instance
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[4.21509616],
       [9.75532293]])
```

绘制模型的预测结果（见图4-2）：

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

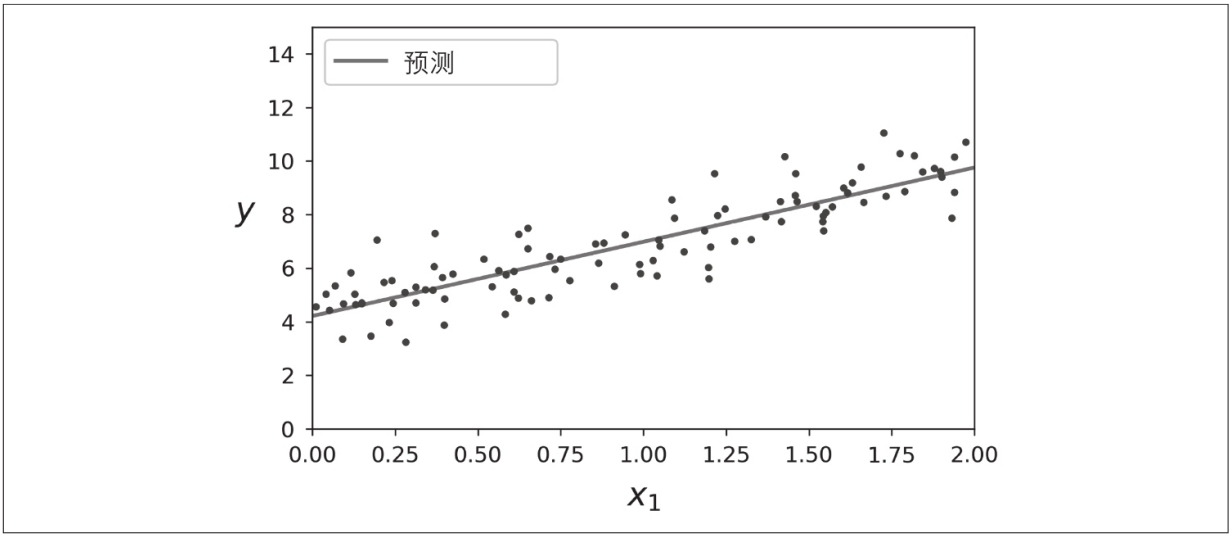


图4-2：线性回归模型预测

使用Scikit-Learn执行线性回归很简单[2]：

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]), array([[2.77011339]]))
>>> lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

LinearRegression类基于scipy.linalg.lstsq（）函数（名称代表“最小二乘”），你可以直接调用它：

```
>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
>>> theta_best_svd
array([[4.21509616],
       [2.77011339]])
```

此函数计算$\hat{\theta} = X^{+}y$，其中X^{+}是X的伪逆（具体来说是Moore-Penrose逆）。你可以使用np.linalg.pinv（）来直接计算这个伪逆：

```
>>> np.linalg.pinv(X_b).dot(y)
array([[4.21509616],
       [2.77011339]])
```

伪逆本身是使用被称为奇异值分解（Singular Value Decomposition，SVD）的标准矩阵分解技术来计算的，可以将训练集矩阵X分解为三个矩阵$U\Sigma V^{T}$的乘积（请参阅numpy.linalg.svd（））。伪逆的计算公式为$X^{+}=V\Sigma^{+}U^{T}$。为了计算矩阵Σ^{+}，该算法取Σ并将所有小于一个小阈值的值设置为零，然后将所有非零值替换成它们的倒数，最后把结果矩阵转置。这种方法比计算标准方程更有效，再加上它可以很好地处理边缘情况：的确，如果矩阵X^{T} X是不可逆的（即奇异的），标准方程可能没有解，例如m<n或某些特征是多余的，但伪逆总是有定义的。

4.1.2 计算复杂度

标准方程计算X^T X的逆，X^T X是一个（n+1）×（n+1）的矩阵（n是特征数量）。对这种矩阵求逆的计算复杂度通常为O（$n^{2.4}$）到O（n^3）之间，取决于具体实现。换句话说，如果将特征数量翻倍，那么计算时间将乘以大约$2^{2.4}$=5.3倍到2^3=8倍之间。

Scikit-Learn的LinearRegression类使用的SVD方法的复杂度约为O（n^2）。如果你将特征数量加倍，那计算时间大约是原来的4倍。

 特征数量比较大（例如100 000）时，标准方程和SVD的计算将极其缓慢。好的一面是，相对于训练集中的实例数量（O（m））来说，两个都是线性的，所以能够有效地处理大量的训练集，只要内存足够。

同样，线性回归模型一经训练（不论是标准方程还是其他算法），预测就非常快速：因为计算复杂度相对于想要预测的实例数量和特征数量来说都是线性的。换句话说，对两倍的实例（或者是两倍的特征数）进行预测，大概需要两倍的时间。

现在，我们再看几个截然不同的线性回归模型的训练方法，这些方法更适合特征数或者训练实例数量大到内存无法满足要求的场景。

[1] 通常情况下，学习算法会尝试优化与评估最终模型的性能指标不同的函数。通常，这是因为该函数更易于计算，它具有性能度量所缺乏的有用的微分特性，或者因为我们希望在训练过程中约束模型，如我们在讨论正则化时所看到的。

[2] 请注意，Scikit-Learn将偏差项（intercept_）与特征权重（coef_）分开。

4.2 梯度下降

梯度下降是一种非常通用的优化算法，能够为大范围的问题找到最优解。梯度下降的中心思想就是迭代地调整参数从而使成本函数最小化。

假设你迷失在山上的浓雾之中，你能感觉到的只有你脚下路面的坡度。快速到达山脚的一个策略就是沿着最陡的方向下坡。这就是梯度下降的做法：通过测量参数向量 θ 相关的误差函数的局部梯度，并不断沿着降低梯度的方向调整，直到梯度降为0，到达最小值！

具体来说，首先使用一个随机的 θ 值（这被称为随机初始化），然后逐步改进，每次踏出一步，每一步都尝试降低一点成本函数（如MSE），直到算法收敛出一个最小值（参见图4-3）。

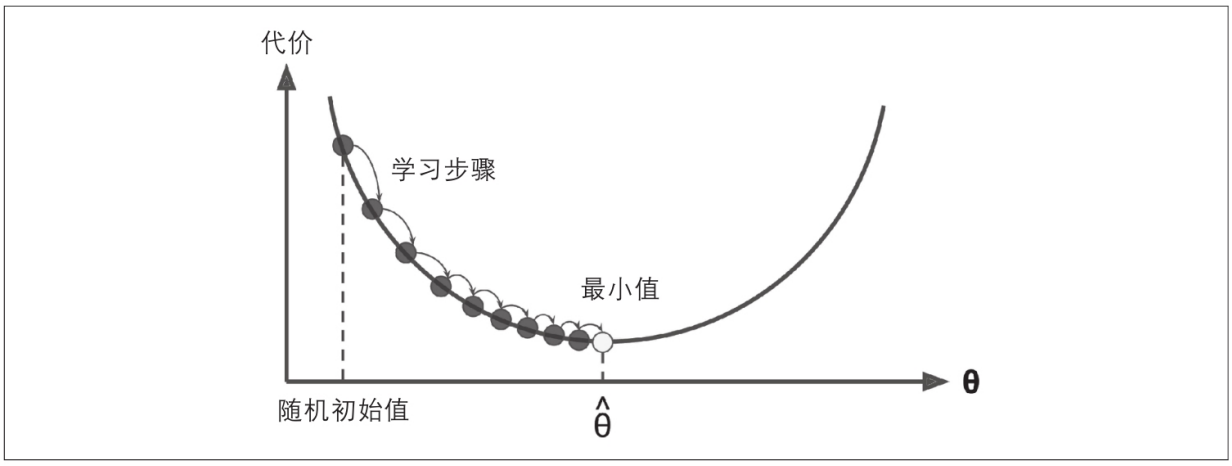


图4-3：在梯度下降的描述中，模型参数被随机初始化并反复调整使成本函数最小化。学习步长与成本函数的斜率成正比，因此，当参数接近最小值时，步长逐渐变小

梯度下降中一个重要参数是每一步的步长，这取决于超参数学习率。如果学习率太低，算法需要经过大量迭代才能收敛，这将耗费很长时间（参见图4-4）。

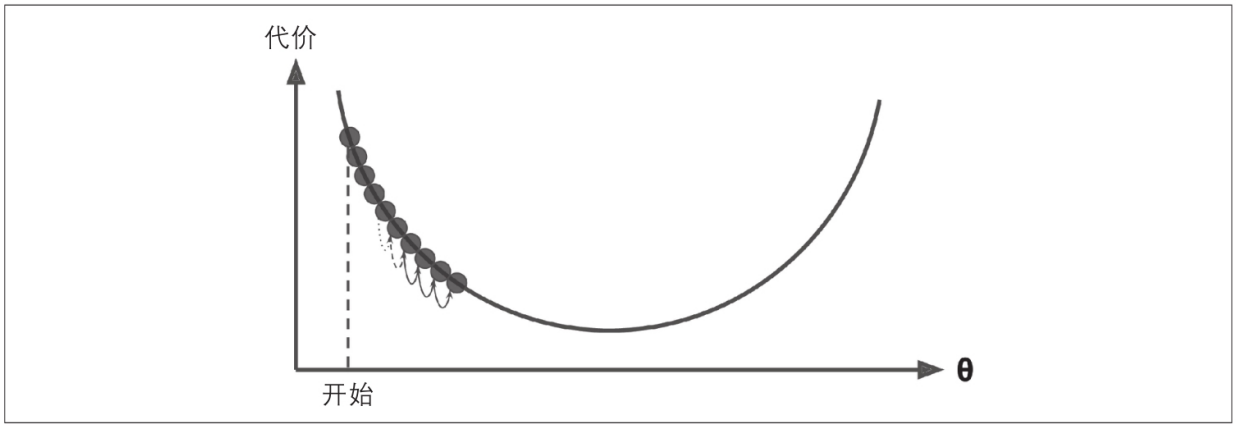


图4-4：学习率太小

反过来说，如果学习率太高，那你可能会越过山谷直接到达另一边，甚至有可能比之前的起点还要高。这会导致算法发散，值越来越大，最后无法找到好的解决方案（参见图4-5）。

最后，并不是所有的成本函数看起来都像一个漂亮的碗。有的可能看着像洞、山脉、高原或者各种不规则的地形，导致很难收敛到最小值。图4-6显示了梯度下降的两个主要挑战：如果随机初始化，算法从左侧起步，那么会收敛到一个局部最小值，而不是全局最小值。如果算法从右侧起步，那么需要经过很长时间才能越过整片高原，如果你停下得太早，将永远达不到全局最小值。

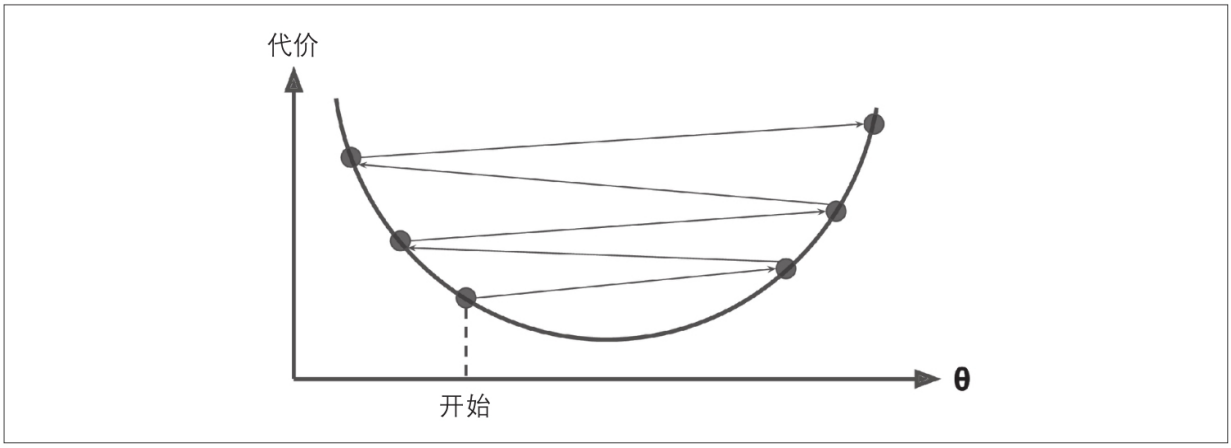


图4-5：学习率太高

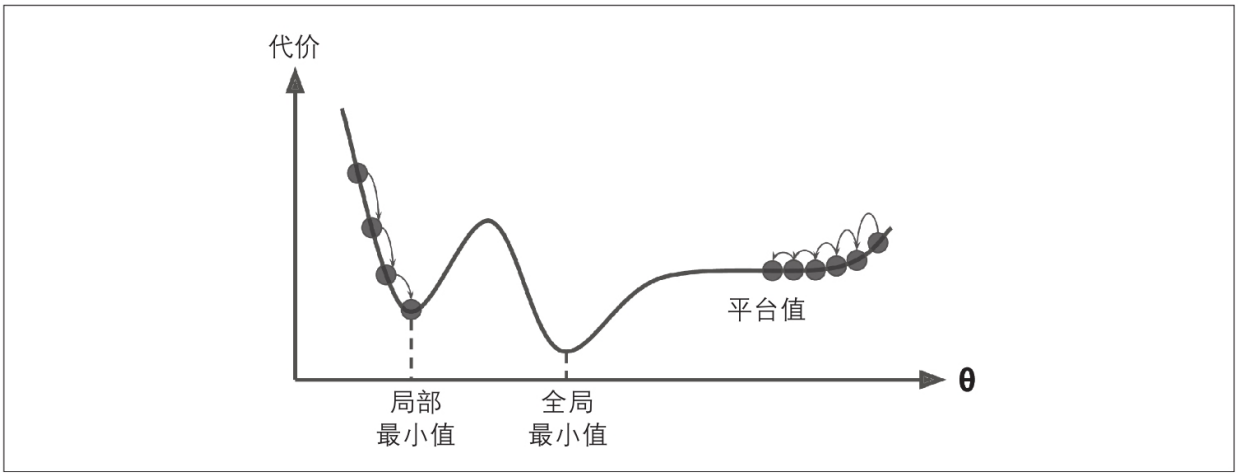


图4-6：梯度下降陷阱

幸好，线性回归模型的MSE成本函数恰好是个凸函数，这意味着连接曲线上任意两点的线段永远不会跟曲线相交。也就是说，不存在局部最小值，只有一个全局最小值。它同时也是一个连续函数，所以斜率不会产生陡峭的变化[1]。这两点保证的结论是：即便是乱走，梯度下降都可以趋近到全局最小值（只要等待时间足够长，学习率也不是太高）。

成本函数虽然是碗状的，但如果不同特征的尺寸差别巨大，那它可能是一个非常细长的碗。如图4-7所示的梯度下降，左边的训练集上特征1和特征2具有相同的数值规模，而右边的训练集上，特征1的值则比特征2要小得多（注：因为特征1的值较小，所以 θ_1 需要更大的变化来影响成本函数，这就是为什么碗形会沿着 θ_1 轴拉长。）。

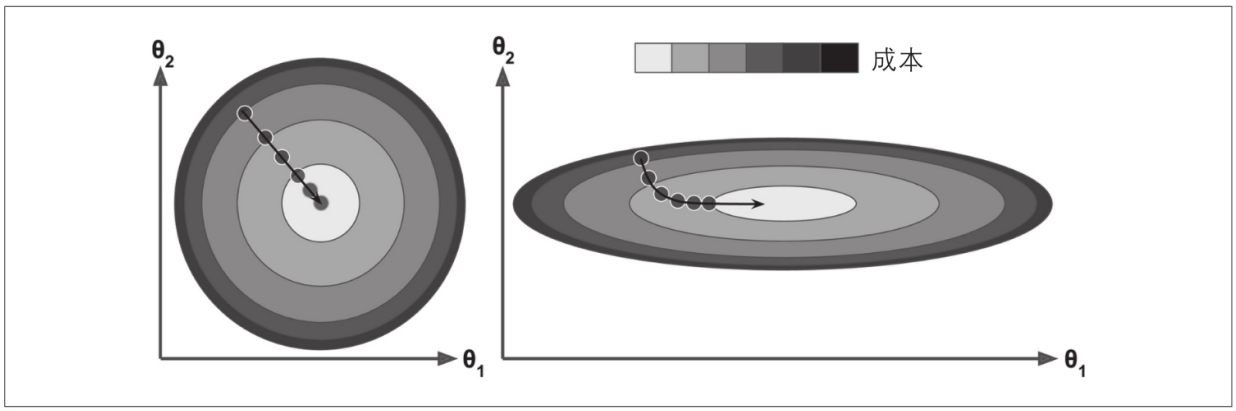


图4-7：有（左）和没有（右）特征缩放的梯度下降

正如你所见，左图的梯度下降算法直接走向最小值，可以快速到达。而在右图中，先是沿着与全局最小值方向近乎垂直的方向前进，接下来是一段几乎平坦的长长的山谷。最终还是会抵达最小值，但是这需要花费大量的时间。

 应用梯度下降时，需要保证所有特征值的大小比例都差不多（比如使用Scikit-Learn的StandardScaler类），否则收敛的时间会长很多。

图4-7也说明，训练模型也就是搜寻使成本函数（在训练集上）最小化的参数组合。这是模型参数空间层面上的搜索：模型的参数越多，这个空间的维度就越多，搜索就越难。同样是在干草堆里寻找一根针，在一个三百维的空间里就比在一个三维空间里要棘手得多。幸运的是，线性回归模型的成本函数是凸函数，针就躺在碗底。

4.2.1 批量梯度下降

要实现梯度下降，你需要计算每个模型关于参数 θ_j 的成本函数的梯度。换言之，你需要计算的是如果改变 θ_j，成本函数会改变多少。这被称为偏导数。这就好比是在问“如果我面向东，我脚下的坡度斜率是多少？”然后面向北问同样的问题（如果你想象超过三个维度的宇宙，对于其他的维度以此类推）。公式4-5计算了关于参数 θ_j 的成本函数的偏导数，计作 $\frac{\partial}{\partial \theta_j}\mathrm{MSE}(\boldsymbol{\theta})$。

公式4-5：成本函数的偏导数

$$\frac{\partial}{\partial \theta_j}\mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m}\sum_{i=1}^{m}(\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{x}^{(i)} - y^{(i)})x_j^{(i)}$$

如果不想单独计算这些偏导数，可以使用公式4-6对其进行一次性计算。梯度向量记作$\nabla_{\theta}\mathrm{MSE}(\theta)$，包含所有成本函数（每个模型参数一个）的偏导数。

公式4-6：成本函数的梯度向量

$$\nabla_{\theta}\mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0}\mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1}\mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n}\mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m}\boldsymbol{X}^{\mathrm{T}}(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y})$$

 请注意，在计算梯度下降的每一步时，都是基于完整的训练集X的。这就是为什么该算法会被称为批量梯度下降：每一步都使用整批训练数据（实际上，全梯度下降可能是个更好的名字）。因此，面对非常庞大的训练集时，算法会变得极慢（不过我们即将看到快得多的梯度下降算法）。但是，梯度下降算法随特征数量扩展的表现比较好。如果要训练的线性模型拥有几十万个特征，使用梯度下降比标准方程或者SVD要快得多。

一旦有了梯度向量，哪个点向上，就朝反方向下坡。也就是从θ中减去∇_{θ}MSE（θ）。这时学习率η就发挥作用了[2]：用梯度向量乘以η确定下坡步长的大小（见公式4-7）。

公式4-7：梯度下降步骤

$$\theta^{(下一步)} = \theta - \eta \nabla_{\theta} \mathrm{MSE}(\theta)$$

让我们看一下该算法的快速实现：

```
eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

不太难！让我们看一下产生的theta：

```
>>> theta
array([[4.21509616],
       [2.77011339]])
```

嘿，这不正是标准方程的发现么！梯度下降表现完美。如果使用了其他的学习率eta呢？图4-8展现了分别使用三种不同的学习率时，梯度下降的前十步（虚线表示起点）。

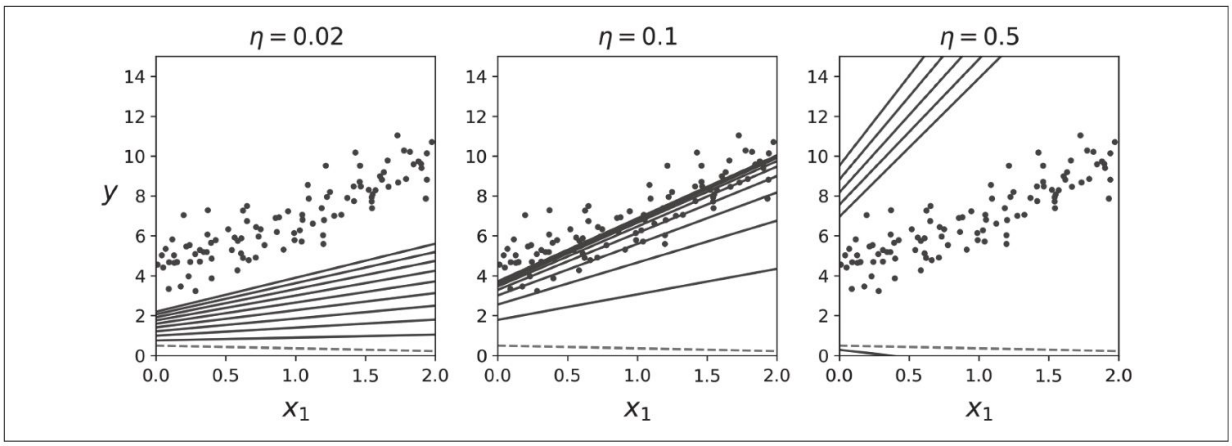


图4-8：各种学习率的梯度下降

左图的学习率太低：算法最终还是能找到解决方法，就是需要太长时间。中间图的学习率看起来非常棒：几次迭代就收敛出了最终解。而右图的学习率太高：算法发散，直接跳过了数据区域，并且每一步都离实际解决方案越来越远。

要找到合适的学习率，可以使用网格搜索（见第2章）。但是你可能需要限制迭代次数，这样网格搜索可以淘汰掉那些收敛耗时太长的模型。

你可能会问，要怎么限制迭代次数呢？如果设置太低，算法可能在离最优解还很远时就停了。但是如果设置得太高，模型达到最优解后，继续迭代则参数不再变化，又会浪费时间。一个简单的办法是在开始时设置一个非常大的迭代次数，但是当梯度向量的值变得很微小时中断算法——也就是当它的范数变得低于（称为容差）时，因为这时梯度下降已经（几乎）到达了最小值。

收敛速度

成本函数为凸函数，并且斜率没有陡峭的变化时（如MSE成本函数），具有固定学习率的批量梯度下降最终会收敛到最佳解，但是你需要等待一段时间：它可以进行O（1/∈）次迭代以在∈的范围内达到最

佳值，具体取决于成本函数的形状。换句话说，如果将容差缩小为原来的1/10（以得到更精确的解），算法将不得不运行10倍的时间。

4.2.2 随机梯度下降

批量梯度下降的主要问题是它要用整个训练集来计算每一步的梯度，所以训练集很大时，算法会特别慢。与之相反的极端是随机梯度下降，每一步在训练集中随机选择一个实例，并且仅基于该单个实例来计算梯度。显然，这让算法变得快多了，因为每次迭代都只需要操作少量的数据。它也可以被用来训练海量的数据集，因为每次迭代只需要在内存中运行一个实例即可（SGD可以作为核外算法实现，见第1章）。

另一方面，由于算法的随机性质，它比批量梯度下降要不规则得多。成本函数将不再是缓缓降低直到抵达最小值，而是不断上上下下，但是从整体来看，还是在慢慢下降。随着时间的推移，最终会非常接近最小值，但是即使它到达了最小值，依旧还会持续反弹，永远不会停止（见图4-9）。所以算法停下来的参数值肯定是足够好的，但不是最优的。

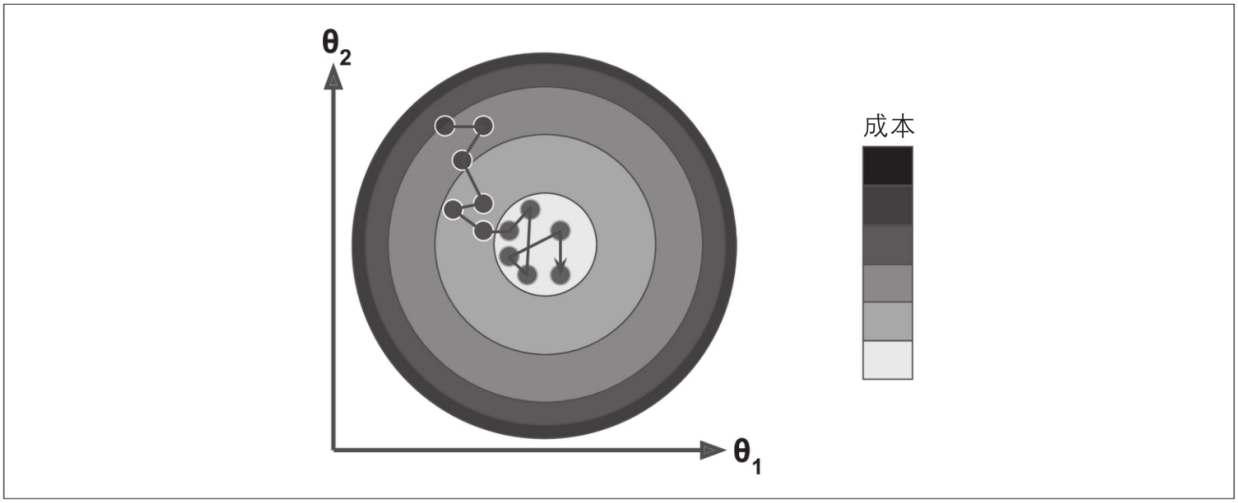


图4-9：与使用批量梯度下降相比，使用随机梯度下降时，每个训练步骤要快得多，但也更加随机

当成本函数非常不规则时（见图4-6），随机梯度下降其实可以帮助算法跳出局部最小值，所以相比批量梯度下降，它对找到全局最小值更有优势。

因此，随机性的好处在于可以逃离局部最优，但缺点是永远定位不出最小值。要解决这个困境，有一个办法是逐步降低学习率。开始的步长比较大（这有助于快速进展和逃离局部最小值），然后越来越小，让算法尽量靠近全局最小值。这个过程叫作模拟退火，因为它类似于冶金时熔化的金属慢慢冷却的退火过程。确定每个迭代学习率的函数叫作学习率调度。如果学习率降得太快，可能会陷入局部最小值，甚至是停留在走向最小值的半途中。如果学习率降得太慢，你需要太长时间才能跳到差不多最小值附近，如果提早结束训练，可能只得到一个次优的解决方案。

下面这段代码使用了一个简单的学习率调度实现随机梯度下降：

```
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)  # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

按照惯例，我们进行m个回合的迭代。每个回合称为一个轮次。虽然批量梯度下降代码在整个训练集中进行了1000次迭代，但此代码仅在训练集中遍历了50次，并达到了一个很好的解决方案：

```
>>> theta
array([[4.21076011],
       [2.74856079]])
```

图4-10显示了训练的前20个步骤（注意这些步骤有多不规则）。

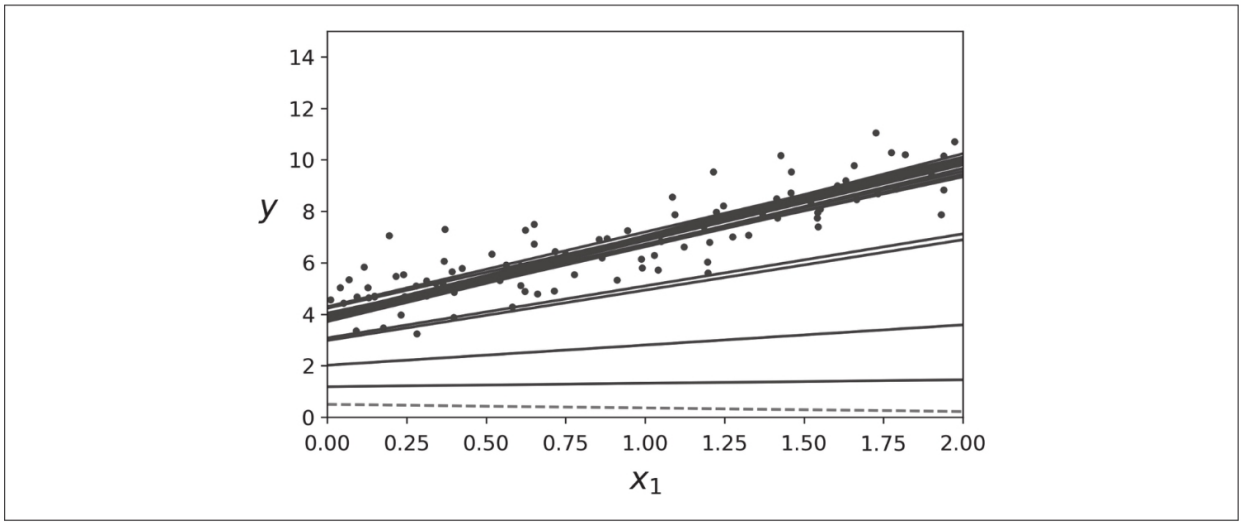


图4-10：随机梯度下降的前20个步骤

请注意，由于实例是随机选取的，因此某些实例可能每个轮次中被选取几次，而其他实例则可能根本不被选取。如果要确保算法在每个轮次都遍历每个实例，则另一种方法是对训练集进行混洗（确保同时对输入特征和标签进行混洗），然后逐个实例进行遍历，然后对其进行再次混洗，以此类推。但是，这种方法通常收敛较慢。

 使用随机梯度下降时，训练实例必须独立且均匀分布（IID），以确保平均而言将参数拉向全局最优值。确保这一点的一种简单方法是在训练过程中对实例进行随机混洗（例如，随机选择每个实例，或者在每个轮次开始时随机混洗训练集）。如果不对实例进行混洗（例如，如果实例按标签排序），那么SGD将首先针对一个标签进行优化，然后针对下一个标签进行优化，以此类推，并且它不会接近全局最小值。

要使用带有Scikit-Learn的随机梯度下降执行线性回归，可以使用SGDRegressor类，该类默认优化平方误差成本函数。以下代码最多可运行1000个轮次，或者直到一个轮次期间损失下降小于0.001为止（max_iter=1000，tol=1e-3）。它使用默认的学习调度（与前一个学习调度不同）以0.1（eta0=0.1）的学习率开始。最后，它不使用任何正则化（penalty=None，稍后将对此进行详细介绍）：

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

你再次找到了一种非常接近于由标准方程返回的解：

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

4.2.3 小批量梯度下降

我们要研究的最后一个梯度下降算法称为小批量梯度下降。只要你了解了批量和随机梯度下降，就很容易理解它：在每一步中，不是根据完整的训练集（如批量梯度下降）或仅基于一个实例（如随机梯度下降）来计算梯度，小批量梯度下降在称为小型批量的随机实例集上计算梯度。小批量梯度下降优于随机梯度下降的主要优点是，你可以通过矩阵操作的硬件优化来提高性能，特别是在使用GPU时。

与随机梯度下降相比，该算法在参数空间上的进展更稳定，尤其是在相当大的小批次中。结果，小批量梯度下降最终将比随机梯度下降走得更接近最小值，但它可能很难摆脱局部最小值（在受局部最小值影响的情况下，不像线性回归）。图4-11显示了训练期间参数空间中三种梯度下降算法所采用的路径。它们最终都接近最小值，但是批量梯度下降

的路径实际上是在最小值处停止，而随机梯度下降和小批量梯度下降都继续走动。但是，不要忘记批量梯度下降每步需要花费很多时间，如果你使用良好的学习率调度，随机梯度下降和小批量梯度下降也会达到最小值。

让我们比较到目前为止讨论的线性回归算法[3]（回想一下，m是训练实例的数量，n是特征的数量）。请参阅表4-1。

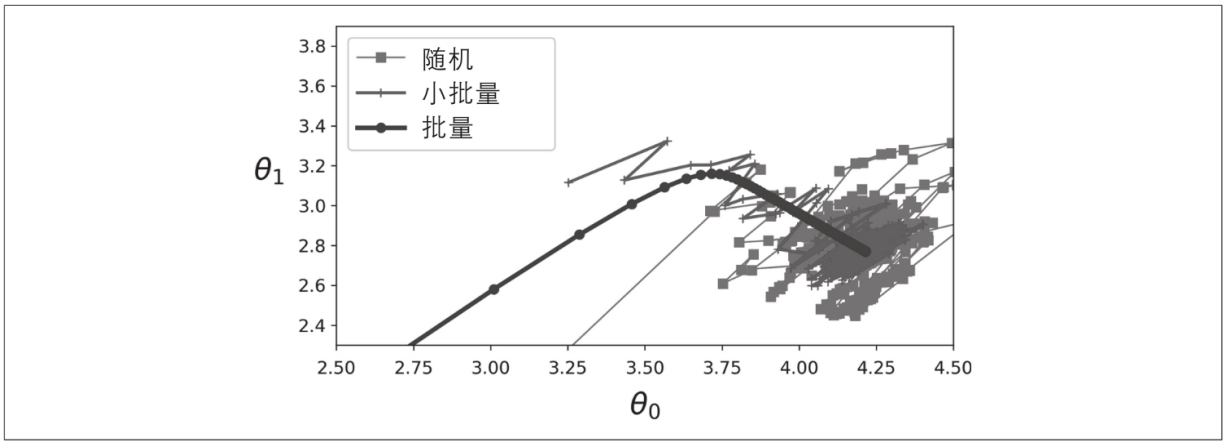


图4-11：参数空间中的梯度下降路径

表4-1：线性回归算法的比较

算法	*m* 很大	核外支持	*n* 很大	超参数	要求缩放	Scikit-Learn
标准方程	快	否	慢	0	否	N/A
SVD	快	否	慢	0	否	LinearRegression
批量 GD	慢	否	快	2	是	SGDRegressor
随机 GD	快	是	快	≥2	是	SGDRegressor
小批量 GD	快	是	快	≥2	是	SGDRegressor

 训练后几乎没有区别：所有这些算法最终都具有非常相似的模型，并且以完全相同的方式进行预测。

[1] 从技术上讲，它的导数是Lipschitz连续的。

[2] Eta（η）是希腊字母的第7个字母。

[3] 标准方程只能执行线性回归，但可以看到，梯度下降算法可用于训练许多其他模型。

4.3 多项式回归

如果你的数据比直线更复杂怎么办？令人惊讶的是，你可以使用线性模型来拟合非线性数据。一个简单的方法就是将每个特征的幂次方添加为一个新特征，然后在此扩展特征集上训练一个线性模型。这种技术称为多项式回归。

让我们看一个示例。首先，让我们基于一个简单的二次方程式（注：二次方程的形式为$y=ax^2+bx+c$。）（加上一些噪声，见图4-12）生成一些非线性数据：

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

显然，一条直线永远无法正确地拟合此数据。因此，让我们使用Scikit-Learn的PolynomialFeatures类来转换训练数据，将训练集中每个特征的平方（二次多项式）添加为新特征（在这种情况下，只有一个特征）：

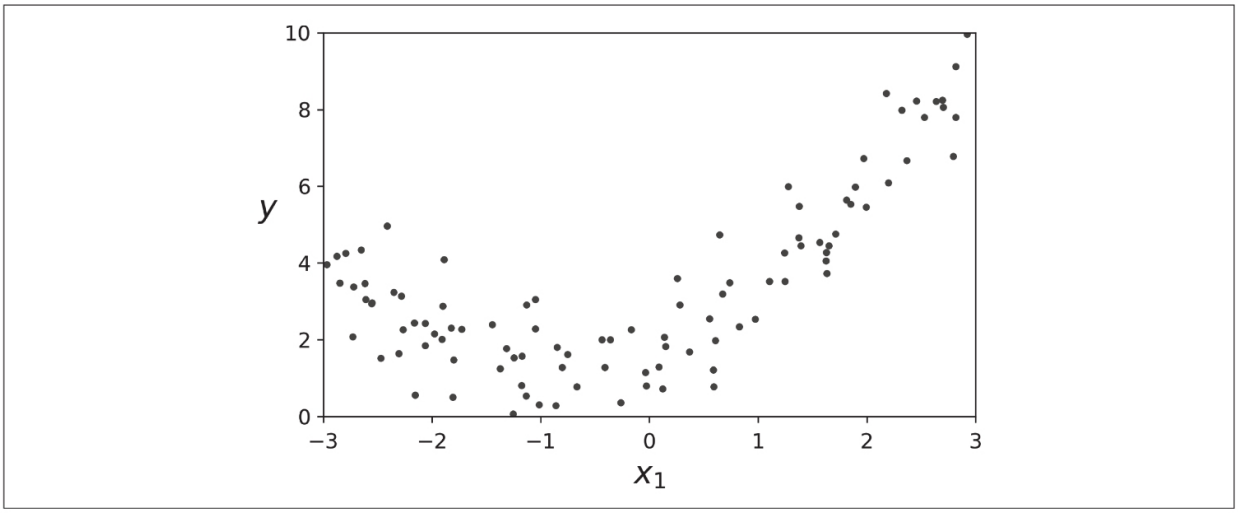


图4-12：生成的非线性和噪声数据集

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

X_poly现在包含X的原始特征以及该特征的平方。现在，你可以将LinearRegression模型拟合到此扩展训练数据中（见图4-13）：

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

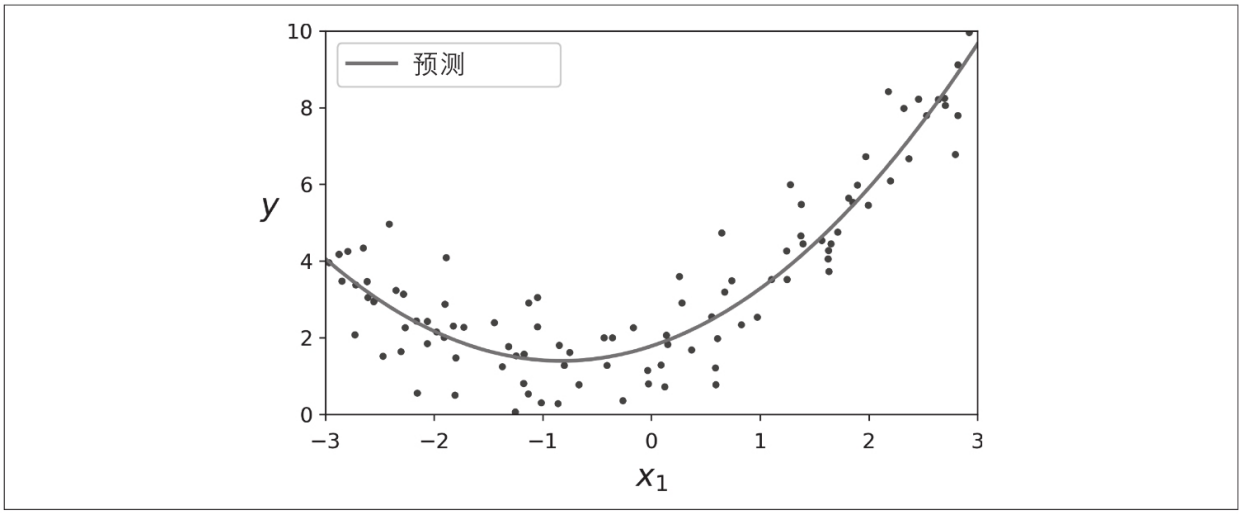


图4-13：多项式回归模型预测

不错：模型估算$\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$，而实际上原始函数为$y = 0.5x_1^2 + 1.0x_1 + 2.0 +$高斯噪声。

请注意，当存在多个特征时，多项式回归能够找到特征之间的关系（这是普通线性回归模型无法做到的）。PolynomialFeatures还可以将特征的所有组合添加到给定的多项式阶数。例如，如果有两个特征a和b，则degree=3的PolynomialFeatures不仅会添加特征a^2、a^3、b^2和b^3，还会添加组合ab、a^2b和ab^2。

 PolynomialFeatures（degree=d）可以将一个包含n个特征的数组转换为包含$\frac{(n+d)!}{d!n!}$个特征的数组，其中n!是n的阶乘，等于1×2×3×…×n。要小心特征组合的数量爆炸。

4.4 学习曲线

你如果执行高阶多项式回归，与普通线性回归相比，拟合数据可能会更好。例如，图4-14将300阶多项式模型应用于先前的训练数据，将结果与纯线性模型和二次模型（二次多项式）进行比较。请注意300阶多项式模型是如何摆动以尽可能接近训练实例的。

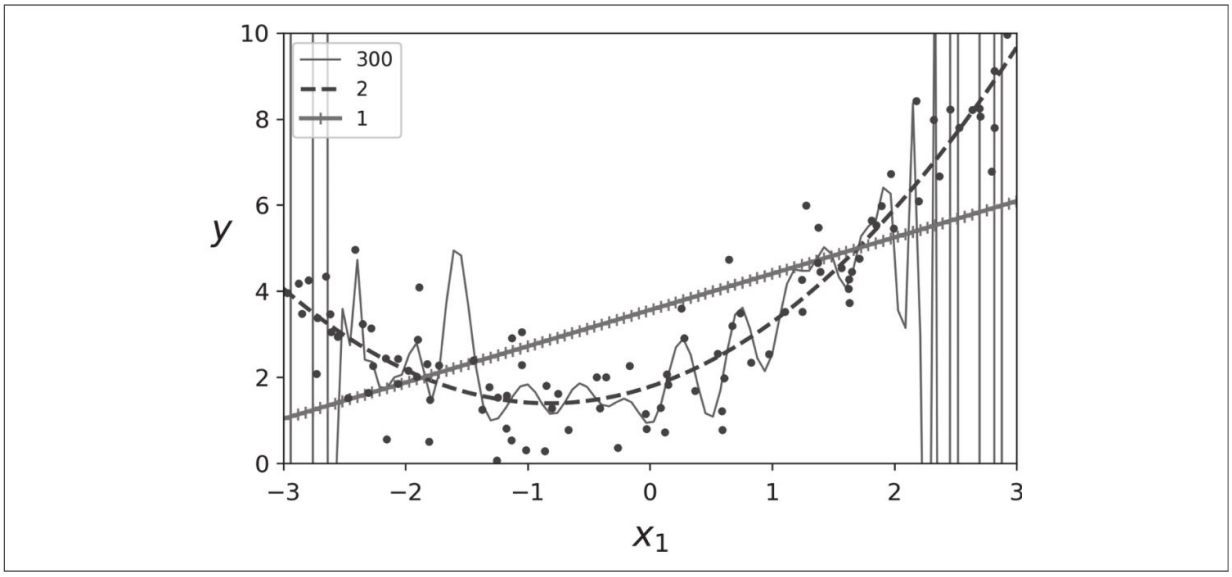


图4-14：高阶多项式回归

这种高阶多项式回归模型严重过拟合训练数据，而线性模型则欠拟合。在这种情况下，最能泛化的模型是二次模型，因为数据是使用二次模型生成的。但是总的来说，你不知道数据由什么函数生成，那么如何确定模型的复杂性呢？你如何判断模型是过拟合数据还是欠拟合数据呢？

在第2章中，你使用交叉验证来估计模型的泛化性能。如果模型在训练数据上表现良好，但根据交叉验证的指标泛化较差，则你的模型过拟合。如果两者的表现均不理想，则说明欠拟合。这是一种区别模型是否过于简单或过于复杂的方法。

还有一种方法是观察学习曲线：这个曲线绘制的是模型在训练集和验证集上关于训练集大小（或训练迭代）的性能函数。要生成这个曲线，只需要在不同大小的训练子集上多次训练模型即可。下面这段代码在给定训练集下定义了一个函数，绘制模型的学习曲线：

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

让我们看一下普通线性回归模型的学习曲线（一条直线，见图4-15）：

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```

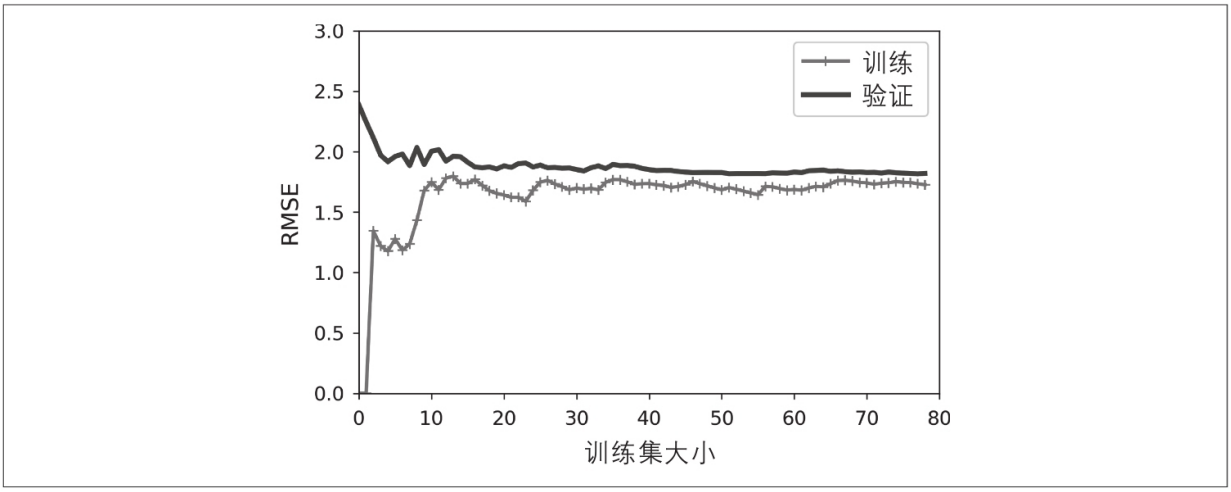


图4-15：学习曲线

这种欠拟合的模型值得解释一下。首先，让我们看一下在训练数据上的性能：当训练集中只有一个或两个实例时，模型可以很好地拟合它们，这就是曲线从零开始的原因。但是，随着将新实例添加到训练集中，模型就不可能完美地拟合训练数据，这既因为数据有噪声，又因为它根本不是线性的。因此，训练数据上的误差会一直上升，直到达到平稳状态，此时在训练集中添加新实例并不会使平均误差变好或变差。现在让我们看一下模型在验证数据上的性能。当在很少的训练实例上训练模型时，它无法正确泛化，这就是验证误差最初很大的原因。然后，随着模型经历更多的训练示例，它开始学习，因此验证错误逐渐降低。但是，直线不能很好地对数据进行建模，因此误差最终达到一个平稳的状态，非常接近另外一条曲线。

这些学习曲线是典型的欠拟合模型。两条曲线都达到了平稳状态。它们很接近而且很高。

 如果你的模型欠拟合训练数据，添加更多训练示例将无济于事。你需要使用更复杂的模型或提供更好的特征。

现在让我们看一下在相同数据上的10阶多项式模型的学习曲线（见图4-16）：

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
```

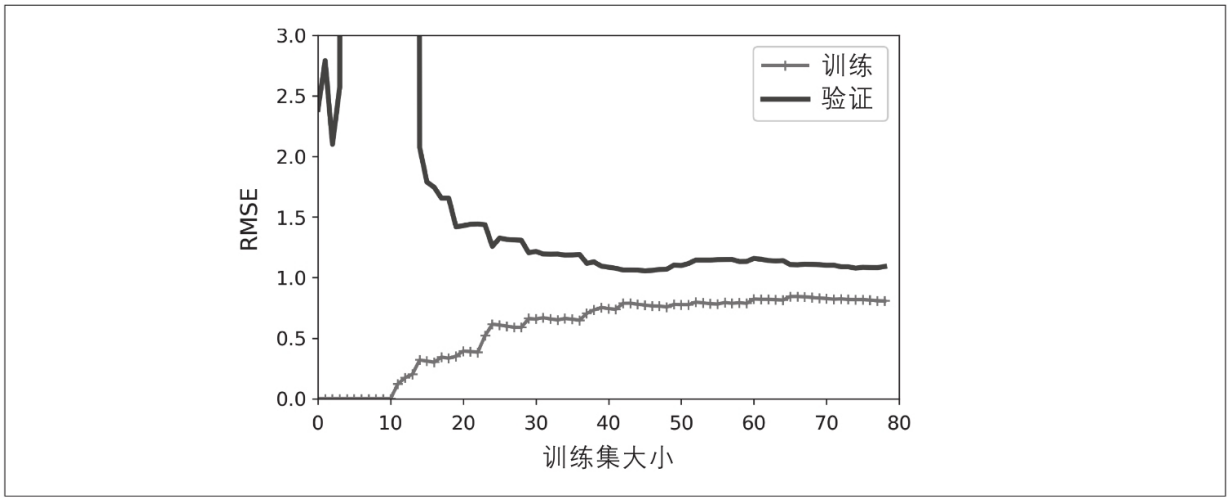


图4-16：10阶多项式模型的学习曲线

这些学习曲线看起来有点像以前的曲线，但是有两个非常重要的区别：

• 与线性回归模型相比，训练数据上的误差要低得多。

• 曲线之间存在间隙。这意味着该模型在训练数据上的性能要比在验证数据上的性能好得多，这是过拟合模型的标志。但是，如果你使用更大的训练集，则两条曲线会继续接近。

 改善过拟合模型的一种方法是向其提供更多的训练数据，直到验证误差达到训练误差为止。

偏差/方差权衡

统计学和机器学习的重要理论成果是以下事实：模型的泛化误差可以表示为三个非常不同的误差之和：

偏差

这部分泛化误差的原因在于错误的假设，比如假设数据是线性的，而实际上是二次的。高偏差模型最有可能欠拟合训练数据[1]。

方差

这部分是由于模型对训练数据的细微变化过于敏感。具有许多自由度的模型（例如高阶多项式模型）可能具有较高的方差，因此可能过拟合训练数据。

不可避免的误差

这部分误差是因为数据本身的噪声所致。减少这部分误差的唯一方法就是清理数据（例如修复数据源（如损坏的传感器），或者检测并移除异常值）。

增加模型的复杂度通常会显著提升模型的方差并减少偏差。反过来，降低模型的复杂度则会提升模型的偏差并降低方差。这就是为什么称其为权衡。

[1] 不要将这里的偏差概念与线性模型中的偏置项概念弄混。

4.5　正则化线性模型

正如我们在第1章和第2章中看到的那样，减少过拟合的一个好方法是对模型进行正则化（即约束模型）：它拥有的自由度越少，则过拟合数据的难度就越大。正则化多项式模型的一种简单方法是减少多项式的次数。

对于线性模型，正则化通常是通过约束模型的权重来实现的。现在，我们看一下岭回归、Lasso回归和弹性网络，它们实现了三种限制权重的方法。

4.5.1　岭回归

岭回归（也称为Tikhonov正则化）是线性回归的正则化版本：将等于 $\alpha\sum_{i=1}^{n}\theta_i^2$ 的正则化项添加到成本函数。这迫使学习算法不仅拟合数据，而且还使模型权重尽可能小。注意仅在训练期间将正则化项添加到成本函数中。训练完模型后，你要使用非正则化的性能度量来评估模型的性能。

 训练过程中使用的成本函数与用于测试的性能指标不同是很常见的。除正则化外，它们可能不同的另一个原因是好的训练成本函数应该具有对优化友好的导数，而用于测试的性能指标应尽可能接近最终目标。例如，通常使用成本函数（例如对数损失（稍后讨论））来训练分类器，但使用精度/召回率对其进行评估。

超参数α控制要对模型进行正则化的程度。如果α=0，则岭回归仅是线性回归。如果α非常大，则所有权重最终都非常接近于零，结果是一条经过数据均值的平线。公式4-8给出了岭回归成本函数[1]。

公式4-8：岭回归成本函数

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

请注意，偏置项θ_0没有进行正则化（总和从i=1开始，而不是0）。如果我们将w定义为特征权重的向量（θ_1至θ_n），则正则项等于$1/2(||\boldsymbol{w}||_2)^2$，其中$||w||_2$表示权重向量的$\ell_2$范数[2]。对于梯度下降，只需将$\alpha w$添加到MSE梯度向量（见公式4-6）。

 在执行岭回归之前缩放数据（例如使用StandardScaler）很重要，因为它对输入特征的缩放敏感。大多数正则化模型都需要如此。

图4-17显示了使用不同的α值对某些线性数据进行训练的几种岭模型。左侧使用普通岭模型，导致了线性预测。在右侧，首先使用PolynomialFeatures（degree=10）扩展数据，然后使用StandardScaler对其进行缩放，最后将岭模型应用于结果特征：这是带有岭正则化的多项式回归。请注意，α的增加会导致更平坦（即不极端，更合理）的预测，从而减少了模型的方差，但增加了其偏差。

与线性回归一样，我们可以通过计算闭合形式的方程或执行梯度下降来执行岭回归。利弊是相同的。公式4-9显示了闭式解，其中A是（n+1）×（n+1）单位矩阵[3]，除了在左上角的单元格中有0（对应于偏置项）。

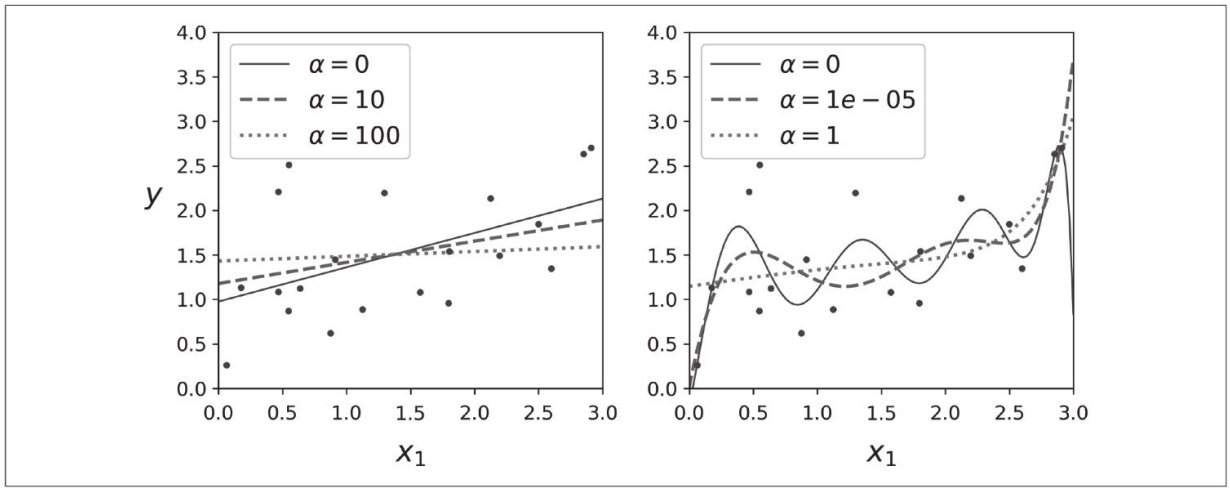


图4-17：线性模型（左）和多项式模型（右），具有各种级别的岭正则化

公式4-9：闭式解的岭回归

$$\hat{\boldsymbol{\theta}} = (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X} + \alpha \boldsymbol{A})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}$$

以下是用Scikit-Learn和闭式解（方程式4-9的一种变体，它使用AndréLouis Cholesky矩阵分解技术）来执行岭回归的方法：

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[1.55071465]])
```

并使用随机梯度下降法[4]：

```
12
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

超参数penalty设置的是使用正则项的类型。设为"l2"表示希望SGD在成本函数中添加一个正则项，等于权重向量的ℓ_2范数的平方的一半，即岭回归。

4.5.2 Lasso回归

线性回归的另一种正则化叫作最小绝对收缩和选择算子回归（Least Absolute Shrinkage and Selection Operator Regression，简称Lasso回归）。与岭回归一样，它也是向成本函数添加一个正则项，但是它增加的是权重向量的ℓ_1范数，而不是ℓ_2范数的平方的一半（参见公式4-10）。

公式4-10：Lasso回归成本函数

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^{n} |\theta_i|$$

图4-18显示了与图4-17相同的东西，但是用Lasso模型替换了岭模型，并使用了较小的α值。

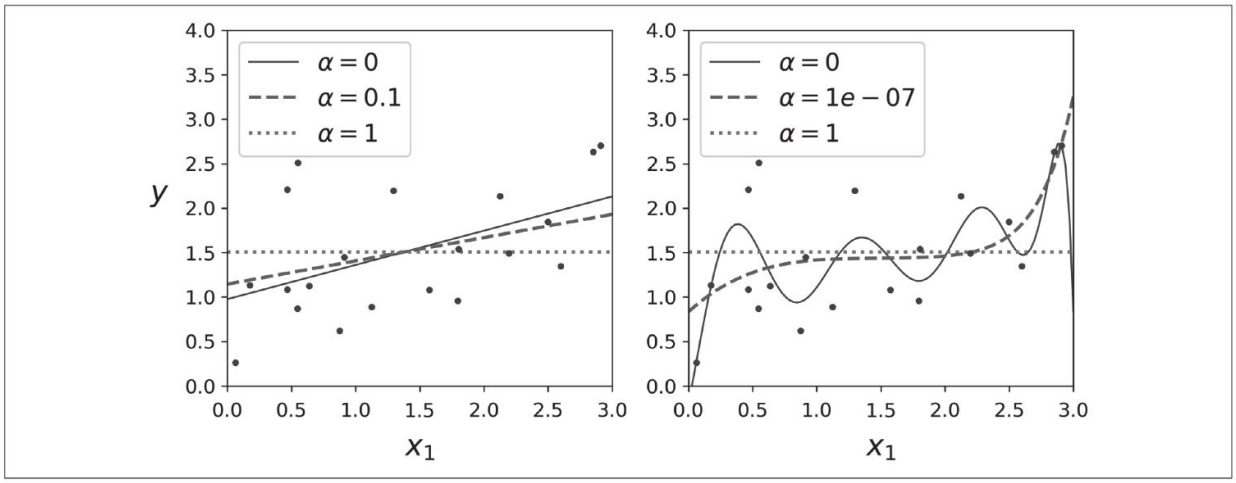


图4-18：线性模型（左）和多项式模型（右），都使用不同级别的Lasso正则化

Lasso回归的一个重要特点是它倾向于完全消除掉最不重要特征的权重（也就是将它们设置为零）。例如，在图4-18的右图中的虚线（$\alpha=10^{-7}$）看起来像是二次的，快要接近于线性：因为所有高阶多项式的特征权重都等于零。换句话说，Lasso回归会自动执行特征选择并输出一个稀疏模型（即只有很少的特征有非零权重）。

你可以通过查看图4-19来了解为什么会这样：轴代表两个模型参数，背景轮廓代表不同的损失函数。在左上图中，轮廓线代表ℓ_1损失（$|\theta_1|+|\theta_2|$），当你靠近任何轴时，该损失呈线性下降。例如，如果将模型参数初始化为$\theta_1=2$和$\theta_1=0.5$，运行梯度下降会使两个参数均等地递减（如黄色虚线所示）。因此θ_2将首先达到0（因为开始时接近0）。之后，梯度下降将沿山谷滚动直到其达到$\theta_1=0$（有一点反弹，因为ℓ_1的梯度永远不会接近0：对于每个参数，它们都是-1或1）。在右上方的图中，轮廓线代表Lasso的成本函数（即MSE成本函数加ℓ_1损失）。白色的小圆圈显示了梯度下降优化某些模型参数的路径，这些参数在$\theta_1=0.25$和$\theta_2=-1$附近初始化：再次注意该路径如何快速到达$\theta_2=0$，然后向下滚动并最终在全局最优值附近反弹（由红色正方形表示）。如果增加α，则全局最优值将沿黄色虚线向左移动；如果减少α，则全局最优值将向右移动（在此示例中，非正则化的MSE的最优参数为$\theta_1=2$和$\theta_2=0.5$）。

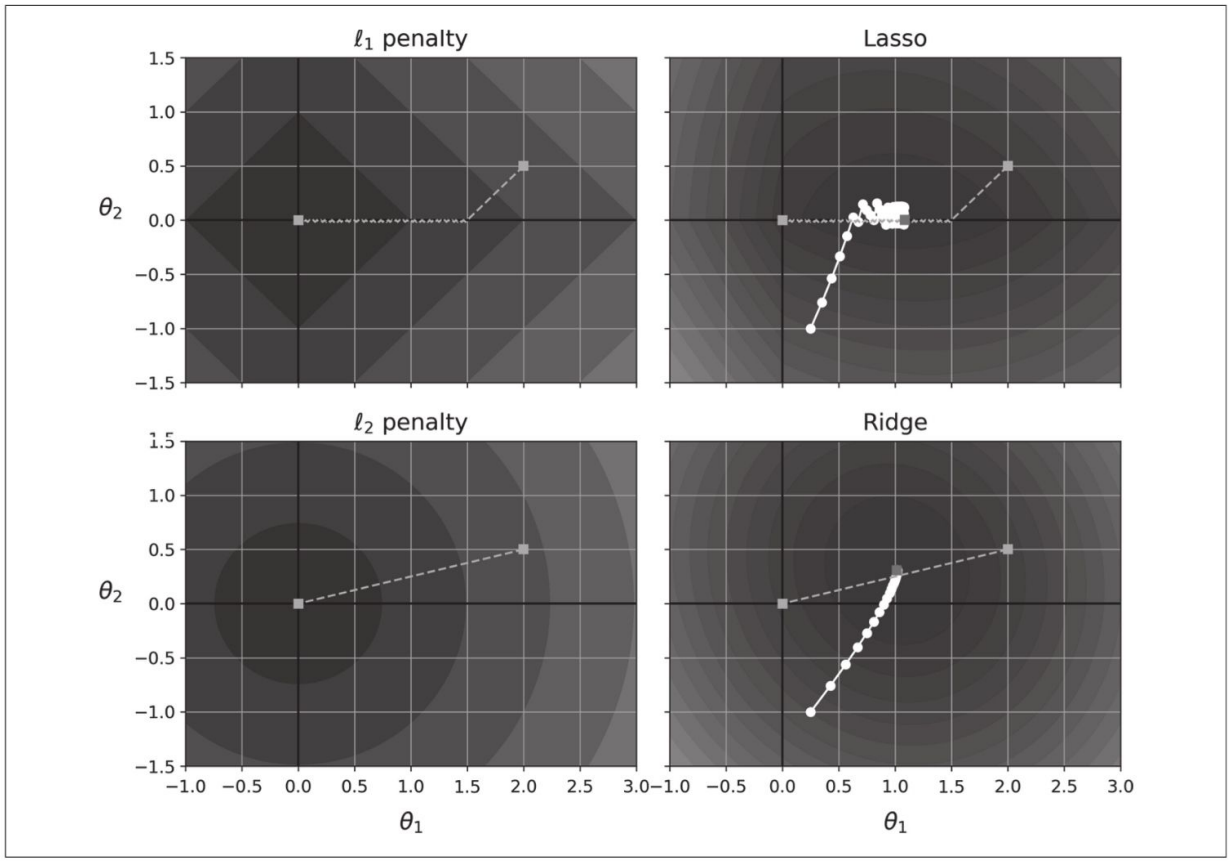


图4-19：Lasso vs岭正则化

底部的两个图显示了相同的内容，但惩罚为ℓ_2。在左下图中，你可以看到ℓ_2损失随距原点的距离而减小，因此梯度下降沿该点直走。在右下图中，轮廓线代表岭回归的成本函数（即MSE成本函数加ℓ_2损失）。Lasso有两个主要区别。首先，随着参数接近全局最优值，梯度会变小，因此，梯度下降自然会减慢，这有助于收敛（因为周围没有反弹）。其次，当你增加α时，最佳参数（用红色正方形表示）越来越接近原点，但是它们从未被完全被消除。

 为了避免在使用Lasso时梯度下降最终在最优解附近反弹，你需要逐渐降低训练期间的学习率（它仍然会在最优解附近反弹，但是步长会越来越小，因此会收敛）。

Lasso成本函数在 θ_i=0（对于i=1，2，…，n）处是不可微的，但是如果你使用子梯度向量g[5]代替任何 θ_i=0，则梯度下降仍然可以正常工作。公式4-11显示了可用于带有Lasso成本函数的梯度下降的子梯度向量方程。

公式4-11：Lasso回归子梯度向量

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}}\mathrm{MSE}(\boldsymbol{\theta}) + \alpha \begin{pmatrix} \sin(\theta_1) \\ \sin(\theta_2) \\ \vdots \\ \sin(\theta_n) \end{pmatrix} \quad \text{其中} \operatorname{sign}(\theta_i) = \begin{cases} -1 & \text{如果} \theta_i < 0 \\ 0 & \text{如果} \theta_i = 0 \\ +1 & \text{如果} \theta_i > 0 \end{cases}$$

这是一个使用Lasso类的Scikit-Learn小示例：

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([1.53788174])
```

请注意，你可以改用SGDRegressor（penalty="l1"）。

4.5.3 弹性网络

弹性网络是介于岭回归和Lasso回归之间的中间地带。正则项是岭和Lasso正则项的简单混合，你可以控制混合比r。当r=0时，弹性网络等效于岭回归，而当r=1时，弹性网络等效于Lasso回归（见公式4-12）。

公式4-12：弹性网络成本函数

$$J(\theta) = \mathrm{MSE}(\theta) + r\alpha\sum_{i=1}^{n}|\theta_i| + \frac{1-r}{2}\alpha\sum_{i=1}^{n}\theta_i^2$$

那么什么时候应该使用普通的线性回归（即不进行任何正则化）、岭、Lasso或弹性网络呢？通常来说，有正则化——哪怕很小，总比没有更可取一些。所以大多数情况下，你应该避免使用纯线性回归。岭回归是个不错的默认选择，但是如果你觉得实际用到的特征只有少数几个，那就应该更倾向于Lasso回归或是弹性网络，因为它们会将无用特征的权重降为零。一般而言，弹性网络优于Lasso回归，因为当特征数量超过训练实例数量，又或者是几个特征强相关时，Lasso回归的表现可能非常不稳定。

这是一个使用Scikit-Learn的ElasticNet的小示例（l1_ratio对应于混合比r）：

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([1.54333232])
```

4.5.4 提前停止

对于梯度下降这一类迭代学习的算法，还有一个与众不同的正则化方法，就是在验证误差达到最小值时停止训练，该方法叫作提前停止法。图4-20展现了一个用批量梯度下降训练的复杂模型（高阶多项式回归模型）。经过一轮一轮的训练，算法不断地学习，训练集上的预测误差（RMSE）自然不断下降，同样其在验证集上的预测误差也随之下降。但是，一段时间之后，验证误差停止下降反而开始回升。这说明模型开始过拟合训练数据。通过早期停止法，一旦验证误差达到最小值就立刻

停止训练。这是一个非常简单而有效的正则化技巧，所以Geoffrey Hinton称其为“美丽的免费午餐”。

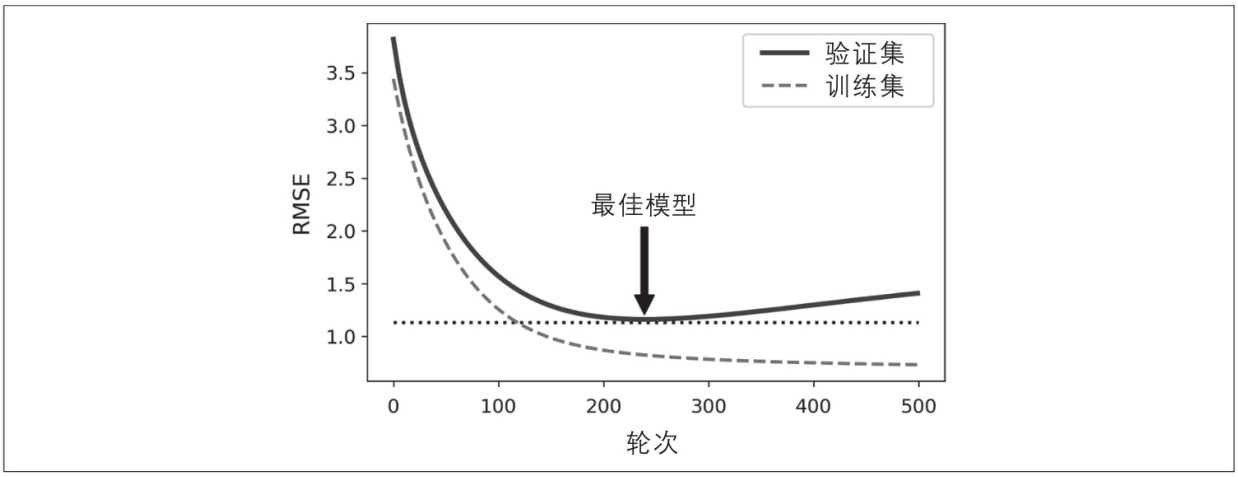


图4-20：提前停止正则化

 使用随机和小批量梯度下降时，曲线不是那么平滑，可能很难知道你是否达到了最小值。一种解决方案是仅在验证错误超过最小值一段时间后停止（当你确信模型不会做得更好时），然后回滚模型参数到验证误差最小的位置。

以下是提前停止法的基本实现：

```
from sklearn.base import clone

# prepare the data
poly_scaler = Pipeline([
        ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
        ("std_scaler", StandardScaler())
    ])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)  # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
```

```
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

请注意，在使用warm_start=True的情况下，当调用fit（）方法时，它将在停止的地方继续训练，而不是从头开始。

[1] 对于没有缩写名称的成本函数，通常使用符号J（θ）。在本书的其余部分中，我们将经常使用这种表示法。上下文将清楚地说明正在讨论哪个成本函数。
[2] 范数在第2章中讨论。
[3] 除主对角线上的1之外，全为0的方阵（从左上到右下）。
[4] 或者，你可以将Ridge类与"sag"求解器一起使用。随机平均梯度下降是随机梯度下降的变体。有关更多详细信息，请参阅来自不列颠哥伦比亚大学Mark Schmidt等人的论文“Minimizing Finite Sums with the Stochastic Average Gradient Algorithm”。
[5] 你可以将不可微分点处的子梯度向量视为该点周围的梯度向量之间的中间向量。

4.6 逻辑回归

正如第1章中提到过的，一些回归算法也可用于分类（反之亦然）。逻辑回归（Logistic回归，也称为Logit回归）被广泛用于估算一个实例属于某个特定类别的概率。（比如，这封电子邮件属于垃圾邮件的概率是多少？）如果预估概率超过50%，则模型预测该实例属于该类别（称为正类，标记为“1”），反之，则预测不是（称为负类，标记为“0”）。这样它就成了一个二元分类器。

4.6.1 估计概率

所以逻辑回归是怎么工作的呢？与线性回归模型一样，逻辑回归模型也是计算输入特征的加权和（加上偏置项），但是不同于线性回归模型直接输出结果，它输出的是结果的数理逻辑值（参见公式4-13）。

公式4-13：逻辑回归模型的估计概率（向量化形式）

$$\hat{p} = h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{\theta})$$

逻辑记为σ（·），是一个sigmoid函数（即S型函数），输出一个介于0和1之间的数字。其定义如公式4-14和图4-21所示。

公式4-14：逻辑函数

$$\sigma(t) = \frac{1}{1+\exp(-t)}$$

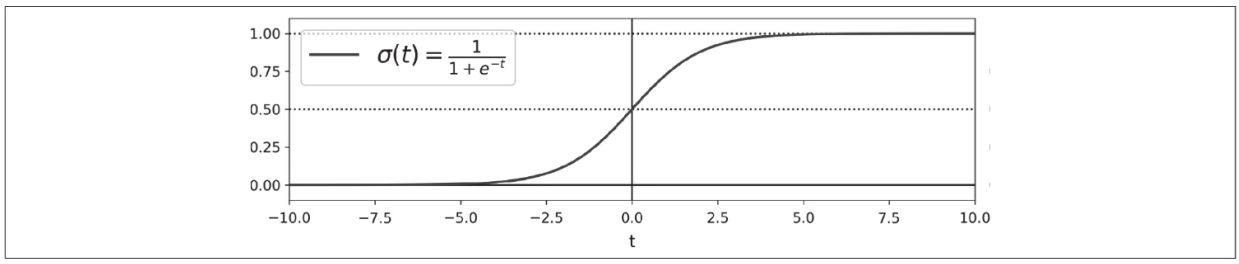


图4-21：逻辑函数

一旦逻辑回归模型估算出实例x属于正类的概率$\hat{p} = h_\theta(\boldsymbol{x})$，就可以轻松做出预测$\hat{y}$（见公式4-15）。

公式4-15：逻辑回归模型预测

$$\hat{y} = \begin{cases} 0, & \text{如果} \hat{p} < 0.5 \\ 1, & \text{如果} \hat{p} \geqslant 0.5 \end{cases}$$

注意，当t＜0时，σ（t）＜0.5；当t≥0时，σ（t）≥0.5。所以如果$x^T\theta$是正类，逻辑回归模型预测结果是1，如果是负类，则预测为0。

 分数t通常称为logit。该名称源于以下事实：定义为logit（p）=log（p/（1－p））的logit函数与logistic函数相反。确实，如果你计算估计概率p的对数，则会发现结果为t。对数也称为对数奇数，因为它是正类别的估计概率与负类别的估计概率之比的对数。

4.6.2　训练和成本函数

现在你知道逻辑回归模型是如何估算概率并做出预测了。但是要怎么训练呢？训练的目的就是设置参数向量θ，使模型对正类实例做出高

概率估算（y=1），对负类实例做出低概率估算（y=0）。公式4-16所示为单个训练实例x的成本函数，正说明了这一点。

公式4-16：单个训练实例的成本函数

$$c(\boldsymbol{\theta}) = \begin{cases} -\log(\hat{p}), & \text{如果} y = 1 \\ -\log(1-\hat{p}), & \text{如果} y = 0 \end{cases}$$

这个成本函数是有道理的，因为当t接近于0时，-log（t）会变得非常大，所以如果模型估算一个正类实例的概率接近于0，成本将会变得很高。同理估算出一个负类实例的概率接近1，成本也会变得非常高。那么反过来，当t接近于1的时候，-log（t）接近于0，所以对一个负类实例估算出的概率接近于0，对一个正类实例估算出的概率接近于1，而成本则都接近于0，这不正好是我们想要的吗？

整个训练集的成本函数是所有训练实例的平均成本。可以用一个称为对数损失的单一表达式来表示，见公式4-17。

公式4-17：逻辑回归成本函数（对数损失）

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(\hat{p}^{(i)}) + (1-y^{(i)})\log(1-\hat{p}^{(i)})]$$

但是坏消息是，这个函数没有已知的闭式方程（不存在一个标准方程的等价方程）来计算出最小化成本函数的θ值。而好消息是这是个凸函数，所以通过梯度下降（或是其他任意优化算法）保证能够找出全局最小值（只要学习率不是太高，你又能长时间等待）。公式4-18给出了成本函数关于第j个模型参数θ_j的偏导数方程。

公式4-18：逻辑成本函数偏导数

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} (\sigma(\boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

该公式与公式4-5非常相似：对于每个实例，它都会计算预测误差并将其乘以第j个特征值，然后计算所有训练实例的平均值。一旦你有了包含所有偏导数的梯度向量就可以使用梯度下降算法了。就是这样，现在你知道如何训练逻辑模型了。对于随机梯度下降，一次使用一个实例；对于小批量梯度下降，一次使用一个小批量。

4.6.3 决策边界

这里我们用鸢尾植物数据集来说明逻辑回归。这是一个非常著名的数据集，共有150朵鸢尾花，分别来自三个不同品种（山鸢尾、变色鸢尾和维吉尼亚鸢尾），数据里包含花的萼片以及花瓣的长度和宽度（见图4-22）。

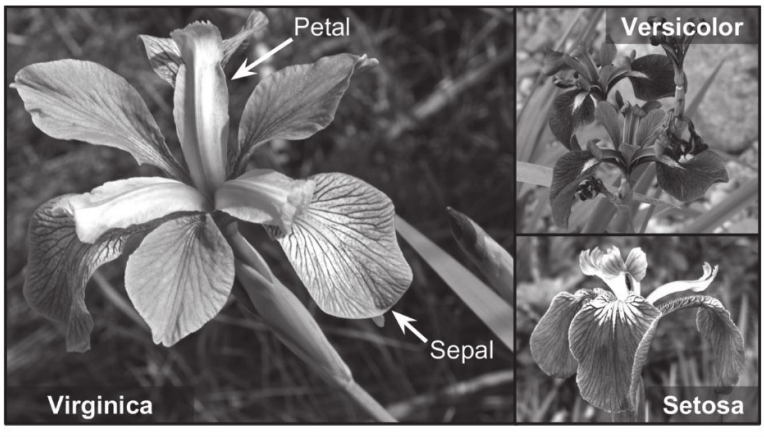


图4-22：三种不同品种的鸢尾花[1]

我们试试仅基于花瓣宽度这一个特征，创建一个分类器来检测维吉尼亚鸢尾花。首先加载数据：

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
>>> X = iris["data"][:, 3:]  # petal width
>>> y = (iris["target"] == 2).astype(np.int)  # 1 if Iris virginica, else 0
```

训练一个逻辑回归模型：

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

我们来看看花瓣宽度在0到3cm之间的鸢尾花，模型估算出的概率（见图4-23）[2]：

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris virginica")
# + more Matplotlib code to make the image look pretty
```

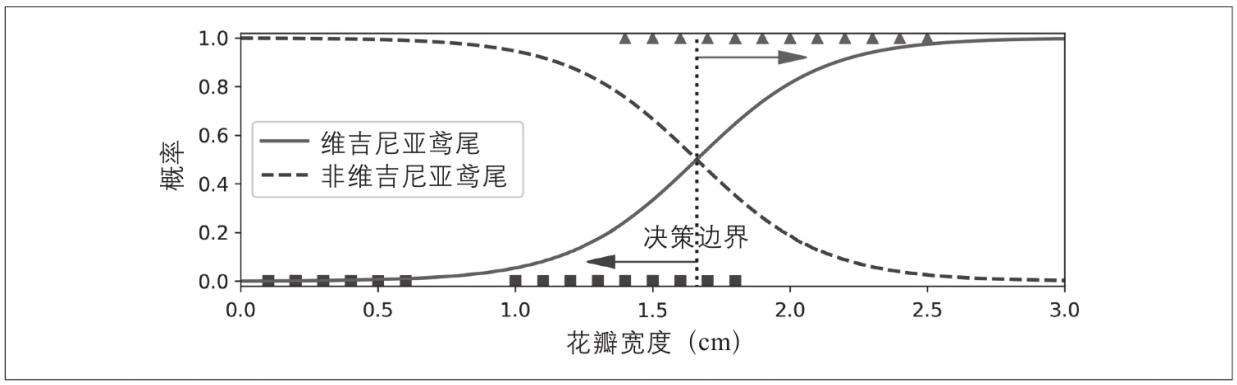


图4-23：估计的概率和决策边界

维吉尼亚鸢尾（三角形所示）的花瓣宽度范围为1.4～2.5cm，而其他两种鸢尾花（正方形所示）花瓣通常较窄，花瓣宽度范围为0.1～1.8cm。注意，这里有一部分重叠。对花瓣宽度超过2cm的花，分类器可以很有信心地说它是一朵维吉尼亚鸢尾花（对该类别输出一个高概率值），对花瓣宽度低于1cm以下的，也可以胸有成竹地说其不是（对“非维吉尼亚鸢尾”类别输出一个高概率值）。在这两个极端之间，分类器则不太有把握。但是，如果你要求它预测出类别（使用predict（）方法而不是predict_proba（）方法），它将返回一个可能性最大的类别。也就是说，在大约1.6cm处存在一个决策边界，这里“是”和“不是”的可能性都是50%，如果花瓣宽度大于1.6cm，分类器就预测它是维吉尼亚鸢尾花，否则就预测不是（即使它没什么把握）：

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

图4-24还是同样的数据集，但是这次显示了两个特征：花瓣宽度和花瓣长度。经过训练，这个逻辑回归分类器就可以基于这两个特征来预测新花朵是否属于维吉尼亚鸢尾。虚线表示模型估算概率为50%的点，即模型的决策边界。注意这里是一个线性的边界（注：这是点x的集合，使得 $\theta_0+\theta_1x_1+\theta_2x_2=0$，它定义了一条直线。）。每条平行线都分别代表一个模型输出的特定概率，从左下的15%到右上的90%。根据这个模型，右上线之上的所有花朵都有超过90%的概率属于维吉尼亚鸢尾。

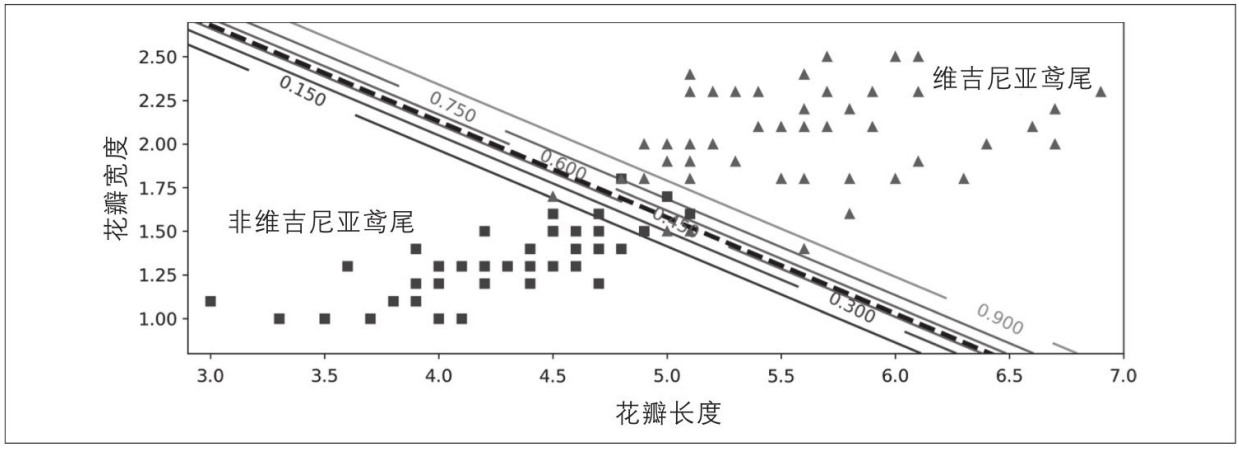


图4-24：线性决策边界

与其他线性模型一样，逻辑回归模型可以用ℓ_1或ℓ_2惩罚函数来正则化。Scikit-Learn默认添加的是ℓ_2函数。

 控制Scikit-Learn LogisticRegression模型的正则化强度的超参数不是alpha（与其他线性模型一样），而是反值C。C值越高，对模型的正则化越少。

4.6.4 Softmax回归

逻辑回归模型经过推广，可以直接支持多个类别，而不需要训练并组合多个二元分类器（如第3章所述）。这就是Softmax回归，或者叫作多元逻辑回归。

原理很简单：给定一个实例x，Softmax回归模型首先计算出每个类k的分数s_k（x），然后对这些分数应用softmax函数（也叫归一化指数），估算出每个类的概率。你应该很熟悉计算s_k（x）分数的公式（见公式4-19），因为它看起来就跟线性回归预测的方程一样。

公式4-19：类k的Softmax分数

$$s_k(\boldsymbol{x}) = \boldsymbol{x}^{\mathrm{T}}\boldsymbol{\theta}^{(k)}$$

请注意，每个类都有自己的特定参数向量θ（k）。所有这些向量通常都作为行存储在参数矩阵Θ中。

一旦为实例x计算了每个类的分数，就可以通过softmax函数来估计实例属于类k的概率$\hat{p}_k$（见公式4-20）。该函数计算每个分数的指数，然后对其进行归一化（除以所有指数的总和）。分数通常称为对数或对数奇数（尽管它们实际上是未归一化的对数奇数）。

公式4-20：Softmax函数

$$\hat{p}_k = \sigma(\boldsymbol{s}(\boldsymbol{x}))_k = \frac{\exp(s_k(\boldsymbol{x}))}{\sum_{j=1}^{K}\exp(s_j(\boldsymbol{x}))}$$

在此等式中：

• K是类数。

• s（x）是一个向量，其中包含实例x的每个类的分数。

• σ（s（x））$_k$是实例x属于类k的估计概率，给定该实例每个类的分数。

就像逻辑回归分类器一样，Softmax回归分类器预测具有最高估计概率的类（简单来说就是得分最高的类），如公式4-21所示。

公式4-21：Softmax回归分类预测

$$\hat{y} = \arg\max_k \sigma(\boldsymbol{s}(\boldsymbol{x}))_k = \arg\max_k s_k(\boldsymbol{x}) = \arg\max_k((\boldsymbol{\theta}^{(k)})^{\mathrm{T}}\boldsymbol{x})$$

argmax运算符返回使函数最大化的变量值。在此等式中，它返回使估计概率σ（s（x））$_k$最大化的k值。

 Softmax回归分类器一次只能预测一个类（即它是多类，而不是多输出），因此它只能与互斥的类（例如不同类型的植物）一起使用。你无法使用它在一张照片中识别多个人。

既然你已经知道了模型如何进行概率估算并做出预测，那我们再来看看怎么训练。训练目标是得到一个能对目标类做出高概率估算的模型（也就是其他类的概率相应要很低）。通过将公式4-22的成本函数（也叫作交叉熵）最小化来实现这个目标，因为当模型对目标类做出较低概率的估算时会受到惩罚。交叉熵经常被用于衡量一组估算出的类概率跟目标类的匹配程度（后面的章节中还会多次用到）。

公式4-22：交叉熵成本函数

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(\hat{p}_k^{(i)})$$

在此等式中：

- $y_k^{(i)}$是属于类k的第i个实例的目标概率。一般而言等于1或0，具体取决于实例是否属于该类。

请注意，当只有两个类（K=2）时，此成本函数等效于逻辑回归的成本函数（对数损失，请参见公式4-17）。

交叉熵

交叉熵源于信息理论。假设你想要有效传递每天的天气信息，选项（晴、下雨等）有8个，那么你可以用3比特对每个选项进行编码，因为$2^3=8$。但是，如果你认为几乎每天都是晴天，那么，对“晴天”用1比特（0），其他7个类用4比特（从1开始）进行编码，显然会更有效率一些。交叉熵测量的是你每次发送天气选项的平均比特数。如果你对天气的假设是完美的，交叉熵将会等于天气本身的熵（也就是其本身固有的不可预测性）。但是如果你的假设是错误的（比如经常下雨），交叉熵将会变大，增加的这一部分我们称之为KL散度（Kullback-Leibler divergence，也叫作相对熵）。

两个概率分布p和q之间的交叉熵定义为$H(p,q)=-\Sigma_x\ p(x)\log q(x)$（至少在离散分布时可以这样定义）。

公式（4-23）给出了该成本函数相对于$\theta^{(k)}$的梯度向量。

公式4-23：类k的交叉熵梯度向量

$$\nabla_{\theta(k)}J(\Theta)=\frac{1}{m}\sum_{i=1}^{m}(\hat{p}_k^{(i)}-y_k^{(i)})\boldsymbol{x}^{(i)}$$

现在，你可以计算每个类的梯度向量，然后使用梯度下降（或任何其他优化算法）来找到最小化成本函数的参数矩阵Θ。

我们来使用Softmax回归将鸢尾花分为三类。当用两个以上的类训练时，Scikit-Learn的LogisticRegressio默认选择使用的是一对多的训练方式，不过将超参数multi_class设置为"multinomial"，可以将其切换成Softmax回归。你还必须指定一个支持Softmax回归的求解器，比如"lbfgs"求解器（详见Scikit-Learn文档）。默认使用ℓ_2正则化，你可以通过超参数C进行控制：

```
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]
softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

所以当你下次碰到一朵鸢尾花，花瓣长5cm宽2cm，你就可以让模型告诉你它的种类，它会回答说：94.2%的概率是维吉尼亚鸢尾（第2类）或者5.8%的概率为变色鸢尾：

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

图4-25展现了由不同背景色表示的决策边界。注意，任何两个类之间的决策边界都是线性的。图中的折线表示属于变色鸢尾的概率（例如，标记为0.45的线代表45%的概率边界）。注意，该模型预测出的类，其估算概率有可能低于50%，比如，在所有决策边界相交的地方，所有类的估算概率都为33%。

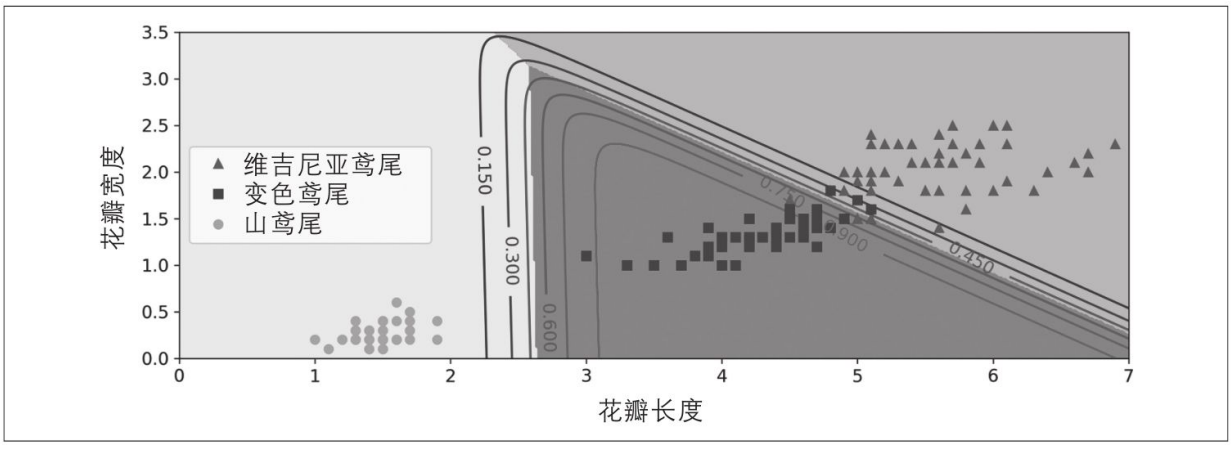


图4-25：Softmax回归决策边界

[1] 图片转自相应的维基百科页面。维吉尼亚鸢尾花的图片出自Frank Mayfield，变色鸢尾花的图片出自D.Gordon E.Robertson（Creative

Commons BY-SA 3.0），山鸢尾花图片来自公共域。

[2] NumPy的reshape（）函数允许将一个维度设为“-1”，这意味着“未指定”：该值是从数组的长度和其余维度推断出来的。

4.7 练习题

1. 如果训练集具有数百万个特征，那么可以使用哪种线性回归训练算法？

2. 如果训练集里特征的数值大小迥异，哪种算法可能会受到影响？受影响程度如何？你应该怎么做？

3. 训练逻辑回归模型时，梯度下降会卡在局部最小值中吗？

4. 如果你让它们运行足够长的时间，是否所有的梯度下降算法都能得出相同的模型？

5. 假设你使用批量梯度下降，并在每个轮次绘制验证误差。如果你发现验证错误持续上升，可能是什么情况？你该如何解决？

6. 当验证错误上升时立即停止小批量梯度下降是个好主意吗？

7. 哪种梯度下降算法（在我们讨论过的算法中）将最快到达最佳解附近？哪个实际上会收敛？如何使其他的也收敛？

8. 假设你正在使用多项式回归。绘制学习曲线后，你会发现训练误差和验证误差之间存在很大的差距。发生了什么？解决此问题的三种方法是什么？

9. 假设你正在使用岭回归，并且你注意到训练误差和验证误差几乎相等且相当高。你是否会说模型存在高偏差或高方差？你应该增加正则化超参数 α 还是减小它呢？

10. 为什么要使用：

a.岭回归而不是简单的线性回归（即没有任何正则化）？

b.Lasso而不是岭回归？

c.弹性网络而不是Lasso？

11.假设你要将图片分类为室外/室内和白天/夜间。你应该实现两个逻辑回归分类器还是一个Softmax回归分类器？

12.用Softmax回归进行批量梯度下降训练，实现提前停止法（不使用Scikit-Learn）。

附录A中提供了这些练习题的解答。

第5章　支持向量机

支持向量机（Support Vector Machine，SVM）是一个功能强大并且全面的机器学习模型，它能够执行线性或非线性分类、回归，甚至是异常值检测任务。它是机器学习领域最受欢迎的模型之一，任何对机器学习感兴趣的人都应该在工具箱中配备一个。SVM特别适用于中小型复杂数据集的分类。

本章将会介绍不同SVM的核心概念、如何使用它们以及它们的工作原理。

5.1 线性SVM分类

SVM的基本思想可以用一些图来说明。图5-1所示的数据集来自第4章末尾引用的鸢尾花数据集的一部分。两个类可以轻松地被一条直线（它们是线性可分离的）分开。左图显示了三种可能的线性分类器的决策边界。其中虚线所代表的模型表现非常糟糕，甚至都无法正确实现分类。其余两个模型在这个训练集上表现堪称完美，但是它们的决策边界与实例过于接近，导致在面对新实例时，表现可能不会太好。相比之下，右图中的实线代表SVM分类器的决策边界，这条线不仅分离了两个类，并且尽可能远离了最近的训练实例。你可以将SVM分类器视为在类之间拟合可能的最宽的街道（平行的虚线所示）。因此这也叫作大间隔分类。

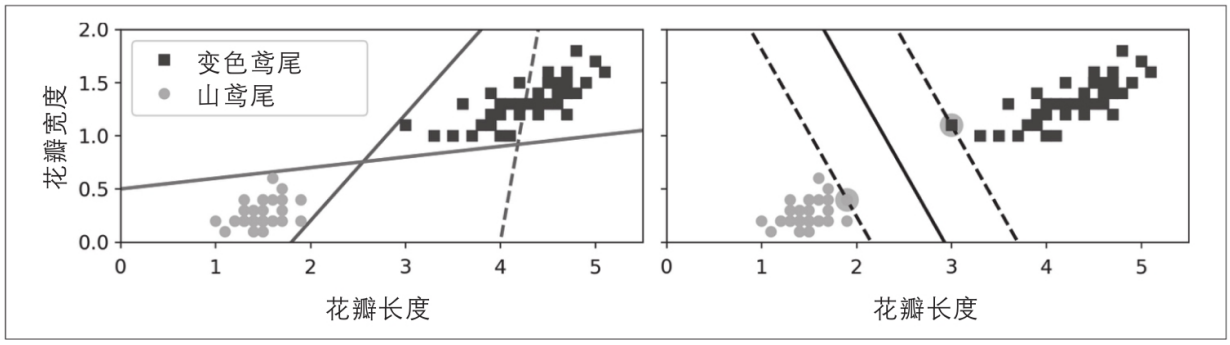


图5-1：大间隔分类

请注意，在“街道以外”的地方增加更多训练实例不会对决策边界产生影响，也就是说它完全由位于街道边缘的实例所决定（或者称之为“支持”）。这些实例被称为支持向量（在图5-1中已圈出）。

SVM对特征的缩放非常敏感，如图5-2所示，在左图中，垂直刻度比水平刻度大得多，因此可能的最宽的街道接近于水平。在特征缩放（例如使用Scikit-Learn的StandardScaler）后，决策边界看起来好了很多（见右图）。

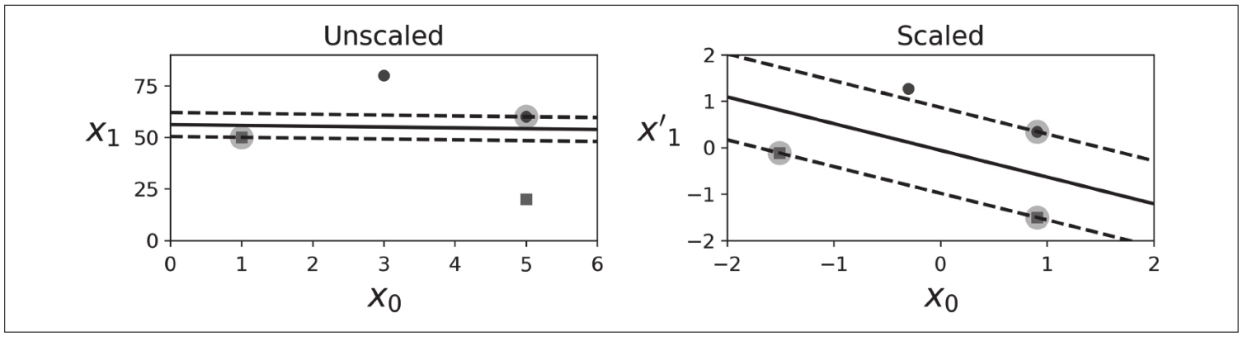


图5-2：特征缩放敏感性

软间隔分类

如果我们严格地让所有实例都不在街道上，并且位于正确的一边，这就是硬间隔分类。硬间隔分类有两个主要问题。首先，它只在数据是线性可分离的时候才有效；其次，它对异常值非常敏感。图5-3显示了有一个额外异常值的鸢尾花数据：左图的数据根本找不出硬间隔，而右图最终显示的决策边界与我们在图5-1中所看到的无异常值时的决策边界也大不相同，可能无法很好地泛化。

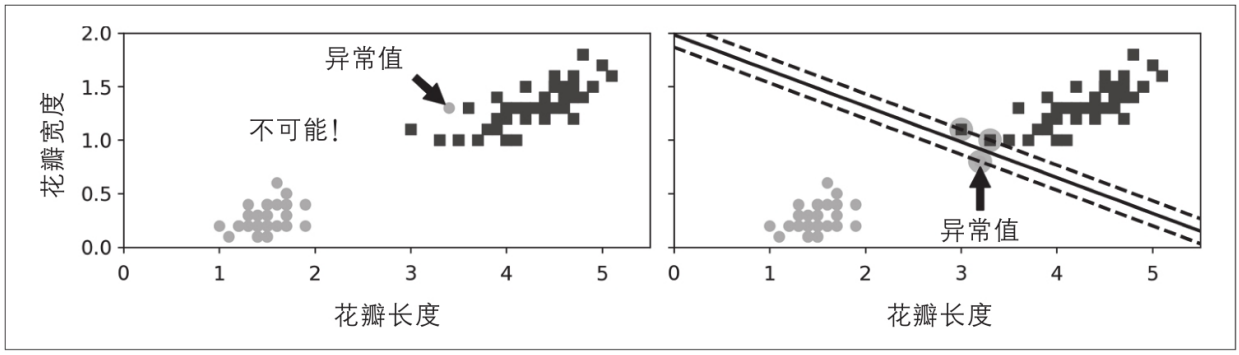


图5-3：硬间隔对异常值的敏感度

要避免这些问题，最好使用更灵活的模型。目标是尽可能在保持街道宽阔和限制间隔违例（即位于街道之上，甚至在错误的一边的实例）之间找到良好的平衡，这就是软间隔分类。

使用Scikit-Learn创建SVM模型时，我们可以指定许多超参数。C是这些超参数之一。如果将其设置为较低的值，则最终得到图5-4左侧的模型。如果设置为较高的值，我们得到右边的模型。间隔冲突很糟糕，通常最好要少一些。但是，在这种情况下，左侧的模型存在很多间隔违例的情况，但泛化效果可能会更好。

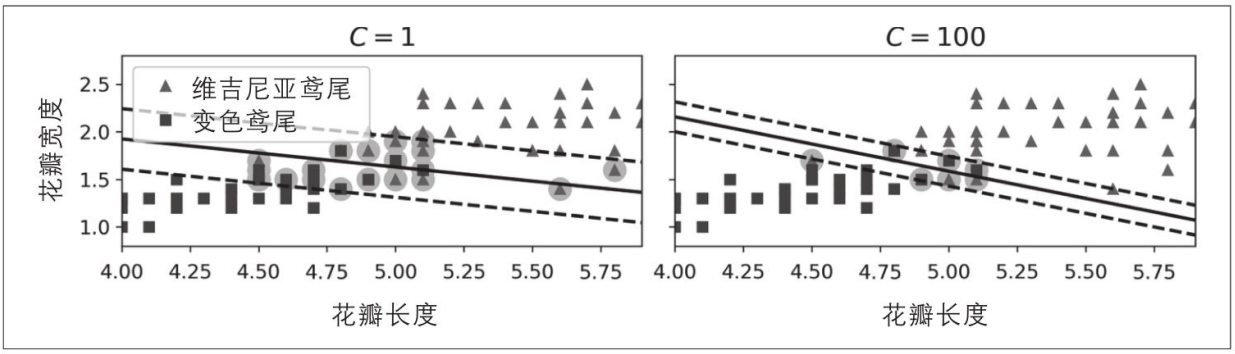


图5-4：大间隔（左）与更少的间隔冲突（右）

 如果你的SVM模型过拟合，可以尝试通过降低C来对其进行正则化。

以下Scikit-Learn代码可加载鸢尾花数据集，缩放特征，然后训练线性SVM模型（使用C=1的LinearSVC类和稍后描述的hinge损失函数）来检测维吉尼亚鸢尾花：

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.float64)  # Iris virginica

svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge")),
    ])

svm_clf.fit(X, y)
```

生成的模型如图5-4左图所示。

你可以像往常一样使用模型进行预测：

```
>>> svm_clf.predict([[5.5, 1.7]])
array([1.])
```

 与Logistic回归分类器不同，SVM分类器不会输出每个类的概率。

我们可以将SVC类与线性内核一起使用，而不使用LinearSVC类。创建SVC模型时，我们可以编写SVC（kernel="linear"，C=1）。或者我们可以将SGDClassifier类与SGDClassifier（loss="hinge"，alpha=1/（m*C））一起使用。这将使用常规的随机梯度下降（见第4章）来训练线性SVM分类器。它的收敛速度不如LinearSVC类，但是对处理在线分类任务或不适合内存的庞大数据集（核外训练）很有用。

 LinearSVC类会对偏置项进行正则化，所以你需要先减去平均值，使训练集居中。如果使用StandardScaler会自动进行这一步。此外，请确保超参数loss设置为"hinge"，因为它不是默认值。最后，为了获得更好的性能，还应该将超参数dual设置为False，除非特征数量比训练实例还多（本章后文将会讨论）。

5.2 非线性SVM分类

虽然在许多情况下，线性SVM分类器是有效的，并且通常出人意料的好，但是，有很多数据集远不是线性可分离的。处理非线性数据集的方法之一是添加更多特征，比如多项式特征（如第4章所述）。某些情况下，这可能导致数据集变得线性可分离。参见图5-5的左图：这是一个简单的数据集，只有一个特征x_1。可以看出，数据集线性不可分。但是如果添加第二个特征$x_2=(x_1)^2$，生成的2D数据集则完全线性可分离。

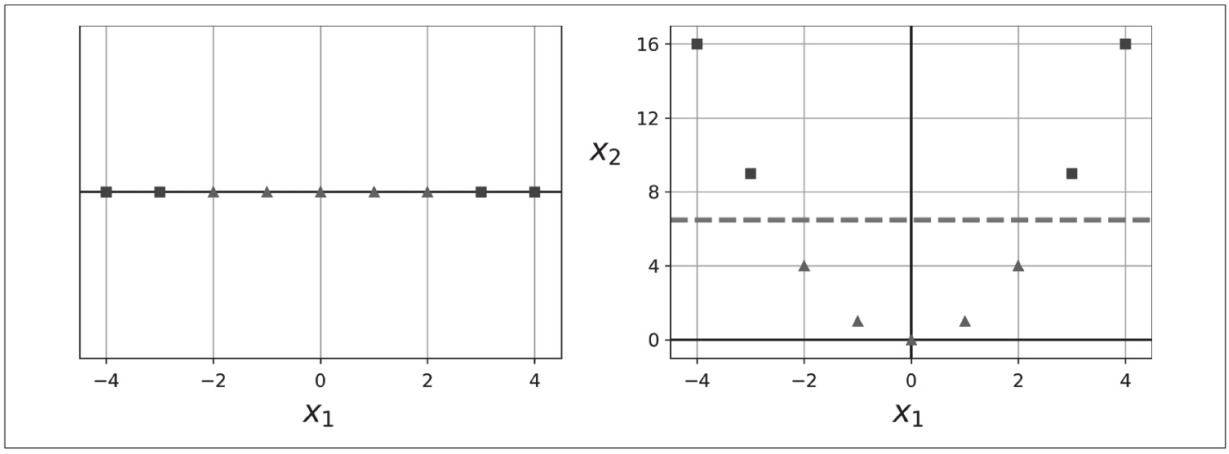


图5-5：通过添加特征使数据集线性可分离

为了使用Scikit-Learn来实现这个想法，创建一个包含PolynomialFeatures转换器（见4.3节）的Pipeline，然后是StandardScaler和LinearSVC。让我们在卫星数据集上进行测试：这是一个用于二元分类的小数据集，其中数据点的形状为两个交织的半圆（见图5-6）。你可以使用make_moons（）函数生成此数据集：

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

X, y = make_moons(n_samples=100, noise=0.15)
polynomial_svm_clf = Pipeline([
        ("poly_features", PolynomialFeatures(degree=3)),
```

```
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge"))
    ])

polynomial_svm_clf.fit(X, y)
```

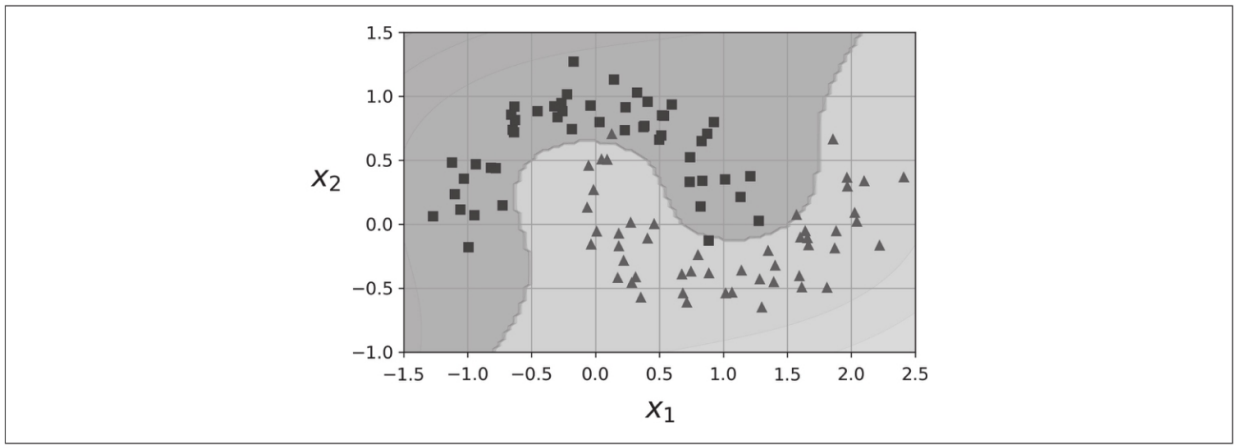


图5-6：使用多项式特征的线性SVM分类器

5.2.1 多项式内核

添加多项式特征实现起来非常简单，并且对所有的机器学习算法（不只是SVM）都非常有效。但问题是，如果多项式太低阶，则处理不了非常复杂的数据集。而高阶则会创造出大量的特征，导致模型变得太慢。

幸运的是，使用SVM时，有一个魔术般的数学技巧可以应用，这就是核技巧（稍后解释）。它产生的结果就跟添加了许多多项式特征（甚至是非常高阶的多项式特征）一样，但实际上并不需要真的添加。因为实际没有添加任何特征，所以也就不存在数量爆炸的组合特征了。这个技巧由SVC类来实现，我们看看在卫星数据集上的测试：

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```

这段代码使用了一个3阶多项式内核训练SVM分类器。如图5-7的左图所示。而右图是另一个使用了10阶多项式核的SVM分类器。显然，如果模型过拟合，你应该降低多项式阶数；反过来，如果欠拟合，则可以尝试使之提升。超参数coef0控制的是模型受高阶多项式还是低阶多项式影响的程度。

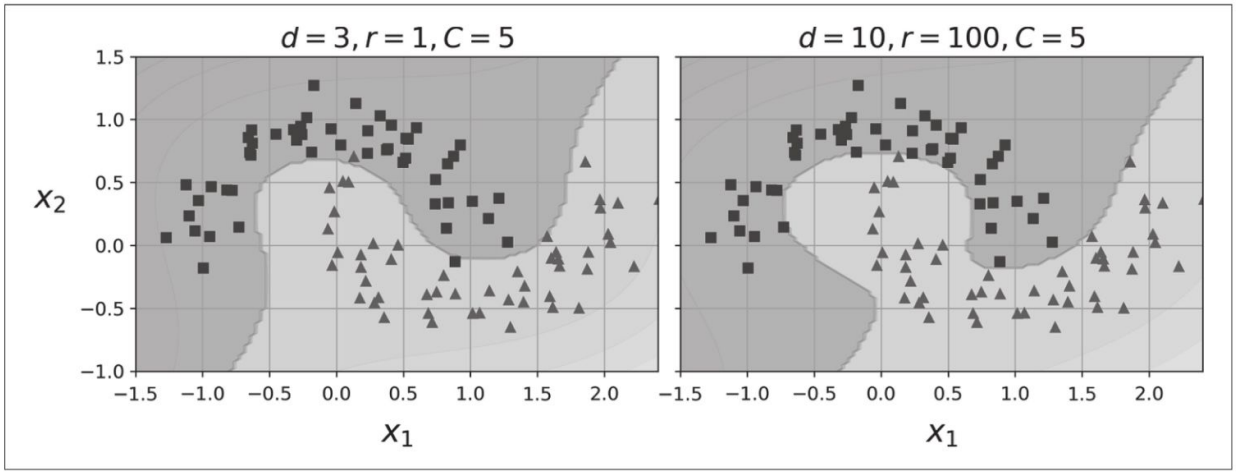


图5-7：多项式核的SVM分类器

 寻找正确的超参数值的常用方法是网格搜索（见第2章）。先进行一次粗略的网格搜索，然后在最好的值附近展开一轮更精细的网格搜索，这样通常会快一些。多了解每个超参数实际上是用来做什么的，有助于你在超参数空间层正确搜索。

5.2.2 相似特征

解决非线性问题的另一种技术是添加相似特征，这些特征经过相似函数计算得出，相似函数可以测量每个实例与一个特定地标之间的相似度。以前面提到过的一维数据集为例，在x_1=-2和x_1=1处添加两个地标（见图5-8中的左图）。接下来，我们采用高斯径向基函数（RBF）作为相似函数，γ=0.3（见等式5-1）。

公式5-1：高斯RBF

$$\phi_\gamma(\boldsymbol{x}, \ell) = \exp(-\gamma \| \boldsymbol{x} - \ell \|^2)$$

这是一个从0（离地标差得非常远）到1（跟地标一样）变化的钟形函数。现在我们准备计算新特征。例如，我们看实例x_1=-1：它与第一个地标的距离为1，与第二个地标的距离为2。因此它的新特征为x_2=eps（-0.3×1^2）≈0.74，x_3=eps（-0.3×2^2）≈0.30。图5-8的右图显示了转换后的数据集（去除了原始特征），现在你可以看出，数据呈线性可分离了。

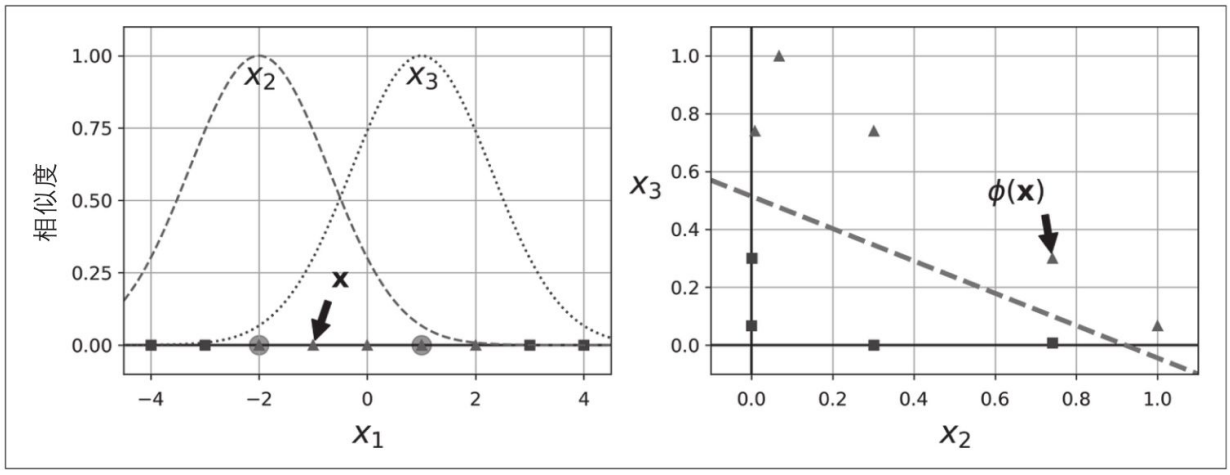


图5-8：使用高斯RBF的相似特征

你可能想知道怎么选择地标。最简单的方法是在数据集里每一个实例的位置上创建一个地标。这会创造出许多维度，因而也增加了转换后的训练集线性可分离的机会。缺点是一个有m个实例n个特征的训练集会被转换成一个m个实例m个特征的训练集（假设抛弃了原始特征）。如果训练集非常大，那就会得到同样大数量的特征。

5.2.3　高斯RBF内核

与多项式特征方法一样，相似特征法也可以用任意机器学习算法，但是要计算出所有附加特征，其计算代价可能非常昂贵，尤其是对大型训练集来说。然而，核技巧再一次施展了它的SVM魔术：它能够产生的结果就跟添加了许多相似特征一样（但实际上也并不需要添加）。我们来使用SVC类试试高斯RBF核：

```
rbf_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ])
rbf_kernel_svm_clf.fit(X, y)
```

图5-9的左下方显示了这个模型。其他图显示了超参数gamma（γ）和C使用不同值时的模型。增加gamma值会使钟形曲线变得更窄（见图5-8的左图），因此每个实例的影响范围随之变小：决策边界变得更不规则，开始围着单个实例绕弯。反过来，减小gamma值使钟形曲线变得更宽，因而每个实例的影响范围增大，决策边界变得更平坦。所以γ就像是一个正则化的超参数：模型过拟合，就降低它的值，如果欠拟合则提升它的值（类似超参数C）。

还有一些其他较少用到的核函数，例如专门针对特定数据结构的核函数。字符串核常用于文本文档或是DNA序列（如使用字符串子序列核或是基于莱文斯坦距离的核函数）的分类。

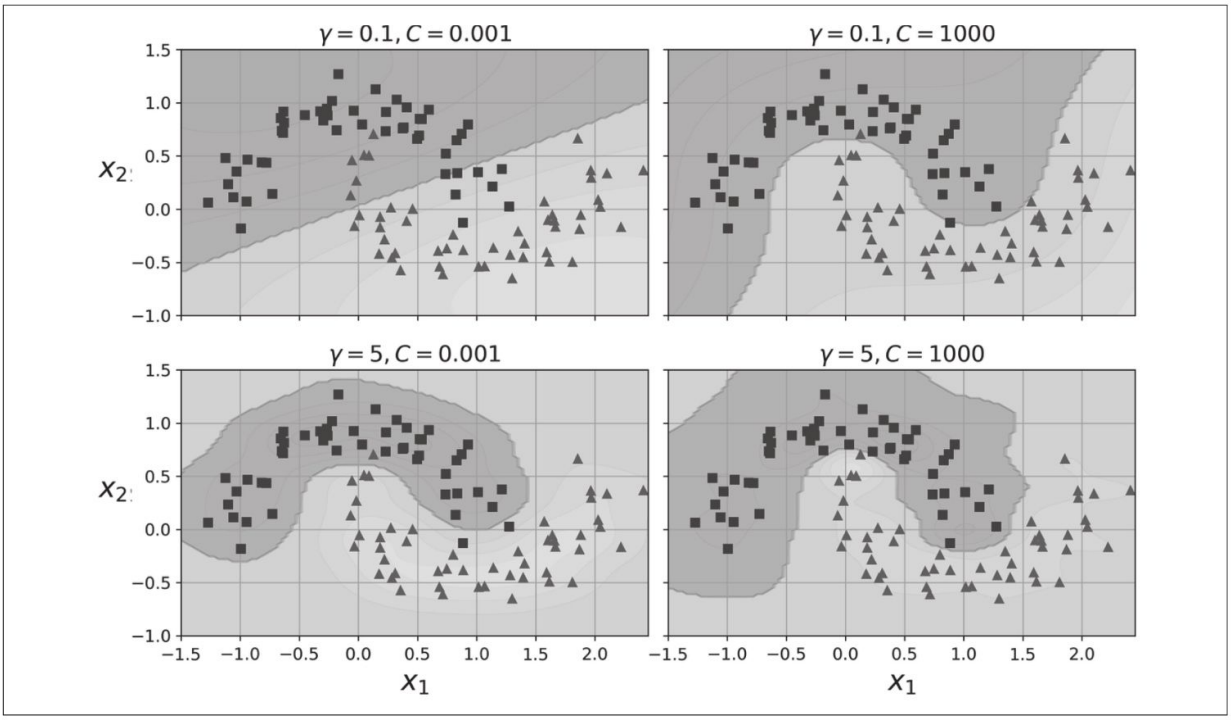


图5-9：使用RBF核的SVM分类器

 有这么多的核函数，该如何决定使用哪一个呢？有一个经验法则是，永远先从线性核函数开始尝试（要记住，LinearSVC比SVC（kernel="linear"）快得多），特别是训练集非常大或特征非常多的时候。如果训练集不太大，你可以试试高斯RBF核，大多数情况下它都非常好用。如果你还有多余的时间和计算能力，可以使用交叉验证和网格搜索来尝试一些其他的核函数，特别是那些专门针对你的数据集数据结构的核函数。

5.2.4　计算复杂度

liblinear库为线性SVM实现了一个优化算法[1]，LinearSVC正是基于该库的。该算法不支持核技巧，不过它与训练实例的数量和特征数量几乎呈线性相关：其训练时间复杂度大致为O（m×n）。

如果你想要非常高的精度，算法需要的时间更长。它由容差超参数ε（在Scikit-Learn中为tol）来控制。大多数分类任务中，默认的容

差就够了。

SVC则是基于libsvm库的，这个库的算法支持核技巧[2]。训练时间复杂度通常在O（$m^2 \times n$）和O（$m^3 \times n$）之间。很不幸，这意味着如果训练实例的数量变大（例如成千上万的实例），它将会慢得可怕，所以这个算法完美适用于复杂但是中小型的训练集。但是，它还是可以良好地适应特征数量的增加，特别是应对稀疏特征（即每个实例仅有少量的非零特征）。在这种情况下，算法复杂度大致与实例的平均非零特征数成比例。表5-1比较了Scikit-Learn的SVM分类器类。

表5-1：用于SVM分类的Scikit-Learn类的比较

类	时间复杂度	核外支持	需要缩放	核技巧
LinearSVC	$O(m \times n)$	否	是	否
SGDClassifier	$O(m \times n)$	是	是	否
SVC	$O(m^2 \times n)$ 到 $O(m^3 \times n)$	否	是	是

[1] Chih-Jen Lin et al.，“A Dual Coordinate Descent Method for Large-Scale Linear SVM”，Proceedings of the 25th International Conference on Machine Learning（2008）：408-415.

[2] John Platt，“Sequential Minimal Optimization：A Fast Algorithm for Training Support Vector Machines”（Microsoft Research technical report，April 21，1998），https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-98-14.pdf.

5.3 SVM回归

正如前面提到的，SVM算法非常全面：它不仅支持线性和非线性分类，而且还支持线性和非线性回归。诀窍在于将目标反转一下：不再尝试拟合两个类之间可能的最宽街道的同时限制间隔违例，SVM回归要做的是让尽可能多的实例位于街道上，同时限制间隔违例（也就是不在街道上的实例）。街道的宽度由超参数 ε 控制。图5-10显示了用随机线性数据训练的两个线性SVM回归模型，一个间隔较大（ ε =1.5），另一个间隔较小（ ε =0.5）。

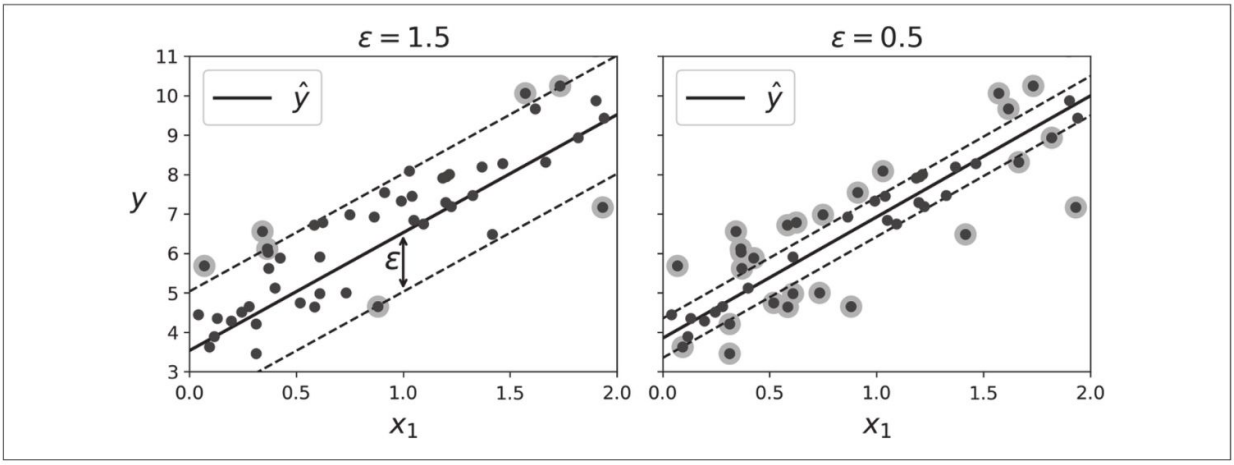


图5-10：SVM回归

在间隔内添加更多的实例不会影响模型的预测，所以这个模型被称为 ε 不敏感。

你可以使用Scikit-Learn的LinearSVR类来执行线性SVM回归。以下代码生成如图5-10左图所示的模型（训练数据需要先缩放并居中）：

```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

要解决非线性回归任务，可以使用核化的SVM模型。例如，图5-11显示了在一个随机二次训练集上使用二阶多项式核的SVM回归。左图几乎没有正则化（C值很大），右图则过度正则化（C值很小）。

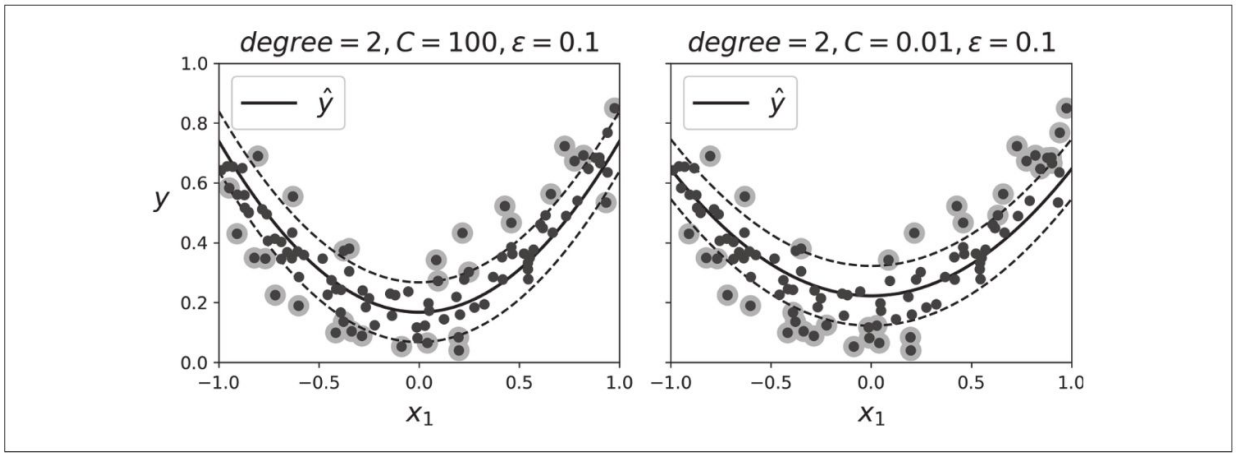


图5-11：使用二阶多项式核的SVM回归

以下代码使用Scikit-Learn的SVR类（支持核技巧）生成如图5-11左图所示的模型：

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

SVR类是SVC类的回归等价物，LinearSVR类也是LinearSVC类的回归等价物。LinearSVR与训练集的大小线性相关（与LinearSVC一样），而SVR则在训练集变大时，变得很慢（SVC也一样）。

 SVM也可用于异常值检测，详细信息请参考Scikit-Learn文档。

5.4 工作原理

本节将会介绍SVM如何进行预测，以及它们的训练算法是如何工作的，从线性SVM分类器开始。如果你刚刚开始接触机器学习，可以安全地跳过本节，直接进入本章末尾的练习，等到想要更深入地了解SVM时再回来也不迟。

首先，说明一下符号。在第4章里，我们使用过一个约定，将所有模型参数放在一个向量 θ 中，包括偏置项 θ_0，以及输入特征的权重 θ_1 到 θ_n，同时在所有实例中添加偏置项 $x_0=1$。在本章中，我们将会使用另一个约定，在处理SVM时它更为方便（也更常见）：偏置项表示为b，特征权重向量表示为w，同时输入特征向量中不添加偏置特征。

5.4.1 决策函数和预测

线性SVM分类器通过简单地计算决策函数 $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b = w_1 x_1 + \cdots + w_n x_n + b$ 来预测新实例x的分类。如果结果为正，则预测类别是正类（1），否则预测其为负类（0），见公式5-2。

公式5-2：线性SVM分类器预测

$$\hat{y} = \begin{cases} 0, & \text{如果} \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b < 0 \\ 1, & \text{如果} \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b \geqslant 0 \end{cases}$$

图5-12显示了图5-4右侧的模型所对应的决策函数：数据集包含两个特征（花瓣宽度和长度），所以是一个二维平面。决策边界是决策函

数等于0的点的集合：它是两个平面的交集，也就是一条直线（加粗实线所示）[1]。

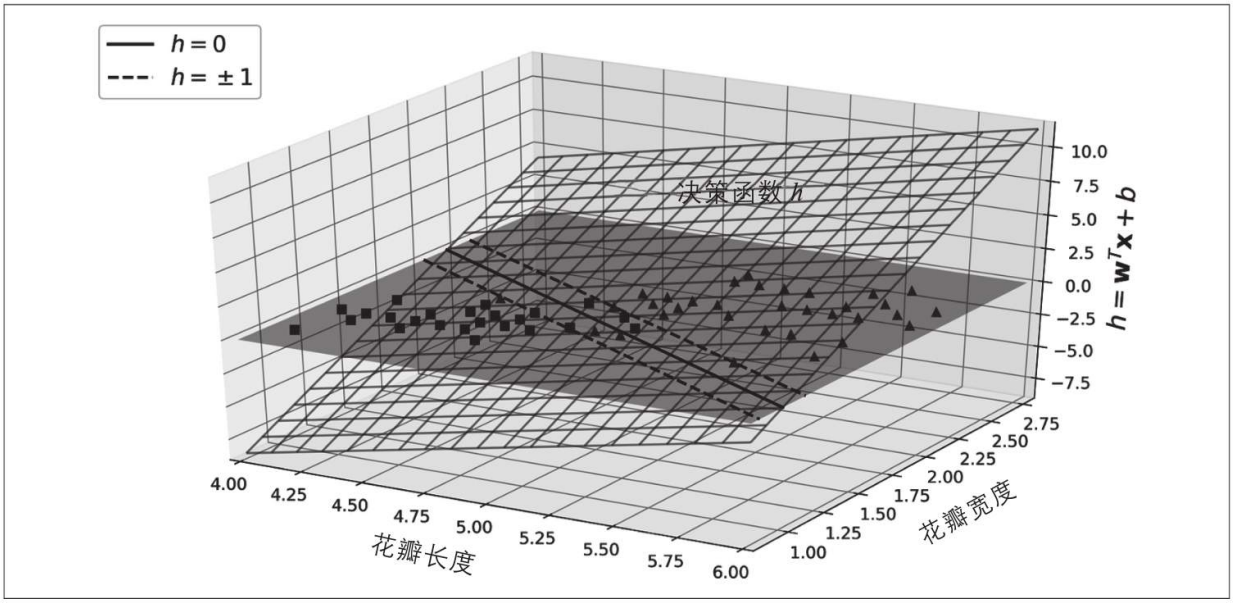


图5-12：鸢尾花数据集的决策函数

虚线表示决策函数等于1或-1的点：它们互相平行，并且与决策边界的距离相等，从而形成了一个间隔。训练线性SVM分类器意味着找到w和b的值，从而使这个间隔尽可能宽的同时，避免（硬间隔）或限制（软间隔）间隔违例。

5.4.2 训练目标

思考一下决策函数的斜率：它等于权重向量的范数，即||w||。如果我们将斜率除以2，那么决策函数等于±1的点也将变得离决策函数两倍远。也就是说，将斜率除以2，将会使间隔乘以2。也许2D图更容易将其可视化，见图5-13。权重向量w越小，间隔越大。

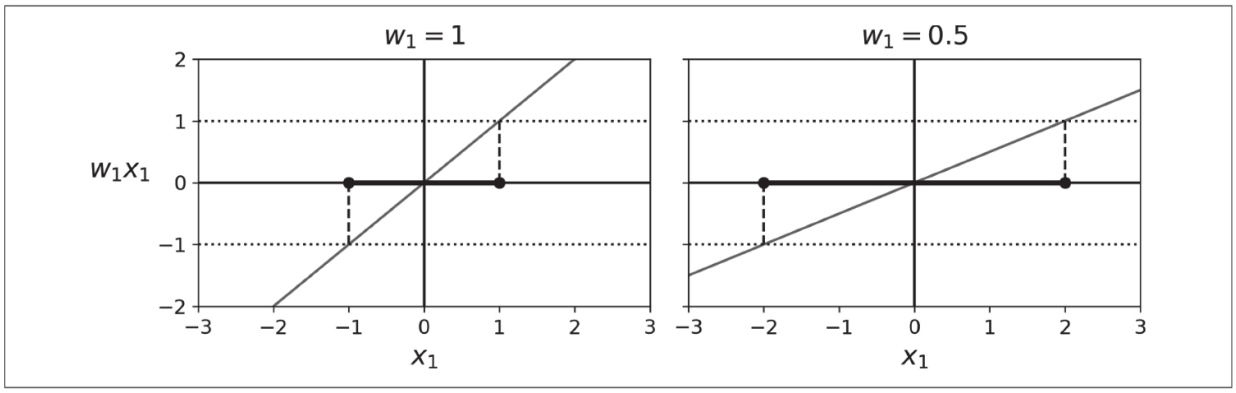


图5-13：权重向量越小，间隔越大

我们要最小化||w||来得到尽可能大的间隔。但是，如果我们想避免任何间隔违例（硬间隔），那么就要使所有正类训练集的决策函数大于1，负类训练集的决策函数小于-1。如果我们定义实例为负类（如果$y^{(i)}=0$）时，$t^{(i)}=-1$；实例为正类（如果$y^{(i)}=1$）时，$t^{(i)}=1$。那么就可以将这个约束条件表示为：对所有实例来说，$t^{(i)}(\boldsymbol{w}^{\mathrm{T}} \cdot \boldsymbol{x}^{(i)} + b) \geqslant 1$。

因此，我们可以将硬间隔线性SVM分类器的目标看作一个约束优化问题，如公式5-3所示。

公式5-3：硬间隔线性SVM分类器的目标

$$\underset{\boldsymbol{w},b}{\mathrm{minimize}} \frac{1}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w}$$

$$\text{使得 } t^{(i)}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}^{(i)} + b) \geqslant 1, \quad i = 1, 2, \cdots, m$$

 我们要最小化$\frac{1}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w}\left(\text{等于}\frac{1}{2}\|\boldsymbol{w}\|^2\right)$，而不是最小化||w||，的确，$\frac{1}{2}\|\boldsymbol{w}\|^2$有一个很好的简单的导数（是w），而||w||在w=0时不可

微。优化算法在可微函数上的工作效果更好。

要达到软间隔的目标，我们需要为每个实例引入一个松弛变量 $\zeta^{(i)} \geqslant 0$[2]，$\zeta^{(i)}$ 衡量的是第i个实例多大程度上允许间隔违例。那么现在我们有了两个互相冲突的目标：使松弛变量越小越好从而减少间隔违例，同时还要使 $w^T \cdot w/2$ 最小化以增大间隔。这正是超参数C的用武之地：允许我们在两个目标之间权衡。公式5-4给出了这个约束优化问题。

公式5-4：软间隔线性SVM分类器目标

$$\underset{\boldsymbol{w},b,\zeta}{\text{minimize}} \frac{1}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w} + C\sum_{i=1}^{m}\zeta^{(i)}$$

$$\text{使得 } t^{(i)}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}^{(i)} + b) \geqslant 1 - \zeta^{(i)} \text{ 和 } \zeta^{(i)} \geqslant 0, i = 1, 2, \cdots, m$$

5.4.3 二次规划

硬间隔和软间隔问题都属于线性约束的凸二次优化问题。这类问题被称为二次规划（QP）问题。要解决二次规划问题有很多现成的求解器，使用到的技术各不相同，这些不在本书的讨论范围之内[3]。

公式5-5给出的是问题的一般形式。

公式5-5：二次规划问题

$$\underset{\boldsymbol{p}}{\text{minimize}} \frac{1}{2}\boldsymbol{p}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{p} + \boldsymbol{f}^{\mathrm{T}}\boldsymbol{p}, \text{使得} \boldsymbol{A}\boldsymbol{p} \leqslant \boldsymbol{b}$$

其中：

• p是一个n_p维向量（n_p是参数数量）

• H是一个$n_p \times n_p$的矩阵

• f是一个n_p维的向量

• A是一个$n_c \times n_p$的矩阵（n_c是约束的数量）

• b是一个n_c维的向量

请注意，表达式$Ap \leqslant b$定义了n_c个约束：$p^T a^{(i)} \leqslant b^{(i)}$，其中i=1，2，…，$n_c$，其中$a^{(i)}$是包含A的第i行元素的向量，$b^{(i)}$是b的第i个元素。

你可以轻松地验证，如果用以下方式设置QP参数，就可以得到硬间隔线性SVM分类器的目标：

• n_p=n+1，其中n是特征数量（+1是偏置项）。

• n_c=m，其中m是训练实例的数量。

• H是$n_p \times n_p$单位矩阵，除了在左上方的元素为零（忽略偏置项）。

• f=0，一个全零的n_p维向量。

• b=-1，一个全是-1的n_c维向量。

• $\boldsymbol{a}^{(i)} = -t^{(i)}\dot{\boldsymbol{x}}^{(i)}$，其中$\dot{x}^{(i)}$等于$x^{(i)}$，并且具有额外的偏差特征$\dot{\boldsymbol{x}}_0 = 1$。

所以，要训练硬间隔线性SVM分类器，有一种办法是直接将上面的参数用在一个现成的二次规划求解器上。得到的向量p将会包括偏置项$b=p_0$，以及特征权重$w_i=p_i$，$i=1，2，\cdots，m$。类似地，你也可以用二次规划求解器来解决软间隔问题（见本章末尾练习）。

但是，为了运用核技巧，接下来我们将要看一个不同的约束优化问题。

5.4.4　对偶问题

针对一个给定的约束优化问题，称之为原始问题，我们常常可以用另一个不同的，但是与之密切相关的问题来表达，这个问题我们称之为对偶问题。通常来说，对偶问题的解只能算是原始问题的解的下限，但是在某些情况下，它也可能跟原始问题的解完全相同。幸运的是，SVM问题刚好就满足这些条件[4]，所以你可以选择是解决原始问题还是对偶问题，二者解相同。公式5-6给出了线性SVM目标的对偶形式（如果你对如何从原始问题导出对偶问题感兴趣，请参阅附录C）。

公式5-6：线性SVM目标的对偶形式

$$\underset{\alpha}{\text{minimize}}\ \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}\boldsymbol{x}^{(i)\mathrm{T}}\boldsymbol{x}^{(j)}-\sum_{i=1}^{m}\alpha^{(i)}$$

$$\text{使得 } \alpha^{(i)}\geqslant 0,\ i=1,2,\cdots,m$$

一旦得到使得该等式最小化（使用二次规划求解器）的向量$\hat{\boldsymbol{\alpha}}$，就可以使用公式5-7来计算使原始问题最小化的$\hat{\boldsymbol{w}}$和$\hat{b}$。

公式5-7：从对偶问题到原始问题

$$\hat{\boldsymbol{w}} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \boldsymbol{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} (t^{(i)} - \hat{\boldsymbol{w}}^{\mathrm{T}} \boldsymbol{x}^{(i)})$$

当训练实例的数量小于特征数量时，解决对偶问题比原始问题更快速。更重要的是，它能够实现核技巧，而原始问题不可能实现。这个核技巧到底是什么呢？

5.4.5 内核化SVM

假设你想要将一个二阶多项式转换为一个二维训练集（例如卫星训练集），然后在转换训练集上训练线性SVM分类器。这个二阶多项式的映射函数Φ如公式5-8所示。

公式5-8：二阶多项式映射

$$\phi(\boldsymbol{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2} x_1 x_2 \\ x_2^2 \end{pmatrix}$$

注意转换后的向量是三维的而不是二维的。现在我们来看看，如果应用这个二阶多项式映射，两个二维向量a和b会发生什么变化，然后计算转换后两个向量的点积（注：如第4章所述，两个向量a和b的点积通

常记为a·b。但是，在机器学习中，向量经常表示为列向量（即单列矩阵），因此点积可通过计算a^Tb获得。为了与本书的其余部分保持一致，我们将在此处使用此表示法，而忽略了这一事实，即从技术上讲会导致一个单元矩阵而不是标量值。）（参见公式5-9）。

公式5-9：二阶多项式映射的核技巧

$$\phi(\boldsymbol{a})^{\mathrm{T}}\phi(\boldsymbol{b}) = \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2$$

$$= (a_1b_1 + a_2b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\boldsymbol{a}^{\mathrm{T}}\boldsymbol{b})^2$$

怎么样？转换后向量的点积等于原始向量的点积的平方：$\phi(\boldsymbol{a})^{\mathrm{T}}\phi(\boldsymbol{b}) = (\boldsymbol{a}^{\mathrm{T}}\boldsymbol{b})^2$。

关键点：如果将转换映射Φ应用于所有训练实例，那么对偶问题（见公式5-6）将包含点积$\phi(x^{(i)})^{\mathrm{T}}\phi(x^{(j)})$的计算。如果Φ是公式5-8所定义的二阶多项式转换，那么可以直接用$(\boldsymbol{x}^{(i)\mathrm{T}}\boldsymbol{x}^{(j)})^2$来替代这个转换向量的点积。所以你根本不需要转换训练实例，只需将公式5-6里的点积换成点积的平方即可。如果你不嫌麻烦，可以动手将训练集进行转换，然后拟合线性SVM算法，你会发现，结果一模一样。但是这个技巧大大提高了整个过程的计算效率，这就是核技巧的本质。

函数K（a，b）=（a^T·b）^{2}被称为二阶多项式核。在机器学习里，核是能够仅基于原始向量a和b来计算点积Φ（a）^{T}Φ（b）的函数，它

不需要计算（甚至不需要知道）转换函数。公式5-10列出了一些最常用的核函数。

公式5-10：常用核函数

$$\text{线性：} \quad K(\boldsymbol{a}, \boldsymbol{b}) = \boldsymbol{a}^{\mathrm{T}}\boldsymbol{b}$$

$$\text{多项式：} \quad K(\boldsymbol{a}, \boldsymbol{b}) = (\gamma \boldsymbol{a}^{\mathrm{T}}\boldsymbol{b} + r)^{d}$$

$$\text{高斯RBF：} \quad K(\boldsymbol{a}, \boldsymbol{b}) = \exp(-\gamma \| \boldsymbol{a} - \boldsymbol{b} \|^2)$$

$$\text{Sigmoid：} \quad K(\boldsymbol{a}, \boldsymbol{b}) = \tanh(\gamma \boldsymbol{a}^{\mathrm{T}}\boldsymbol{b} + r)$$

Mercer定理

根据Mercer定理，如果函数K（a，b）符合几个数学条件——也就是Mercer条件（K必须是连续的，并且在其参数上对称，所以K（a，b）=K（b，a），等等），则存在函数Φ将a和b映射到另一空间（可能是更高维度的空间），使得K（a，b）=Φ（a）^{T}Φ（b）。所以你可以将K用作核函数，因为你知道Φ是存在的，即使你不知道它是什么。对于高斯RBF核函数，可以看出，Φ实际上将每个训练实例映射到了一个无限维空间，幸好不用执行这个映射。

注意，也有一些常用的核函数（如S型核函数）不符合Mercer条件的所有条件，但是它们在实践中通常也表现不错。

还有一个未了结的问题需要说明。公式5-7显示了用线性SVM分类器如何从对偶解走到原始解，但是如果你应用了核技巧，最终得到的是包含Φ（$x^{(i)}$）的方程。而$\hat{\boldsymbol{w}}$的维度数量必须与Φ（$x^{(i)}$）相同，后者很有可能是巨大甚至是无穷大的，所以你根本没法计算。可是不知道$\hat{\boldsymbol{w}}$该如何做出预测呢？你可以将公式5-7中$\hat{\boldsymbol{w}}$的公式插入新实例$x^{(n)}$的

决策函数中，这样就得到了一个只包含输入向量之间点积的公式。这时你就可以再次运用核技巧了（见公式5-11）。

公式5-11：使用核化SVM做出预测

$$h_{\hat{\boldsymbol{w}},\hat{b}}(\phi(\boldsymbol{x}^{(n)})) = \hat{\boldsymbol{w}}^{\mathrm{T}}\phi(\boldsymbol{x}^{(n)}) + \hat{b} = \left(\sum_{i=1}^{m}\hat{\alpha}^{(i)}t^{(i)}\phi(\boldsymbol{x}^{(i)})\right)^{\mathrm{T}}\phi(\boldsymbol{x}^{(n)}) + \hat{b}$$

$$= \sum_{i=1}^{m}\hat{\alpha}^{(i)}t^{(i)}(\phi(\boldsymbol{x}^{(i)})^{\mathrm{T}}\phi(\boldsymbol{x}^{(n)})) + \hat{b}$$

$$= \sum_{\substack{i=1\\ \hat{\alpha}^{(i)}>0}}^{m}\hat{\alpha}^{(i)}t^{(i)}K(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(n)}) + \hat{b}$$

注意，因为仅对于支持向量才有α（i）≠0，所以预测时，计算新输入向量x（n）的点积，使用的仅仅是支持向量而不是全部训练实例。当然，你还需要使用同样的技巧来计算偏置项$\hat{b}$（见公式5-12）。

公式5-12：使用核技巧来计算偏置项

$$\hat{b} = \frac{1}{n_s}\sum_{\substack{i=1\\ \hat{\alpha}^{(i)}>0}}^{m}(t^{(i)} - \hat{\boldsymbol{w}}^{\mathrm{T}}\phi(\boldsymbol{x}^{(i)})) = \frac{1}{n_s}\sum_{\substack{i=1\\ \hat{\alpha}^{(i)}>0}}^{m}\left(t^{(i)} - \left(\sum_{j=1}^{m}\hat{\alpha}^{(j)}t^{(j)}\phi(\boldsymbol{x}^{(j)})\right)^{\mathrm{T}}\phi(\boldsymbol{x}^{(i)})\right)$$

$$= \frac{1}{n_s}\sum_{\substack{i=1\\ \hat{\alpha}^{(i)}>0}}^{m}\left(t^{(i)} - \sum_{\substack{j=1\\ \hat{\alpha}^{(j)}>0}}^{m}\hat{\alpha}^{(j)}t^{(j)}K(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})\right)$$

如果你现在开始觉得头痛，完全正常：这正是核技巧的副作用。

5.4.6　在线SVM

在结束本章之前，让我们快速看一下在线SVM分类器（回想一下，在线学习意味着增量学习，通常是随着新实例的到来而学习）。

对于线性SVM分类器，一种实现在线SVM分类器的方法是使用梯度下降（例如，使用SGDClassifier）来最小化源自原始问题的公式5-13中的成本函数。不幸的是，梯度下降比基于QP的方法收敛慢得多。

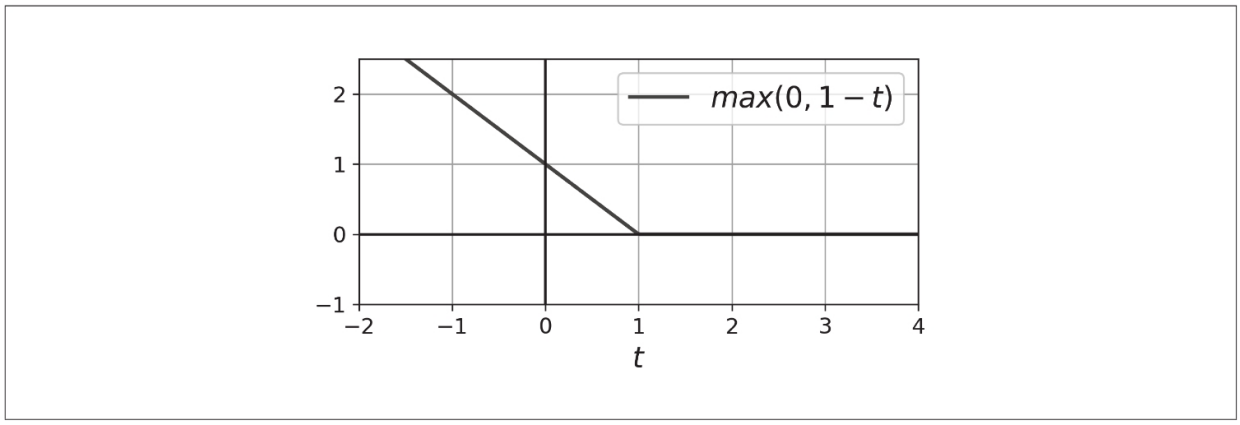
公式5-13：线性SVM分类器成本函数

$$J(\boldsymbol{w}, b) = \frac{1}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w} + C\sum_{i=1}^{m}\max(0, 1 - t^{(i)}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}^{(i)} + b))$$

成本函数中的第一项会推动模型得到一个较小的权重向量w，从而使间隔更大。第二项则计算全部的间隔违例。如果没有一个示例位于街道之上，并且都在街道正确的一边，那么这个实例的间隔违例为0；否则，该实例的违例大小与其到街道正确一边的距离成正比。所以将这个项最小化，能够保证模型使间隔违例尽可能小，也尽可能少。

Hinge损失函数

函数max（0，1-t）被称为hinge损失函数（如图所示）。当t≥1时，函数等于0。如果t<1，其导数（斜率）等于-1；如果t>1，则导数（斜率）为0；t=1时，函数不可导。但是，在t=0处可以使用任意次导数（即-1到0之间的任意值），你还是可以使用梯度下降，就跟Lasso回归一样。



在线SVM也可以实现核技巧，可参考“Incremental and Decremental SVM Learning”[5]，以及“Fast Kernel Classifiers with Online and Active Learning”[6]。但是这些核SVM是在Matlab和C++上实现的。对于大规模非线性问题，你可能需要使用神经网络（见本书第二部分）。

[1] 更一般而言，当存在n个特征时，决策函数为n维超平面，决策边界为（n-1）维超平面。
[2] Zeta（ζ）是希腊字母的第6个字母。
[3] 要了解有关二次规划的更多信息，可以先阅读Stephen Boyd和Lieven Vandenberghe的书Convex Optimization（剑桥大学出版社，2004年）或观看Richard Brown的一系列视频讲座。
[4] 目标函数是凸函数，不等式约束是连续可微和凸函数。
[5] Gert Cauwenberghs 和 Tomaso Poggio，“Incremental and Decremental Support Vector Machine Learning”，Proceedings of the 13th International Conference on Neural Information Processing Systems（2000）：388–394.
[6] Antoine Bordes et al.，“Fast Kernel Classifiers with Online and Active Learning”，Journal of Machine Learning Research 6（2005）：1579–1619.

5.5 练习题

1. 支持向量机的基本思想是什么？

2. 什么是支持向量？

3. 使用SVM时，对输入值进行缩放为什么重要？

4. SVM分类器在对实例进行分类时，会输出信心分数吗？概率呢？

5. 如果训练集有成百万个实例和几百个特征，你应该使用SVM原始问题还是对偶问题来训练模型？

6. 假设你用RBF核训练了一个SVM分类器，看起来似乎对训练集欠拟合，你应该提升还是降低γ（gamma）？C呢？

7. 如果使用现成二次规划求解器，你应该如何设置QP参数（H、f、A和b）来解决软间隔线性SVM分类器问题？

8. 在一个线性可分离数据集上训练LinearSVC。然后在同一数据集上训练SVC和SGDClassifier。看看你是否可以用它们产生大致相同的模型。

9. 在MNIST数据集上训练SVM分类器。由于SVM分类器是个二元分类器，所以你需要使用一对多来为10个数字进行分类。你可能还需要使用小型验证集来调整超参数以加快进度。最后看看达到的准确率是多少？

10. 在加州住房数据集上训练一个SVM回归模型。

附录A中提供了这些练习题的解答。

第6章　决策树

与SVM一样，决策树是通用的机器学习算法，可以执行分类和回归任务，甚至多输出任务。它们是功能强大的算法，能够拟合复杂的数据集。例如，在第2章中，你在加州房屋数据集中训练了DecisionTreeRegressor模型，使其完全拟合（实际上是过拟合）。

决策树也是随机森林的基本组成部分（见第7章），它们是当今最强大的机器学习算法之一。

在本章中，我们将从讨论如何使用决策树进行训练、可视化和做出预测开始。然后，我们将了解Scikit-Learn使用的CART训练算法，并将讨论如何对树进行正则化并将其用于回归任务。最后，我们将讨论决策树的一些局限性。

6.1 训练和可视化决策树

为了理解决策树，让我们建立一个决策树，然后看看它是如何做出预测的。以下代码在鸢尾花数据集上训练了一个DecisionTreeClassifier（见第4章）：

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

要将决策树可视化，首先，使用export_graphviz（）方法输出一个图形定义文件，命名为iris_tree.dot：

```
from sklearn.tree import export_graphviz

export_graphviz(
        tree_clf,
        out_file=image_path("iris_tree.dot"),
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```

然后，你可以使用Graphviz软件包中的dot命令行工具将此.dot文件转换为多种格式，例如PDF或PNG[1]。此命令行将.dot文件转换为.png图像文件：

```
$ dot -Tpng iris_tree.dot -o iris_tree.png
```

你的第一个决策树如图6-1所示。

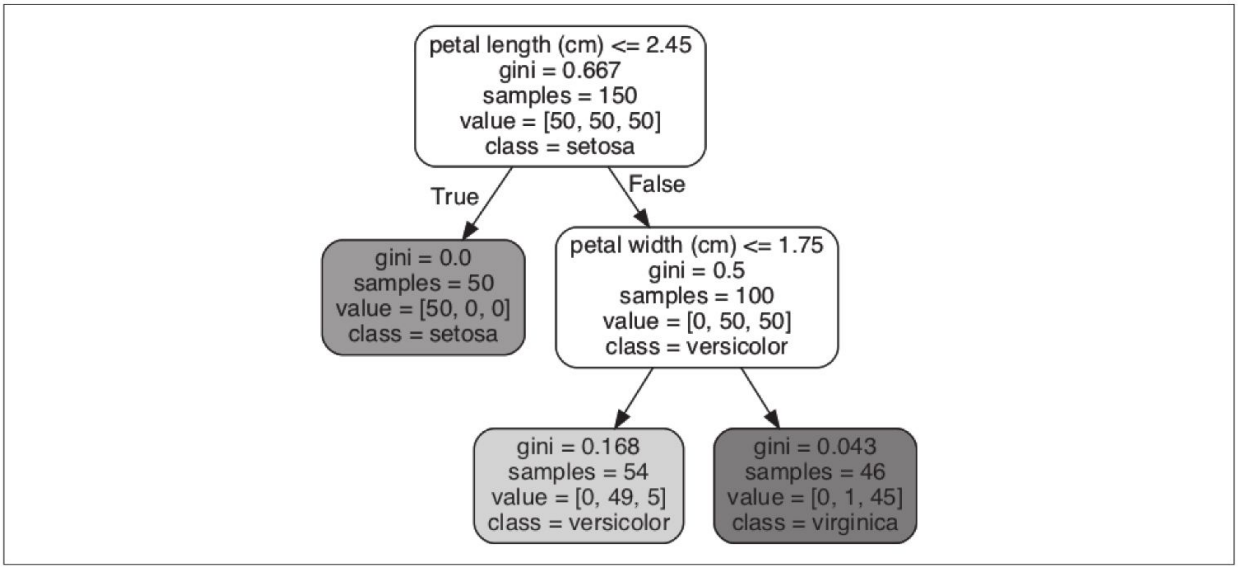


图6-1：鸢尾花决策树

[1] graphviz是一个开源图形可视化软件包，可从http://wwwgraphviz.org/获取。

6.2 做出预测

让我们看看图6-1中的树是如何进行预测的。假设你找到一朵鸢尾花，要对其进行分类。你从根节点开始（深度为0，在顶部）：该节点询问花的花瓣长度是否小于2.45cm。如果是，则向下移动到根的左子节点（深度1，左）。在这种情况下，它是一片叶子节点（即它没有任何子节点），因此它不会提出任何问题：只需查看该节点的预测类，然后决策树就可以预测花朵是山鸢尾花（class=setosa）。

现在假设你发现了另一朵花，这次花瓣的长度大于2.45cm，你必须向下移动到根的右子节点（深度1，右），该子节点不是叶子节点，因此该节点会问另一个问题：花瓣宽度是否小于1.75cm？如果是，则你的花朵很可能是变色鸢尾花（深度2，左）。如果不是，则可能是维吉尼亚鸢尾花（深度2，右）。就是这么简单。

 决策树的许多特质之一就是它们几乎不需要数据准备。实际上，它们根本不需要特征缩放或居中。

节点的samples属性统计它应用的训练实例数量。例如，有100个训练实例的花瓣长度大于2.45cm（深度1，右），其中54个花瓣宽度小于1.75cm（深度2，左）。节点的value属性说明了该节点上每个类别的训练实例数量。例如，右下节点应用在0个山鸢尾、1个变色鸢尾和45个维吉尼亚鸢尾实例上。最后，节点的gini属性衡量其不纯度（impurity）：如果应用的所有训练实例都属于同一个类别，那么节点就是“纯”的（gini=0）。例如，深度1左侧节点仅应用于山鸢尾花训练实例，所以它就是纯的，并且gini值为0。公式6-1说明了第i个节点的基尼系数Gi的计算方式。例如，深度2左侧节点，基尼系数等于$1-(0/54)^2-(49/54)^2-(5/54)^2\approx 0.168$。

公式6-1：基尼不纯度

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2$$

在此等式中：

- $p_{i,k}$是第i个节点中训练实例之间的k类实例的比率。

 Scikit-Learn使用的是CART算法，该算法仅生成二叉树：非叶节点永远只有两个子节点（即问题答案仅有是或否）。但是，其他算法（比如ID3生成的决策树），其节点可以拥有两个以上的子节点。

图6-2显示了决策树的决策边界。加粗直线表示根节点（深度0）的决策边界：花瓣长度=2.45cm。因为左侧区域是纯的（只有山鸢尾花），所以它不可再分。但是右侧区域是不纯的，所以深度1右侧的节点在花瓣宽度=1.75cm处（虚线所示）再次分裂。因为这里最大深度max_depth设置为2，所以决策树在此停止。但是如果你将max_depth设置为3，那么两个深度为2的节点将各自再产生一条决策边界（点线所示）。

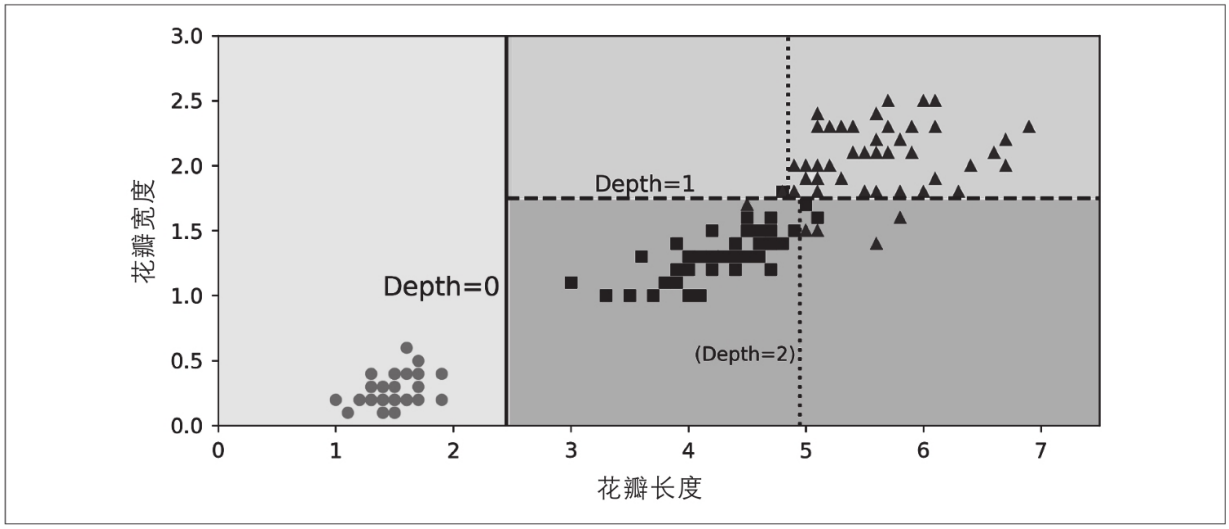


图6-2：决策树决策边界

模型解释：白盒子与黑盒子

决策树是直观的，其决策也易于解释。这种模型通常称为白盒模型。相反，正如我们将看到的，通常将随机森林或神经网络视为黑盒模型。它们做出了很好的预测，你可以轻松地检查它们为做出这些预测而执行的计算。但是，通常很难用简单的话语来解释为什么做出这样的预测。例如，如果神经网络说某个人出现在图片上，那么很难知道是什么因素促成了这一预测：该模型识别该人的眼睛、嘴、鼻子、鞋子，甚至他们坐的沙发？相反，决策树提供了很好的、简单的分类规则，如果需要的话，甚至可以手动应用这些规则（例如，用于花的分类）。

6.3 估计类概率

决策树同样可以估算某个实例属于特定类k的概率：首先，跟随决策树找到该实例的叶节点，然后返回该节点中类k的训练实例占比。例如，假设你发现一朵花，其花瓣长5cm，宽1.5cm。相应的叶节点为深度2左侧节点，因此决策树输出如下概率：山鸢尾花，0%（0/54）；变色鸢尾花，90.7%（49/54）；维吉尼亚鸢尾花，9.3%（5/54）。当然，如果你要求它预测类，那么它应该输出变色鸢尾花（类别1），因为它的概率最高。我们试一下：

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[0.        , 0.90740741, 0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

完美！注意，在图6-2的右下角矩形中，任意点的估计概率都是相同的，例如，如果花瓣长6cm，宽1.5cm（即使看起来很可能是维吉尼亚鸢尾花）。

6.4 CART训练算法

Scikit-Learn使用分类和回归树（Classification and Regression Tree，CART）算法来训练决策树（也称为“增长树”）。该算法的工作原理是：首先使用单个特征k和阈值t_k（例如，“花瓣长度”≤2.45cm”）将训练集分为两个子集。如何选择k和t_k？它搜索产生最纯子集（按其大小加权）的一对（k，t_k）。公式6-2给出了算法试图最小化的成本函数。

公式6-2：CART分类成本函数

$$J(k,t_k)=\frac{m_{\text{left}}}{m}G_{\text{left}}+\frac{m_{\text{right}}}{m}G_{\text{right}}$$

$$\text{其中}\begin{cases}G_{\text{left/right}}\text{测量左右子集的不纯度}\\ m_{\text{left/right}}\text{测量左右子集的实例数}\end{cases}$$

一旦CART算法成功地将训练集分为两部分，它就会使用相同的逻辑将子集进行分割，然后再分割子集，以此类推。一旦达到最大深度（由超参数max_depth定义），或者找不到可减少不纯度的分割，它将停止递归。其他一些超参数（稍后描述）可以控制其他一些停止条件（min_samples_split、min_samples_leaf、min_weight_fraction_leaf和max_leaf_nodes）。

 如你所见，CART是一种贪婪算法：从顶层开始搜索最优分裂，然后每层重复这个过程。几层分裂之后，它并不会检视这个分裂的不纯度是否为可能的最低值。贪婪算法通常会产生一个相当不错的解，但是不能保证是最优解。

而不幸的是，寻找最优树是一个已知的NP完全问题[1]：需要的时间是O（exp（m）），所以即使是很小的训练集，也相当棘手。这就是为什么我们必须接受一个“相当不错”的解。

[1] P是可以在多项式时间内解决的一组问题。NP是可以在多项式时间内验证其解的一组问题。NP难问题是可以在多项式时间内将任何NP问题减少的问题。一个NP完全问题是NP和NP难。一个主要的开放数学问题是P=NP是否相等。如果P≠NP（这似乎是可能的），那么不会找到针对任何NP完全问题的多项式算法（也许在量子计算机上除外）。

6.5 计算复杂度

进行预测需要从根到叶遍历决策树。决策树通常是近似平衡的，因此遍历决策树需要经过大约O（$\log_2$（m））个节点（注：$\log_2$是二进制对数。它等于$\log_2$（m）=log（m）/log（2）。）。由于每个节点仅需要检查一个特征值，因此总体预测复杂度为O（$\log_2$（m））。与特征数量无关。因此，即使处理大训练集，预测也非常快。

训练算法比较每个节点上所有样本上的所有特征（如果设置了max_features，则更少）。比较每个节点上所有样本的所有特征会导致训练复杂度为O（n×m $\log_2$（m））。对于小训练集（少于几千个实例），Scikit-Learn可以通过对数据进行预排序（设置presort=True）来加快训练速度，但是这样做会大大降低大训练集的训练速度。

6.6 基尼不纯度或熵

默认使用的是基尼不纯度来进行测量，但是，你可以将超参数criterion设置为"entropy"来选择熵作为不纯度的测量方式。熵的概念源于热力学，是一种分子混乱程度的度量：如果分子保持静止和良序，则熵接近于零。后来这个概念推广到各个领域，其中包括香农的信息理论，它衡量的是一条信息的平均信息内容[1]：如果所有的信息都相同，则熵为零。在机器学习中，它也经常被用作一种不纯度的测量方式：如果数据集中仅包含一个类别的实例，其熵为零。公式6-3显示了第i个节点的熵值的计算方式。例如，图6-1中深度2左侧节点的熵值等于 $-\frac{49}{54}\log_2\left(\frac{49}{54}\right)-\frac{5}{54}\log_2\left(\frac{5}{54}\right)\approx 0.445$ 。

公式6-3：熵

$$H_i = -\sum_{\substack{k=1 \\ p_{i,k}\neq 0}}^{n} p_{i,k}\log_2(p_{i,k})$$

那么到底应该使用基尼不纯度还是熵呢？其实，大多数情况下，它们并没有什么大的不同，产生的树都很相似。基尼不纯度的计算速度略微快一些，所以它是个不错的默认选择。它们的不同在于，基尼不纯度倾向于从树枝中分裂出最常见的类别，而熵则倾向于生产更平衡的树[2]。

[1] 熵的减少通常称为信息增益。

[2] 有关更多详细信息，请参见Sebastian Raschka的有趣分析（https://homl.info/19）。

6.7 正则化超参数

决策树极少对训练数据做出假设（比如线性模型就正好相反，它显然假设数据是线性的）。如果不加以限制，树的结构将跟随训练集变化，严密拟合，并且很可能过拟合。这种模型通常被称为非参数模型，这不是说它不包含任何参数（事实上它通常有很多参数），而是指在训练之前没有确定参数的数量，导致模型结构自由而紧密地贴近数据。相反，参数模型（比如线性模型）则有预先设定好的一部分参数，因此其自由度受限，从而降低了过拟合的风险（但是增加了欠拟合的风险）。

为避免过拟合，需要在训练过程中降低决策树的自由度。现在你应该知道，这个过程被称为正则化。正则化超参数的选择取决于使用的模型，但是通常来说，至少可以限制决策树的最大深度。在Scikit-Learn中，这由超参数max_depth控制（默认值为None，意味着无限制）。减小max_depth可使模型正则化，从而降低过拟合的风险。

DecisionTreeClassifier类还有一些其他的参数，同样可以限制决策树的形状：min_samples_split（分裂前节点必须有的最小样本数）、min_samples_leaf（叶节点必须有的最小样本数量）、min_weight_fraction_leaf（与min_samples_leaf一样，但表现为加权实例总数的占比）、max_leaf_nodes（最大叶节点数量），以及max_features（分裂每个节点评估的最大特征数量）。增大超参数min_*或减小max_*将使模型正则化。

 还可以先不加约束地训练模型，然后再对不必要的节点进行剪枝（删除）。如果一个节点的子节点全部为叶节点，则该节点可被认为不必要，除非它所表示的纯度提升有重要的统计意义。标准统计测试（比如 χ^2 测试）用来估算“提升纯粹是出于偶然”（被称为零假设）的概率。如果这个概率（称之为p值）高于一个给定阈值（通常是5%，

由超参数控制），那么这个节点可被认为不必要，其子节点可被删除。直到所有不必要的节点都被删除，剪枝过程结束。

图6-3显示了在卫星数据集上训练的两个决策树（在第5章中介绍）。左侧使用默认的超参数（即无限制）训练决策树，右侧使用min_samples_leaf=4进行训练。显然，左边的模型过拟合，右边的模型可能会更好地泛化。

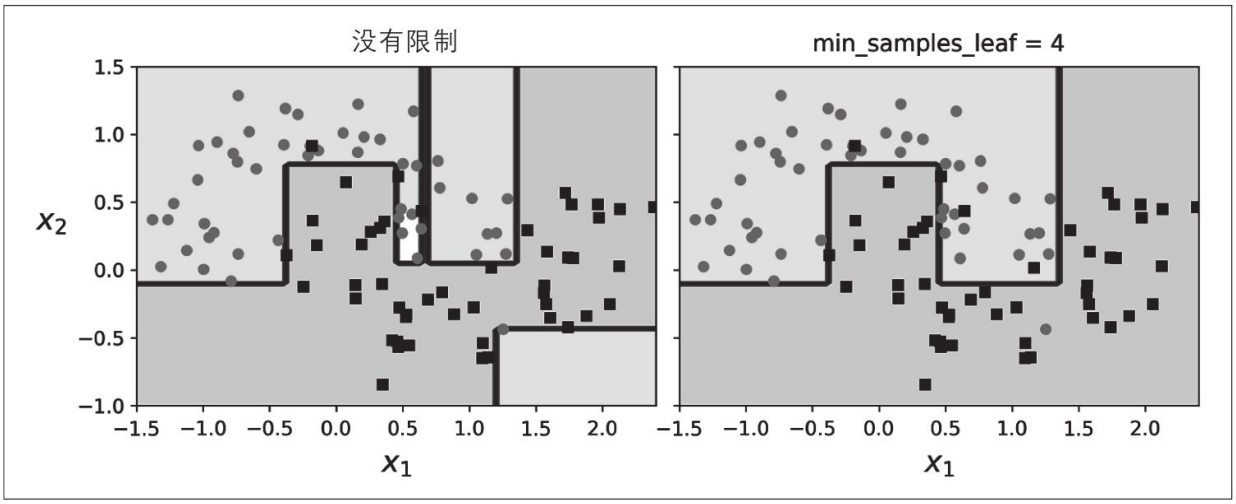


图6-3：使用min_samples_leaf进行正则化

6.8 回归

决策树还能够执行回归任务。让我们使用Scikit-Learn的DecisionTreeRegressor类构建一个回归树，并用max_depth=2在一个有噪声的二次数据集上对其进行训练：

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

生成的树如图6-4所示。

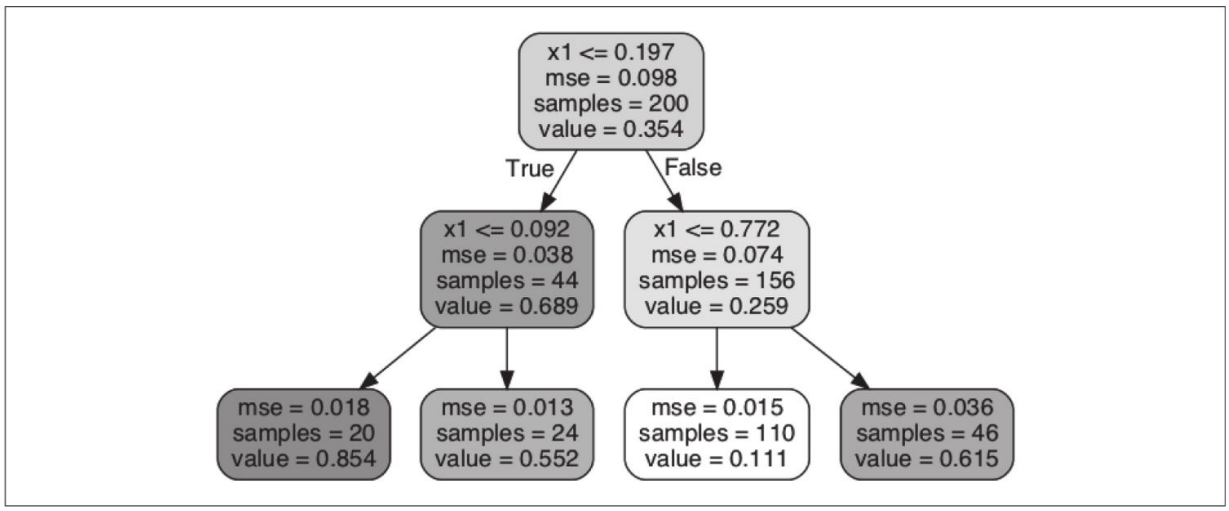


图6-4：回归决策树

这棵树看起来与之前建立的分类树很相似。主要差别在于，每个节点上不再预测一个类别而是预测一个值。例如，如果你想要对一个x1=0.6的新实例进行预测，那么从根节点开始遍历，最后到达预测value=0.111的叶节点。这个预测结果其实就是与这个叶节点关联的110个实例的平均目标值。在这110个实例上，预测产生的均方误差（MSE）等于0.015。

图6-5的左侧显示了该模型的预测。如果设置max_depth=3，将得到如图6-5右侧所示的预测。注意看，每个区域的预测值永远等于该区域内实例的目标平均值。算法分裂每个区域的方法就是使最多的训练实例尽可能接近这个预测值。

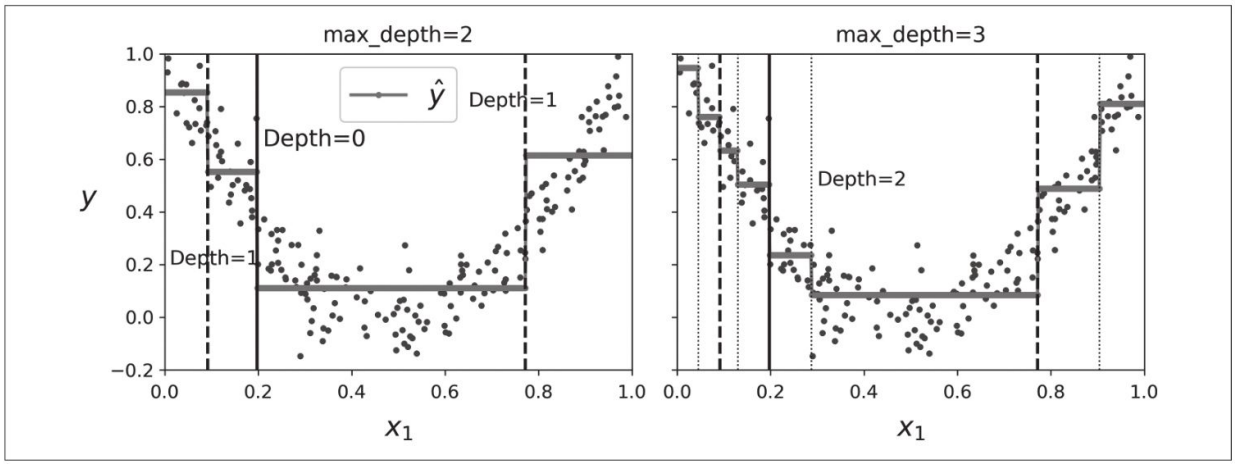


图6-5：两种决策树回归模型的预测

CART算法的工作原理与以前的方法大致相同，不同之处在于，它不再尝试以最小化不纯度的方式来拆分训练集，而是以最小化MSE的方式来拆分训练集。公式6-4给出了算法试图最小化的成本函数。

公式6-4：CART回归成本函数

$$J(k, t_k) = \frac{m_{\text{left}}}{m} \text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m} \text{MSE}_{\text{right}}$$

$$\text{其中} \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} (\hat{y}_{\text{node}} - y^{(i)})^2 \\ \hat{y}_{\text{node}} = \dfrac{\sum_{i \in \text{node}} y^{(i)}}{m_{\text{node}}} \end{cases}$$

就像分类任务一样，决策树在处理回归任务时容易过拟合。如果不进行任何正则化（如使用默认的超参数），你会得到图6-6左侧的预测。这些预测显然非常过拟合训练集。只需设置min_samples_leaf=10就可以得到一个更合理的模型，如图6-6右侧所示。

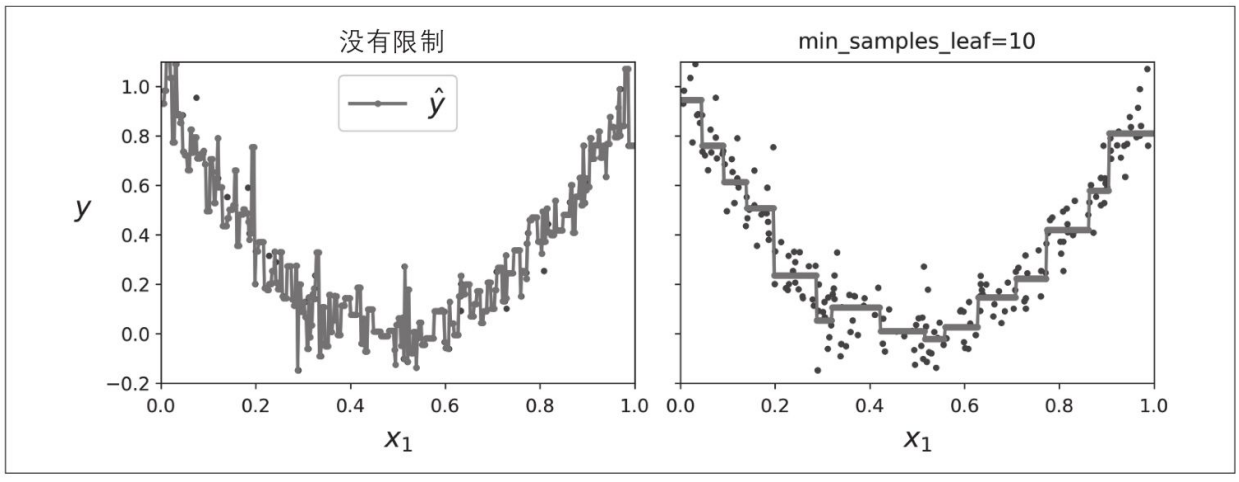


图6-6：正则化一个回归决策树

6.9 不稳定性

希望到目前为止，你已经确信决策树有很多用处：它们易于理解和解释、易于使用、用途广泛且功能强大。但是，它们确实有一些局限性。首先，你可能已经注意到，决策树喜欢正交的决策边界（所有分割都垂直于轴），这使它们对训练集旋转敏感。例如，图6-7显示了一个简单的线性可分离数据集：在左侧，决策树可以轻松地对其进行拆分，而在右侧，将数据集旋转45度后，决策边界看起来复杂了（没有必要）。尽管两个决策树都非常拟合训练集，但右侧的模型可能无法很好地泛化。限制此问题的一种方法是使用主成分分析（见第8章），这通常会使训练数据的方向更好。

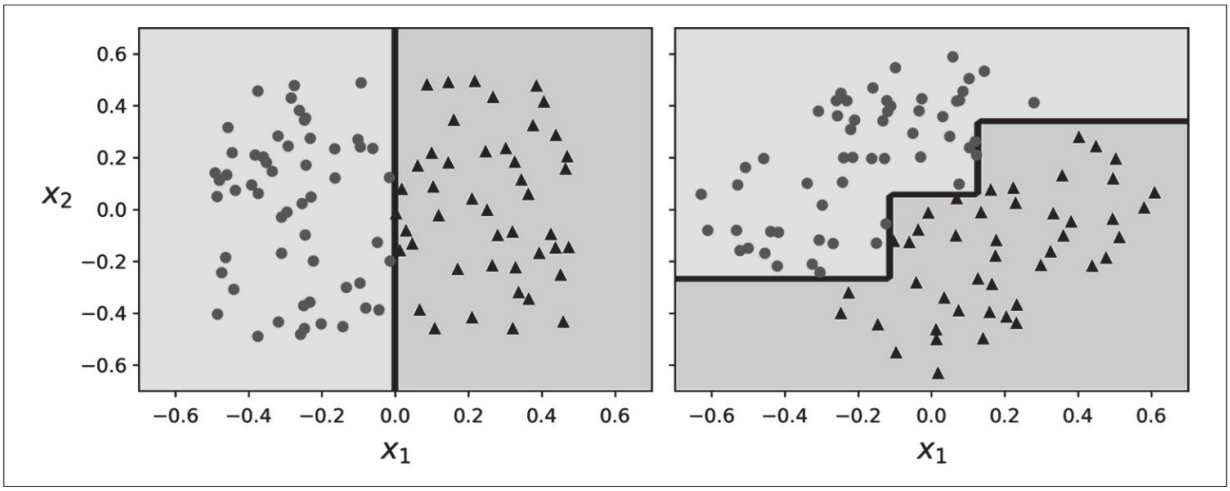


图6-7：对训练集旋转敏感

更概括地说，决策树的主要问题是它们对训练数据中的小变化非常敏感。例如，如果你从鸢尾花数据集中移除花瓣最宽的变色鸢尾花（花瓣长4.8cm，宽1.8cm），然后重新训练一个决策树，你可能得到如图6-8所示的模型。这跟之前图6-2的决策树看起来截然不同。事实上，由于Scikit-Learn所使用的算法是随机的[1]，即使是在相同的训练数据上，你也可能得到完全不同的模型（除非你对超参数random_state进行设置）。

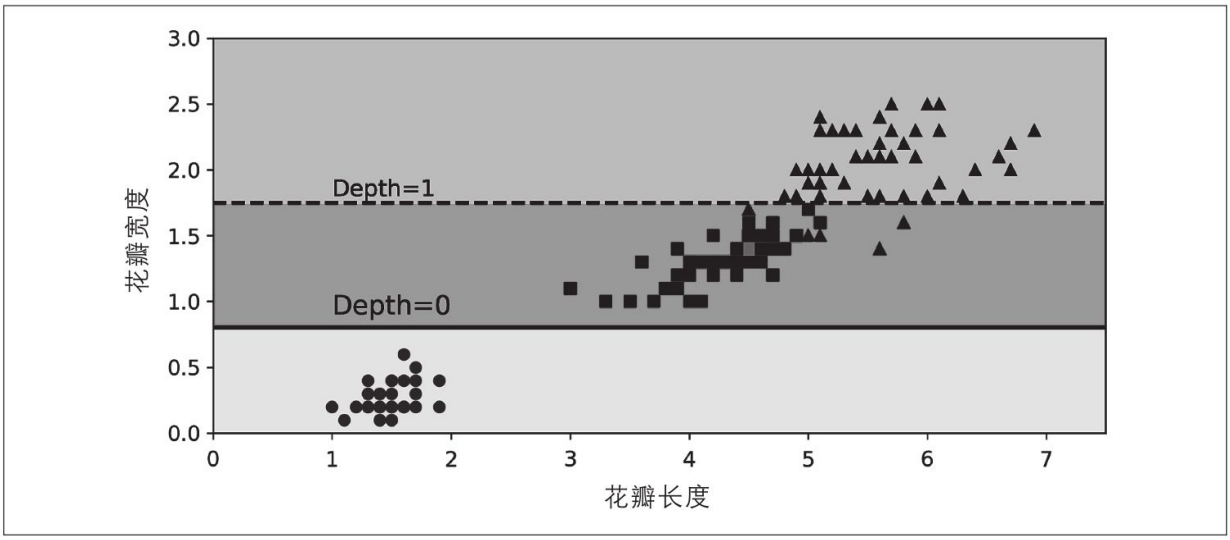


图6-8：对训练集细节敏感

随机森林可以通过对许多树进行平均预测来限制这种不稳定性，正如我们将在第7章中看到的那样。

[1] 每个节点随机选择特征集进行评估。

6.10 练习题

1.如果训练集有100万个实例，训练决策树（无约束）大致的深度是多少？

2.通常来说，子节点的基尼不纯度是高于还是低于其父节点？是通常更高/更低？还是永远更高/更低？

3.如果决策树过拟合训练集，减少max_depth是否为一个好主意？

4.如果决策树对训练集欠拟合，尝试缩放输入特征是否为一个好主意？

5.如果在包含100万个实例的训练集上训练决策树需要一个小时，那么在包含1000万个实例的训练集上训练决策树，大概需要多长时间？

6.如果训练集包含10万个实例，设置presort=True可以加快训练吗？

7.为卫星数据集训练并微调一个决策树。

a.使用make_moons（n_samples=10000，noise=0.4）生成一个卫星数据集。

b.使用train_test_split（）拆分训练集和测试集。

c.使用交叉验证的网格搜索（在GridSearchCV的帮助下）为DecisionTreeClassifier找到适合的超参数。提示：尝试max_leaf_nodes的多种值。

d.使用超参数对整个训练集进行训练，并测量模型在测试集上的性能。你应该得到约85%～87%的准确率。

8.按照以下步骤种植森林。

a.继续之前的练习，生产1000个训练集子集，每个子集包含随机挑选的100个实例。提示：使用Scikit-Learn的ShuffleSplit来实现。

b.使用前面得到的最佳超参数值，在每个子集上训练一个决策树。在测试集上评估这1000个决策树。因为训练集更小，所以这些决策树的表现可能比第一个决策树要差一些，只能达到约80%的准确率。

c.见证奇迹的时刻到了。对于每个测试集实例，生成1000个决策树的预测，然后仅保留次数最频繁的预测（可以使用SciPy的mode（）函数）。这样你在测试集上可获得大多数投票的预测结果。

d.评估测试集上的这些预测，你得到的准确率应该比第一个模型更高（高出0.5%～1.5%）。恭喜，你已经训练出了一个随机森林分类器！

附录A中提供了这些练习题的解答。

第7章　集成学习和随机森林

如果你随机向几千个人询问一个复杂问题，然后汇总他们的回答。在许多情况下，你会发现，这个汇总的回答比专家的回答还要好，这被称为群体智慧。同样，如果你聚合一组预测器（比如分类器或回归器）的预测，得到的预测结果也比最好的单个预测器要好。这样的一组预测器称为集成，所以这种技术也被称为集成学习，而一个集成学习算法则被称为集成方法。

例如，你可以训练一组决策树分类器，每一棵树都基于训练集不同的随机子集进行训练。做出预测时，你只需要获得所有树各自的预测，然后给出得票最多的类别作为预测结果（见第6章练习题8）。这样一组决策树的集成被称为随机森林，尽管很简单，但它是迄今可用的最强大的机器学习算法之一。

此外，正如我们在第2章讨论过的，在项目快要结束时，你可能已经构建好了一些不错的预测器，这时就可以通过集成方法将它们组合成一个更强的预测器。事实上，在机器学习竞赛中获胜的解决方案通常都涉及多种集成方法（最知名的是Nerflix大奖赛）。

本章我们将探讨最流行的几种集成方法，包括bagging、boosting、stacking等，也将探索随机森林。

7.1 投票分类器

假设你已经训练好了一些分类器，每个分类器的准确率约为80%。大概包括一个逻辑回归分类器、一个SVM分类器、一个随机森林分类器、一个K-近邻分类器，或许还有更多（见图7-1）。

这时，要创建出一个更好的分类器，最简单的办法就是聚合每个分类器的预测，然后将得票最多的结果作为预测类别。这种大多数投票分类器被称为硬投票分类器（见图7-2）。

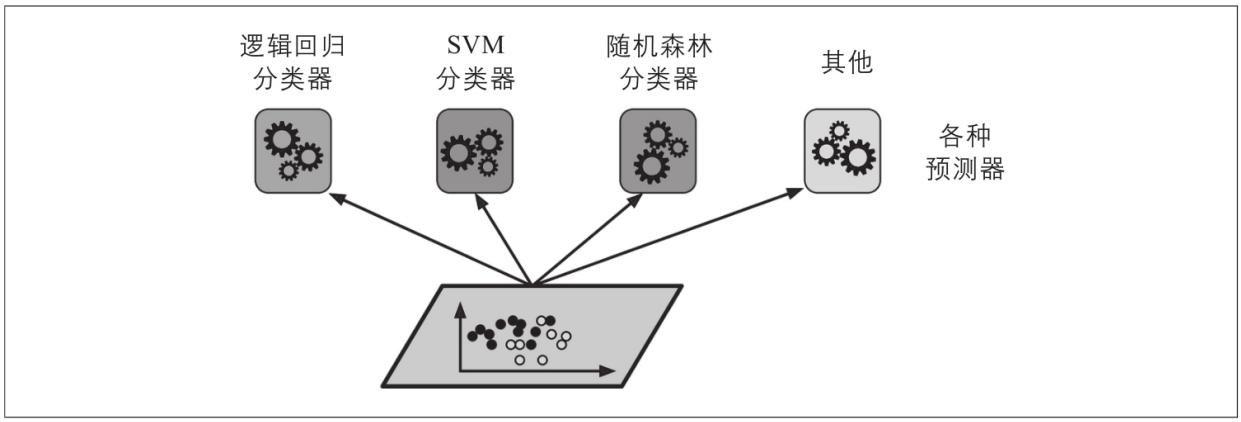


图7-1：训练多种分类器

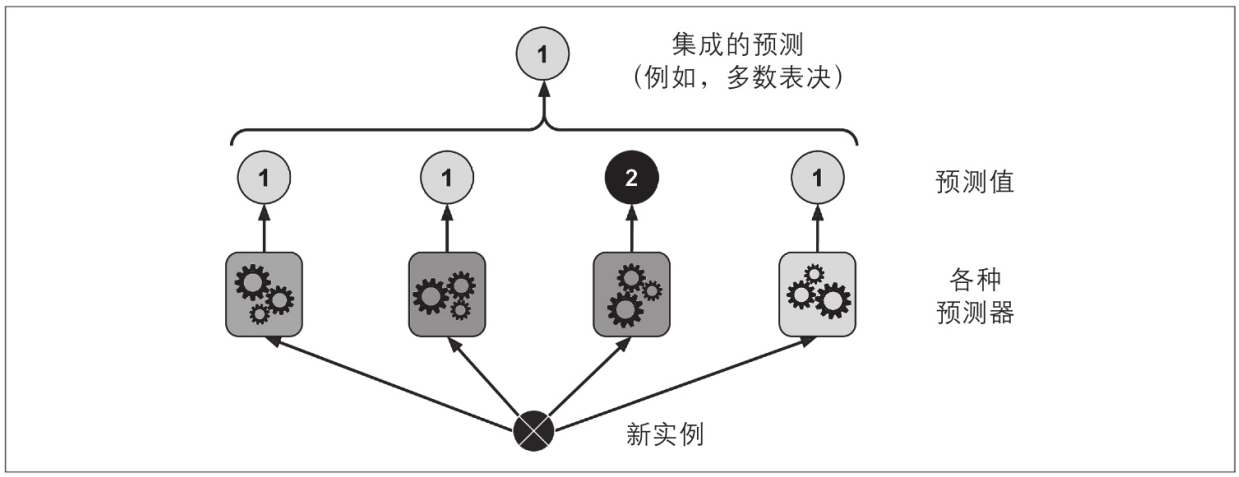


图7-2：硬投票分类器预测

你会多少有点惊讶地发现，这个投票法分类器的准确率通常比集成中最好的分类器还要高。事实上，即使每个分类器都是弱学习器（意味着它仅比随机猜测好一点），通过集成依然可以实现一个强学习器（高准确率），只要有足够大数量并且足够多种类的弱学习器即可。

这怎么可能呢？下面这个类比可以帮助你掀开这层神秘面纱。假设你有一个略微偏倚的硬币，它有51%的可能正面数字朝上，49%的可能背面花朝上。如果你掷1000次，你大致会得到差不多510次数字和490次花，所以正面是大多数。而如果你做数学题，你会发现，“在1000次投掷后，大多数为正面朝上”的概率接近75%。投掷硬币的次数越多，这个概率越高（例如，投掷10 000次后，这个概率攀升至97%）。这是因为大数定理导致的：随着你不断投掷硬币，正面朝上的比例越来越接近于正面的概率（51%）。图7-3显示了10条偏倚硬币的投掷结果。可以看出随着投掷次数的增加，正面的比例逐渐接近51%，最终所有10条线全都接近51%，并且始终位于50%以上。

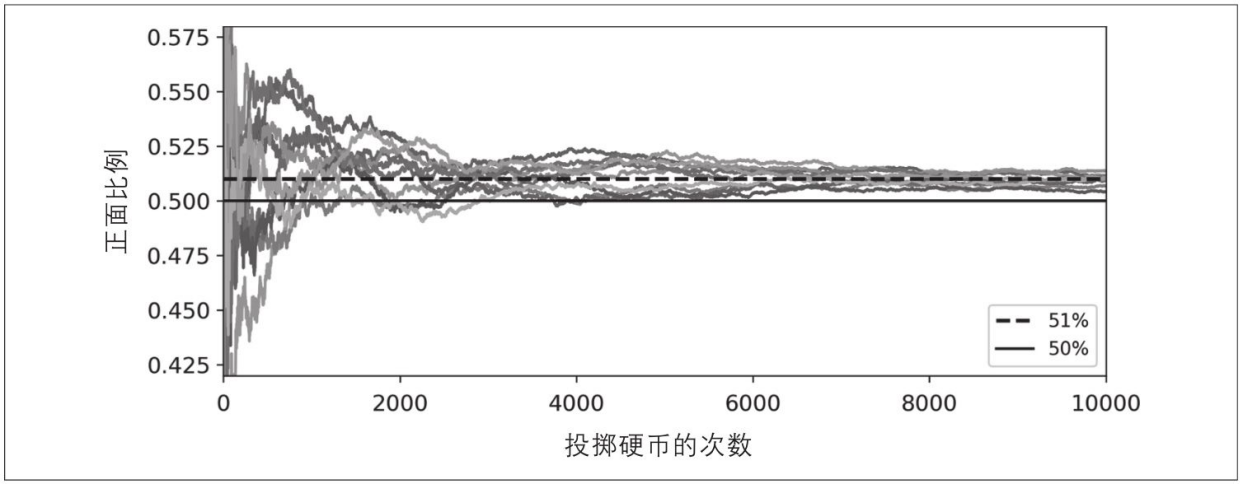


图7-3：大数定理

同样，假设你创建了一个包含1000个分类器的集成，每个分类器都只有51%的概率是正确的（几乎不比随机猜测强多少）。如果你以大多数投票的类别作为预测结果，可以期待的准确率高达75%。但是，这基于的前提是所有的分类器都是完全独立的，彼此的错误毫不相关。显然这是不可能的，因为它们都是在相同的数据上训练的，很可能会犯相同

的错误，所以也会有很多次大多数投给了错误的类别，导致集成的准确率有所降低。

 当预测器尽可能互相独立时，集成方法的效果最优。获得多种分类器的方法之一就是使用不同的算法进行训练。这会增加它们犯不同类型错误的机会，从而提升集成的准确率。

下面的代码用Scikit-Learn创建并训练一个投票分类器，由三种不同的分类器组成（训练集是卫星数据集，见第5章）：

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

我们来看一下测试集上每个分类器的精度：

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.888
VotingClassifier 0.904
```

投票分类器略胜于所有单个分类器。

如果所有分类器都能够估算出类别的概率（即有predict_proba（）方法），那么你可以将概率在所有单个分类器上平均，然后让Scikit-Learn给出平均概率最高的类别作为预测。这被称为软投票法。通常来说，它比硬投票法的表现更优，因为它给予那些高度自信的投票更高的权重。而所有你需要做的就是用voting="soft"代替voting="hard"，并确保所有分类器都可以估算出概率。默认情况下，SVC类是不行的，所以你需要将其超参数probability设置为True（这会导致SVC使用交叉验证来估算类别概率，减慢训练速度，并会添加predict_proba（）方法）。如果修改上面代码为使用软投票，你会发现投票分类器的准确率达到91.2%以上！

7.2 bagging和pasting

前面提到，获得不同种类分类器的方法之一是使用不同的训练算法。还有另一种方法是每个预测器使用的算法相同，但是在不同的训练集随机子集上进行训练。采样时如果将样本放回，这种方法叫作bagging[1]（bootstrap aggregating[2]的缩写，也叫自举汇聚法）。采样时样本不放回，这种方法则叫作pasting[3]。

换句话说，bagging和pasting都允许训练实例在多个预测器中被多次采样，但是只有bagging允许训练实例被同一个预测器多次采样。采样过程和训练过程如图7-4所示。

一旦预测器训练完成，集成就可以通过简单地聚合所有预测器的预测来对新实例做出预测。聚合函数通常是统计法（即最多数的预测与硬投票分类器一样）用于分类，或是平均法用于回归。每个预测器单独的偏差都高于在原始训练集上训练的偏差，但是通过聚合，同时降低了偏差和方差[4]。总体来说，最终结果是，与直接在原始训练集上训练的单个预测器相比，集成的偏差相近，但是方差更低。

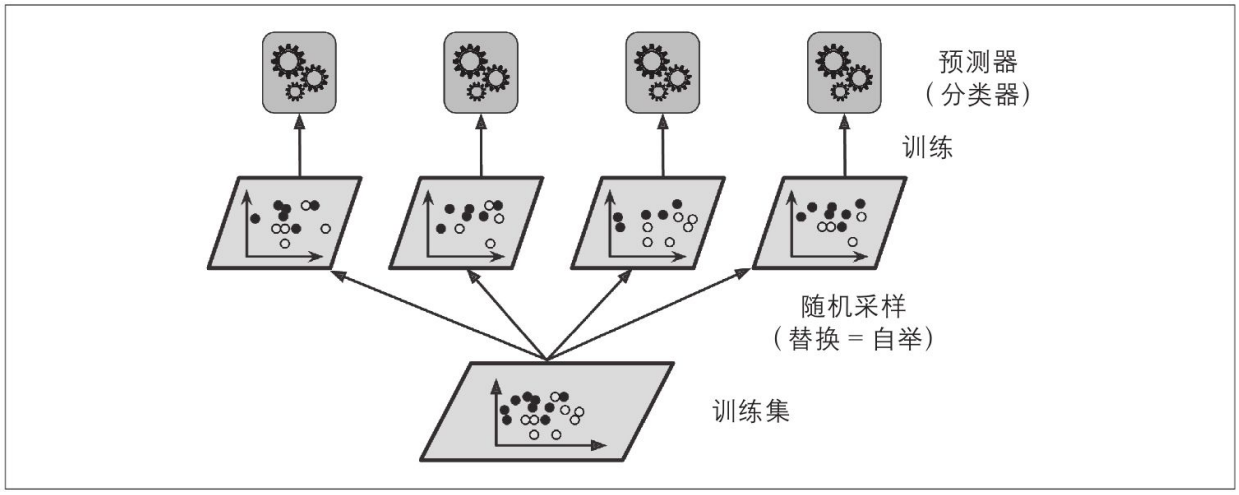


图7-4：bagging和pasting，在训练集的不同随机样本上训练几个预测器

如图7-4所示，你可以通过不同的CPU内核甚至不同的服务器并行地训练预测器。类似地，预测也可以并行。这正是bagging和pasting方法如此流行的原因之一，它们非常易于扩展。

7.2.1 Scikit-Learn中的bagging和pasting

Scikit-Learn提供了一个简单的API，可用BaggingClassifier类进行bagging和pasting（或BaggingRegressor用于回归）。以下代码训练了一个包含500个决策树分类器[5]的集成，每次从训练集中随机采样100个训练实例进行训练，然后放回（这是一个bagging的示例，如果你想使用pasting，只需要设置bootstrap=False即可）。参数n_jobs用来指示Scikit-Learn用多少CPU内核进行训练和预测（-1表示让Scikit-Learn使用所有可用内核）：

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

 如果基本分类器可以估计类别概率（如果它具有predict_proba（）方法），则BaggingClassifier自动执行软投票而不是硬投票，在决策树分类器中就是这种情况。

图7-5比较了两种决策边界，一种是单个决策树，一种是由500个决策树组成的bagging集成（来自前面的代码），二者均在卫星数据集上训练完成。可以看出，集成预测的泛化效果很可能会比单独的决策树要好一些：二者偏差相近，但是集成的方差更小（两边训练集上的错误数量差不多，但是集成的决策边界更规则）。

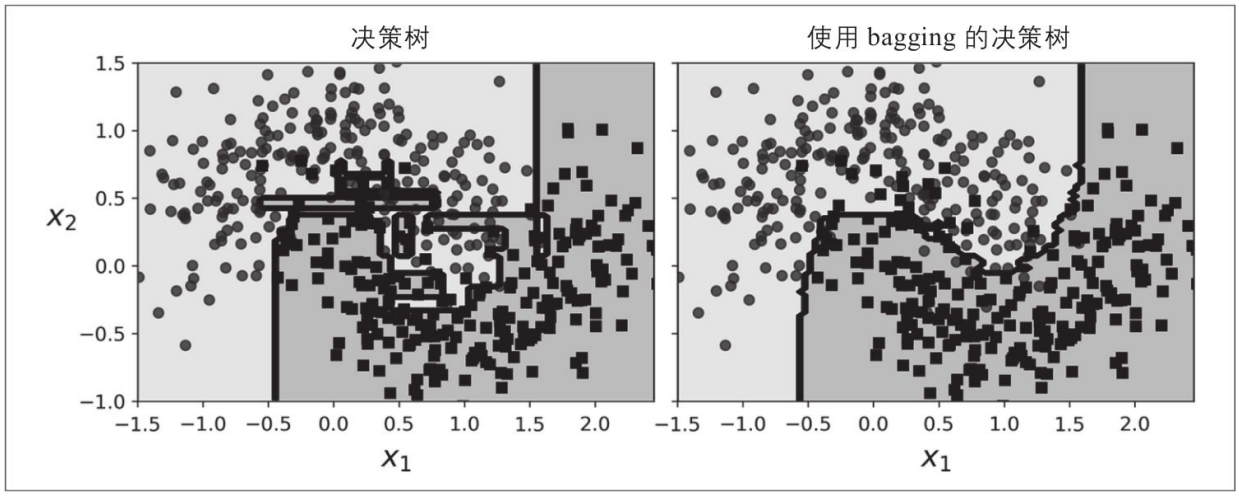


图7-5：单个决策树（左）与500个决策树组成的bagging集成（右）

由于自举法给每个预测器的训练子集引入了更高的多样性，所以最后bagging比pasting的偏差略高，但这也意味着预测器之间的关联度更低，所以集成的方差降低。总之，bagging生成的模型通常更好，这也就是为什么它更受欢迎。但是，如果你有充足的时间和CPU资源，可以使用交叉验证来对bagging和pasting的结果进行评估，再做出最合适的选择。

7.2.2 包外评估

对于任意给定的预测器，使用bagging，有些实例可能会被采样多次，而有些实例则可能根本不被采样。BaggingClassifier默认采样m个训练实例，然后放回样本（bootstrap=True），m是训练集的大小。这意味着对每个预测器来说，平均只对63%的训练实例进行采样[6]。剩余37%未被采样的训练实例称为包外（oob）实例。注意，对所有预测器来说，这是不一样的37%。

由于预测器在训练过程中从未看到oob实例，因此可以在这些实例上进行评估，而无须单独的验证集。你可以通过平均每个预测器的oob评估来评估整体。

在Scikit-Learn中，创建BaggingClassifier时，设置oob_score=True就可以请求在训练结束后自动进行包外评估。下面的代码演示了这一点。通过变量oob_score_可以得到最终的评估分数：

```
>>> bag_clf = BaggingClassifier(
...     DecisionTreeClassifier(), n_estimators=500,
...     bootstrap=True, n_jobs=-1, oob_score=True)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.90133333333333332
```

根据此oob评估，此BaggingClassifier能在测试集上达到约90.1%的准确率。让我们验证一下：

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.91200000000000003
```

我们在测试集上获得91.2%的准确率——足够接近！

每个训练实例的包外决策函数也可以通过变量oob_decision_function_获得。本例中（基本预测器有predict_proba（）方法），决策函数返回的是每个实例的类别概率。例如，包外评估估计，第二个训练实例有68.25%的概率属于正类（以及31.75%的概率属于负类）：

```
>>> bag_clf.oob_decision_function_
array([[0.31746032, 0.68253968],
       [0.34117647, 0.65882353],
       [1.        , 0.        ],
       ...
       [1.        , 0.        ],
       [0.03108808, 0.96891192],
       [0.57291667, 0.42708333]])
```

[1] Leo Breiman，“Bagging Predictors”，Machine Learning 24，no.2（1996）：123－140.
[2] 在统计信息中，有替换地进行重采样称为自举。
[3] Leo Breiman，“Pasting Small Votes for Classification in Large Databases and On-Line”，Machine Learning 36，no.1－2（1999）：85－103。
[4] 第4章介绍了偏差和方差。
[5] 可以将max_samples设置为介于0.0和1.0之间的浮点数，在这种情况下，要采样的最大实例数等于训练集的大小乘以max_samples。
[6] 随着m的增长，该比率接近1－exp（-1）≈63.212%。

7.3 随机补丁和随机子空间

BaggingClassifier类也支持对特征进行采样。采样由两个超参数控制：max_features和bootstrap_features。它们的工作方式与max_samples和bootstrap相同，但用于特征采样而不是实例采样。因此，每个预测器将用输入特征的随机子集进行训练。

这对于处理高维输入（例如图像）特别有用。对训练实例和特征都进行抽样，这称为随机补丁方法[1]。而保留所有训练实例（即bootstrap=False并且max_samples=1.0）但是对特征进行抽样（即bootstrap_features=True并且/或max_features<1.0），这被称为随机子空间法[2]。

对特征抽样给预测器带来更大的多样性，所以以略高一点的偏差换取了更低的方差。

[1] Gilles Louppe 和 Pierre Geurts，“Ensembles on Random Patches”，Lecture Notes in Computer Science 7523（2012）：346–361.

[2] Tin Kam Ho，“The Random Subspace Method for Constructing Decision Forests”，IEEE Transactions on Pattern Analysis and Machine Intelligence 20，no.8（1998）：832–844.

7.4 随机森林

前面已经提到，随机森林[1]是决策树的集成，通常用bagging（有时也可能是pasting）方法训练，训练集大小通过max_samples来设置。除了先构建一个BaggingClassifier然后将其传输到DecisionTreeClassifier，还有一种方法就是使用RandomForestClassifier类，这种方法更方便，对决策树更优化[2]（同样，对于回归任务也有一个RandomForestRegressor类）。以下代码使用所有可用的CPU内核，训练了一个拥有500棵树的随机森林分类器（每棵树限制为最多16个叶节点）：

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

除少数例外，RandomForestClassifier具有DecisionTreeClassifier的所有超参数（以控制树的生长方式），以及BaggingClassifier的所有超参数来控制集成本身[3]。

随机森林在树的生长上引入了更多的随机性：分裂节点时不再是搜索最好的特征（见第6章），而是在一个随机生成的特征子集里搜索最好的特征。这导致决策树具有更大的多样性，（再一次）用更高的偏差换取更低的方差，总之，还是产生了一个整体性能更优的模型。下面的BaggingClassifier大致与前面的RandomForestClassifier相同：

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

7.4.1 极端随机树

如前所述，在随机森林里单棵树的生长过程中，每个节点在分裂时仅考虑到了一个随机子集所包含的特征。如果我们对每个特征使用随机阈值，而不是搜索得出的最佳阈值（如常规决策树），则可能让决策树生长得更加随机。

这种极端随机的决策树组成的森林称为极端随机树集成[4]（或简称Extra-Trees）。同样，它也是以更高的偏差换取了更低的方差。极端随机树训练起来比常规随机森林要快很多，因为在每个节点上找到每个特征的最佳阈值是决策树生长中最耗时的任务之一。

使用Scikit-Learn的ExtraTreesClassifier类可以创建一个极端随机树分类器。它的API与RandomForestClassifier类相同。同理，ExtraTreesRegressor类与RandomForestRegressor类的API也相同。

 通常来说，很难预先知道一个RandomForestClassifier类是否会比一个ExtraTreesClassifier类更好或是更差。唯一的方法是两种都尝试一遍，然后使用交叉验证（还需要使用网格搜索调整超参数）进行比较。

7.4.2 特征重要性

随机森林的另一个好特性是它们使测量每个特征的相对重要性变得容易。Scikit-Learn通过查看使用该特征的树节点平均（在森林中的所有树上）减少不纯度的程度来衡量该特征的重要性。更准确地说，它是一个加权平均值，其中每个节点的权重等于与其关联的训练样本的数量（见第6章）。

Scikit-Learn会在训练后为每个特征自动计算该分数，然后对结果进行缩放以使所有重要性的总和等于1。你可以使用feature_importances_变量来访问结果。例如，以下代码在鸢尾花数据集上训练了RandomForestClassifier（在第4章中介绍），并输出每个特征的重要性。看起来最重要的特征是花瓣长度（44%）和宽度（42%），而花萼的长度和宽度则相对不那么重要（分别是11%和2%）：

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

同样，如果在MNIST数据集上训练随机森林分类器（在第3章中介绍）并绘制每个像素的重要性，则会得到如图7-6所示的图像。

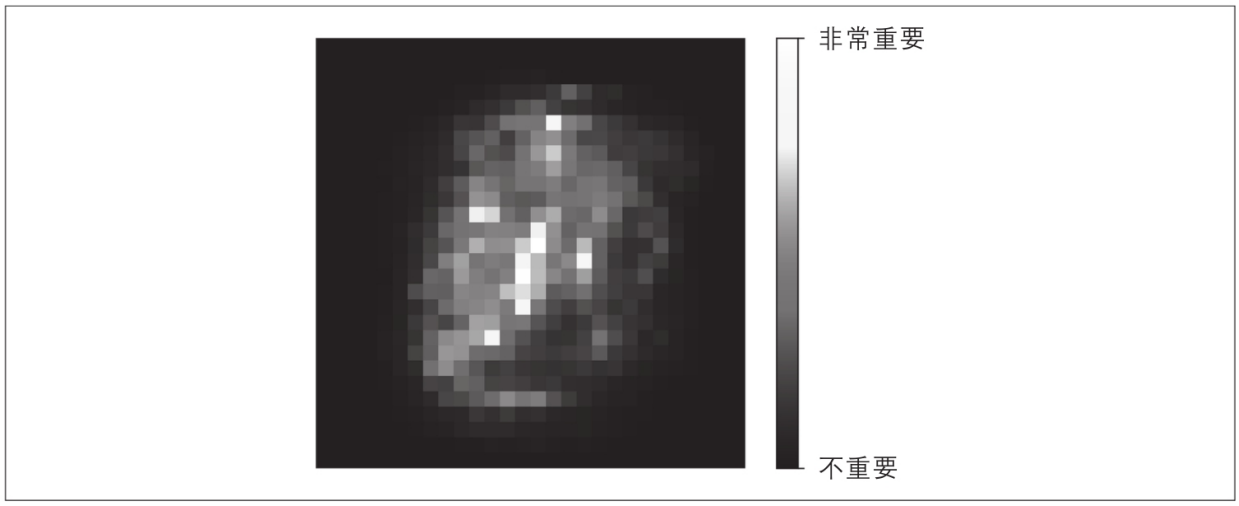


图7-6：MNIST像素的重要性（根据随机森林分类器）

随机森林非常便于你快速了解哪些特征是真正重要的，特别是在需要执行特性选择时。

[1] Tin Kam Ho，“Random Decision Forests”，Proceedings of the Third International Conference on Document Analysis and Recognition 1（1995）：278.
[2] 如果你想要对决策树之外的东西进行装袋（bag），BaggingClassifier还是有用的。
[3] 有几个值得注意的例外：没有splitter（强制为random），没有presort（强制为False），没有max_samples（强制为1.0），没有base_estimator（给定超参数，强制为DecisionTreeClassifier）。
[4] Pierre Geurts et al.，“Extremely Randomized Trees”，Machine Learning 63，no.1（2006）：3－42.

7.5 提升法

提升法（boosting，最初被称为假设提升）是指可以将几个弱学习器结合成一个强学习器的任意集成方法。大多数提升法的总体思路是循环训练预测器，每一次都对其前序做出一些改正。可用的提升法有很多，但目前最流行的方法是AdaBoost[1]（Adaptive Boosting的简称）和梯度提升。我们先从AdaBoost开始介绍。

7.5.1 AdaBoost

新预测器对其前序进行纠正的方法之一就是更多地关注前序欠拟合的训练实例，从而使新的预测器不断地越来越专注于难缠的问题，这就是AdaBoost使用的技术。

例如，当训练AdaBoost分类器时，该算法首先训练一个基础分类器（例如决策树），并使用它对训练集进行预测。然后，该算法会增加分类错误的训练实例的相对权重。然后，它使用更新后的权重训练第二个分类器，并再次对训练集进行预测，更新实例权重，以此类推（见图7-7）。

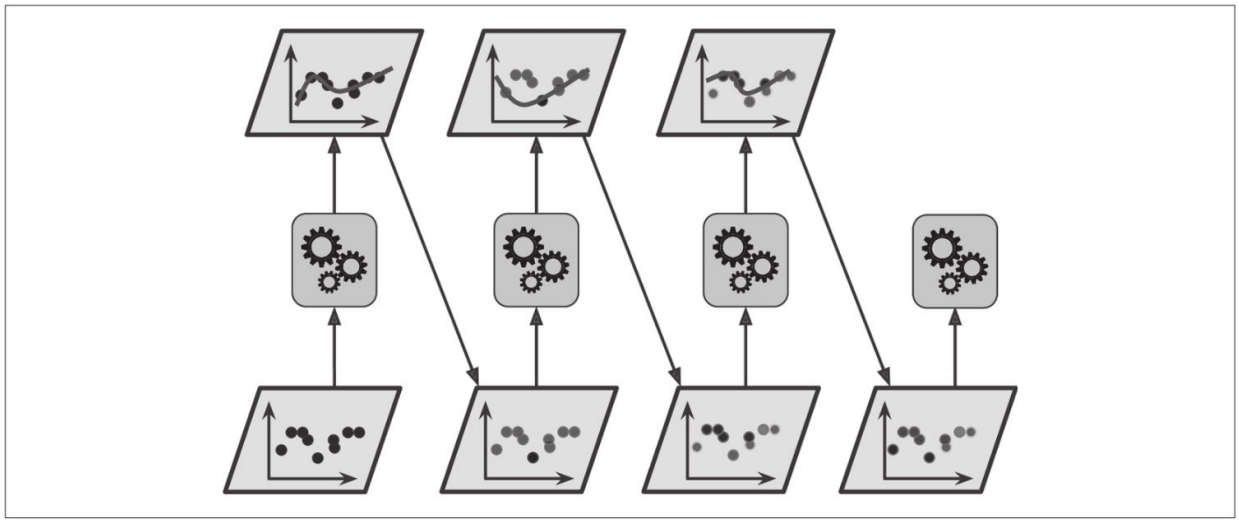


图7-7：AdaBoost循环训练，实例权重不断更新

图7-8显示了在卫星数据集上5个连续的预测器的决策边界（在本例中，每个预测器都使用RBF核函数[2]的高度正则化的SVM分类器）。第一个分类器产生了许多错误实例，所以这些实例的权重得到提升。因此第二个分类器在这些实例上的表现有所提升，然后第三个、第四个……右图绘制的是相同预测器序列，唯一的差别在于学习率减半（即每次迭代仅提升一半错误分类的实例的权重）。可以看出，AdaBoost这种依序循环的学习技术跟梯度下降有一些异曲同工之处，差别只在于——不再是调整单个预测器的参数使成本函数最小化，而是不断在集成中加入预测器，使模型越来越好。

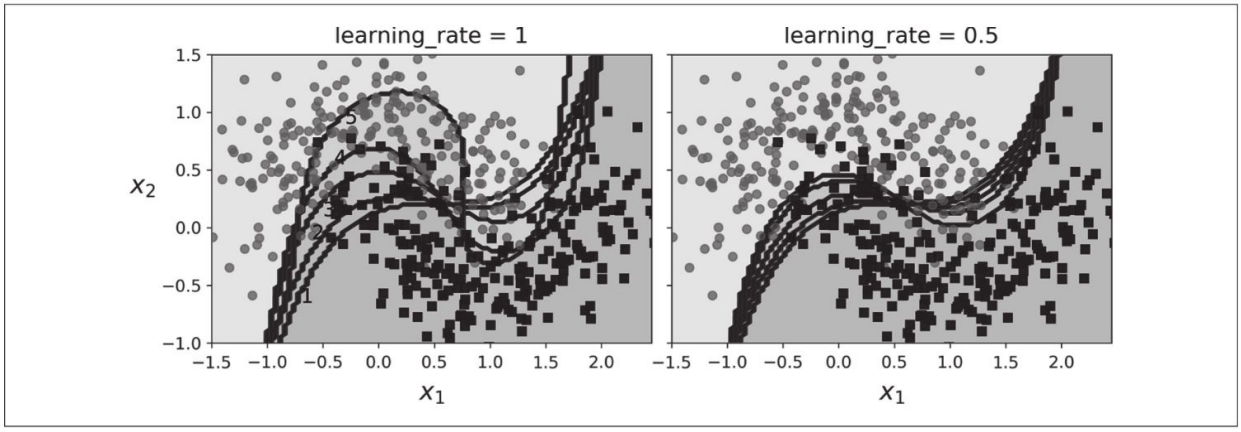


图7-8：连续预测器的决策边界

一旦全部预测器训练完成，集成整体做出预测时就跟bagging或pasting方法一样了，除非预测器有不同的权重，因为它们总的准确率是基于加权后的训练集。

 这种依序学习技术有一个重要的缺陷就是无法并行（哪怕只是一部分），因为每个预测器只能在前一个预测器训练完成并评估之后才能开始训练。因此，在扩展方面，它的表现不如bagging和pasting方法。

让我们仔细看看AdaBoost算法。每个实例权重$w^{(i)}$最初设置为$\frac{1}{m}$。对第一个预测器进行训练，并根据训练集计算其加权误差率r1，请参见公式7-1。

公式7-1：第j个预测器的加权误差率

$$r_j = \frac{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$

$\hat{y}_j^{(i)}$是第i个实例的第j个预测器的预测。

预测器的权重α_j通过公式7-2来计算，其中η是学习率超参数（默认为1）[3]。预测器的准确率越高，其权重就越高。如果它只是随机猜测，则其权重接近于零。但是，如果大部分情况下它都是错的（也就是准确率比随机猜测还低），那么它的权重为负。

公式7-2：预测器权重

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

接下来，AdaBoost算法使用公式7-3更新实例权重，从而提高了误分类实例的权重。

公式7-3：权重更新规则

$$\text{对于 } i = 1, 2, \cdots, m$$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)}, & \text{如果} \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j), & \text{如果} \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

然后对所有实例权重进行归一化（即除以$\sum_{i=1}^{m} w^{(i)}$）。

最后，使用更新后的权重训练一个新的预测器，然后重复整个过程（计算新预测器的权重，更新实例权重，然后对另一个预测器进行训练，等等）。当到达所需数量的预测器或得到完美的预测器时，算法停止。

预测的时候，AdaBoost就是简单地计算所有预测器的预测结果，并使用预测器权重α_j对它们进行加权。最后，得到大多数加权投票的类就是预测器给出的预测类（见公式7-4）。

公式7-4：AdaBoost预测

$$\hat{y}(\boldsymbol{x}) = \underset{k}{\arg\max} \sum_{\substack{j=1 \\ \hat{y}_j(\boldsymbol{x})=k}}^{N} \alpha_j$$

其中N是预测器的数量。

Scikit-Learn使用的其实是AdaBoost的一个多分类版本，叫作SAMME（基于多类指数损失函数的逐步添加模型）[4]。当只有两类时，SAMME即等同于AdaBoost。此外，如果预测器可以估算类概率（即具有predict_proba（）方法），Scikit-Learn会使用一种SAMME的变体，称为SAMME.R（R代表“Real”），它依赖的是类概率而不是类预测，通常表现更好。

下面的代码使用Scikit-Learn的AdaBoostClassifier（正如你猜想的，还有一个AdaBoostRegressor类）训练了一个AdaBoost分类器，它基于200个单层决策树。顾名思义，单层决策树就是max_depth=1的决策树，换言之，就是一个决策节点加两个叶节点。这是AdaBoostClassifier默认使用的基础估算器。

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
```

 如果你的AdaBoost集成过度拟合训练集，你可以试试减少估算器数量，或是提高基础估算器的正则化程度。

7.5.2 梯度提升

另一个非常受欢迎的提升法是梯度提升[5]。与AdaBoost一样，梯度提升也是逐步在集成中添加预测器，每一个都对其前序做出改正。不同之处在于，它不是像AdaBoost那样在每个迭代中调整实例权重，而是让新的预测器针对前一个预测器的残差进行拟合。

我们来看一个简单的回归示例，使用决策树作为基础预测器（梯度提升当然也适用于回归任务），这被称为梯度树提升或者是梯度提升回

归树（GBRT）。首先，在训练集（比如带噪声的二次训练集）上拟合一个DecisionTreeRegressor：

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

针对第一个预测器的残差，训练第二个DecisionTreeRegressor：

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

针对第二个预测器的残差，训练第三个回归器：

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

现在，我们有了一个包含三棵树的集成。它将所有树的预测相加，从而对新实例进行预测：

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

图7-9的左侧表示这三棵树单独的预测，右侧表示集成的预测。第一行，集成只有一棵树，所以它的预测与第一棵树的预测完全相同。第二行是在第一棵树的残差上训练的一棵新树，从右侧可见，集成的预测

等于前面两棵树的预测之和。类似地，第三行又有一棵在第二棵树的残差上训练的新树，集成的预测随着新树的添加逐渐变好。

训练GBRT集成有个简单的方法，就是使用Scikit-Learn的GradientBoostingRegressor类。与RandomForestRegressor类似，它具有控制决策树生长的超参数（例如max_depth、min_samples_leaf等），以及控制集成训练的超参数，例如树的数量（n_estimators）。以下代码可创建上面的集成：

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbrt.fit(X, y)
```

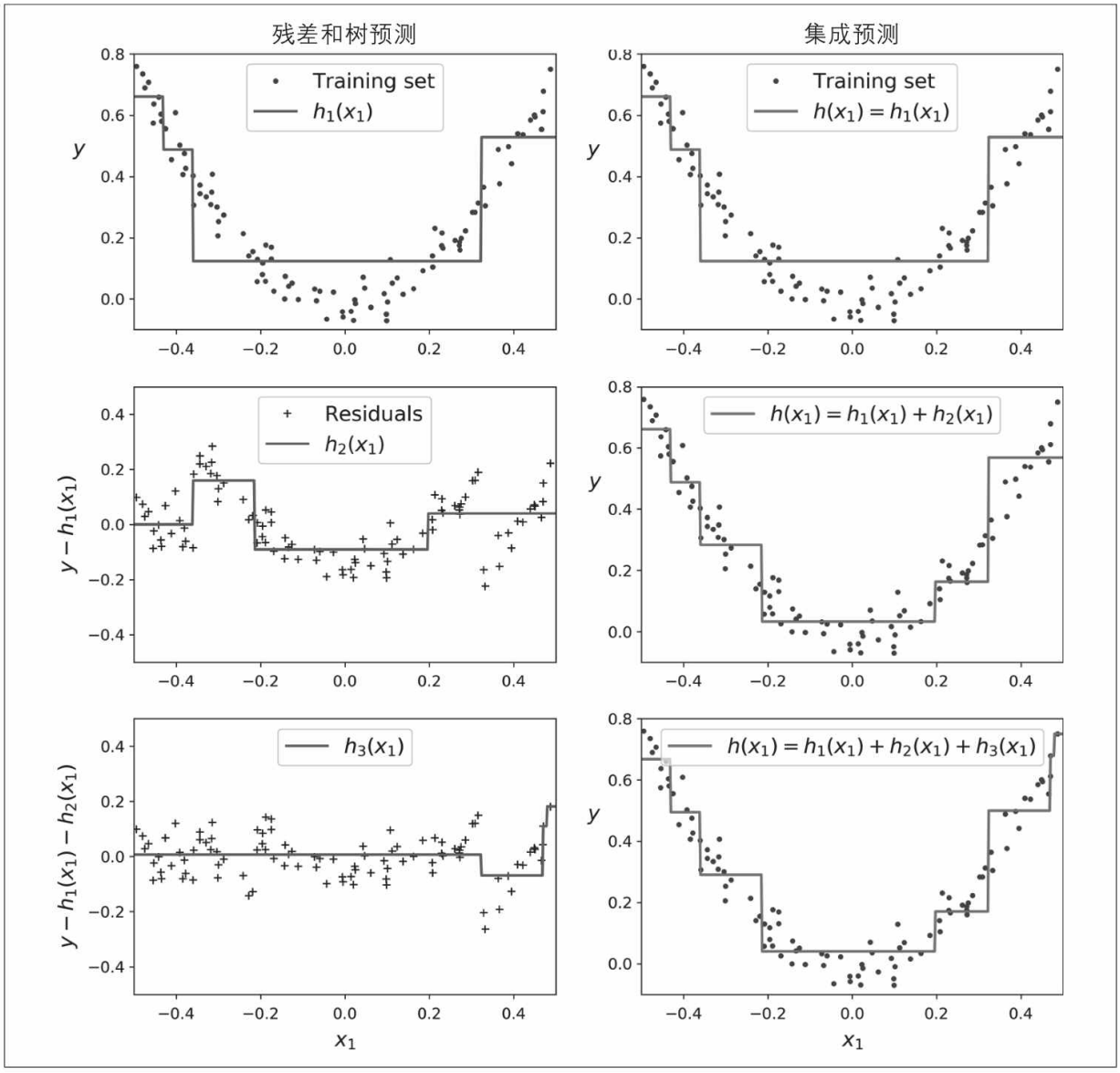


图7-9：在此梯度提升的描述中，第一个预测器（左上）进行正常训练，然后对每个连续的预测器（左中和左下）针对先前预测器的残差进行训练。右列显示了集成的预测结果

超参数learning_rate对每棵树的贡献进行缩放。如果你将其设置为低值，比如0.1，则需要更多的树来拟合训练集，但是预测的泛化效果通常更好，这是一种被称为收缩的正则化技术。图7-10显示了用低学习率训练的两个GBRT集成：左侧拟合训练集的树数量不足，而右侧拟合训练集的树数量过多从而导致过拟合。

要找到树的最佳数量，可以使用提前停止法（参见第4章）。简单的实现方法就是使用staged_predict（）方法：它在训练的每个阶段（一棵树时，两棵树时，等等）都对集成的预测返回一个迭代器。以下代码训练了一个拥有120棵树的GBRT集成，然后测量每个训练阶段的验证误差，从而找到树的最优数量，最后使用最优树数重新训练了一个GBRT集成：

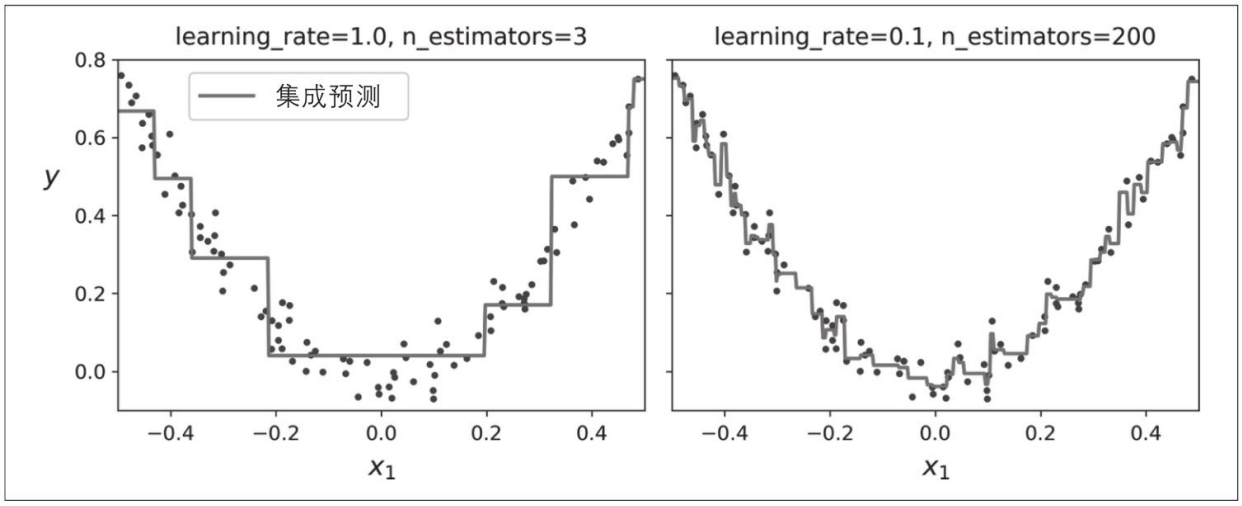


图7-10：GBRT集成——预测器太少（左图）和预测器太多（右图）

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors) + 1

gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

验证误差显示在图7-11的左侧，最佳模型的预测显示在右侧。

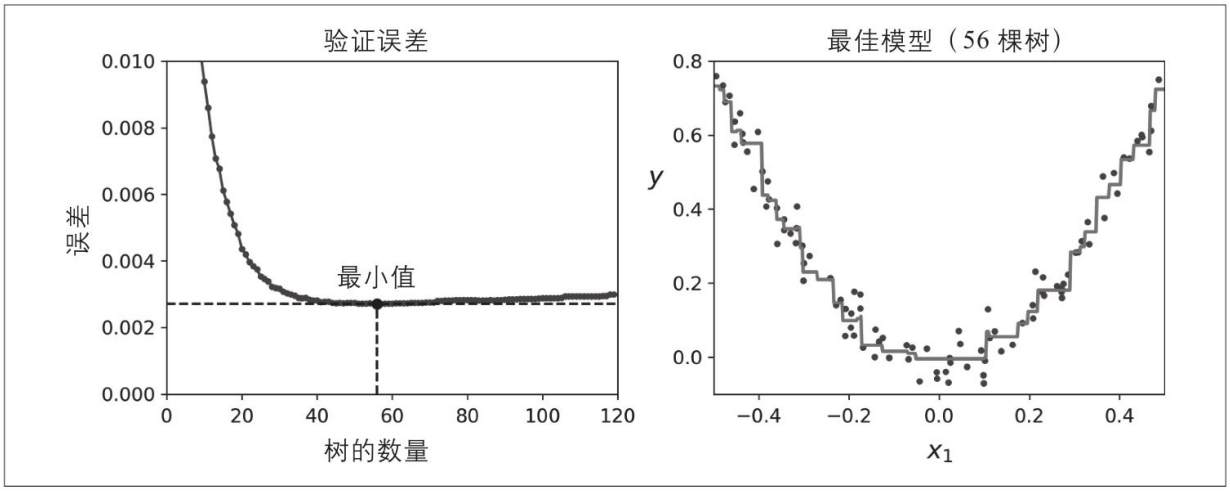


图7-11：通过提前停止法调整树的数量

实际上，要实现提前停止法，不一定需要先训练大量的树，然后再回头找最优的数字，还可以提前停止训练。设置warm_start=True，当fit（）方法被调用时，Scikit-Learn会保留现有的树，从而允许增量训练。以下代码会在验证误差连续5次迭代未改善时，直接停止训练：

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)

min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break  # early stopping
```

GradientBoostingRegressor类还可以支持超参数subsample，指定用于训练每棵树的实例的比例。例如，如果subsample=0.25，则每棵树用25%的随机选择的实例进行训练。现在你可以猜到，这也是用更高的偏差换取了更低的方差，同时在相当大的程度上加速了训练过程。这种技术被称为随机梯度提升。

 梯度提升也可以使用其他成本函数，通过超参数loss来控制（有关详细信息，请参阅Scikit-Learn的文档）。

值得注意的是，流行的Python库XGBoost（该库代表Extreme Gradient Boosting）中提供了梯度提升的优化实现，该软件包最初是由Tianqi Chen作为分布式（深度）机器学习社区（DMLC）的一部分开发的，其开发目标是极快、可扩展和可移植。实际上，XGBoost通常是ML竞赛中获胜的重要组成部分。XGBoost的API与Scikit-Learn的非常相似：

```
import xgboost

xgb_reg = xgboost.XGBRegressor()
xgb_reg.fit(X_train, y_train)
y_pred = xgb_reg.predict(X_val)
```

XGBoost还提供了一些不错的特性，例如自动处理提前停止：

```
xgb_reg.fit(X_train, y_train,
            eval_set=[(X_val, y_val)], early_stopping_rounds=2)
y_pred = xgb_reg.predict(X_val)
```

你一定要看一下！

[1] Yoav Freund和Robert E.Schapire，“A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting”，Journal of Computer and System Sciences 55，no.1（1997）：119－139.
[2] 这只是出于说明目的。SVM通常不是AdaBoost的良好基本预测器。它们很慢，并且趋于不稳定。
[3] 原始的AdaBoost算法不使用学习率超参数。

[4] 更多详细信息，请参见Ji Zhu等人的论文“Multi-Class AdaBoost”，Statistics and Its Interface 2，no.3（2009）：349－360.

[5] 梯度提升在Leo Breiman 1997年的论文“Arcing the Edge”中首次引入，在1999年Jerome H.Friedman的论文“Greedy Function Approximation：A Gradient Boosting Machine”中得到了进一步发展。

7.6 堆叠法

本章我们要讨论的最后一个集成方法叫作堆叠法（stacking），又称层叠泛化法[1]。它基于一个简单的想法：与其使用一些简单的函数（比如硬投票）来聚合集成中所有预测器的预测，我们为什么不训练一个模型来执行这个聚合呢？图7-12显示了在新实例上执行回归任务的这样一个集成。底部的三个预测器分别预测了不同的值（3.1、2.7和2.9），然后最终的预测器（称为混合器或元学习器）将这些预测作为输入，进行最终预测（3.0）。

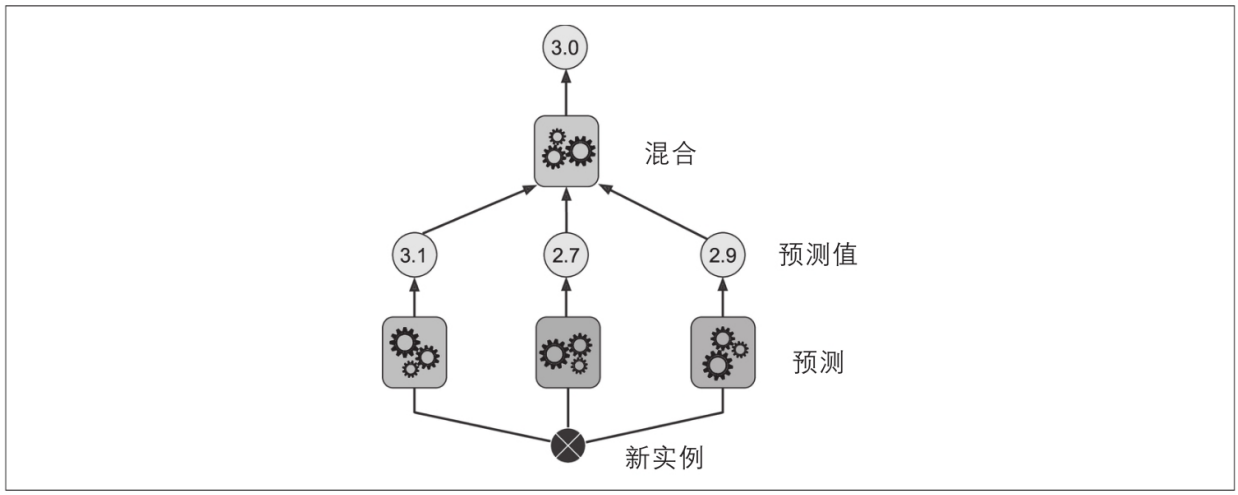


图7-12：通过混合预测器聚合预测

训练混合器的常用方法是使用留存集[2]。我们看看它是如何工作的。首先，将训练集分为两个子集，第一个子集用来训练第一层的预测器（见图7-13）。

然后，用第一层的预测器在第二个（留存）子集上进行预测（见图7-14）。因为预测器在训练时从未见过这些实例，所以可以确保预测是“干净的”。那么现在对于留存集中的每个实例都有了三个预测值。我们可以使用这些预测值作为输入特征，创建一个新的训练集（新的训练

集有三个维度），并保留目标值。在这个新的训练集上训练混合器，让它学习根据第一层的预测来预测目标值。

事实上，通过这种方法可以训练多种不同的混合器（例如，一个使用线性回归，另一个使用随机森林回归，等等）。于是我们可以得到一个混合器层。诀窍在于将训练集分为三个子集：第一个用来训练第一层，第二个用来创造训练第二层的新训练集（使用第一层的预测），而第三个用来创造训练第三层的新训练集（使用第二层的预测）。一旦训练完成，我们可以按照顺序遍历每层来对新实例进行预测，如图7-15所示。

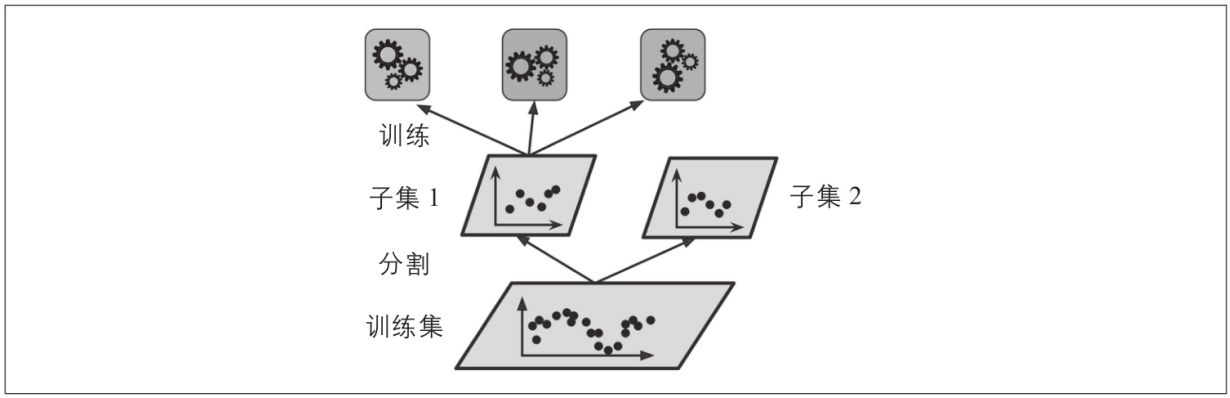


图7-13：训练第一层

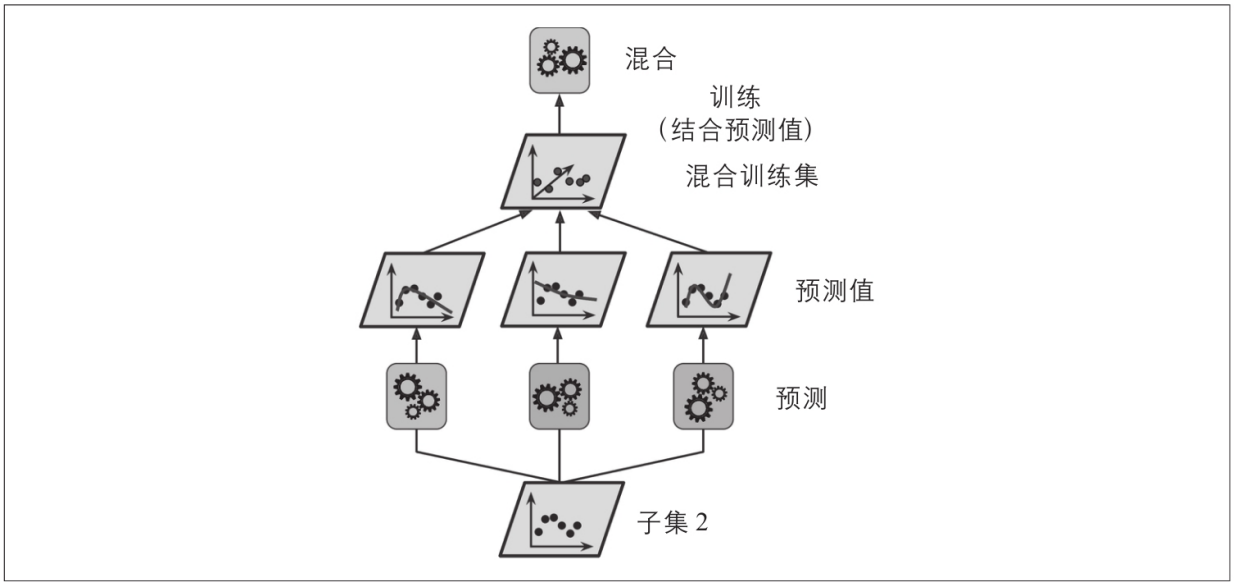


图7-14：训练混合器

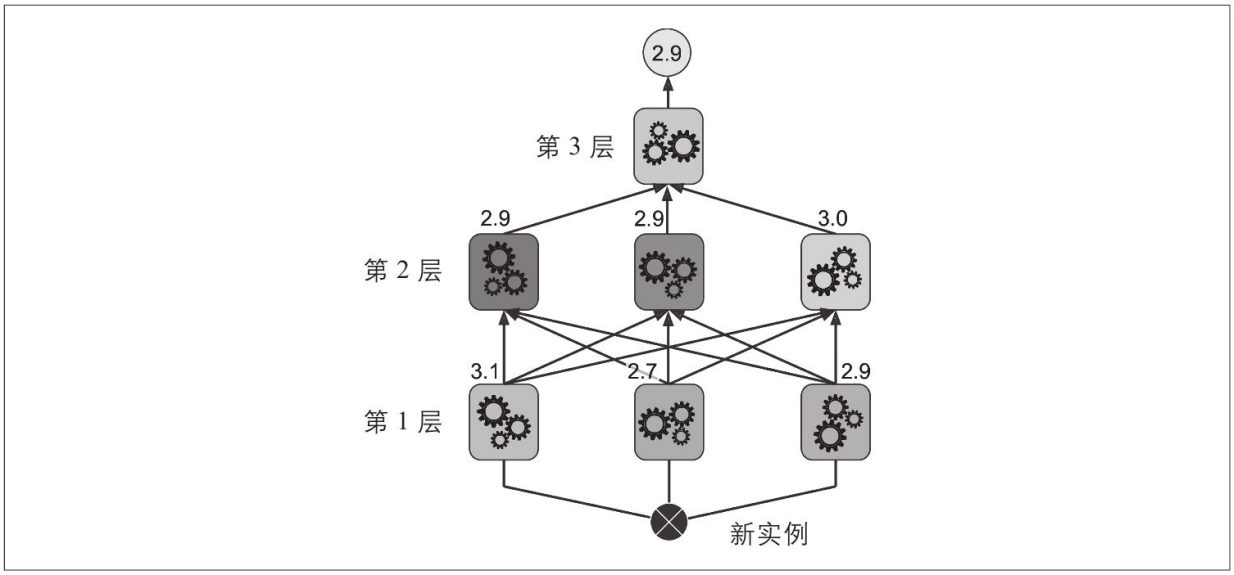


图7-15：一个多层堆叠集成的预测

不幸的是，Scikit-Learn不直接支持堆叠，但是推出自己的实现并不太难（参见接下来的练习题）。或者，你也可以使用开源的实现方案，例如DESlib。

[1] David H.Wolpert，“Stacked Generalization”，Neural Networks 5，no.2（1992）：241－259.

[2] 也可以使用折外（out-of-fold）预测。在某些情况下，这才被称为堆叠（stacking），而使用留存集被称为混合（blending）。但是对多数人而言，这二者是同义词。

7.7 练习题

1. 如果你已经在完全相同的训练集上训练了5个不同的模型，并且它们都达到了95%的准确率，是否还有机会通过结合这些模型来获得更好的结果？如果可以，该怎么做？如果不行，为什么？

2. 硬投票分类器和软投票分类器有什么区别？

3. 是否可以通过在多个服务器上并行来加速bagging集成的训练？pasting集成呢？boosting集成呢？随机森林或stacking集成呢？

4. 包外评估的好处是什么？

5. 是什么让极端随机树比一般随机森林更加随机？这部分增加的随机性有什么用？极端随机树比一般随机森林快还是慢？

6. 如果你的AdaBoost集成对训练数据欠拟合，你应该调整哪些超参数？怎么调整？

7. 如果你的梯度提升集成对训练集过拟合，你是应该提升还是降低学习率？

8. 加载MNIST数据集（第3章中有介绍），将其分为一个训练集、一个验证集和一个测试集（例如，使用50 000个实例训练、10 000个实例验证、10 000个实例测试）。然后训练多个分类器，比如一个随机森林分类器、一个极端随机树分类器和一个SVM分类器。接下来，尝试使用软投票法或者硬投票法将它们组合成一个集成，这个集成在验证集上的表现要胜过它们各自单独的表现。成功找到集成后，在测试集上测试。与单个的分类器相比，它的性能要好多少？

9.运行练习题8中的单个分类器，用验证集进行预测，然后用预测结果创建一个新的训练集：新训练集中的每个实例都是一个向量，这个向量包含所有分类器对于一张图像的一组预测，目标值是图像的类。恭喜，你成功训练了一个混合器，结合第一层的分类器，它们一起构成了一个stacking集成。现在在测试集上评估这个集成。对于测试集中的每张图像，使用所有的分类器进行预测，然后将预测结果提供给混合器，得到集成的预测。与前面训练的投票分类器相比，这个集成的结果如何？

附录A中提供了这些练习题的解答。

第8章　降维

许多机器学习问题涉及每个训练实例的成千上万甚至数百万个特征。正如我们将看到的那样，所有这些特征不仅使训练变得极其缓慢，而且还会使找到好的解决方案变得更加困难。这个问题通常称为维度的诅咒。

幸运的是，在实际问题中，通常可以大大减少特征的数量，从而将棘手的问题转变为易于解决的问题。例如，考虑MNIST图像（在第3章中介绍）：图像边界上的像素几乎都是白色，因此你可以从训练集中完全删除这些像素而不会丢失太多信息。图7-6确认了这些像素对于分类任务而言完全不重要。另外，两个相邻的像素通常是高度相关的，如果将它们合并为一个像素（例如，通过取两个像素的平均值），不会丢失太多信息。

 数据降维确实会丢失一些信息（就好比将图像压缩为JPEG会降低其质量一样），所以，它虽然能够加速训练，但是也会轻微降低系统性能。同时它也让流水线更为复杂，维护难度上升。因此，如果训练太慢，你首先应该尝试的还是继续使用原始数据，然后再考虑数据降维。不过在某些情况下，降低训练数据的维度可能会滤除掉一些不必要的噪声和细节，从而导致性能更好（但通常来说不会，它只会加速训练）。

除了加快训练，降维对于数据可视化（或称DataViz）也非常有用。将维度降到两个（或三个），就可以在图形上绘制出高维训练集，通过视觉来检测模式，常常可以获得一些十分重要的洞察，比如聚类。此外，DataViz对于把你的结论传达给非数据科学家至关重要，尤其是将使用你的结果的决策者。

本章将探讨维度的诅咒，简要介绍高维空间中发生的事情。然后，我们将介绍两种主要的数据降维方法（投影和流形学习），并学习现在最流行的三种数据降维技术：PCA、Kernal PCA以及LLE。

8.1 维度的诅咒

我们太习惯三维空间[1]的生活，所以当我们试图去想象一个高维空间时，直觉思维很难成功。即使是一个基本的四维超立方体（见图8-1），我们也很难在脑海中想象出来，更不用说在一个千维空间中弯曲的二百维椭圆体。

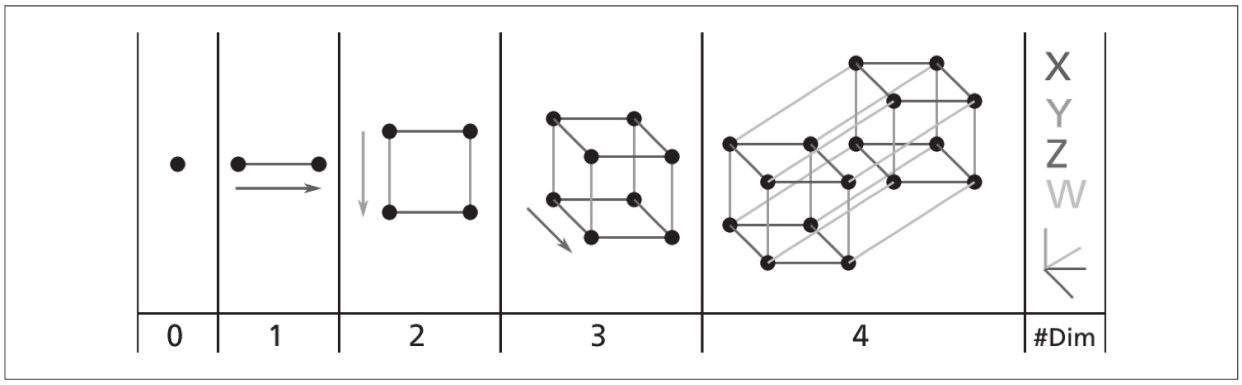


图8-1：点、线段、正方形、立方体和网格（零维至四维超立方体）[2]

事实证明，在高维空间中，许多事物的行为都迥然不同。例如，如果你在一个单位平面（1×1的正方形）内随机选择一个点，那么这个点离边界的距离小于0.001的概率只有约0.4%（也就是说，一个随机的点不大可能刚好位于某个维度的“极端”）。但是，在一个10 000维的单位超立方体（1×1…×1立方体，一万个1）中，这个概率大于99.99999%。高维超立方体中大多数点都非常接近边界[3]。

还有一个更麻烦的区别：如果你在单位平面中随机挑两个点，这两个点之间的平均距离大约为0.52。如果在三维的单位立方体中随机挑两个点，两点之间的平均距离大约为0.66。但是，如果在一个100万维的超立方体中随机挑两个点呢？不管你相信与否，平均距离大约为408.25（约等于$\sqrt{1\,000\,000/6}$）！这是非常违背直觉的：位于同一个单位超立方体中的两个点，怎么可能距离如此之远？这个事实说明高维数据集有很大可能是非常稀疏的：大多数训练实例可能彼此之间相距很远。当然，这也意味着新的实例很可能远离任何一个训练实例，导致跟低维度相比，预测更加不可靠，因为它们基于更大的推测。简而言之，训练集的维度越高，过拟合的风险就越大。

理论上来说，通过增大训练集，使训练实例达到足够的密度，是可以解开维度的诅咒的。然而不幸的是，实践中，要达到给定密度，所需要的训练实例数量随着维度的增加呈指数式上升。仅仅100个特征下（远小于MNIST问题），要让所有训练实例（假设在所有维度上平均分布）之间的平均距离小于0.1，你需要的训练实例数量就比可观察宇宙中的原子数量还要多。

[1] 好吧，如果算上时间就是四维，或者如果你是个弦物理学家，还可以再高几个维度。
[2] 在https://homl.info/30上可以观看一个投影到三维空间的旋转超立方体。图片由维基百科用户NerdBoy1932提供（Creative Commons BY-SA 3.0），转载自https://en.wikipedia.org/wiki/Tesseract。
[3] 趣味事实：只要你考虑足够多的维度，你所知道的每个人，至少在某一个维度上，都可能算是个极端主义者（比如，他们在咖啡里放多少糖）。

8.2 降维的主要方法

在深入研究特定的降维算法之前，让我们看一下减少维度的两种主要方法：投影和流形学习。

8.2.1 投影

在大多数实际问题中，训练实例并不是均匀地分布在所有维度上。许多特征几乎是恒定不变的，而其他特征则是高度相关的（如之前针对MNIST所述）。结果，所有训练实例都位于（或接近于）高维空间的低维子空间内。这听起来很抽象，所以让我们看一个示例。在图8-2中，你可以看到由圆圈表示的3D数据集。

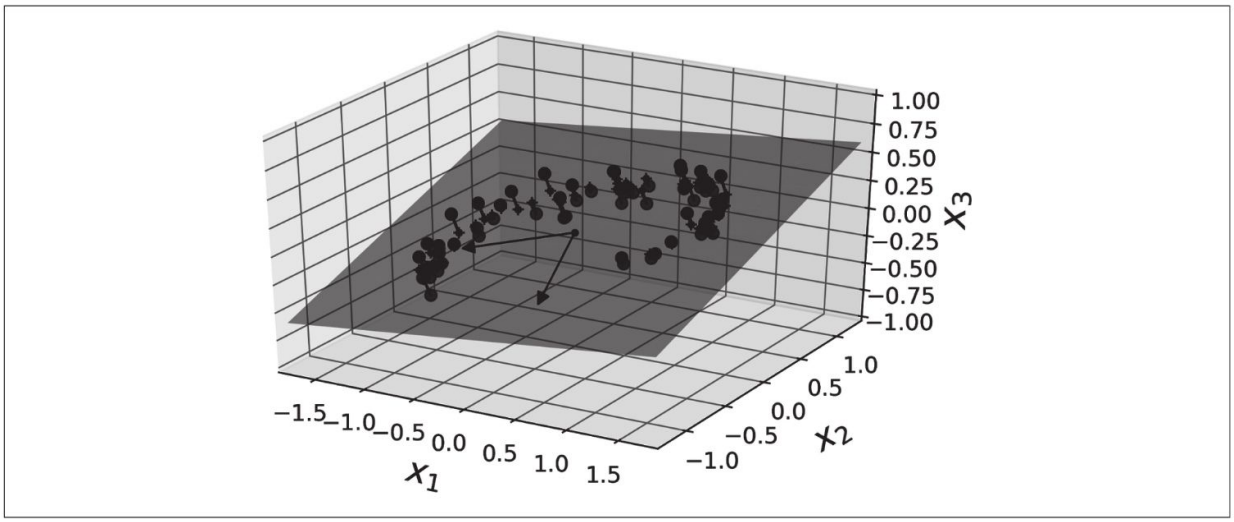


图8-2：靠近2D子空间的3D数据集

请注意，所有训练实例都位于一个平面附近：这是高维（3D）空间的低维（2D）子空间。如果我们将每个训练实例垂直投影到该子空间上（如实例连接到平面的短线所示），我们将获得如图8-3所示的新2D数据集——我们刚刚将数据集的维度从3D减少到2D。注意，轴对应于新特征z_1和z_2（平面上投影的坐标）。

但是，投影并不总是降低尺寸的最佳方法。在许多情况下，子空间可能会发生扭曲和转动，例如在图8-4中所示的著名的瑞士卷小数据集中。

如图8-5左侧所示，简单地投影到一个平面上（例如，去掉x_3维度）会将瑞士卷的不同层挤压在一起。你真正想要的是展开瑞士卷，得到图8-5右侧的2D数据集。

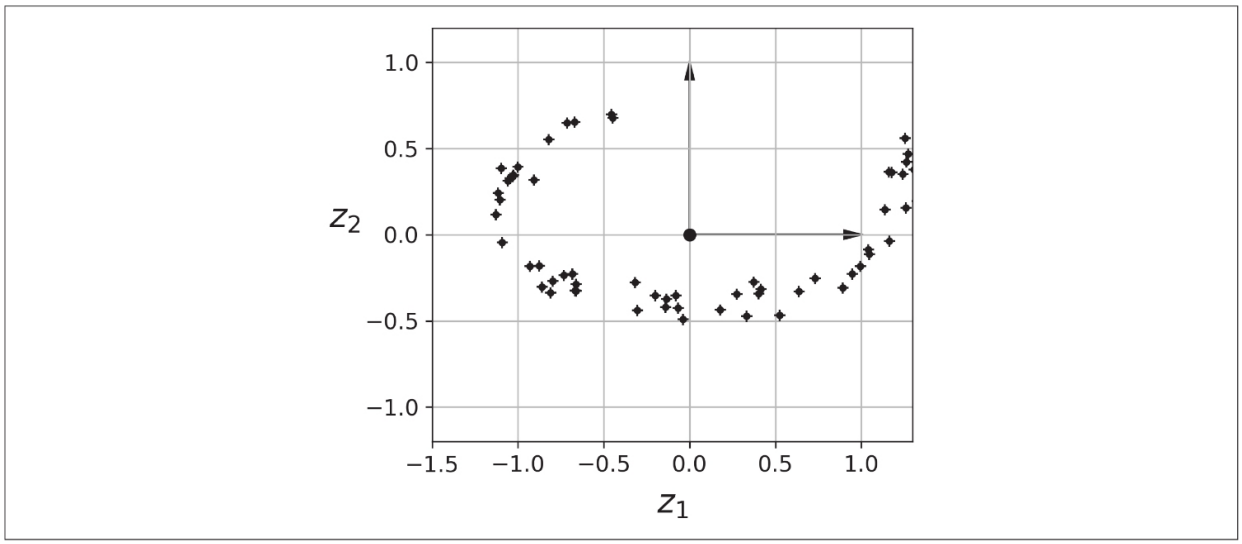


图8-3：投影后的新2D数据集

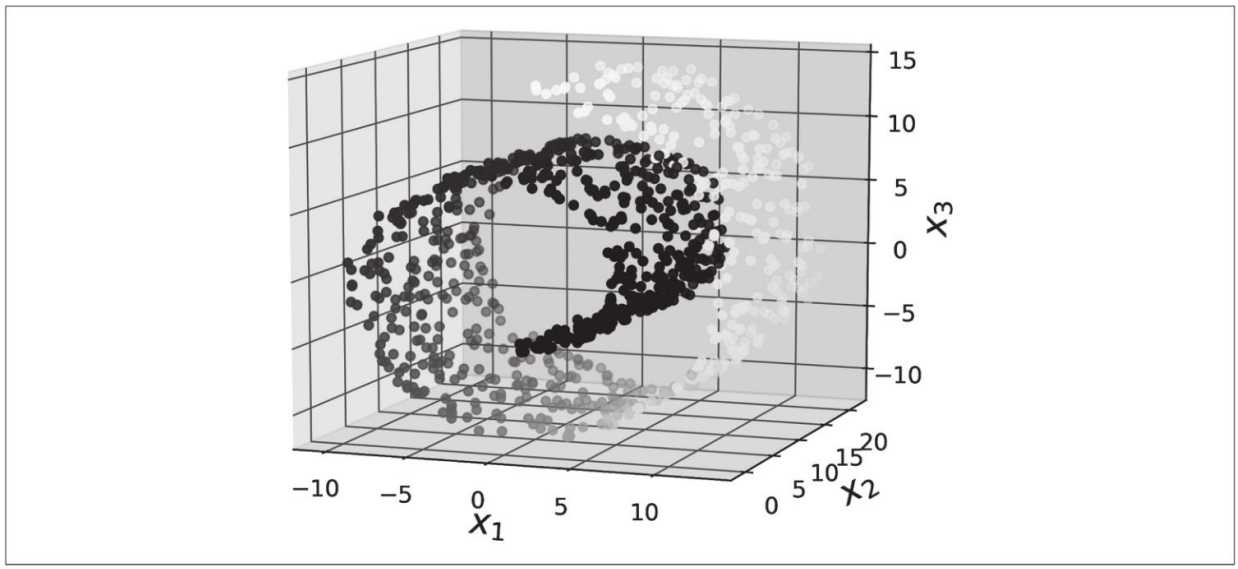


图8-4：瑞士卷数据集

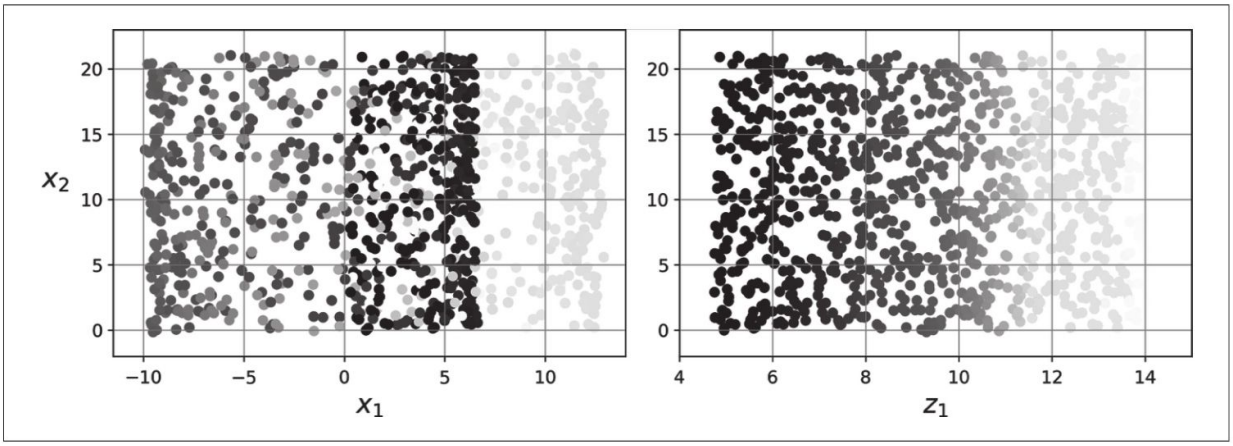


图8-5：投影到平面上（左）与展开瑞士卷（右）

8.2.2 流形学习

瑞士卷是2D流形的一个示例。简而言之，2D流形是可以在更高维度的空间中弯曲和扭曲的2D形状。更一般而言，d维流形是n维空间（其中d<n）的一部分，局部类似于d维超平面。在瑞士卷的情况下，d=2且n=3时，它局部类似于2D平面，但在第三维中弯曲。

许多降维算法通过对训练实例所在的流形进行建模来工作。这称为流形学习。它依赖于流形假设（也称为流形假说），该假设认为大多数现实世界的高维数据集都接近于低维流形。通常这是根据经验观察到的这种假设。

再次考虑一下MNIST数据集：所有手写数字图像都有一些相似之处。它们由连接的线组成，边界为白色，并且或多或少居中。如果你随机生成图像，那么其中只有一小部分看起来像手写数字。换句话说，如果你试图创建数字图像，可用的自由度大大低于允许你生成任何图像的自由度。这些约束倾向于将数据集压缩为低维流形。

流形假设通常还伴随着另一个隐式假设：如果用流形的低维空间表示，手头的任务（例如分类或回归）将更加简单。例如，在图8-6的上

面一行中，瑞士卷分为两类：在3D空间（左侧）中，决策边界会相当复杂，而在2D展开流形空间中（右侧），决策边界是一条直线。

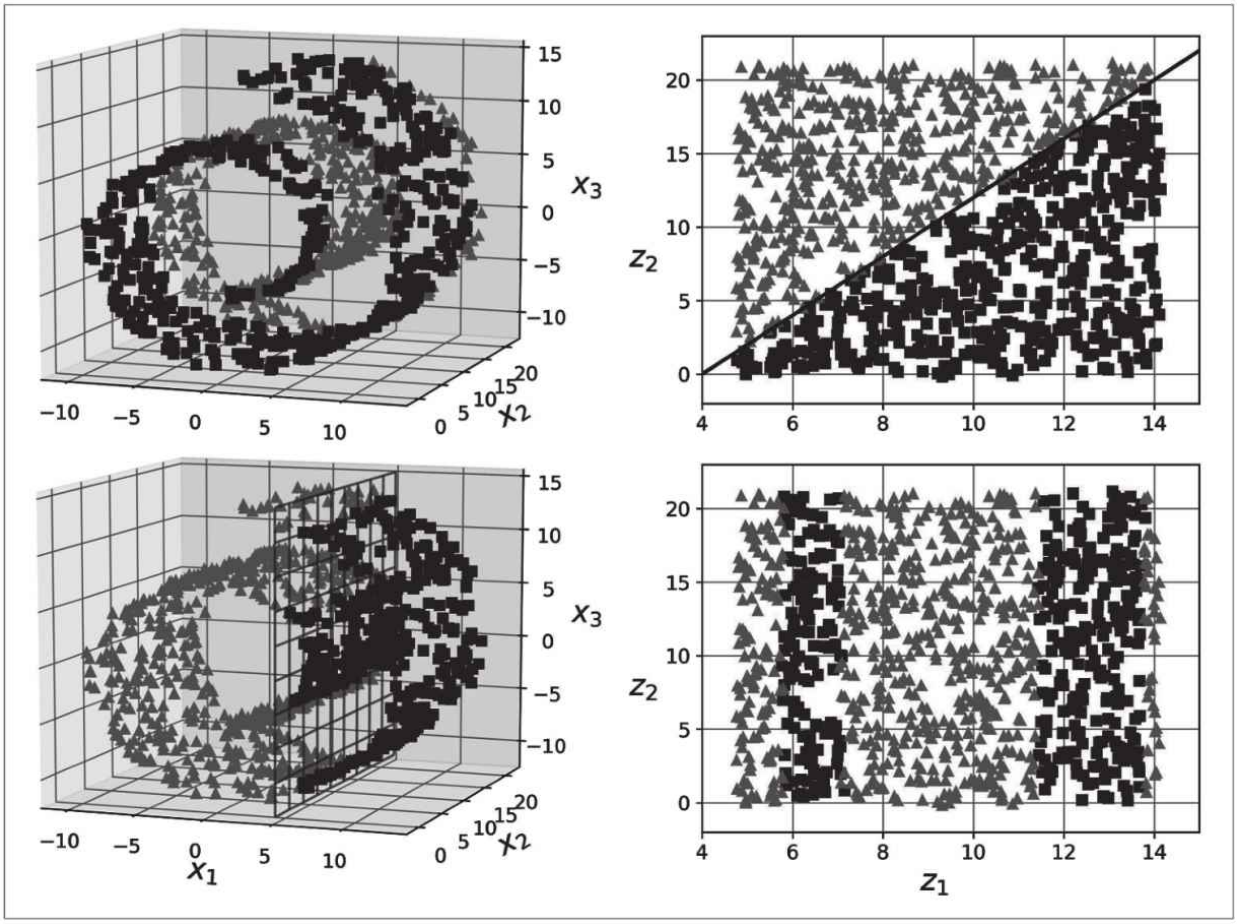


图8-6：决策边界不总是维度越低越简单

但是，这种隐含假设并不总是成立。例如，在图8-6的下面一行中，决策边界位于x1=5处。此决策边界在原始3D空间（垂直平面）中看起来非常简单，但在展开流形中看起来更加复杂（四个独立线段的集合）。

简而言之，在训练模型之前降低训练集的维度肯定可以加快训练速度，但这并不总是会导致更好或更简单的解决方案，它取决于数据集。

希望现在你对于维度的诅咒有了一个很好的理解，也知道降维算法是怎么解决它的，特别是当流形假设成立的时候应该怎么处理。本章剩

余部分将逐一介绍几个最流行的算法。

8.3 PCA

主成分分析（PCA）是迄今为止最流行的降维算法。首先，它识别最靠近数据的超平面，然后将数据投影到其上，如图8-2所示。

8.3.1 保留差异性

将训练集投影到低维超平面之前需要选择正确的超平面。例如图8-7的左图代表一个简单的2D数据集，沿三条不同的轴（即一维超平面）。右图是将数据集映射到每条轴上的结果。正如你所见，在实线上的投影保留了最大的差异性，而点线上的投影只保留了非常小的差异性，虚线上的投影的差异性居中。

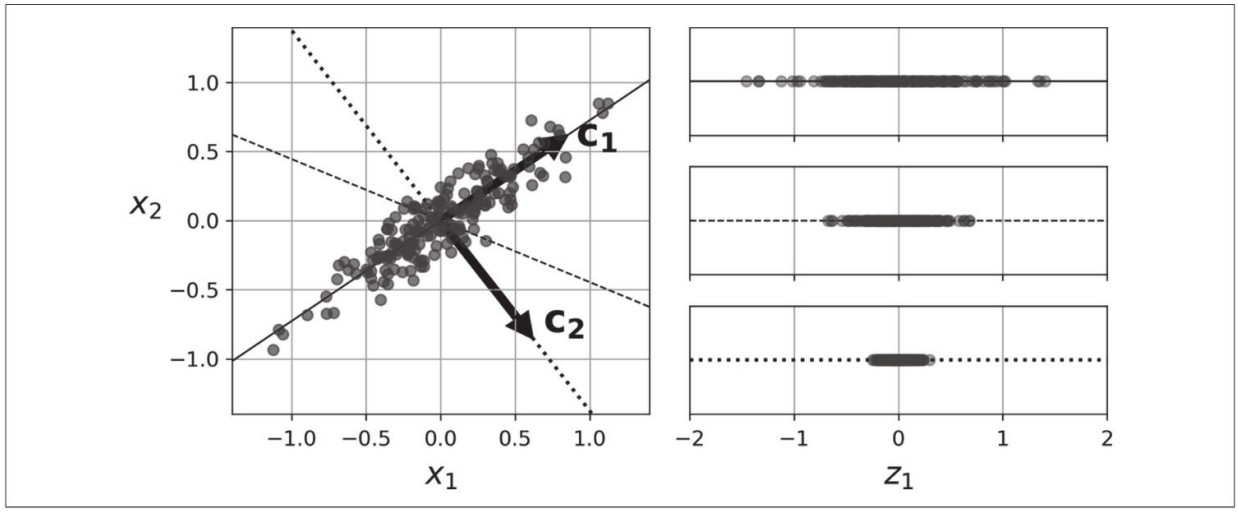


图8-7：选择要投影的子空间

选择保留最大差异性的轴看起来比较合理，因为它可能比其他两种投影丢失的信息更少。要证明这一选择，还有一种方法，即比较原始数据集与其轴上的投影之间的均方距离，使这个均方距离最小的轴是最合理的选择，也就是实线代表的轴。这也正是PCA背后的简单思想[1]。

8.3.2 主要成分

主成分分析可以在训练集中识别出哪条轴对差异性的贡献度最高。在图8-7中是由实线表示的轴。同时它也找出了第二条轴，与第一条轴垂直，它对剩余差异性的贡献度最高。因为这个示例是二维的，所以除了这条点线再没有其他。如果是在更高维数据集中，PCA还会找到与前两条都正交的第三条轴，以及第四条、第五条，等等——轴的数量与数据集维度数量相同。

第i个轴称为数据的第i个主要成分（PC）。在图8-7中，第一个PC是向量c_1所在的轴，第二个PC是向量c_2所在的轴。在图8-2中，前两个PC是平面上两个箭头所在的正交轴，第三个PC是与该平面正交的轴。

 对于每个主要成分，PCA都找到一个指向PC方向的零中心单位向量。由于两个相对的单位向量位于同一轴上，因此PCA返回的单位向量的方向不稳定：如果稍微扰动训练集并再次运行PCA，则单位向量可能会指向原始向量的相反方向。但是，它们通常仍位于相同的轴上。在某些情况下，一对单位向量甚至可以旋转或交换（如果沿这两个轴的方差接近），但是它们定义的平面通常保持不变。

那么如何找到训练集的主要成分呢？幸运的是，有一种称为奇异值分解（SVD）的标准矩阵分解技术，该技术可以将训练集矩阵X分解为三个矩阵$U\Sigma V^T$的矩阵乘法，其中V包含定义所有主要成分的单位向量。如公式8-1所示。

公式8-1：主成分矩阵

$$\boldsymbol{V} = \begin{pmatrix} | & | & & | \\ \boldsymbol{c}_1 & \boldsymbol{c}_2 & \cdots & \boldsymbol{c}_n \\ | & | & & | \end{pmatrix}$$

以下Python代码使用NumPy的svd（）函数来获取训练集的所有主要成分，然后提取定义前两个PC的两个单位向量：

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

 PCA假定数据集以原点为中心。正如我们将看到的，Scikit-Learn的PCA类负责为你居中数据。如果你自己实现PCA（如上例所示），或者使用其他库，请不要忘记首先将数据居中。

8.3.3 向下投影到d维度

一旦确定了所有主要成分，你就可以将数据集投影到前d个主要成分定义的超平面上，从而将数据集的维度降低到d维。选择这个超平面可确保投影将保留尽可能多的差异性。例如，在图8-2中，将3D数据集投影到由前两个主成分定义的2D平面上，从而保留了数据集大部分的差异性。最终，2D投影看起来非常类似于原始3D数据集。

要将训练集投影到超平面上并得到维度为d的简化数据集$X_{d\text{-proj}}$，计算训练集矩阵X与矩阵W_d的矩阵相乘，矩阵W_d定义为包含V的前d列的矩阵，如公式8-2所示。

公式8-2：将训练集投影到d维度

$$\boldsymbol{X}_{d\text{-proj}} = \boldsymbol{X}\,\boldsymbol{W}_d$$

以下Python代码将训练集投影到由前两个主要成分定义的平面上：

```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

你现在知道了如何将任何数据集的维度减少到任意数量的维度，同时保留尽可能多的差异性。

8.3.4 使用Scikit-Learn

就像我们在本章前面所做的那样，Scikit-Learn的PCA类使用SVD分解来实现PCA。以下代码应用PCA将数据集的维度降到二维（请注意，它会自动处理数据居中的问题）：

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

将PCA转换器拟合到数据集后，其components_属性是W_d的转置（例如，定义第一个主成分的单位向量等于pca.components_.T[：，0]）。

8.3.5 可解释方差比

另一个有用的信息是每个主成分的可解释方差比，可以通过explained_variance_ratio_变量来获得。该比率表示沿每个成分的数据集方差的比率。例如，让我们看一下图8-2中表示的3D数据集的前两个成分的可解释方差比：

```
>>> pca.explained_variance_ratio_
array([0.84248607, 0.14631839])
```

此输出告诉你，数据集方差的84.2%位于第一个PC上，而14.6%位于第二个PC上。对于第三个PC，这还不到1.2%，因此可以合理地假设第三个PC携带的信息很少。

8.3.6 选择正确的维度

与其任意选择要减小到的维度，不如选择相加足够大的方差部分（例如95%）的维度。当然，如果你是为了数据可视化而降低维度，这种情况下，需要将维度降低到2或3。

以下代码在不降低维度的情况下执行PCA，然后计算保留95%训练集方差所需的最小维度：

```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

然后，你可以设置n_components=d并再次运行PCA。但是还有一个更好的选择：将n_components设置为0.0到1.0之间的浮点数来表示要保留的方差率，而不是指定要保留的主成分数：

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

另一个选择是将可解释方差绘制成维度的函数（简单地用cumsum绘制，见图8-8）。曲线上通常会出现一个拐点，其中可解释方差会停止快速增大。在这种情况下，你可以看到将维度降低到大约100而不会损失太多的可解释方差。

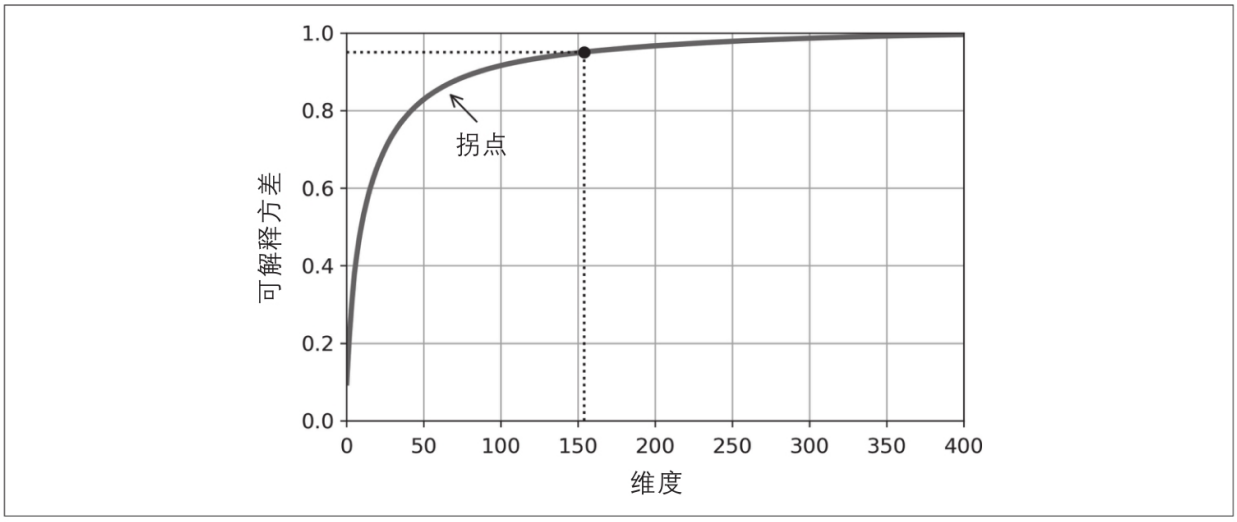


图8-8：可解释方差作为维度的函数

8.3.7 PCA压缩

降维后，训练集占用的空间要少得多。例如，将PCA应用于MNIST数据集，同时保留其95%的方差。你会发现每个实例将具有150多个特征，而不是原始的784个特征。因此，尽管保留了大多数方差，但数据集现在不到其原始大小的20%！这是一个合理的压缩率，你可以看到这种维度减小极大地加速了分类算法（例如SVM分类器）。

通过应用PCA投影的逆变换，还可以将缩减后的数据集解压缩回784维。由于投影会丢失一些信息（在5%的方差被丢弃），因此这不会给你原始的数据，但可能会接近原始数据。原始数据与重构数据（压缩后再解压缩）之间的均方距离称为重构误差。

以下代码将MNIST数据集压缩为154个维度，然后使用inverse_transform（）方法将其解压缩回784个维度：

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

图8-9显示了原始训练集的一些数字（左侧），以及压缩和解压缩后的相应数字。你会看到图像质量略有下降，但是数字仍然大部分保持完好。

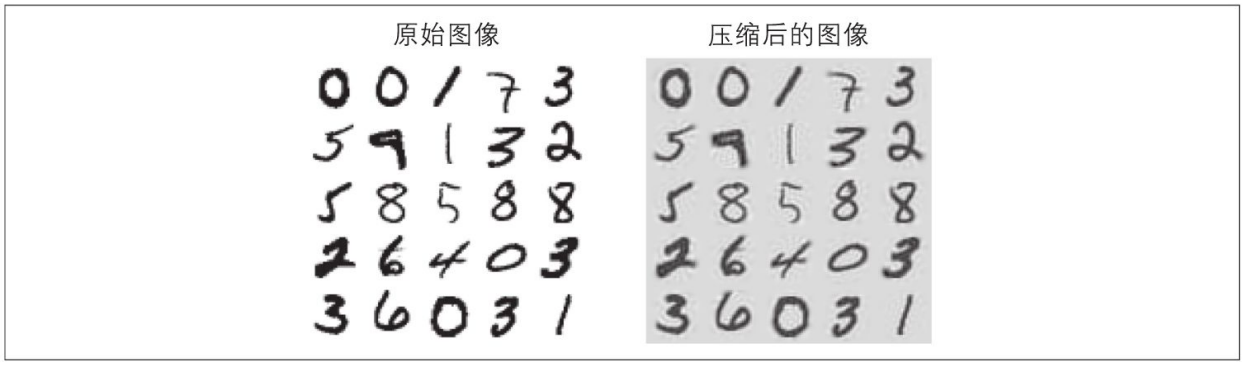


图8-9：保留95%方差的MNIST压缩

逆变换的公式如公式8-3所示。

公式8-3：PCA逆变换，回到原始数量的维度

$$\boldsymbol{X}_{\text{recovered}} = \boldsymbol{X}_{d\text{-proj}}\,\boldsymbol{W}_d^{\mathrm{T}}$$

8.3.8 随机PCA

如果将超参数svd_solver设置为"randomized"，则Scikit-Learn将使用一种称为Randomized PCA的随机算法，该算法可以快速找到前d个主成分的近似值。它的计算复杂度为O（m×d^2）+O（d^3），而不是完全SVD方法的O（m×n^2）+O（n^3），因此，当d远远小于n时，它比完全的SVD快得多：

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

默认情况下，svd_solver实际上设置为"auto"：如果m或n大于500并且d小于m或n的80%，则Scikit-Learn自动使用随机PCA算法，否则它将使用完全的SVD方法。如果要强制Scikit-Learn使用完全的SVD，可以将svd_solver超参数设置为"full"。

8.3.9 增量PCA

前面的PCA实现的一个问题是，它们要求整个训练集都放入内存才能运行算法。幸运的是已经开发了增量PCA（IPCA）算法，它们可以使你把训练集划分为多个小批量，并一次将一个小批量送入IPCA算法。这对于大型训练集和在线（即在新实例到来时动态运行）应用PCA很有用。

以下代码将MNIST数据集拆分为100个小批量（使用NumPy的array_split（）函数），并将其馈送到Scikit-Learn的IncrementalPCA类[2]，来把MNIST数据集的维度降低到154（就像之前做的那样）。请注意，你必须在每个小批量中调用partial_fit（）方法，而不是在整个训练集中调用fit（）方法：

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

另外，你可以使用NumPy的memmap类，该类使你可以将存储在磁盘上的二进制文件中的大型数组当作完全是在内存中一样来操作，该类仅在需要时才将数据加载到内存中。由于IncrementalPCA类在任何给定时间仅使用数组的一小部分，因此内存使用情况处于受控状态。如以下代码所示，这使得调用通常的fit（）方法成为可能：

```
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))

batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mm)
```

---

[1] Karl Pearson，“On Lines and Planes of Closest Fit to Systems of Points in Space”，The London，Edinburgh，and Dublin Philosophical Magazine and Journal of Science 2，no.11（1901）：559-572，https://homl.info/pca.
[2] Scikit-Learn使用David A.Ross等人描述的算法，“Incremental Learning for Robust Visual Tracking”，International Journal of Computer Vision 77，no.1－3（2008）：125－141。

8.4 内核PCA

在第5章中，我们讨论了内核，这是一种数学技术，它可以将实例隐式映射到一个高维空间（称为特征空间），从而可以使用支持向量机来进行非线性分类和回归。回想一下，高维特征空间中的线性决策边界对应于原始空间中的复杂非线性决策边界。

事实证明，可以将相同的技术应用于PCA，从而可以执行复杂的非线性投影来降低维度。这叫作内核PCA（kPCA）[1]。它通常擅长在投影后保留实例的聚类，有时甚至可以展开位于扭曲流形附近的数据集。

下面的代码使用Scikit-Learn的KernelPCA类以及用RBF内核来执行kPCA（有关RBF内核和其他内核的更多详细信息，请参见第5章）：

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

图8-10显示了瑞士卷，它使用线性内核（相当于简单地使用PCA类）、RBF内核和sigmoid内核减小为二维。

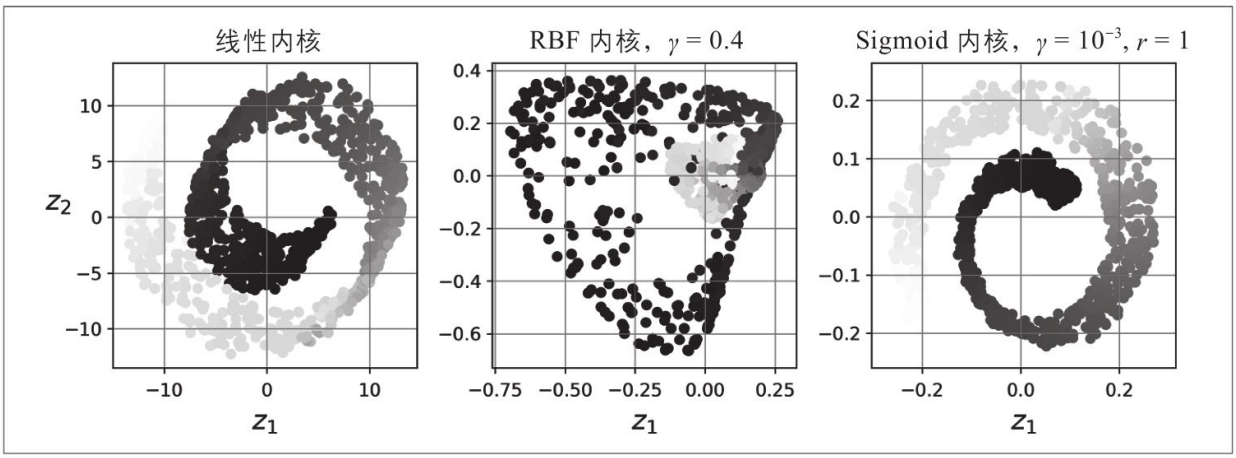


图8-10：使用各种内核的kPCA将瑞士卷缩减为2D

选择内核并调整超参数

由于kPCA是一种无监督学习算法，因此没有明显的性能指标可以帮助你选择最好的内核和超参数值。也就是说，降维通常是有监督学习任务（例如分类）的准备步骤，因此你可以使用网格搜索来选择在该任务上能获得最佳性能的内核和超参数。以下代码创建了一个两步流水线，首先使用kPCA将维度减少到二维，然后使用逻辑回归来分类。它使用GridSearchCV来查找kPCA的最佳内核和gamma值，以便在流水线的最后得到最好的分类准确率：

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
        ("kpca", KernelPCA(n_components=2)),
        ("log_reg", LogisticRegression())
    ])

param_grid = [{
        "kpca__gamma": np.linspace(0.03, 0.05, 10),
        "kpca__kernel": ["rbf", "sigmoid"]
    }]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

然后，可以通过best_params_变量来得到最佳内核和超参数：

```
>>> print(grid_search.best_params_)
{'kpca__gamma': 0.043333333333333335, 'kpca__kernel': 'rbf'}
```

另一种完全无监督的方法是选择产生最低重构误差的内核和超参数。请注意，重建并不像使用线性PCA那样容易。以下是原因。图8-11显示了原始的瑞士卷3D数据集（左上）和使用RBF内核应用kPCA之后得

到的2D数据集（右上）。多亏了内核技术，此变换在数学上等效于使用特征图 φ 将训练集映射到无限维特征空间（右下），然后使用线性PCA将变换后的训练集投影到2D。

请注意，如果我们可以对一个给定的实例在缩小的空间中反转线性PCA，则重构点将位于特征空间中，而不是原始空间中（例如，如图中的X所示）。由于特征空间是无限维的，因此我们无法计算重构点，无法计算真实的重构误差。幸运的是，有可能在原始空间中找到一个点，该点将映射到重建点附近，这一点称为重建原像。一旦你有了原像，就可以测量其与原始实例的平方距离。然后可以选择内核和超参数，最大限度地减少此重构原像误差。

你可能想知道如何执行这个重构。一种解决方案是训练有监督的回归模型，其中将投影实例作为训练集，将原始实例作为目标值。如果设置fit_inverse_transform=True，Scikit-Learn会自动执行此操作，如以下代码所示[2]：

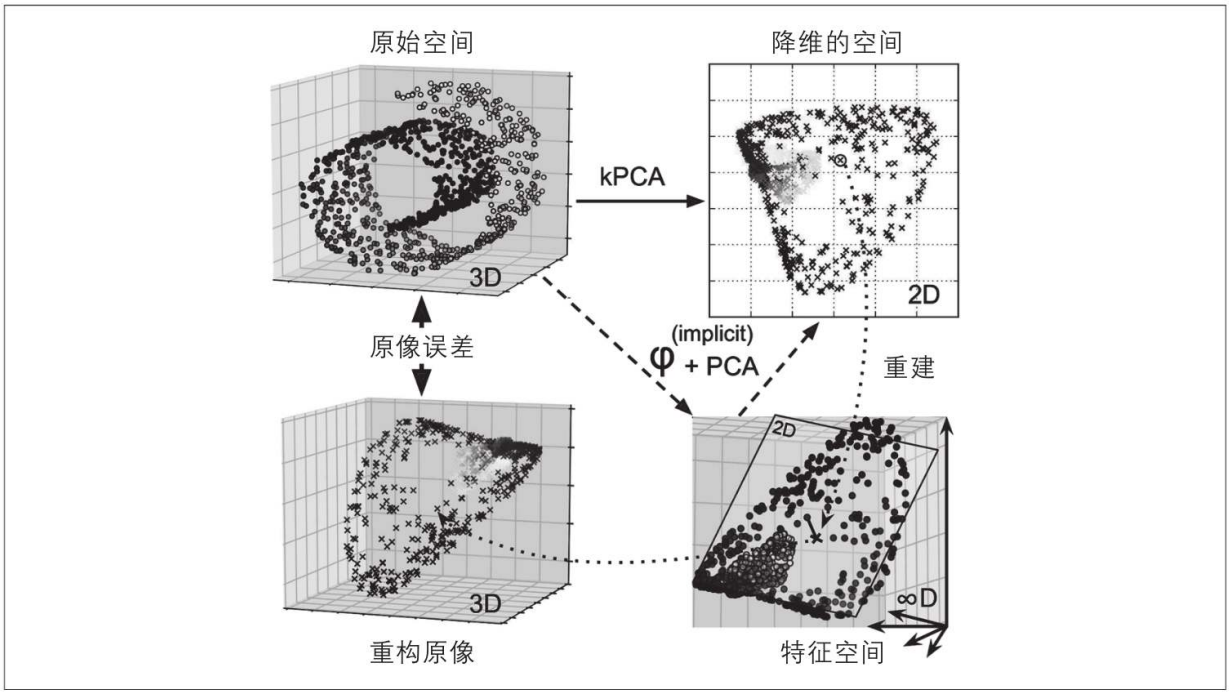


图8-11：内核PCA和重构原像误差

```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,
                    fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
```

 默认情况下，fit_inverse_transform=False，并且KernelPCA没有inverse_transform（）方法。仅当你设置fit_inverse_transform=True时，才会创建此方法。

计算重建原像误差：

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(X, X_preimage)
32.786308795766132
```

现在，你可以使用网格搜索与交叉验证来找到可最大限度减少此错误的内核和超参数。

[1] Bernhard Schölkopf et al.，“Kernel Principal Component Analysis”，in Lecture Notes in Computer Science 1327（Berlin：Springer，1997）：583－588.
[2] 如果设置fit_inverse_transform=True，Scikit-Learn将使用Gokhan H.Bakır等人的算法（基于Kernel Ridge Regression）。“Learning to Find Pre-Images”，Proceedings of the 16th International Conference on Neural Information Processing Systems（2004）：449－456。

8.5 LLE

局部线性嵌入（LLE）是另一种强大的非线性降维（NLDR）技术[1]。它是一种流形学习技术，不像以前的算法那样依赖于投影。简而言之，LLE的工作原理是首先测量每个训练实例如何与其最近的邻居（c.n.）线性相关，然后寻找可以最好地保留这些局部关系的训练集的低维表示形式（稍后会详细介绍）。这种方法特别适合于展开扭曲的流形，尤其是在没有太多噪声的情况下。

以下代码使用Scikit-Learn的LocallyLinearEmbedding类来展开瑞士卷：

```
from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```

生成的2D数据集如图8-12所示。如你所见，瑞士卷已完全展开，并且实例之间的距离在局部得到了很好的保留。但是，距离并没有在更大范围内保留：展开的瑞士卷的左侧部分被拉伸，而右侧部分被挤压。尽管如此，LLE在流形建模方面做得很好。

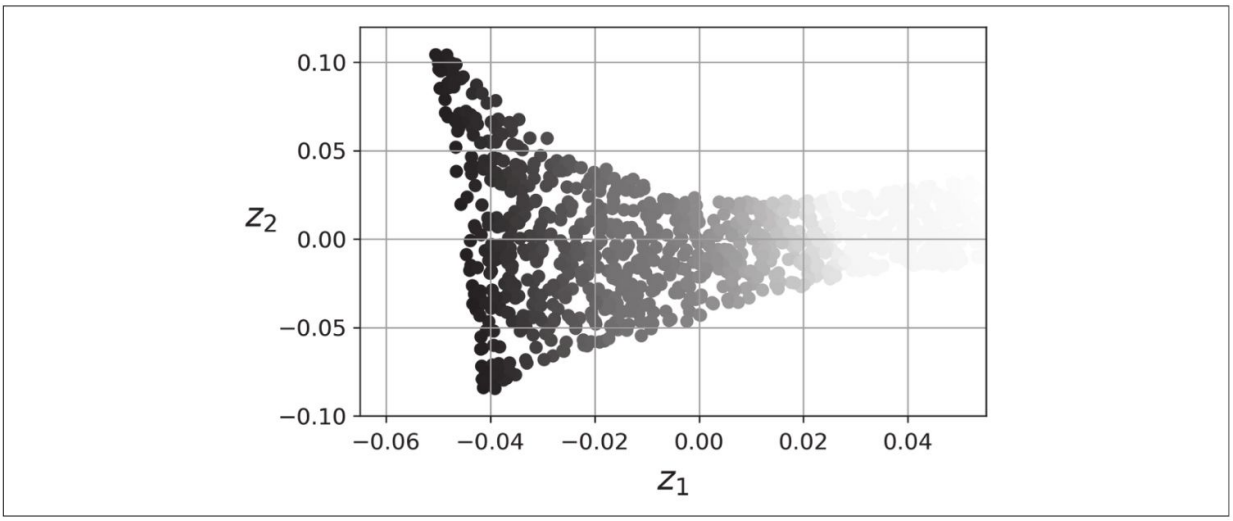


图8-12：使用LLE展开的瑞士卷

LLE的工作方式如下：对于每个训练实例$x^{(i)}$，算法都会识别出其k个最近的邻居（在前面的代码k=10中），然后尝试将$x^{(i)}$重构为这些邻居的线性函数。更具体地说，它找到权重$w_{i,j}$，使得$x^{(i)}$与$\sum_{j=1}^{m} w_{i,j} x^{(j)}$之间的平方距离尽可能小，并假设如果$x^{(j)}$不是$x^{(i)}$的k个最接近的邻居之一则$w_{i,j}=0$。因此，LLE的第一步是公式8-4中描述的约束优化问题，其中W是包含所有权重$w_{i,j}$的权重矩阵。第二个约束条件只是简单地将每个训练实例$x^{(i)}$的权重归一化。

公式8-4：LLE第一步：局部关系线性建模

$$\hat{\boldsymbol{W}} = \underset{\boldsymbol{W}}{\arg\min} \sum_{i=1}^{m} \left(\boldsymbol{x}^{(i)} - \sum_{j=1}^{m} w_{i,j} \boldsymbol{x}^{(j)} \right)^2$$

$$\text{满足} \begin{cases} w_{i,j} = 0 & x^{(j)} \text{不属于} x^{(i)} \text{的} k \text{ c.n.} \\ \sum_{j=1}^{m} w_{i,j} = 1 & \text{其中} i = 1, 2, \cdots, m \end{cases}$$

在此步骤之后，权重矩阵$\hat{\boldsymbol{W}}$（包含权重$\hat{w}_{i,j}$）对训练实例之间的局部线性关系进行编码。第二步是将训练实例映射到d维空间（其中d<n），同时尽可能保留这些局部关系。如果$z^{(i)}$是此d维空间中$x^{(i)}$的图像，则我们希望$z^{(i)}$与$\sum_{j=1}^{m} \hat{w}_{i,j} z^{(j)}$之间的平方距离尽可能小。

这种想法导致了公式8-5中描述的无约束优化问题。它看起来与第一步非常相似，但是我们没有保持实例固定并找到最佳权重，而是相反：保持权重固定并找到实例图像在低维空间中的最佳位置。注意Z是包含所有$z^{(i)}$的矩阵。

公式8-5：LLE第二步：在保持关系的同时减少维度

$$\hat{\boldsymbol{Z}} = \underset{\boldsymbol{Z}}{\arg\min} \sum_{i=1}^{m} \left(\boldsymbol{z}^{(i)} - \sum_{j=1}^{m} \hat{w}_{i,j} \boldsymbol{z}^{(j)} \right)^2$$

Scikit-Learn的LLE实现具有以下计算复杂度：$O(m \log(m) n \log(k))$用于找到k个最近的邻居，$O(mnk^3)$用于优化权重，$O(dm^2)$用于构造低维表示。不幸的是，最后一项中的m^2使该算法很难扩展到非常大的数据集。

[1] Sam T.Roweis and Lawrence K.Saul，“Nonlinear Dimensionality Reduction by Locally Linear Embedding”，Science 290，no.5500（2000）：2323－2326.

8.6 其他降维技术

还有许多其他降维技术，Scikit-Learn中提供了其中几种。以下是一些很受欢迎的降维技术：

随机投影

顾名思义，使用随机线性投影将数据投影到较低维度的空间。这听起来可能很疯狂，但事实证明，这样的随机投影实际上很有可能很好地保持距离，就如William B.Johnson和Joram Lindenstrauss在著名引理中的数学证明。降维的质量取决于实例数目和目标维度，令人惊讶的不取决于初始维度。请查看sklearn.random_projection软件包的文档以获取更多详细信息。

多维缩放（MDS）

当尝试保留实例之间的距离时降低维度。

Isomap

通过将每个实例与其最近的邻居连接来创建一个图，然后在尝试保留实例之间的测地距离[1]的同时降低维度。

t分布随机近邻嵌入（t-SNE）

降低了维度，同时使相似实例保持接近，异类实例分开。它主要用于可视化，特别是在高维空间中可视化实例的聚类（例如，以2D可视化MNIST图像）。

线性判别分析（LDA）

LDA是一种分类算法，但是在训练过程中，它会学习各类之间最有判别力的轴，然后可以使用这些轴来定义要在其上投影数据的超平面。这种方法的好处是投影将使类保持尽可能远的距离，因此LDA是在运行其他分类算法（例如SVM分类器）之前降低维度的好技术。

图8-13显示了其中一些技术的结果。

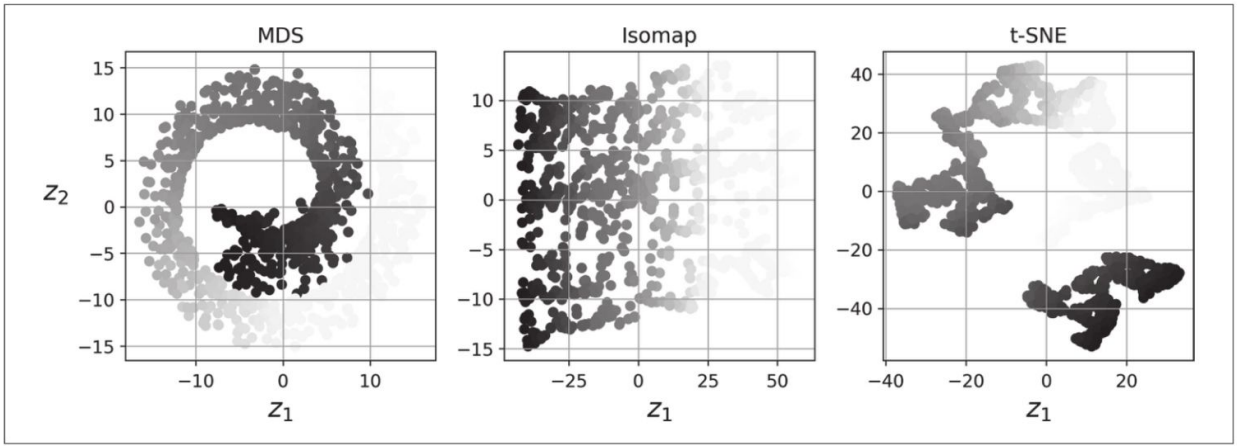


图8-13：使用各种技术将瑞士卷降至2D

[1] 一个图中两个节点之间的测地距离是这些节点之间最短路径上的节点数。

8.7 练习题

1. 减少数据集维度的主要动机是什么？主要缺点是什么？

2. 维度的诅咒是什么？

3. 一旦降低了数据集的维度，是否可以逆操作？如果可以，怎么做？如果不能，为什么？

4. 可以使用PCA来减少高度非线性的数据集的维度吗？

5. 假设你在1000维的数据集上执行PCA，将可解释方差比设置为95%。结果数据集将具有多少个维度？

6. 在什么情况下，你将使用常规PCA、增量PCA、随机PCA或内核PCA？

7. 如何评估数据集中的降维算法的性能？

8. 链接两个不同的降维算法是否有意义？

9. 加载MNIST数据集（在第3章中介绍），并将其分为训练集和测试集（使用前60 000个实例进行训练，其余10 000个进行测试）。在数据集上训练随机森林分类器，花费多长时间，然后在测试集上评估模型。接下来，使用PCA来减少数据集的维度，可解释方差率为95%。在精简后的数据集上训练新的随机森林分类器，查看花费了多长时间。训练速度提高了吗？接下来，评估测试集上的分类器。与之前的分类器相比如何？

10.使用t-SNE将MNIST数据集降至两个维度，然后用Matplotlib绘制结果。你可以通过散点图用10个不同的颜色来代表每个图像的目标类，或者也可以用对应实例的类（从0到9的数字）替换散点图中的每个点，甚至你还可以绘制数字图像本身的缩小版（如果你绘制所有数字，视觉效果会太凌乱，所以你要么绘制一个随机样本，要么选择单个实例，但是这个实例的周围最好没有其他绘制的实例）。现在你应该得到了一个很好的可视化结果及各自分开的数字集群。尝试使用其他降维算法，如PCA、LLE或MDS等，比较可视化结果。

附录A中提供了这些练习题的解答。

第9章　无监督学习技术

尽管今天机器学习的大多数应用都是基于有监督学习的（因此，这是大多数投资的方向），但是绝大多数可用数据都没有标签：我们具有输入特征X，但是没有标签y。计算机科学家Yann LeCun曾有句著名的话：“如果智能是蛋糕，无监督学习将是蛋糕本体，有监督学习是蛋糕上的糖霜，强化学习是蛋糕上的樱桃。”换句话说，无监督学习具有巨大的潜力，我们才刚刚开始研究。

假设你要创建一个系统，该系统将在制造生产线上为每个产品拍摄几张图片，并检测哪些产品有缺陷。你可以相当容易地创建一个自动拍照系统，这可能每天为你提供数千张图片。然后，你可以在几周内构建一个相当大的数据集。但是，等等，没有标签！如果你想训练一个常规的二元分类器来预测某件产品是否有缺陷，则需要将每张图片标记为“有缺陷”或“正常”。这通常需要人类专家坐下来并手动浏览所有图片。这是一项漫长、昂贵且烦琐的任务，因此通常只能在可用图片的一部分上完成。因此，标记的数据集将非常小，并且分类器的性能将令人失望。而且，公司每次对其产品进行任何更改时，都需要从头开始整个过程。如果该算法只需要利用未标记的数据而无须人工标记每张图片，那不是很好吗？让我们进入无监督学习。

在第8章中，我们研究了最常见的无监督学习任务：降维。在本章中，我们将研究其他一些无监督的学习任务和算法：

聚类

目标是将相似的实例分组到集群中。聚类是很好的工具，用于数据分析、客户细分、推荐系统、搜索引擎、图像分割、半监督学习、降维等。

异常检测

目的是学习“正常”数据看起来是什么样的，然后将其用于检测异常情况，例如生产线上的缺陷产品或时间序列中的新趋势。

密度估算

这是估计生成数据集的随机过程的概率密度函数（PDF）的任务，密度估算通常用于异常检测：位于非常低密度区域的实例很可能是异常。它对于数据分析和可视化也很有用。

准备好蛋糕了吗？我们将从使用K-Means和DBSCAN进行聚类开始，然后讨论高斯混合模型，并了解如何将它们用于密度估计、聚类和异常检测。

9.1 聚类

你在山中徒步旅行时，偶然发现了从未见过的植物。你环顾四周，发现还有很多。它们并不完全相同，但是它们足够相似，你可能知道它们有可能属于同一物种（或至少属于同一属）。你可能需要植物学家告诉你什么是物种，但你当然不需要专家来识别外观相似的物体组。这称为聚类：识别相似实例并将其分配给相似实例的集群或组。

就像在分类中一样，每个实例都分配给一个组。但是与分类不同，聚类是一项无监督任务。考虑图9-1：左侧是鸢尾花数据集（在第4章中介绍），其中每个实例的种类（即类）用不同的标记表示。它是一个标记的数据集，非常适合使用逻辑回归、SVM或随机森林分类器等分类算法。右侧是相同的数据集，但是没有标签，因此你不能再使用分类算法。这就是聚类算法的引入之处，它们中的许多算法都可以轻松检测左下角的集群。肉眼也很容易看到，但是右上角的集群由两个不同的子集群组成，并不是很明显。也就是说，数据集具有两个附加特征（萼片长度和宽度），此处未表示，并且聚类算法可以很好地利用所有的特征，因此实际上它们可以很好地识别三个聚类（例如，使用高斯混合模型，在150个实例中，只有5个实例分配给错误的集群）。

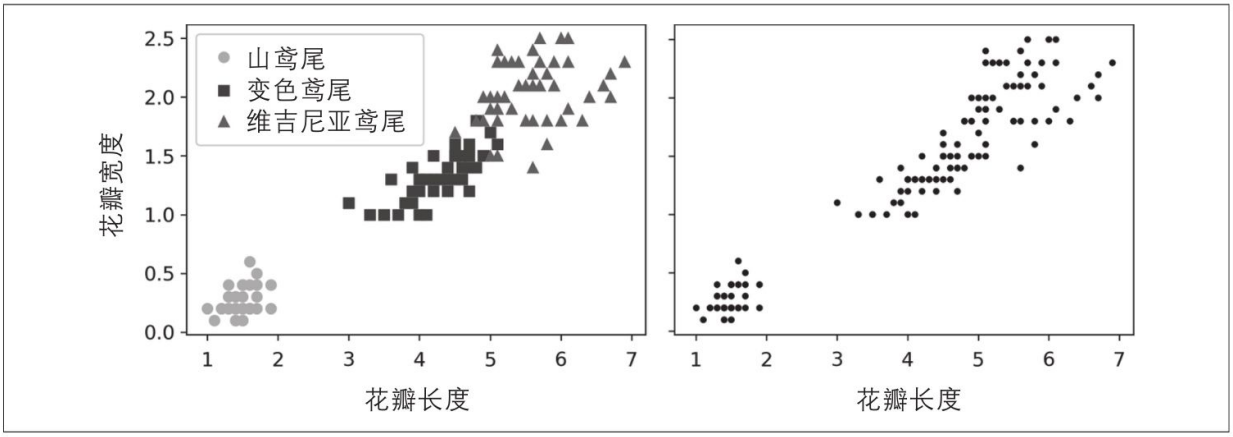


图9-1：分类（左）与聚类（右）

聚类可用于各种应用程序，包括：

客户细分

你可以根据客户的购买记录和他们在网站上的活动对客户进行聚类。这对于了解你的客户是谁以及他们的需求很有用，因此你可以针对每个细分客户调整产品和营销活动。例如，客户细分在推荐系统中可以很有用，可以推荐同一集群中其他用户喜欢的内容。

数据分析

在分析新数据集时，运行聚类算法然后分别分析每个集群。

降维技术

数据集聚类后，通常可以测量每个实例与每个集群的相似度（相似度是衡量一个实例和一个集群的相似程度）。然后可以将每个实例的特征向量x替换为其集群的向量。如果有k个集群，则此向量为k维。此向量的维度通常比原始特征向量低得多，但它可以保留足够的信息以进行进一步处理。

异常检测（也称为离群值检测）

对所有集群具有低相似度的任何实例都可能是异常。例如，如果你已根据用户行为对网站用户进行了聚类，则可以检测到具有异常行为的用户，例如每秒的请求数量异常。异常检测在检测制造生产线中的缺陷或欺诈检测中特别有用。

半监督学习

如果你只有几个标签，则可以执行聚类并将标签传播到同一集群中的所有实例。该技术可以大大增加可用于后续有监督学习算法的标签数

量，从而提高其性能。

搜索引擎

一些搜索引擎可让你搜索与参考图像相似的图像。要构建这样的系统，首先要对数据库中的所有图像应用聚类算法，相似的图像最终会出现在同一集群中。然后，当用户提供参考图像时，你需要做的就是使用训练好的聚类模型找到该图像的集群，然后可以简单地从该集群中返回所有的图像。

分割图像

通过根据像素的颜色对像素进行聚类，然后用其聚类的平均颜色替换每个像素的颜色，可以显著减少图像中不同颜色的数量。图像分割用于许多物体检测和跟踪系统中，因为它可以更轻松地检测每个物体的轮廓。

关于聚类什么没有统一的定义，它实际上取决于上下文，并且不同的算法会得到不同种类的集群。一些算法会寻找围绕特定点（称为中心点）的实例。其他人则寻找密集实例的连续区域：这些集群可以呈现任何形状。一些算法是分层的，寻找集群中的集群。这样的示例不胜枚举。

在本节中，我们将研究两种流行的聚类算法——K-Means和DBSCAN，并探讨它们的一些应用，例如非线性降维、半监督学习和异常检测。

9.1.1 K-Means

考虑图9-2中所示的未标记数据集：你可以清楚地看到5组实例。K-Means算法是一种简单的算法，能够非常快速、高效地对此类数据集进行聚类，通常只需几次迭代即可。它是由贝尔实验室的Stuart Lloyd在

1957年提出的，用于脉冲编码调制，但直到1982[1]年才对外发布。1965年，Edward W.Forgy发布了相同的算法，因此K-Means有时被称为Lloyd－Forgy。

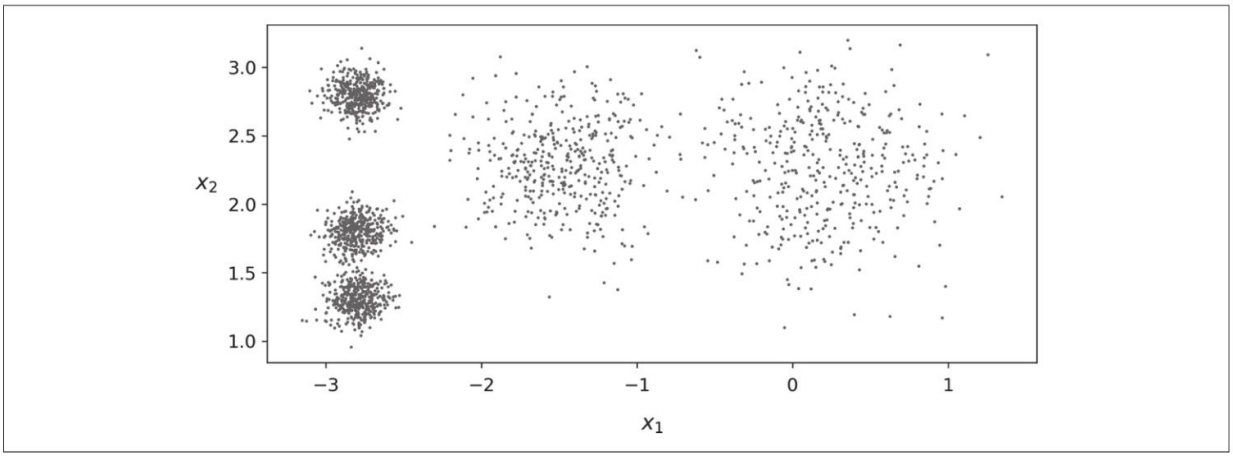


图9-2：由5组实例集群组成的未标记数据集

让我们在该数据集上训练一个K-Means聚类器。它将尝试找到每个集群的中心，并将每个实例分配给最近的集群：

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)
```

请注意，你必须指定这个算法必须要找到的集群数k。在此例中，通过查看数据可以明显看出k应该设置为5，但总的来说并不是那么容易。我们会简短讨论这个问题。

每个实例都会分配给5个集群之一。在聚类的上下文中，实例的标签是该实例被算法分配给该集群的索引：不要与分类中的类标签相混淆（请记住，聚类是无监督学习任务）。KMeans实例保留了经过训练的实例的标签副本，可通过labels_实例变量得到该副本：

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
>>> y_pred is kmeans.labels_
True
```

我们还可以看一下算法发现的5个中心点：

```
>>> kmeans.cluster_centers_
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

可以很容易地将新实例分配给中心点最接近的集群：

```
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 2, 2], dtype=int32)
```

如果绘制集群的边界，则会得到Voronoi图（参见图9-3，其中每个中心点都用×表示）。

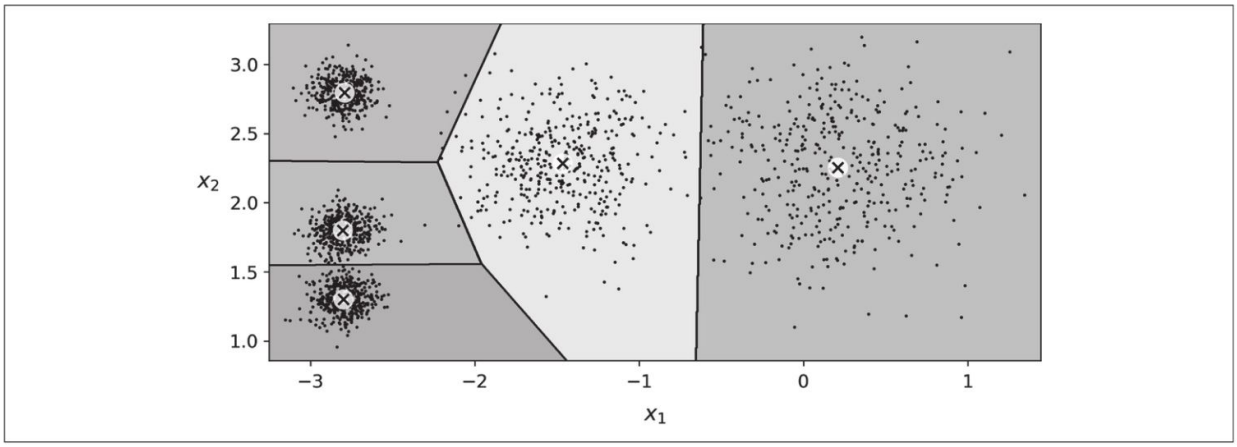


图9-3：K-Means决策边界（Voronoi图）

绝大多数实例已正确分配给适当的集群，但少数实例可能贴错了标签（尤其是在左上集群和中央集群之间的边界附近）。的确，当集群具有不同的直径时，K-Means算法的性能不是很好，因为将实例分配给某个集群时，它所关心的只是与中心点的距离。

与其将每个实例分配给一个单独的集群（称为硬聚类），不如为每个实例赋予每个集群一个分数（称为软聚类）会更有用。分数可以是实例与中心点之间的距离。相反，它可以是相似性分数（或远近程度），例如高斯径向基函数（在第5章中介绍）。在KMeans类中，transform（）方法测量每个实例到每个中心点的距离：

```
>>> kmeans.transform(X_new)
array([[2.81093633, 0.32995317, 2.9042344 , 1.49439034, 2.88633901],
       [5.80730058, 2.80290755, 5.84739223, 4.4759332 , 5.84236351],
       [1.21475352, 3.29399768, 0.29040966, 1.69136631, 1.71086031],
       [0.72581411, 3.21806371, 0.36159148, 1.54808703, 1.21567622]])
```

在此例中，X_new中的第一个实例与第一个中心点的距离为2.81，与第二个中心点的距离为0.33，与第三个中心点的距离为2.90，与第四个中心点的距离为1.49，与第五个中心点的距离为2.89。如果你有一个高维数据集并以这种方式进行转换，那么最终将得到一个k维数据集：这种转换是一种非常有效的非线性降维技术。

K-Means算法

那么，该算法如何工作？好吧，假设你得到了中心点。你可以通过将每个实例分配给中心点最接近的集群来标记数据集中的所有实例。相反，如果你有了所有实例的标签，则可以通过计算每个集群的实例平均值来定位所有的中心点。但是你既没有标签也没有中心点，那么如何进行呢？好吧，只要从随机放置中心点开始（例如，随机选择k个实例并将其位置用作中心点）。然后标记实例，更新中心点，标记实例，更新

中心点，以此类推，直到中心点停止移动。这个算法能保证在有限数量的步骤内收敛（通常很小），不会永远振荡[2]。

你可以在图9-4中看到正在使用的算法：中心点被随机初始化（左上），然后实例被标记（右上），中心点被更新（左中心），实例被重新标记（右中），以此类推。如你所看到的，仅仅三个迭代，该算法就达到了近似于最佳状态的聚类。

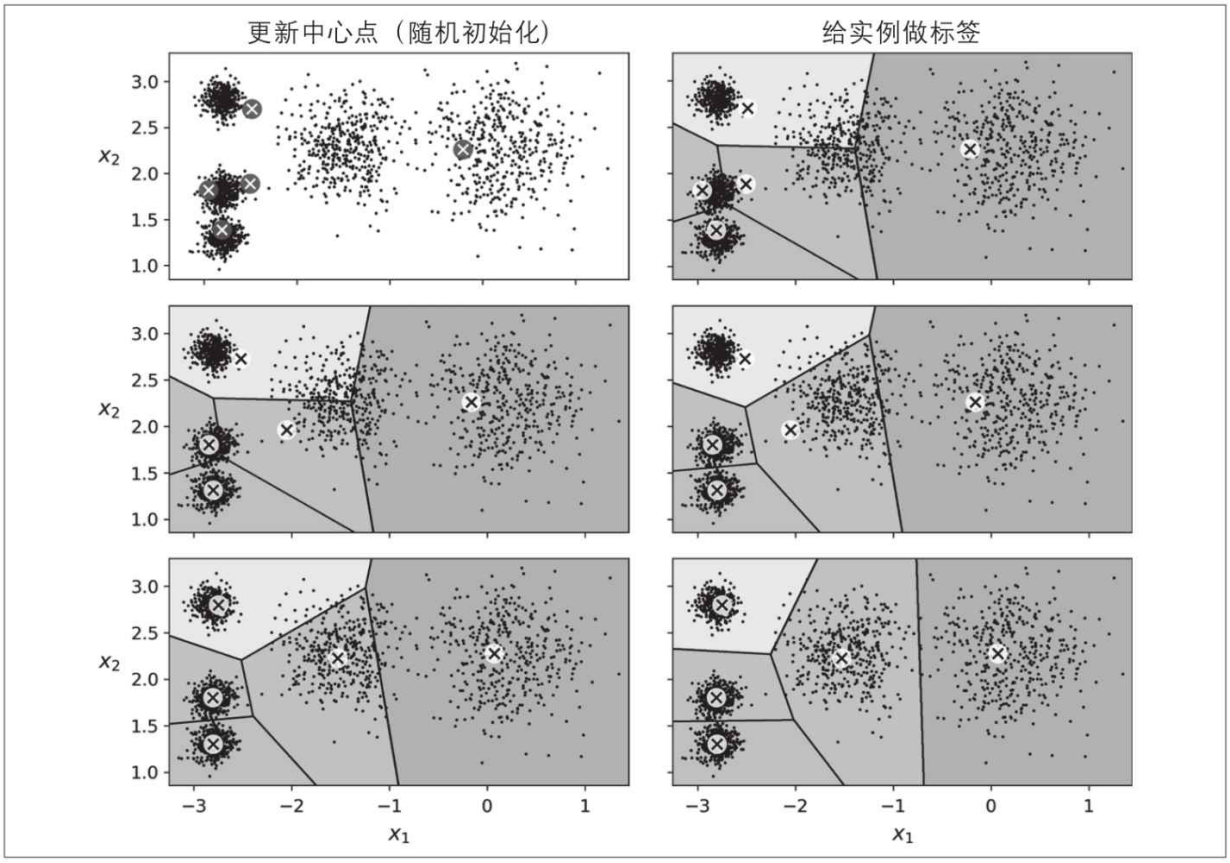


图9-4：K-Means算法

 该算法的计算复杂度在实例数m，集群数k和维度n方面通常是线性的。但是，仅当数据具有聚类结构时才如此。如果不是这样，那么在最坏的情况下，复杂度会随着实例数量的增加而呈指数增加。实际上，这种情况很少发生，并且K-Means通常是最快的聚类算法之一。

尽管算法保证会收敛，但它可能不会收敛到正确解（即可能会收敛到局部最优值）：这取决于中心点的初始化。图9-5显示了两个次优解，如果你不幸地使用随机初始化步骤，算法会收敛到它们。

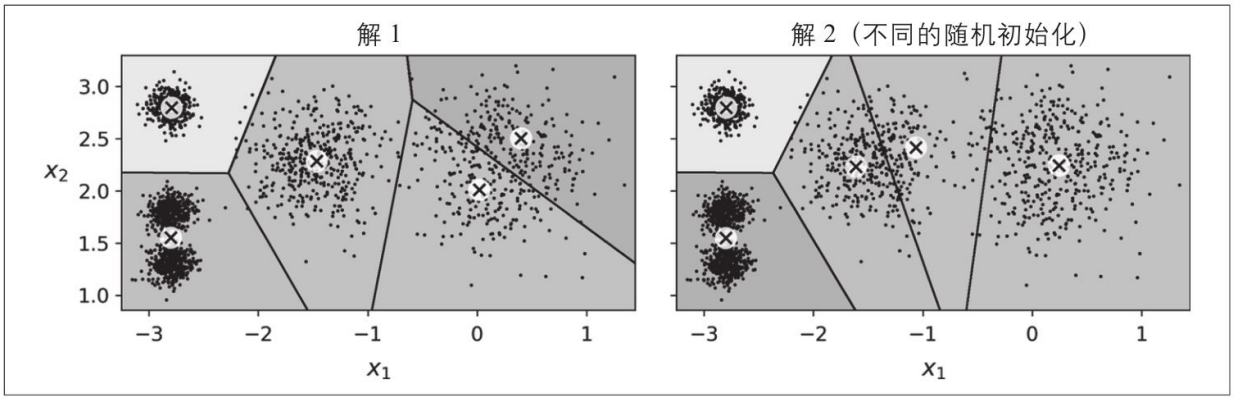


图9-5：由于中心点初始化不好而导致的次优解

让我们看一下通过改善中心点初始化来减低这种风险的几种方法。

中心点初始化方法

如果你碰巧知道了中心点应该在哪里（例如，你之前运行了另一种聚类算法），则可以将超参数init设置为包含中心点列表的NumPy数组，并将n_init设置为1：

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1)
```

另一种解决方案是使用不同的随机初始化多次运行算法，并保留最优解。随机初始化的数量由超参数n_init控制：默认情况下，它等于10，这意味着当你调用fit（）时，前述算法运行10次，Scikit-Learn会保留最优解。但是，如何确切知道哪种解答是最优解呢？它使用性能指标！这个指标称为模型的惯性，即每个实例与其最接近的中心点之间的均方距离。对于图9-5中左边的模型，它大约等于223.3；对于图9-5

中右边的模型，它等于237.5；对于图9-3中的模型，它等于211.6。KMeans类运行n_init次算法，并保留模型的最低惯性。在此例中，将选择图9-3中的模型（除非我们非常不幸地使用n_init连续随机初始化）。如果你有意，可以通过inertia_instance变量来访问模型的惯性：

```
>>> kmeans.inertia_
211.59853725816856
```

score（）方法返回负惯性。为什么是负的？因为预测器的score（）方法必须始终遵循Scikit-Learn的“越大越好”的规则，即如果一个预测器比另一个更好，则其score（）方法应该返回更大的数：

```
>>> kmeans.score(X)
-211.59853725816856
```

David Arthur和Sergei Vassilvitskii[3]在2006年的论文中提出了对K-Means算法的重要改进——K-Means++。他们引入了一种更智能的初始化步骤，该步骤倾向于选择彼此相距较远的中心点，这一改进使得K-Means算法收敛到次优解的可能性很小。他们表明，更智能的初始化步骤所需的额外计算量是值得的，因为它可以大大减少寻找最优解所需运行算法的次数。以下是K-Means++的初始化算法：

1.取一个中心点$c^{(1)}$，从数据集中随机选择一个中心点。

2.取一个新中心点$c^{(i)}$，选择一个概率为$D(\boldsymbol{x}^{(i)})^2 / \sum_{j=1}^{m} D(\boldsymbol{x}^{(j)})^2$的实例$x^{(i)}$，其中$D(x^{(i)})$是实例$x^{(i)}$与已经选择的最远中心点的距

离。这种概率分布确保距离已选择的中心点较远的实例有更大可能性被选择为中心点。

3.重复上一步，直到选择了所有k个中心点。

默认情况下，KMeans类使用这种初始化方法。如果要强制使用原始方法（即随机选择k个实例来初始化中心点），则可以将超参数init设置为"random"。你很少需要这样做。

加速的K-Means和小批量K-Means

Charles Elkan[4]在2003年的一篇论文中提出了对K-Means算法的另一个重要改进。该改进后的算法通过避免许多不必要的距离计算，大大加快了算法的速度。Elkan通过利用三角不等式（即一条直线是两点之间的最短距离[5]）并通过跟踪实例和中心点之间的上下限距离来实现这一目的。这是KMeans类默认使用的算法（你可以通过将超参数algorithm设置为"full"来强制使用原始算法，尽管你可能永远也不需要）。

David Sculley在2010年的一篇论文中提出了K-Means算法的另一个重要变体[6]。该算法能够在每次迭代中使用小批量K-Means稍微移动中心点，而不是在每次迭代中使用完整的数据集。这将算法的速度提高了3到4倍，并且可以对不容纳内存的大数据集进行聚类。Scikit-Learn在MiniBatchKMeans类中实现了此算法。你可以像KMeans类一样使用此类：

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(X)
```

如果数据集无法容纳在内存中，则最简单的选择是使用memmap类，就像我们在第8章中对增量PCA所做的那样。或者你可以一次将一个小批量传给partial_fit（）方法，但这需要更多的工作，因为你需要执行多次初始化并选择一个最好的初始化（有关示例请参阅notebook的小批量处理K-Means部分）。

尽管小批量K-Means算法比常规K-Means算法快得多，但其惯性通常略差一些，尤其是随着集群的增加。你可以在图9-6中看到这一点：左侧的图比较了小批量K-Means和常规K-Means模型的惯性，这是在先前数据集上使用各种不同k训练而来的。两条曲线之间的差异保持相当恒定，但是随着k的增加，该差异变得越来越大，因为惯性变得越来越小。在右边的图中，你可以看到小批量K-Means比常规K-Means快得多，并且该差异随k的增加而增加。

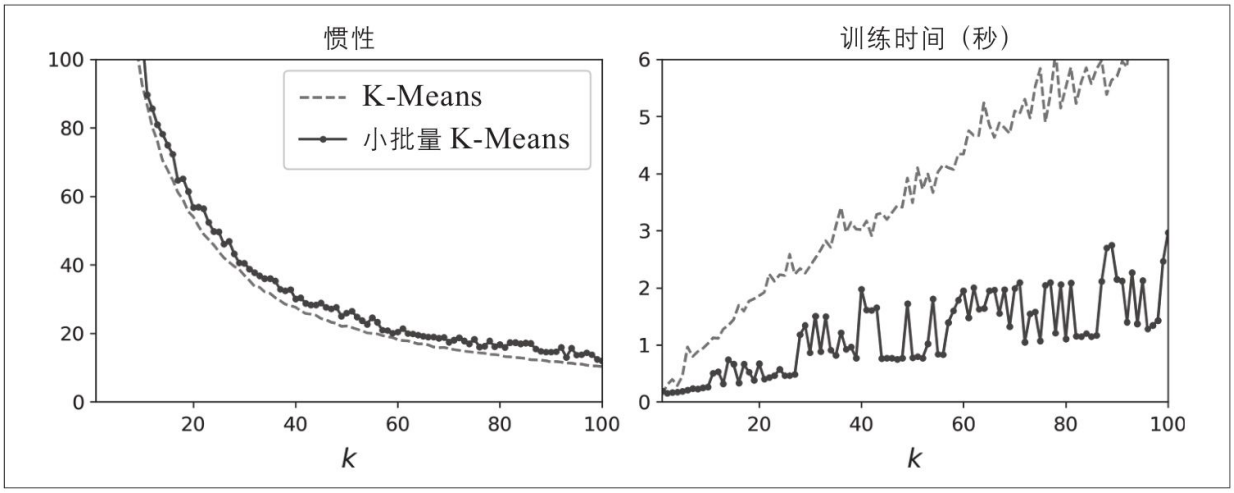


图9-6：小批量K-Means的惯性要高于K-Means（左），但要快得多（右），尤其是当k增加时

寻找最佳聚类数

到目前为止，我们将集群数k设置为5，因为通过查看数据可以明显看出这是正确的集群数。但是总的来说，知道如何设置k并不容易，如

果将k设置为错误值，结果可能会很糟糕。如图9-7所示，将k设置为3或8会导致非常糟糕的模型。

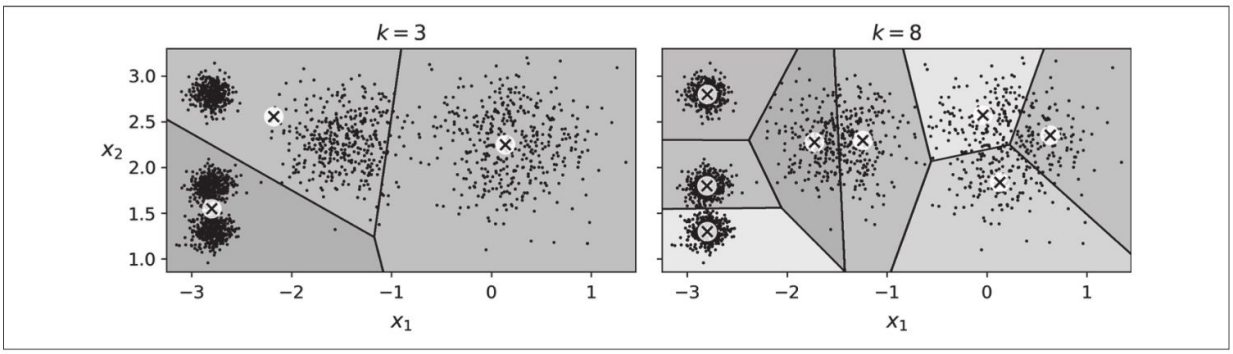


图9-7：集群数量的错误选择：当k太小时，单独的集群会合并（左），而当k太大时，某些集群会被分成多个（右）

你可能会认为我们可以选择惯性最低的模型，对吗？不是那么简单！k=3的惯性是653.2，比k=5（211.6）大得多。但是在k=8的情况下，惯性仅为119.1。尝试选择k时，惯性不是一个好的性能指标，因为随着k的增加，惯性会不断降低。实际上，集群越多，每个实例将越接近其最接近的中心点，因此惯性将越低。让我们将惯性作为k的函数作图（见图9-8）。

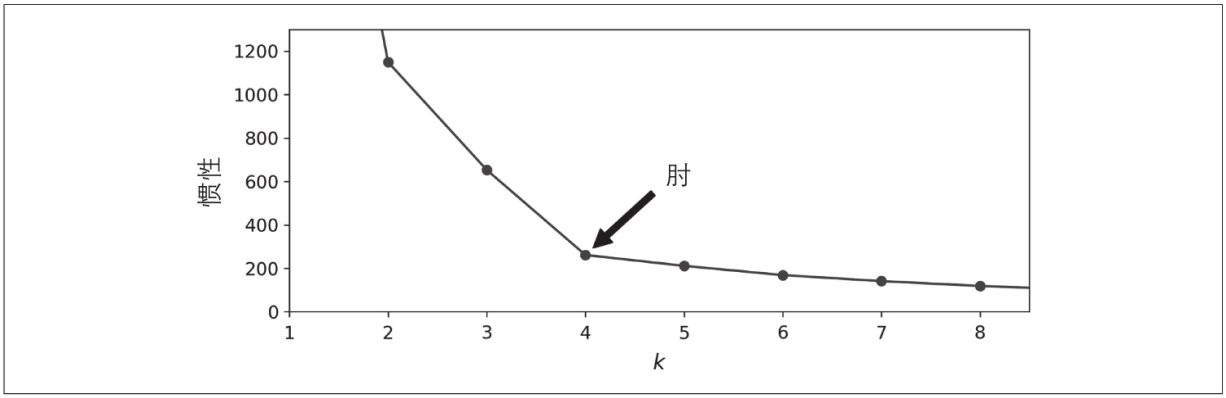


图9-8：当绘制惯性作为集群k的函数曲线时，曲线通常包含一个称为“肘”的拐点

如你所见，随着k增大到4，惯性下降得很快，但随着k继续增大，惯性下降得很慢。该曲线大致像手臂形状，在k=4处有一个“肘”形。因此，如果我们不知道更好的选择，则4是一个不错的选择：较低的值变化比较大，而较高的值没有多大帮助，我们有时可以把集群再细分成两半。

为集群数选择最佳值的技术相当粗糙。一种更精确的方法（但也需要更多的计算）是使用轮廓分数，它是所有实例的平均轮廓系数。实例的轮廓系数等于（b-a）/max（a，b），其中a是与同一集群中其他实例的平均距离（即集群内平均距离），b是平均最近集群距离（即到下一个最近集群实例的平均距离，定义为使b最小的那个，不包括实例自身的集群）。轮廓系数可以在-1和+1之间变化。接近+1的系数表示该实例很好地位于其自身的集群中，并且远离其他集群，而接近0的系数表示该实例接近一个集群的边界，最后一个接近-1的系数意味着该实例可能已分配给错误的集群。

要计算轮廓分数，可以使用Scikit-Learn的silhouette_score（）函数，为它输入数据集中的所有实例以及为其分配的标签：

```
>>> from sklearn.metrics import silhouette_score
>>> silhouette_score(X, kmeans.labels_)
0.655517642572828
```

让我们比较不同集群数量的轮廓分数（见图9-9）。

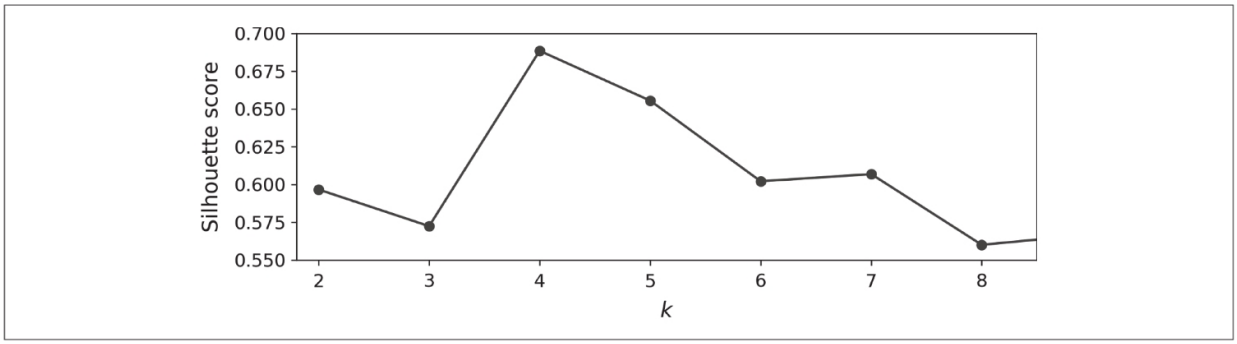


图9-9：使用轮廓分数选择集群数k

如你所见，这个可视化效果比上一个更加丰富：尽管它确认k=4是一个很好的选择，但它也说明一个事实，即k=5也非常好，并且比k=6或者k=7要好得多。这个在比较惯性时看不到。

当你绘制每个实例的轮廓系数时，可以获得更加丰富的可视化效果，这些轮廓系数按它们分配的集群和系数值进行排序，这称为轮廓图（见图9-10）。每个图包含一个刀形的聚类。形状的高度表示集群包含的实例数，宽度表示集群中实例的排序轮廓系数（越宽越好）。虚线表示平均轮廓系数。

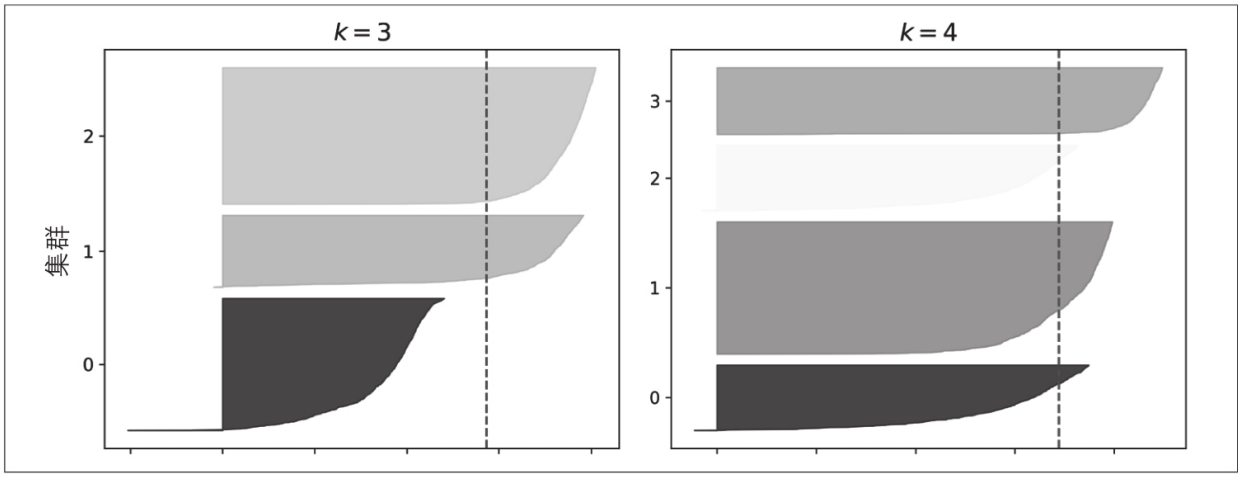


图9-10：各种K值的轮廓图分析

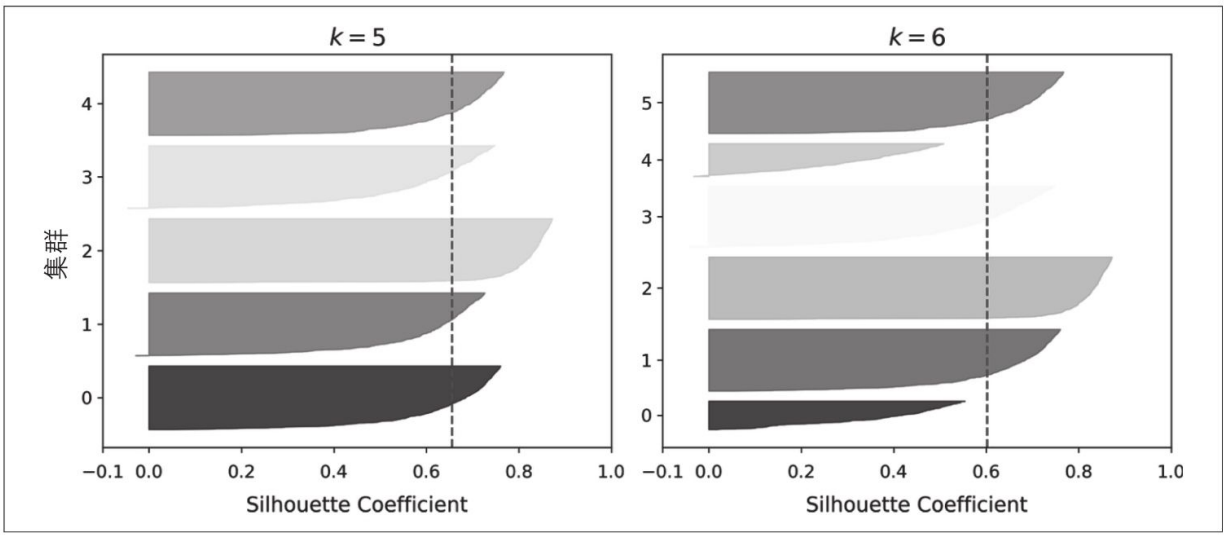


图9-10：各种K值的轮廓图分析（续）

垂直虚线表示每个集群的轮廓分数。当集群中的大多数实例的系数均低于此分数时（如果许多实例在虚线附近停止，在其左侧结束），则该集群比较糟糕，因为这意味着其实例太接近其他集群了。可以看到，当k=3或者k=6时，我们得到了不好的集群。但是当k=4或k=5时，集群看起来很好：大多数实例都超出虚线，向右延伸并接近1.0。当k=4时，索引为1（从顶部开始的第三个）的集群很大。当k=5时，所有集群的大小都相似。因此，即使k=4的整体轮廓得分略大于k=5的轮廓得分，使用k=5来获得相似大小的集群似乎也是一个好主意。

9.1.2 K-Means的局限

尽管K-Means有许多优点，尤其是快速且可扩展，但它并不完美。如我们所见，必须多次运行该算法才能避免次优解，此外，你还需要指定集群数，这很麻烦。此外，当集群具有不同的大小、不同的密度或非球形时，K-Means的表现也不佳。例如，图9-11显示了K-Means如何对包含三个不同尺寸、密度和方向的椭圆形集群的数据集进行聚类。

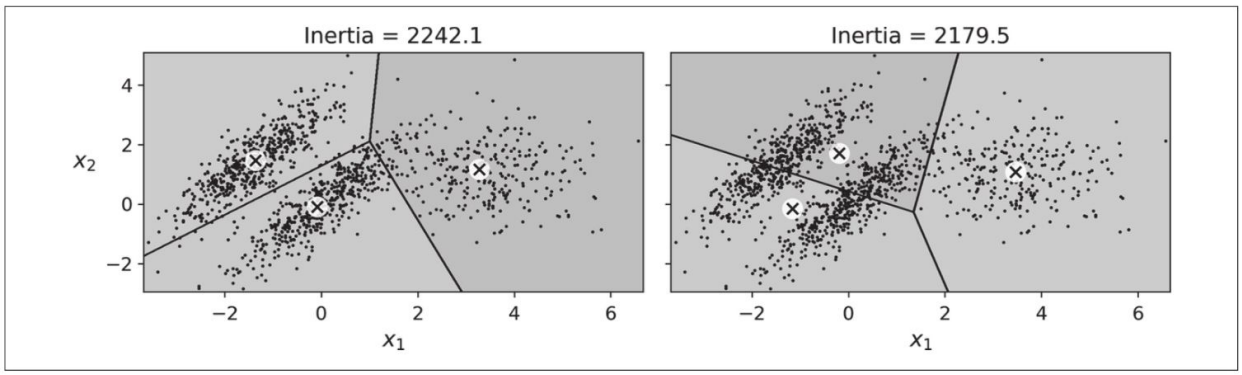


图9-11：K-Means无法正确聚类这些椭圆形集群

如你所见，这些聚类的结果都不好。左边的结果好一点，但是它仍然砍掉了中间集群的25%，并将其分配给右边的集群。即使其惯性较

低，右侧的结果也很糟糕。因此，根据数据，不同的聚类算法可能会表现好一点。在这种类型的椭圆簇上，高斯混合模型非常有效。

 在运行K-Means之前，请先缩放输入特征，否则集群可能会变形，这样K-Means的性能会很差。缩放特征并不能保证所有的集群都很好，但是通常可以改善结果。

现在让我们看一些可以从聚类中受益的方法。我们将使用K-Means，但可以随时试验其他的聚类算法。

9.1.3 使用聚类进行图像分割

图像分割是将图像分成多个分割的任务。在语义分割中，属于同一对象类型的所有像素均被分配给同一分割。例如，在无人驾驶汽车的视觉系统中，可能会将行人图像中的所有像素都分配给“行人”（会有一个包含所有行人的分割）。在实例分割中，属于同一单个对象的所有像素都分配给同一分割。在这种情况下，每个行人会有不同的分割。如今，使用最新技术的基于卷积神经网络的复杂结构可以实现语义分割或实例分割（见第14章）。在这里，我们做一些简单的事情：颜色分割。如果像素具有相似的颜色，我们将简单地将它们分配给同一分割。在某些应用中，这可能就足够了。例如，如果要分析卫星图像以测量某个区域中有多少森林总面积，则颜色分割就很好了。

首先，使用Matplotlib的imread（）函数加载图像（见图9-12的左上方图像）：

```
>>> from matplotlib.image import imread  # or `from imageio import imread`
>>> image = imread(os.path.join("images","unsupervised_learning","ladybug.png"))
>>> image.shape
(533, 800, 3)
```

图像表示为3D数组。第一维的大小是高度，第二个是宽度，第三个是颜色通道的数量，在这种情况下为红色、绿色和蓝色（RGB）。换句话说，对于每个像素都有一个3D向量，包含红色、绿色和蓝色的强度，强度在0.0和1.0之间（如果使用imageio.imread（），则在0和255之间）。某些图像可能具有较少的通道，例如灰度图像（一个通道）。某些图像可能具有更多的通道，例如具有透明度的alpha通道的图像或卫星图像，这些图像通常包含许多光频率（例如红外）的通道。以下代码对数组进行重构以得到RGB颜色的长列表，然后使用K-Means将这些颜色聚类：

```
X = image.reshape(-1, 3)
kmeans = KMeans(n_clusters=8).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)
```

例如，它可以识别所有绿色阴影的颜色聚类。接下来，对于每种颜色（例如深绿色），它会寻找像素颜色集群的平均颜色。例如，所有绿色阴影都可以用相同的浅绿色代替（假设绿色集群的平均颜色是浅绿色）。最后，它会重新调整颜色的长列表，使其形状与原始图像相同。我们完成了！

这样就输出了图9-12右上方所示的图像。你可以尝试各种数量的集群，如图9-12所示。当你使用少于8个集群时，请注意，瓢虫的闪亮红色不能聚成单独的一类，它将与环境中的颜色合并。这是因为K-Means更喜欢相似大小的集群。瓢虫很小，比图像的其余部分要小得多，所以即使它的颜色是鲜明的，K-Means也不能为它指定一个集群。

图9-12：使用各种数量的颜色聚类K-Means进行图像分割

不是很难，不是吗？现在让我们看一下聚类的另一个应用：预处理。

9.1.4 使用聚类进行预处理

聚类是一种有效的降维方法，特别是作为有监督学习算法之前的预处理步骤。作为使用聚类进行降维的示例，让我们处理数字数据集，它是一个类似于MNIST的简单数据集，其中包含了表示从0到9数字的1797个灰度8×8图像。首先，加载数据集：

```
from sklearn.datasets import load_digits
X_digits, y_digits = load_digits(return_X_y=True)
```

将其分为训练集和测试集：

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits)
```

拟合一个逻辑回归模型：

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
```

在测试集上评估其精度：

```
>>> log_reg.score(X_test, y_test)
0.9688888888888889
```

这就是我们的基准：准确率为96.9%。让我们看看通过使用K-Means作为预处理是否可以做得更好。我们创建一个流水线，该流水线首先将训练集聚类为50个集群，并将图像替换为其与这50个集群的距离，然后应用逻辑回归模型：

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50)),
    ("log_reg", LogisticRegression()),
])
pipeline.fit(X_train, y_train)
```

 由于有10个不同的数字，因此很容易将集群数量设置为10。但是，每个数字可以用几种不同的方式写入，因此最好使用更大的集群数量，例如50。评估这个分类流水线：

```
>>> pipeline.score(X_test, y_test)
0.9777777777777777
```

怎么样？我们把错误率降低了近30%（从大约3.1%降低到大约2.2%）！

但是我们是任意选择集群k，我们当然可以做得更好。由于K-Means只是分类流水线中的预处理步骤，因此为k找到一个好的值要比以前简单得多。无须执行轮廓分析或最小化惯性。k的最优解是在交叉验证过程中的最佳分类性能的值。我们可以使用GridSearchCV来找到集群的最优解：

```
from sklearn.model_selection import GridSearchCV

param_grid = dict(kmeans__n_clusters=range(2, 100))
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)
grid_clf.fit(X_train, y_train)
```

让我们看一下k的最优解和流水线的性能：

```
>>> grid_clf.best_params_
{'kmeans__n_clusters': 99}
>>> grid_clf.score(X_test, y_test)
0.9822222222222222
```

在k=99个聚类的情况下，我们获得了显著的精度提升，在测试集上达到了98.22%的精度。你可以继续探索更高值的k，因为99是我们探索范围内的最大值。

9.1.5　使用聚类进行半监督学习

聚类的另一个应用是在半监督学习中，我们有很多未标记的实例，而很少带标签的实例。让我们在digits数据集的50个带标签实例的样本上训练逻辑回归模型：

```
n_labeled = 50
log_reg = LogisticRegression()
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

该模型在测试集上的性能如何呢？

```
>>> log_reg.score(X_test, y_test)
0.8333333333333334
```

精度仅为83.3%。毫不奇怪，这比我们在完整的训练集上训练模型时要低得多。让我们看看如何能做得更好。首先，我们将训练集聚类为50个集群。然后，对于每个集群，找到最接近中心点的图像。我们将这些图像称为代表性图像：

```
k = 50
kmeans = KMeans(n_clusters=k)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

图9-13显示了这50张代表性图像。

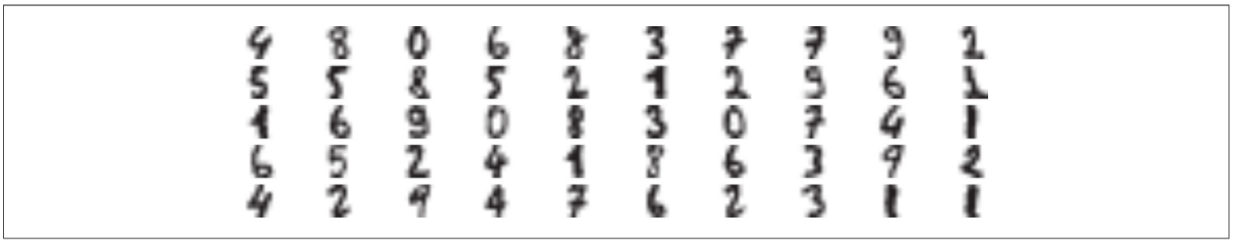


图9-13：50个代表性数字图像（每个集群一个）

让我们看一下每个图像并手动标记它：

```
y_representative_digits = np.array([4, 8, 0, 6, 8, 3, ..., 7, 6, 2, 3, 1, 1])
```

现在，我们有一个只有50个带标签实例的数据集，但是它们不是随机实例，每个实例都是其集群的代表图像。让我们看看性能是否更好：

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_representative_digits, y_representative_digits)
>>> log_reg.score(X_test, y_test)
0.9222222222222223
```

哇！尽管我们仍然仅在50个实例上训练模型，但准确性从83.3%跃升至92.2%。由于标记实例通常很昂贵且很痛苦，尤其是当必须由专家手动完成时，标记代表性实例（而不是随机实例）是一个好主意。

但是也许我们可以更进一步：如果将标签传播到同一集群中的所有其他实例，会怎么样？这称为标签传播：

```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    y_train_propagated[kmeans.labels_==i] = y_representative_digits[i]
```

再次训练模型并查看其性能：

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train, y_train_propagated)
```

```
>>> log_reg.score(X_test, y_test)
0.9333333333333333
```

我们获得了合理的精度提升，这不让人惊奇。问题在于，我们把每个代表性实例的标签传播到同一集群中的所有实例，包括位于集群边界附近的实例，这些实例更容易被错误标记。让我们看看如果仅将标签传播到最接近中心点的20%的实例，会发生什么情况：

```
percentile_closest = 20

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
```

在这个部分传播的数据集上再次训练模型：

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.94
```

很好！仅使用50个带标签的实例（平均每个类只有5个实例！），我们获得了94.0%的准确率，与整个带标签的数字数据集上逻辑回归的性能非常接近（96.9%）。之所以如此出色，是因为传播的标签实际上非常好，它们的准确率非常接近99%，如以下代码所示：

```
>>> np.mean(y_train_partially_propagated == y_train[partially_propagated])
0.9896907216494846
```

主动学习

为了继续改进模型和训练集，下一步可能是进行几轮主动学习，即当人类专家与学习算法进行交互时，在算法要求时为特定实例提供标签。主动学习有许多不同的策略，但最常见的策略之一称为不确定性采样。下面是它的工作原理：

1. 该模型是在目前为止收集到的带标签的实例上进行训练，并且该模型用于对所有未标记实例进行预测。

2. 把模型预测最不确定的实例（当估计的概率最低时）提供给专家来做标记。

3. 重复此过程，直到性能改进到不值得做标记为止。

其他策略包括标记那些会导致模型最大变化的实例或模型验证错误下降最大的实例，或不同模型不一致的实例（例如SVM或随机森林）。

在进入高斯混合模型之前，让我们看一下DBSCAN，这是另一种流行的聚类算法，它说明了一种基于局部密度估计的不同的方法。这种方法允许算法识别任意形状的聚类。

9.1.6 DBSCAN

该算法将集群定义为高密度的连续区域。下面是它的工作原理：

• 对于每个实例，该算法都会计算在距它一小段距离 ε 内有多少个实例。该区域称为实例的 ε -邻域。

• 如果一个实例在其 ε 邻域中至少包含min sam_ples个实例（包括自身），则该实例被视为核心实例。换句话说，核心实例是位于密集区

域中的实例。

• 核心实例附近的所有实例都属于同一集群。这个邻域可能包括其他核心实例。因此，一长串相邻的核心实例形成一个集群。

• 任何不是核心实例且邻居中没有实例的实例都被视为异常。

如果所有集群足够密集并且被低密度区域很好地分隔开，则该算法效果很好。Scikit-Learn中的DBSCAN类就像你期望的那样易于使用。让我们在第5章介绍的moons数据集上进行测试：

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

在labels_实例变量中可以得到所有实例的标签：

```
>>> dbscan.labels_
array([ 0,  2, -1, -1,  1,  0,  0,  0, ...,  3,  2,  3,  3,  4,  2,  6,  3])
```

请注意，某些实例的集群索引等于-1，这意味着该算法将它们视为异常。core_sample_indices_变量可以得到核心实例的索引，components_变量可以得到核心实例本身：

```
>>> len(dbscan.core_sample_indices_)
808
>>> dbscan.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, ..., 992, 993, 995, 997, 998, 999])
>>> dbscan.components_
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
                  ...
```

```
       [-0.94355873,  0.3278936 ],
       [ 0.79419406,  0.60777171]])
```

这种聚类在图9-14的左侧图中表示。如你所见，它识别出很多异常以及7个不同的集群。很令人失望！幸运的是，如果通过将eps增大到0.2来扩大每个实例的邻域，则可以得到右边的聚类，看起来很完美。让我们继续这个模型。

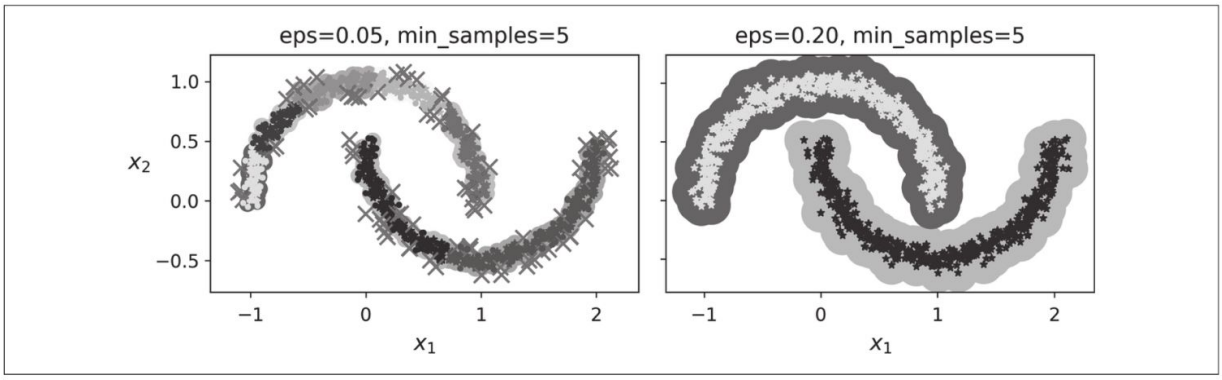


图9-14：使用两个不同邻域半径的DBSCAN聚类

出乎意料的是，尽管DBSCAN类具有fit_predict（）方法，但它没有predict（）方法。换句话说，它无法预测新实例属于哪个集群。之所以做出这个决定是因为不同的分类算法可以更好地完成不同的任务，因此作者决定让用户选择使用哪种算法。而且实现起来并不难。例如，让我们训练一个KNeighborsClassifier：

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])
```

现在，给出一些新实例，我们可以预测它们最有可能属于哪个集群，甚至可以估计每个集群的概率：

```
>>> X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
>>> knn.predict(X_new)
array([1, 0, 1, 0])
>>> knn.predict_proba(X_new)
array([[0.18, 0.82],
       [1.  , 0.  ],
       [0.12, 0.88],
       [1.  , 0.  ]])
```

请注意，我们仅在核心实例上训练了分类器，但是我们也可以选择在所有实例或除异常之外的所有实例上训练分类器，这取决于你的任务。

决策边界如图9-15所示（十字表示X_new中的四个实例）。注意，由于训练集中没有异常，分类器总是选择一个集群，即使该集群很远也是如此。直接引入最大距离的概念，在这种情况下，距离两个集群均较远的两个实例被归类为异常。为此，请使用KNeighborsClassifier的kneighbors（）方法。给定一组实例，它返回训练集中k个最近邻居的距离和索引（两个矩阵，每个有k列）：

```
>>> y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
>>> y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]
>>> y_pred[y_dist > 0.2] = -1
>>> y_pred.ravel()
array([-1,  0,  1, -1])
```

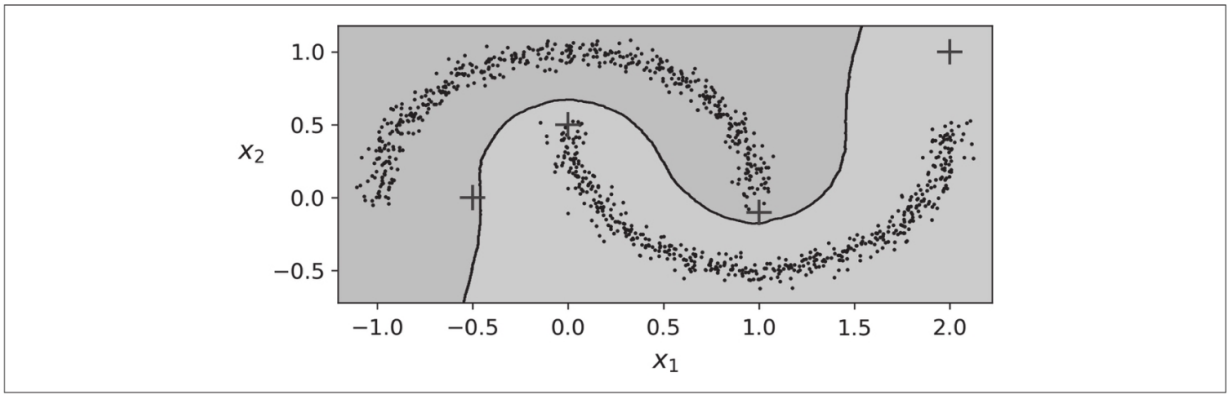


图9-15：两个聚类之间的决策边界

简而言之，DBSCAN是一种非常简单但功能强大的算法，能够识别任何数量的任何形状的集群。它对异常值具有鲁棒性，并且只有两个超参数（eps和min_samples）。但是，如果密度在整个集群中变化很大，则可能无法正确识别所有的集群。它的计算复杂度大约为O（mlogm），使其在实例数量上非常接近线性，但是如果eps大，Scikit-Learn的实现可能需要多达O（m^2）的内存。

 你可能也需要尝试在scikit-learn-contrib项目中实现的Hierarchical DBSCAN（HDBSCAN）。

9.1.7　其他聚类算法

Scikit-Learn还实现了另外几种你应该了解的聚类算法。我们无法在此处详细介绍所有内容，以下是一个简要概述：

聚集聚类

集群的层次结构是自下而上构建的。想想许多微小的气泡漂浮在水上并逐渐相互附着在一起，直到有一大群气泡。同样，在每次迭代中，聚集聚类连接最近的一对集群（从单个实例开始）。如果为每对合并的集群绘制一个带有分支的树，将得到一个二叉树集群，其中叶子是各个实例。这种方法可以很好地扩展到大量实例或集群。它可以识别各种形状的集群，生成灵活且信息丰富的集群树，而不强迫你选择特定的集群规模，并且可以与任何成对的距离一起使用。如果你有一个连通矩阵，它可以很好地扩展到大量实例，该矩阵是一个稀疏m×m矩阵，它表明哪些实例对是相邻的（例如，由sklearn.neighbors.kneighbors_graph（）返回）。没有连通矩阵，该算法就无法很好地扩展到大型数据集。

BIRCH

BIRCH（使用层次结构的平衡迭代约简和聚类）算法是专门为非常大的数据集设计的，并且只要特征数量不太大（<20），它可以比批量处理的K-Means更快，结果相似。在训练期间，它构建了一个树结构，该树结构仅包含足以将每个新实例快速分配给集群的信息，而不必将所有实例存储在树中，这种方法使它在处理大型数据集时使用有限的内存。

均值漂移

此算法从开始在每个实例上居中放置一个圆圈。然后，对于每个圆，它会计算位于其中的所有实例的均值，然后移动圆，使其以均值为中心。接下来，迭代此均值移动步骤，直到所有圆都停止移动（直到每个圆都以其包含的实例的均值为中心）为止。均值漂移将圆向较高密度的方向移动，直到每个圆都找到了局部最大密度。最后，所有圆已经落在同一位置（或足够近）的实例都分配给同一集群。均值漂移具有与DBSCAN相同的特性，例如如何找到具有任何形状的任意数量的集群，它的超参数很少（仅一个圆的半径，称为带宽），并且依赖于局部密度估计。但是，与DBSCAN不同，均值漂移在有内部密度变化时，集群会分裂成小块。不幸的是，其计算复杂度为O（m^2），因此不适用于大型数据集。

相似性传播

此算法使用投票机制，实例对相似的实例进行投票将其作为代表，一旦该算法收敛，每个代表及其投票者将组成一个集群。相似性传播可以检测任意数量的不同大小的集群。不幸的是，该算法的计算复杂度为O（m^2），因此也不适用于大型数据集。

谱聚类

该算法采用实例之间的相似度矩阵，并由其创建低维嵌入（即降维），然后在该低维空间中使用另一种聚类算法（Scikit-Learn使用K-

Means实现）。谱聚类可以识别复杂的集群结构，也可以用于分割图（例如，识别社交网络上的朋友集群）。它不能很好地扩展到大量实例，并且当集群的大小不同时，它的效果也不好。

现在让我们深入研究高斯混合模型，该模型可用于密度估计、聚类和异常检测。

[1] Stuart P.Lloyd，“Least Squares Quantization in PCM”，IEEE Transactions on Information Theory 28，no.2（1982）：129－137.
[2] 这是因为实例与其最接近的中心点之间的均方根距离只能在每一步下降。
[3] David Arthur and Sergei Vassilvitskii，“k-Means++：The Advantages of Careful Seeding”，Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms（2007）：1027－1035.
[4] Charles Elkan，“Using the Triangle Inequality to Accelerate k-Means”，Proceedings of the 20th International Conference on Machine Learning（2003）：147－153.
[5] 三角形不等式为AC≤AB+BC，其中A、B和C是三个点，AB、AC和BC是这些点之间的距离。
[6] David Sculley，“Web-Scale K-Means Clustering”，Proceedings of the 19th International Conference on World Wide Web（2010）：1177－1178.

9.2 高斯混合模型

高斯混合模型（GMM）是一种概率模型，它假定实例是由多个参数未知的高斯分布的混合生成的。从单个高斯分布生成的所有实例都形成一个集群，通常看起来像一个椭圆。每个集群可以具有不同的椭圆形状、大小、密度和方向，如图9-11所示。当你观察到一个实例时，知道它是从一个高斯分布中生成的，但是不知道是哪个高斯分布，并且你也不知道这些分布的参数是什么。

有几种GMM变体。在GaussianMixture类中实现的最简单变体中，你必须事先知道高斯分布的数量k。假定数据集X是通过以下概率过程生成的：

- 对于每个实例，从k个集群中随机选择一个集群。选择第j个集群的概率由集群的权重 $\phi^{(j)}$[1]定义。第i个实例选择的集群的索引记为 $z^{(i)}$。

- 如果 $z^{(i)}=j$，表示第i个实例已分配给第j个集群，则从高斯分布中随机采样该实例的位置 $x^{(i)}$，其中均值 $\mu^{(j)}$ 和协方差矩阵 $\Sigma^{(j)}$。这记为 $\boldsymbol{x}^{(i)} \sim \mathcal{N}(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)})$。

该生成过程可以表示为图模型。图9-16表示随机变量之间的条件依存关系的结构。

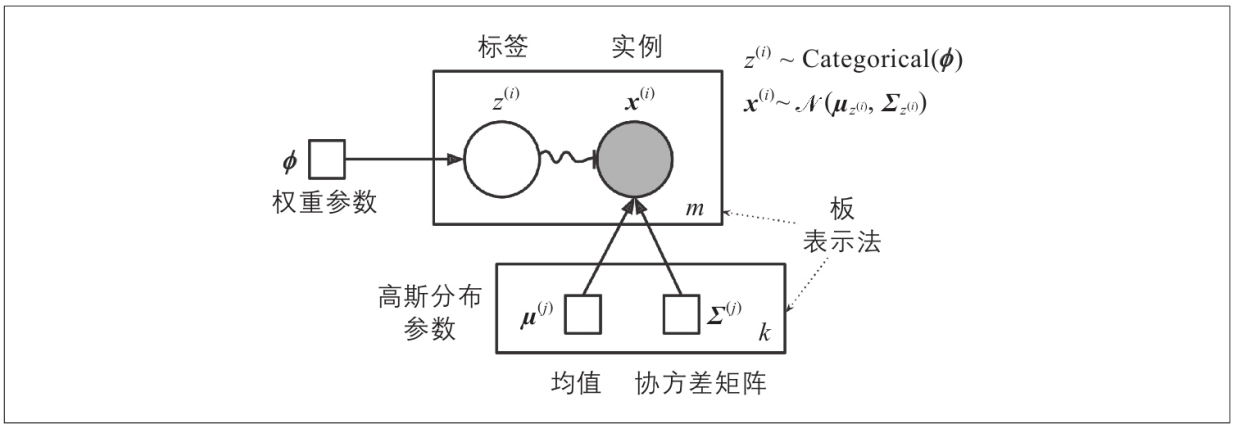


图9-16：高斯混合模型的图形表示，包括其参数（正方形）、随机变量（圆形）及其条件依存关系（实线箭头）

这是该图的解释[2]：

• 圆圈代表随机变量。

• 正方形代表固定值（即模型的参数）。

• 大矩形称为平板。它们表示其内容重复了几次。

• 每个平板右下角的数字表示其内容重复了多少次。因此，有m个随机变量$z^{(i)}$（从$z^{(1)}$到$z^{(m)}$）和m个随机变量$x^{(i)}$，还有k个均值$\mu^{(j)}$和k个协方差矩阵$\Sigma^{(j)}$。最后，只有一个权重向量Φ（包含所有权重$\phi^{(1)}$到$\phi^{(k)}$）。

• 每个变量$z^{(i)}$从具有权重Φ的分类分布中得出。每个变量$x^{(i)}$从正态分布中得出，均值和协方差矩阵由其集群$z^{(i)}$定义。

• 实线箭头表示条件依赖性。例如，每个随机变量z（i）的概率分布取决于权重向量Φ。请注意，当箭头越过平板边界时，这意味着它适用于该平板的所有重复。例如，权重向量Φ确定所有随机变量$x^{(1)}$到$x^{(m)}$的概率分布。

• 从$z^{(i)}$到$x^{(i)}$的弯曲箭头表示一个切换：根据$z^{(i)}$的值，将从不同的高斯分布中采样实例$x^{(i)}$。例如，如果$z^{(i)}=j$，则$x^{(i)} \sim (\mu^{(j)}, \Sigma^{(j)})$。

• 阴影节点表示该值是已知的。因此，在这种情况下，只有随机变量$x^{(i)}$具有已知值，它们被称为已观察变量。未知的随机变量$z^{(i)}$称为潜在变量。

那么，你可以用这种模型做什么？好吧，给定数据集X，你通常希望首先估计权重ϕ以及所有分布参数$\mu^{(1)}$至$\mu^{(k)}$和$\Sigma^{(1)}$至$\Sigma^{(k)}$。Scikit-Learn的GaussianMixture类使这个超级简单：

```
from sklearn.mixture import GaussianMixture

gm = GaussianMixture(n_components=3, n_init=10)
gm.fit(X)
```

算法估计的参数：

```
>>> gm.weights_
array([0.20965228, 0.4000662 , 0.39028152])
>>> gm.means_
array([[ 3.39909717,  1.05933727],
       [-1.40763984,  1.42710194],
       [ 0.05135313,  0.07524095]])
>>> gm.covariances_
array([[[ 1.14807234, -0.03270354],
        [-0.03270354,  0.95496237]],

       [[ 0.63478101,  0.72969804],
        [ 0.72969804,  1.1609872 ]],

       [[ 0.68809572,  0.79608475],
        [ 0.79608475,  1.21234145]]])
```

效果很好！实际上，用于生成数据的权重分别为0.2、0.4和0.4。同样，均值和协方差矩阵与该算法发现的非常接近。但这是怎么做到

的？这个类依赖于期望最大化（Expectation-Maximization，EM）算法，该算法与K-Means算法有很多相似之处：它随机初始化集群参数，然后重复两个步骤直到收敛，首先将实例分配给集群（这称为期望步骤），然后更新集群（这称为最大化步骤）。听起来很熟悉，对吧？在聚类的上下文中，你可以将EM看作是K-Means的推广，它不仅可以找到集群中心（$\mu^{(1)}$到$\mu^{(k)}$），而且可以找到它们的大小、形状和方向（$\Sigma^{(1)}$到$\Sigma^{(k)}$）以及它们的相对权重（$\phi^{(1)}$到$\phi^{(k)}$）。但是与K-Means不同，EM使用软集群分配而不是硬分配。对于每个实例，在期望步骤中，该算法（基于当前的集群参数）估计它属于每个集群的概率。然后，在最大化步骤中，使用数据集中的所有实例来更新每个聚类，并通过每个实例属于该集群的估计概率对其进行加权。这些概率称为实例集群的职责。在最大化步骤中，每个集群的更新主要受到其最负责的实例的影响。

 不幸的是，就像K-Means一样，EM最终可能会收敛到较差的解，因此它需要运行几次，仅仅保留最优解。这就是我们将n_init设置为10的原因。请注意，默认情况下，n_init设置为1。

检查算法是否收敛以及迭代次数：

```
>>> gm.converged_
True
>>> gm.n_iter_
3
```

现在，你已经估算了每个集群的位置、大小、形状、方向和相对权重，该模型可以容易地将每个实例分配给最可能的集群（硬聚类）或估算它属于特定集群的概率（软聚类）。只需对硬聚类使用predict（）方法，对软聚类使用predict_proba（）方法：

```
>>> gm.predict(X)
array([2, 2, 1, ..., 0, 0, 0])
>>> gm.predict_proba(X)
array([[2.32389467e-02, 6.77397850e-07, 9.76760376e-01],
       [1.64685609e-02, 6.75361303e-04, 9.82856078e-01],
       [2.01535333e-06, 9.99923053e-01, 7.49319577e-05],
       ...,
       [9.99999571e-01, 2.13946075e-26, 4.28788333e-07],
       [1.00000000e+00, 1.46454409e-41, 5.12459171e-16],
       [1.00000000e+00, 8.02006365e-41, 2.27626238e-15]])
```

高斯混合模型是一个生成模型，这意味着你可以从中采样新实例（请注意，它们按集群索引排序）：

```
>>> X_new, y_new = gm.sample(6)
>>> X_new
array([[ 2.95400315,  2.63680992],
       [-1.16654575,  1.62792705],
       [-1.39477712, -1.48511338],
       [ 0.27221525,  0.690366  ],
       [ 0.54095936,  0.48591934],
       [ 0.38064009, -0.56240465]])

>>> y_new
array([0, 1, 2, 2, 2, 2])
```

也可以在任何给定位置估计模型的密度。这可以使用score_samples（）方法实现的：对于每个给定的实例，该方法估算该位置的概率密度函数（PDF）的对数。分数越高，密度越高：

```
>>> gm.score_samples(X)
array([-2.60782346, -3.57106041, -3.33003479, ..., -3.51352783,
       -4.39802535, -3.80743859])
```

如果你计算这些分数的指数，可以在给定实例的位置得到PDF的值。这些不是概率，而是概率密度，它们可以取任何正值（不只是0到1之间的值）。要估计实例落入特定区域的概率，你必须在该区域上积分PDF（如果你在可能的实例位置的整个空间中执行此操作，结果将为1）。

图9-17显示了该模型的集群均值、决策边界（虚线）和密度等值线。

很好！该算法显然找到了一个很好的解答。当然，我们通过使用一组二维高斯分布生成数据来简化其任务（不幸的是，现实生活中的数据并不总是那么高斯和低维的）。我们还为算法提供了正确数量的集群。当存在多个维度、多个集群或很少实例时，EM可能难以收敛到最佳解决方案。你可能需要通过限制算法必须学习的参数数量来降低任务的难度。一种方法是限制集群可能具有的形状和方向的范围，这可以通过对协方差矩阵施加约束来实现。为此，将covariance_type超参数设置为以下值之一：

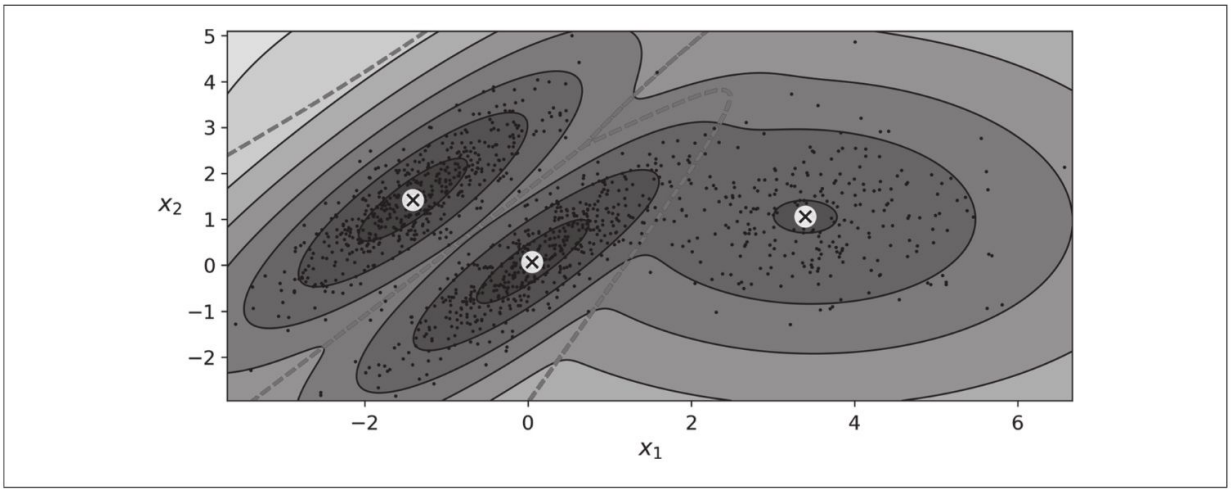


图9-17：训练过的高斯混合模型的均值、决策边界和密度等值线

"spherical"

所有集群都必须是球形的，但它们可以具有不同的直径（即不同的方差）。

"diag"

集群可以采用任何大小的任意椭圆形，但是椭圆形的轴必须平行于坐标轴（即协方差矩阵必须是对角线）。

"tied"

所有集群必须具有相同的椭圆形状、大小和方向（即所有集群共享相同的协方差矩阵）。

默认情况下，covariance_type等于"full"，意味着每个集群可以采用任何形状、大小和方向（它具有自己无约束的协方差矩阵）。图9-18绘制了当covariance_type设置为"tied"或"spherical"时，通过EM算法找到的解。

 训练高斯混合模型的计算复杂度取决于实例数m、维度n、集群数k和协方差矩阵的约束。如果covariance_type为"spherical"或"diag"，假定数据具有聚类结构，则为O（kmn）；如果cova riance_type为"tied"或"full"，则为O（kmn2+kn3），因此不会扩展到具有大量特征。

高斯混合模型也可以用于异常检测。让我们看看它是如何做的。

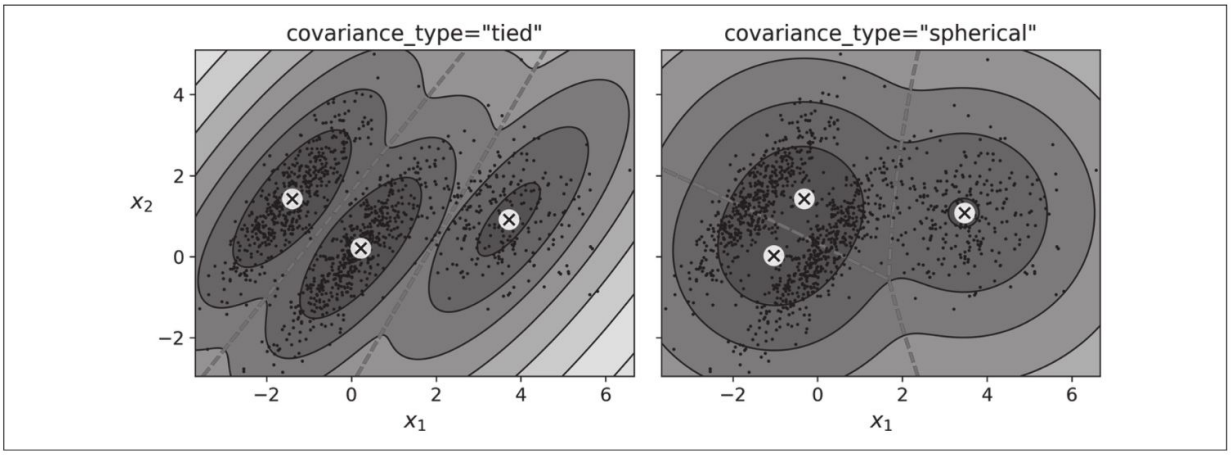


图9-18：束状集群（左）和球形集群（右）的高斯混合

9.2.1 使用高斯混合进行异常检测

异常检测（也称为离群值检测）是检测严重偏离标准的实例的任务。这些实例称为异常或离群值，而正常实例称为内值。异常检测在各种应用中很有用，例如欺诈检测，在制造业中检测有缺陷的产品，或在训练一个模型之前从数据集中删除异常值（这可以显著提高所得模型的性能）。

使用高斯混合模型进行异常检测非常简单：位于低密度区域的任何实例都可以被视为异常。你必须定义要使用的密度阈值。例如，在试图检测缺陷产品的制造业公司中，缺陷产品的比例通常是已知的，假设它等于4%。将密度阈值设置为导致4%的实例位于该阈值密度以下的区域中的值。如果你发现误报过多（即标记为有缺陷的好产品），则可以降低阈值。相反，如果假负过多（即系统未将其标记为次品的次品），则可以提高阈值。这是通常的精度/召回的权衡（见第3章）。以下是使用最低4%的密度作为阈值来识别异常值的方法（即大约4%的实例被标记为异常）：

```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```

图9-19中将这些异常值表示为星号。

一个与之密切相关的任务是新颖性检测，它与异常检测不同之处在于，该算法被假定为在“干净”的数据集上训练的，不受异常值的污染，而异常检测并没有这个假设。实际上，离群值检测通常用于清理数据集。

 高斯混合模型会尝试拟合所有数据，包括异常数据，因此，如果你有太多异常值，这将使模型的“正常性”产生偏差，某些离群值可能被错误地视为正常数据。如果发生这种情况，你可以尝试拟合模型一次，使用它来检测并去除最极端的离群值，然后再次将模型拟合到清理后的数据集上。另一种方法是使用鲁棒的协方差估计方法（请参阅EllipticEnvelope类）。

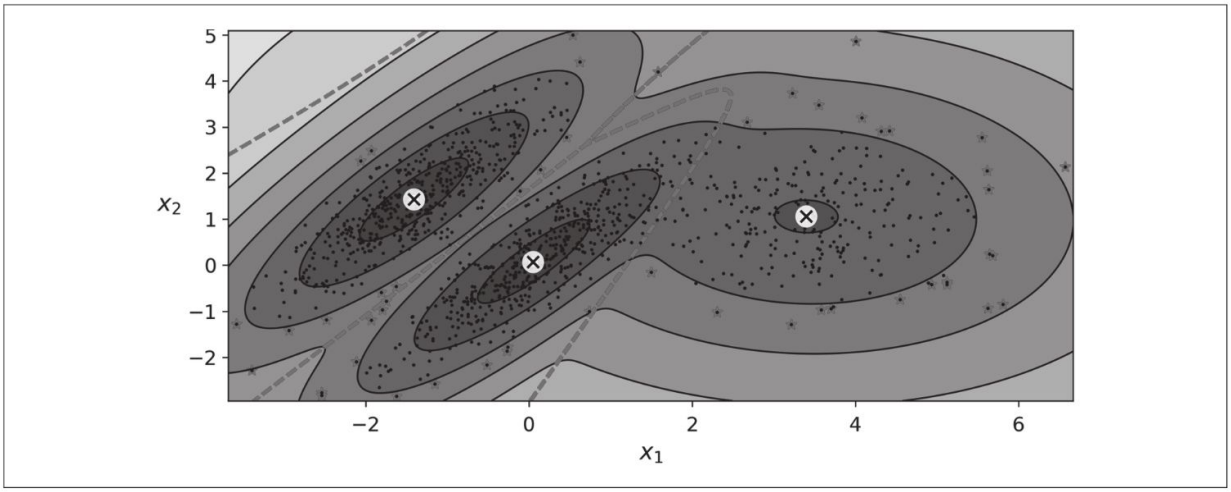


图9-19：使用高斯混合模型进行异常检测

就像K-Means一样，GaussianMixture算法要求你指定集群数目。那么，我们如何找到它呢？

9.2.2　选择聚类数

使用K-Means，你可以使用惯性或轮廓分数来选择适当的集群数。但是对于高斯混合，则无法使用这些度量标准，因为当集群不是球形或具有不同大小时，它们不可靠。相反，你可以尝试找到最小化理论信息量准则的模型，例如公式9-1中定义的贝叶斯信息准则（BIC）或赤池信息准则（AIC）。

公式9-1：贝叶斯信息准则（BIC）和赤池信息准则（AIC）

$$\mathrm{BIC} = \log(m)p - 2\log(\hat{L})$$

$$\mathrm{AIC} = 2p - 2\log(\hat{L})$$

在这些等式中：

- m总是实例数。
- p是模型学习的参数数量。
- $\hat{L}$是模型的似然函数的最大值。

BIC和AIC都对具有更多要学习的参数（例如，更多的集群）的模型进行惩罚，并奖励非常拟合数据的模型。它们常常最终会选择相同的模型。当它们不同时，BIC选择的模型往往比AIC选择的模型更简单（参数更少），但往往不太拟合数据（对于较大的数据集尤其如此）。

似然函数

术语“概率”和“似然”在英语中经常互换使用，但是它们在统计学中的含义却大不相同。给定具有一些参数 θ 的统计模型，用“概率”一词描述未来的结果x的合理性（知道参数值 θ ），而用“似然”一词表示描述在知道结果x之后，一组特定的参数值 θ 的合理性。

考虑两个以-4和+1为中心的高斯分布一维混合模型。为简单起见，此玩具模型具有一个控制两个分布的标准差的单个参数 θ 。图9-20的左上方轮廓图显示了以x和 θ 为函数的模型f（x； θ ）。要估计未来结果x的概率分布，需要设置模型参数 θ 。例如，如果将 θ 设置为1.3（水平线），则会获得左下图中所示的概率密度函数f（x； θ =1.3）。假设你要估算x落在-2和+2之间的概率，则必须在此范围内（即阴影区域的表面）计算PDF的积分。但是，如果你不知道 θ ，而是观察到单个实例

x=2.5（左上图中的垂直线），该怎么办？在这种情况下，你得到似然函数…（θ|x=2.5）=f（x=2.5；θ），如右上图所示。

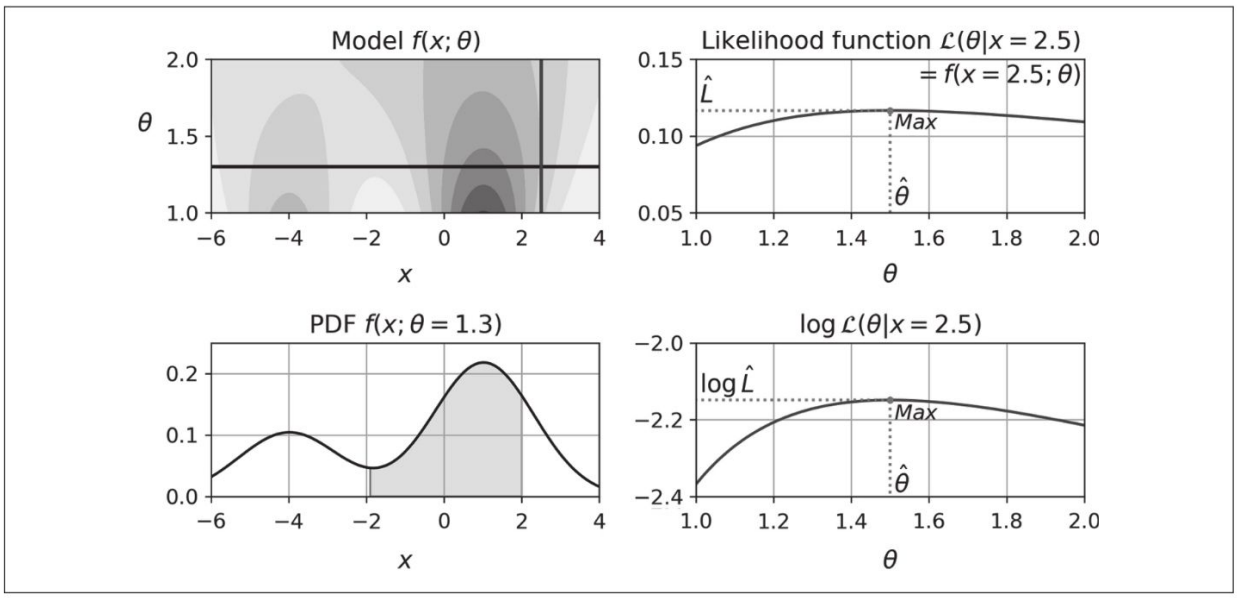


图9-20：模型的参数函数（左上）和一些派生函数：PDF（左下）、似然函数（右上）和对数似然函数（右下）

简而言之，PDF是x的函数（固定了θ），而似然函数是θ的函数（固定了x）。重要的是要理解似然函数不是概率分布：如果你对x的所有可能值积分其概率分布，则总为1；但是，如果对所有可能的θ值积分其似然函数，则结果可以是任何正值。

给定数据集X，一项常见的任务是尝试估计模型参数最有可能的值。为此，你必须找到给定X的最大化似然函数的值。在此示例中，如果观察到单个实例x=2.5，则最大似然估计（MLE）为$\hat{\theta}=1.5$。如果存在关于θ的先验概率分布g，则可以通过最大化L（θ|x）g（θ）来考虑，而不仅仅是最大化L（θ|x）。这称为最大后验（MAP）估计。由于MAP会约束参数值，因此你可以将其视为MLE的正则化版本。

注意，最大化似然函数等效于最大化其对数（在图9-20的右下方图中表示）。实际上，对数是严格增加的函数，因此，如果θ使对数可能

性最大化，那么对数也就使似然最大化。事实证明，通常使对数可能性最大化更容易。例如，如果观察到几个独立的实例x（1）到x（m），则你需要找到使单个似然函数的乘积最大化的θ值。但是，由于对数可以将乘积转换为和，所以最大化对数似然函数的和（而不是乘积）是等效的，而且要简单得多：log（ab）=log（a）+log（b）。

一旦你估算出$\hat{\theta}$，即似然函数最大化的θ值，便可以计算$\hat{L} = \mathcal{L}(\hat{\theta}, \boldsymbol{X})$，这是用于计算AIC和BIC的值，你可以将其视为模型拟合数据的程度的度量。

要计算BIC和AIC，请调用bic（）和aic（）方法：

```
>>> gm.bic(X)
8189.74345832983
>>> gm.aic(X)
8102.518178214792
```

图9-21显示了不同数目的集群k的BIC。如你所见，当k=3时，BIC和AIC都最低，因此它很可能是最佳选择。请注意，我们也可以为超参数covariance_type搜索最佳值。例如，如果它是"spherical"而不是"full"，则该模型要学习的参数要少得多，但它不是很拟合数据。

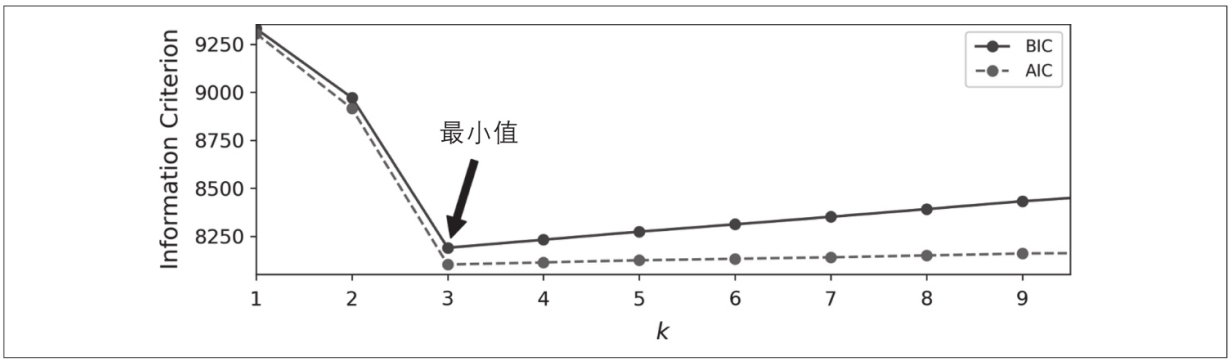


图9-21：不同数目的集群k的AIC和BIC

9.2.3 贝叶斯高斯混合模型

你可以使用BayesianGaussianMixture类，而不是手动搜索集群的最佳数目，该类能够为不必要的集群赋予等于（或接近于）零的权重。将集群数目n_components设置为一个你有充分理由相信的值，该值大于最佳集群的数量（这假定你对当前问题有一些了解），算法会自动消除不必要的集群。例如，让我们将集群数设置为10，然后看看会发生什么：

```
>>> from sklearn.mixture import BayesianGaussianMixture
>>> bgm = BayesianGaussianMixture(n_components=10, n_init=10)
>>> bgm.fit(X)
>>> np.round(bgm.weights_, 2)
array([0.4 , 0.21, 0.4 , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ])
```

完美：算法自动检测到只需要三个集群，并且所得集群几乎与图9-17中的相同。

在此模型中，集群参数（包括权重、均值和协方差矩阵）不再被视为固定的模型参数，而是被视为潜在的随机变量，像集群分配（请参见图9-22）。因此，z现在同时包括集群参数和集群分配。

Beta分布通常用于对值在固定范围内的随机变量进行建模。在这种情况下，范围是0到1。最好通过一个示例来说明“突破性过程”（SBP）：假设Φ=[0.3，0.6，0.5，…]，那么分配30%的实例到集群0，剩余实例的60%分配给集群1，然后将剩余实例的50%分配给集群2，以此类推。在这种数据集中，新实例相比小集群更可能加入大集群（例如人们更可能迁移到较大的城市）。如果α集中度高，则Φ值很可能接近0，并且SBP会生成许多集群。相反，如果集中度低，则Φ值很可能接近1，并且集群很少。最后，使用Wishart分布对协方差矩阵进行采样：参数d和V控制集群形状的分布。

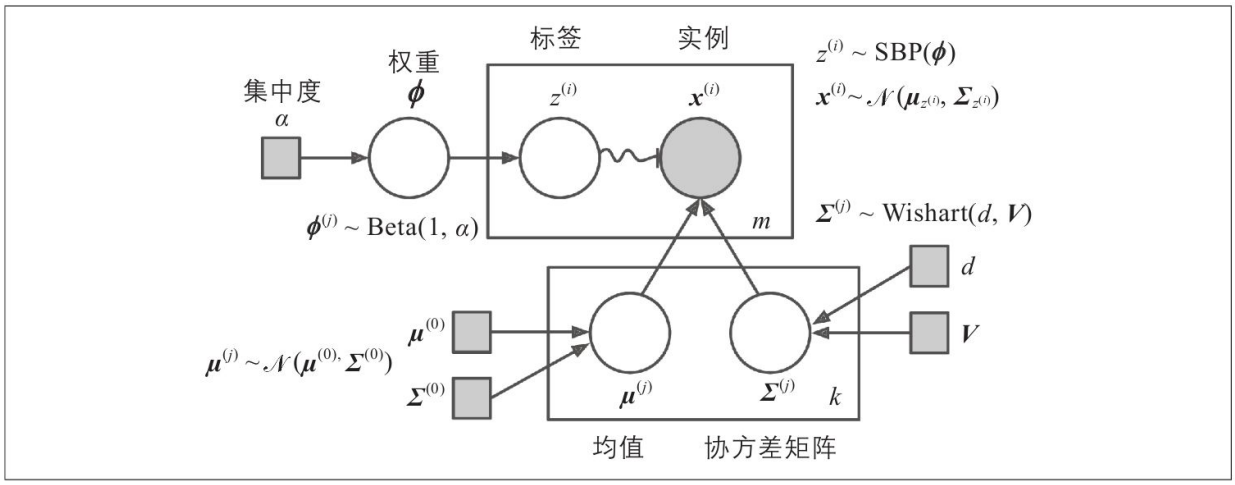


图9-22：贝叶斯高斯混合模型

可以在概率分布p（z）中编码有关潜在变量z的先验知识称为先验。例如，我们可能先验地认为聚类可能很少（低集中度），或者相反，它们可能很多（高集中度）。可以使用超参数weight_concentration_prior来调整有关集群数的先验知识。将其设置为0.01或10 000可得到非常不同的聚类（见图9-23）。但是，我们拥有的数据越多，先验就越不重要。实际上，要绘制具有如此大差异的图表，你必须使用非常强的先验知识和少量数据。

贝叶斯定理（见公式9-2）告诉我们在观察到一些数据X之后如何更新潜在变量的概率分布。它计算后验分布p（z|X），这是给定X的z的条件概率。

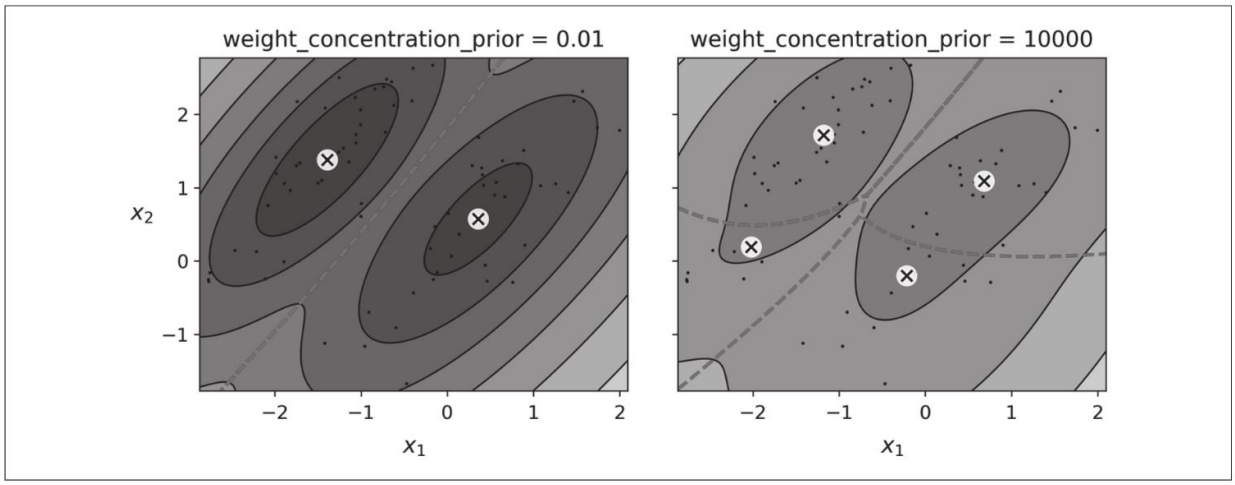


图9-23：在同一数据上使用不同的先验集中度会导致不同数量的集群

公式9-2：贝叶斯定理

$$p(z|X) = 后验 = \frac{概率 \times 先验}{证据} = \frac{p(X|z)p(z)}{p(X)}$$

不幸的是，在高斯混合模型（以及许多其他问题）中，分母p（X）很难处理，因为它需要对z的所有可能值进行积分（见公式9-3），这需要考虑所有可能的集群参数和集群分配的组合。

公式9-3：证据p（X）是难处理的

$$p(X) = \int P(X|z)p(z)\mathrm{d}z$$

这种难处理性是贝叶斯统计中的核心问题之一，有几种解决方法。其中之一是变分推理，它选择具有变分参数λ的分布族q（z；λ），然后优化这些参数以使q（z）成为p（z|X）的良好近似值。这是通过找到最小化从q（z）到p（z|X）的KL散度的λ值来实现的，记为DKL（q||p）。KL散度方程如公式9-4所示，可以将其重写为证据的对数（log p（X））减去证据的下界（ELBO）。由于证据的对数不依赖于q，它是一个常数项，因此，使KL散度最小化只需要使ELBO最大化。

公式9-4：从q（z）到p（z|X）的KL散度

$$
\begin{aligned}
D_{\mathrm{KL}}(q \| p) &= \mathbb{E}_q\left[\log \frac{q(\boldsymbol{z})}{p(\boldsymbol{z} \mid \boldsymbol{X})}\right] \\
&= \mathbb{E}_q[\log q(\boldsymbol{z}) - \log p(\boldsymbol{z} \mid \boldsymbol{X})] \\
&= \mathbb{E}_q\left[\log q(\boldsymbol{z}) - \log \frac{p(\boldsymbol{z} \mid \boldsymbol{X})}{p(\boldsymbol{X})}\right] \\
&= \mathbb{E}_q[\log q(\boldsymbol{z}) - \log p(\boldsymbol{z} \mid \boldsymbol{X}) + \log p(\boldsymbol{X})] \\
&= \mathbb{E}_q[\log q(\boldsymbol{z})] - \mathbb{E}_q[\log p(\boldsymbol{z} \mid \boldsymbol{X})] + \mathbb{E}_q[\log p(\boldsymbol{X})] \\
&= \mathbb{E}_q[\log p(\boldsymbol{X})] - (\mathbb{E}_q[\log p(\boldsymbol{z} \mid \boldsymbol{X})] - \mathbb{E}_q[\log q(\boldsymbol{z})] \\
&= \log p(\boldsymbol{X}) - \mathrm{ELBO} \\
\text{其中} \mathrm{ELBO} &= \mathbb{E}_q[\log p(\boldsymbol{z}, \boldsymbol{X})] - \mathbb{E}_q[\log q(\boldsymbol{z})]
\end{aligned}
$$

在实践中，存在多种使ELBO最大化的不同技术。在均值场变分推理中，有必要非常仔细地选择分布族q（z； λ）和先验p（z），确保ELBO的方程能简化为可计算的形式。不幸的是，没有一般的方法可以做到这一点。选择正确的分布族和正确的先验取决于任务并需要一些数学技能。例如，文档中介绍了Scikit-Learn的BayesianGaussianMixture类中使用的分布和下界等式。从这些方程式中可以得出集群参数和赋值变量的更新方程式。然后，这些方程式的使用非常类似于Expectation-Maximization算法。实际上，BayesianGaussianMixture类的计算复杂度类似于GaussianMixture类的计算复杂度（但通常要慢得多）。一种最大化ELBO的更简单方法称为黑盒随机变分推理（BBSVI）：在每次迭代中，从q中抽取一些样本，然后使用它们来估计ELBO关于变分参数λ的梯度，然后将其用于梯度上升步骤。这种方法使得可以将贝叶斯推理

用于任何类型的模型（假设它是可微分的），甚至是深度神经网络。将贝叶斯推理与深度神经网络结合使用称为贝叶斯深度学习。

 如果你想更深入地了解贝叶斯统计，请查阅Andrew Gelman等人的Bayesian Data Analysis一书。

高斯混合模型在具有椭圆形状的集群上效果很好，但是如果你尝试拟合具有不同形状的数据集，可能会感到意外。例如，让我们看看如果使用贝叶斯高斯混合模型对moons数据集进行聚类会发生什么情况（见图9-24）。

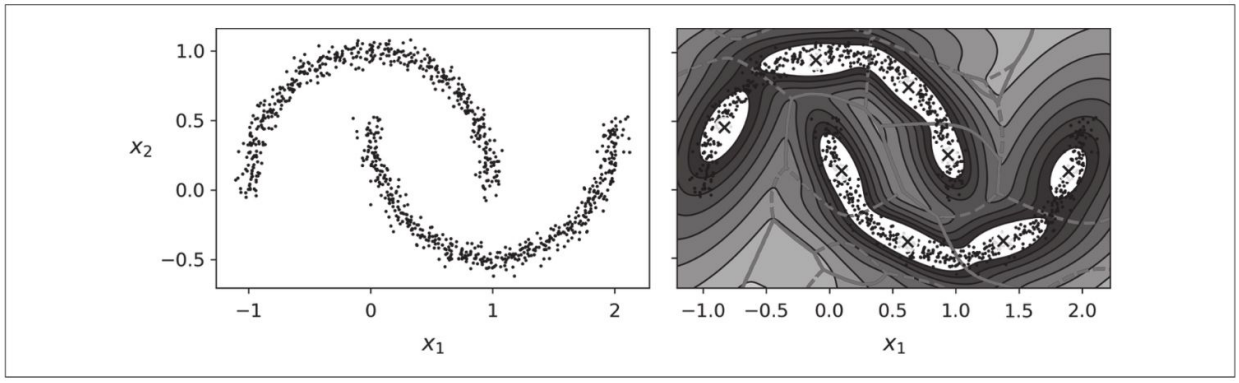


图9-24：将高斯混合模型拟合到非椭圆形集群

很糟糕！该算法拼命搜索椭圆形，因此找到了8个不同的集群（而不是两个）。密度估计还算不错，因此该模型可能可以用于异常检测，但未能识别出两个卫星。现在让我们看一些能够处理任意形状集群的聚类算法。

9.2.4　其他用于异常检测和新颖性检测的算法

Scikit-Learn实现了专用于异常检测或新颖性检测的其他算法：

PCA（以及其他使用inverse_transform（）方法的降维技术）

如果你将正常实例的重建误差与异常的重建误差进行比较，则后者通常会大得多。这是一种简单且通常非常有效的异常检测方法（有关此方法的应用，请参阅本章的练习）。

Fast-MCD（最小协方差决定）

由EllipticEnvelope类实现，该算法对于异常值检测（特别是清理数据集）很有用。假定正常实例（内部值）是根据一个单个高斯分布（而不是混合）生成的。它还假定数据集被不是由该高斯分布生成的异常值所污染。当算法估算高斯分布的参数（即围绕椭圆的椭圆形包络线的形状）时，请小心忽略最可能是异常值的实例。

该技术可以更好地估计椭圆形包络，从而使算法在识别异常值方面更好。

隔离森林

这是一种用于异常值检测的有效算法，尤其是在高维数据集中。该算法构建一个随机森林，其中每个决策树都是随机生长的：在每个节点上，它随机选择一个特征，然后选择一个随机阈值（介于最小值和最大值之间）来将数据集分成两部分。数据集以这种方式逐渐切成分支，直到所有实例最终都与其他实例隔离开来。异常通常与其他实例相距甚远，因此平均而言（在所有决策树中），与正常实例相比，它们倾向于用更少的步骤被隔离。

局部离群因子（Local Outlier Factor，LOF）

此算法也可用于离群值检测。它将给定实例周围的实例密度与其相邻实例周围的密度进行比较。异常值通常比k个最近的邻居更孤立。

单类SVM

此算法更适合新颖性检测。回想一下，一个内核化的SVM分类器通过首先（隐式地）将所有的实例映射到一个高维空间，然后在这个高维空间中使用线性SVM分类器来分离这两个类（见第5章）。由于我们只有单类实例，因此单类SVM算法会尝试将高维空间中的实例与原点分开。在原始空间中，这将对应于找到一个包含所有实例的小区域。如果新实例不在该区域内，则为异常。有一些超参数需要调整：用于内核化SVM的超参数，还有一个余量超参数，它对应于一个新实例实际上是正常的而被误认为是新颖实例的可能性。它非常有用，特别是对于高维数据集，但是像所有SVM一样，它不能扩展到大型数据集。

[1] Phi（Φ or φ）是希腊字母的第21个字母。

[2] 这些符号中的大多数是标准符号，但其他一些符号则取自Wikipedia上有关板符号的文章。

9.3 练习题

1. 如何定义聚类？你能列举几种聚类算法吗？

2. 聚类算法的主要应用有哪些？

3. 描述两种使用K-Means时选择正确数目的集群的技术。

4. 什么是标签传播？为什么要实施它，如何实现？

5. 你能否说出两种可以扩展到大型数据集的聚类算法？两个寻找高密度区域的算法？6. 你能想到一个主动学习有用的示例吗？你将如何实施它？

7. 异常检测和新颖性检测有什么区别？

8. 什么是高斯混合模型？你可以将其用于哪些任务？

9. 使用高斯混合模型时，你能否列举两种技术来找到正确数量的集群？

10. 经典的Olivetti人脸数据集包含400张灰度的64×64像素的人脸图像。每个图像被展平为大小为4096的一维向量。40个不同的人被拍照（每个10次），通常的任务是训练一个模型来预测每个图片中代表哪个人。使用sklearn.datasets.fetch_olivetti_faces（）函数来加载数据集，然后将其拆分为训练集、验证集和测试集（请注意，数据集已缩放到0到1之间）。由于数据集非常小，你可能希望使用分层抽样来确保每组中每个人的图像数量相同。接下来，使用K-Means对图像进行聚类，确保你拥有正确的集群数（使用本章中讨论的一种技术）。可视化集群：你在每个集群中看到了相似面孔吗？

11.继续使用Olivetti人脸数据集，训练分类器来预测每张图片代表哪个人，并在验证集上对其进行评估。接下来，将K-Means用作降维工具，然后在简化集上训练分类器。搜索使分类器获得最佳性能的集群数量：可以达到什么性能？如果将简化集中的特征附加到原始特征上（再次搜索最佳数目的集群）怎么办？

12.在Olivetti人脸数据集上训练高斯混合模型。为了加快算法的速度，你可能应该降低数据集的维度（例如，使用PCA，保留99%的方差）。使用该模型来生成一些新面孔（使用sample（）方法），并对其进行可视化（如果使用PCA，需要使用其inverse_transform（）方法）。尝试修改一些图像（例如，旋转、翻转、变暗），然后查看模型是否可以检测到异常（即比较score_samples（）方法对正常图像和异常图像的输出）。

13.一些降维技术也可以用于异常检测。例如，使用Olivetti人脸数据集并使用PCA对其进行降维，保留99%的方差。然后计算每个图像的重建误差。接下来，使用一些你在练习12中构建的一些修改后的图像，查看其重建误差：注意重建误差的大小。如果画出重建图像，你会看到原因：它尝试重建一张正常的脸。

附录A中提供了这些练习题的解答。

第二部分　神经网络与深度学习

第10章　Keras人工神经网络简介

我们从鸟类那里得到启发，学会了飞翔，从牛蒡那里得到启发，发明了魔术贴，还有很多其他的发明都是被自然所启发。这么说来看看大脑的组成，并期望因此而得到启发来构建智能机器就显得很合乎逻辑了。这也是人工神经网络（ANN）思想的根本来源。不过，虽然飞机的发明受鸟类的启发，但是它并不用扇动翅膀来飞翔。同样，人工神经网络和它的生物版本也有很大差异。甚至有些研究者认为应该放弃对生物类比的使用（比如，称其为“单元”而不是“神经元”），以免我们将创造力限制在生物学看似合理的系统[1]上。

人工神经网络是深度学习的核心。它们用途广泛、功能强大且可扩展，使其非常适合处理大型和高度复杂的机器学习任务，例如对数十亿张图像进行分类（例如Google Images），为语音识别服务（例如Apple的Siri）提供支持，每天向成千上万的用户推荐（例如YouTube）观看的最佳视频，或学习在围棋游戏（DeepMind的AlphaGo）中击败世界冠军。

本章的第一部分介绍了人工神经网络，首先是对第一个ANN架构的快速浏览，然后是今天广泛使用的多层感知机（MLP）（其他架构将在第11章中进行探讨）。在第二部分中，我们将研究如何使用流行的Keras API实现神经网络。这是设计精巧、简单易用的用于构建、训练、评估和运行神经网络的API。但是，不要被它的简单性所迷惑：它的表现力和灵活性足以让你构建各种各样的神经网络架构。实际上，对于大多数示例而言，这可能就足够了。如果你需要额外的灵活性，可以随时使用其较低级的API编写自定义的Keras组件，这将在第12章中讨论。

但是首先，让我们回到过去，看看人工神经网络是如何发展的！

[1] 你可以通过接受生物学灵感而获得两全其美，只要它们运行良好，就不必害怕创建生物学上不切实际的模型。

10.1 从生物神经元到人工神经元

令人惊讶的是，人工神经网络已经存在很长一段时间了：它们于1943年由神经生理学家Warren McCulloch和数学家Walter Pitts首次提出。McCulloch和Pitts在其具有里程碑意义的论文“A Logical Calculus of Ideas Immanent in Nervous Activity[1]”中，提出了一种简化的计算模型，该模型计算了生物神经元如何在动物大脑中协同工作，利用命题逻辑进行复杂的计算。这是第一个人工神经网络架构。从那时起，我们看到许多其他架构被发明出来。

人工神经网络的早期成功使得人们普遍相信，我们很快将与真正的智能机器进行对话。当在20世纪60年代我们清楚地知道不能兑现这一承诺（至少相当长一段时间）时，资金流向了其他地方，人工神经网络进入了漫长的冬天。在20世纪80年代初期，发明了新的架构，并开发了更好的训练技术，从而激发了人们对连接主义（神经网络的研究）的兴趣。但是进展缓慢，到了20世纪90年代，发明了其他强大的机器学习技术，例如支持向量机（见第5章）。这些技术似乎提供了比人工神经网络更好的结果和更坚实的理论基础，神经网络的研究再次被搁置。

我们现在目睹了对人工神经网络的另一波兴趣。这波浪潮会像以前一样消灭吗？好吧，这里有一些充分的理由使我们相信这次是不同的，人们对人工神经网络重新充满兴趣将对我们的生活产生更深远的影响：

- 现在有大量数据可用于训练神经网络，并且在非常大和复杂的问题上，人工神经网络通常优于其他机器学习技术。

- 自20世纪90年代以来，计算能力的飞速增长使得现在有可能在合理的时间内训练大型神经网络。这部分是由于摩尔定律（集成电路中的器件数量在过去的50年中，每两年大约增加一倍），这还要归功于游戏产业——刺激了数百万计强大的GPU卡的生产。此外，云平台已使所有人都可以使用这个功能。

• 训练算法已得到改进。公平地说，它们仅与20世纪90年代使用的略有不同，但是这些相对较小的调整产生了巨大的积极影响。

• 在实践中，人工神经网络的一些理论局限性被证明是良性的。例如，许多人认为ANN训练算法注定要失败，因为它们可能会陷入局部最优解，但事实证明，这在实践中相当罕见（而且在这种情况下，它们通常与全局最优解相当接近）。

• 人工神经网络似乎已经进入了资金和发展的良性循环。基于人工神经网络的好产品会成为头条新闻，这吸引了越来越多的关注和资金，从而带来了越来越多的进步甚至产生了惊人的产品。

10.1.1 生物神经元

在讨论人工神经元之前，让我们快速看一下生物神经元（见图10-1）。它是一种看起来不寻常的细胞，主要存在于动物的大脑中。它由包含核和大多数细胞复杂成分的细胞体组成，其中许多分支延伸称为树突，再加上一个很长的延伸称为轴突。轴突的长度可能比细胞体长几倍，或者长几万倍。轴突在其末端附近分裂成许多分支，称为端粒，在这些分支的顶端是称为突触末端（或简称为突触）的微小结构，与其他神经元的树突或细胞体相连[2]。生物神经元产生短的电脉冲称为动作电位（AP，或只是信号），它们沿着轴突传播，使突触释放称为神经递质的化学信号。当神经元在几毫秒内接收到足够数量的这些神经递质时，它会激发自己的电脉冲（实际上，它取决于神经递质，因为其中一些会抑制神经元的发射）。

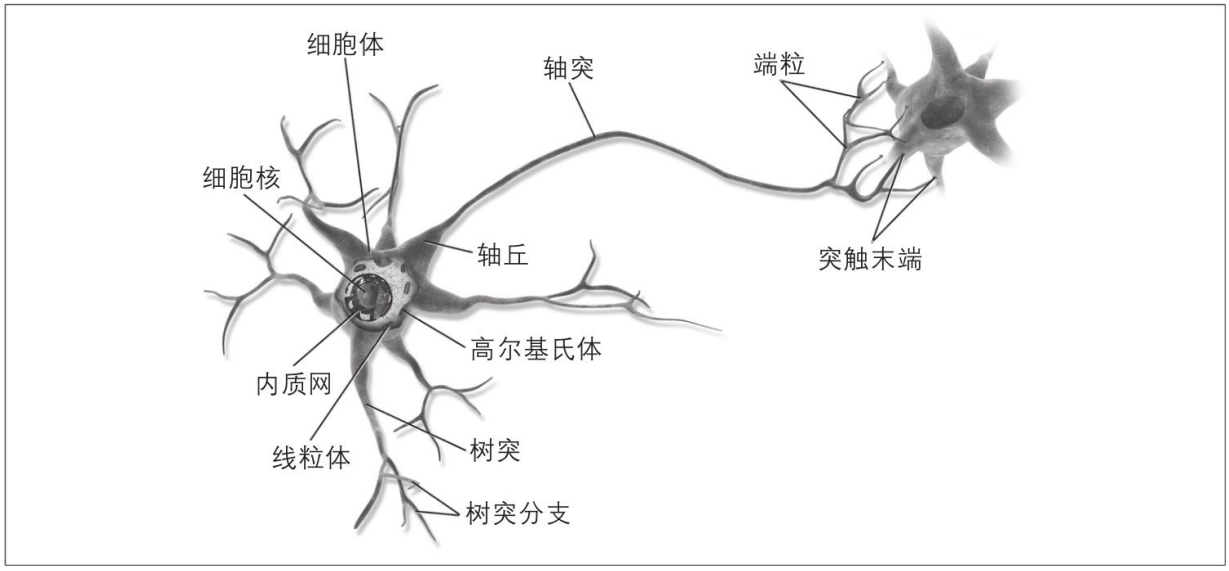


图10-1：生物神经元[3]

因此，单个生物神经元的行为似乎很简单，但是它们组成了数十亿个庞大的网络，每个神经元都与数千个其他神经元相连。高度复杂的计算可以通过相当简单的神经元网络来执行，就像复杂的蚁丘可以通过简单蚂蚁的共同努力而出现一样。生物神经网络（BNN）[4]的架构仍是活跃的研究主题，但大脑的某些部分已被绘制成图，似乎神经元通常组织成连续的层，尤其是在大脑皮层中（大脑的外层），如图10-2所示。

图10-2：生物神经网络（人类皮层）中的很多层[5]

10.1.2　神经元的逻辑计算

McCulloch和Pitts提出了一个非常简单的生物神经元模型，该模型后来被称为神经元。它具有一个或多个二进制（开/关）输入和一个二进制输出。当超过一定数量的输入处于激活状态时，人工神经元将激活其输出。他们的论文表明即使使用这样的简化模型，也可以构建一个人工神经元网络来计算所需的任何逻辑命题。为了了解这种网络的工作原理，让我们构建一些执行各种逻辑计算的ANN（见图10-3），假设神经元至少两个输入处于激活状态时，神经元就会被激活。

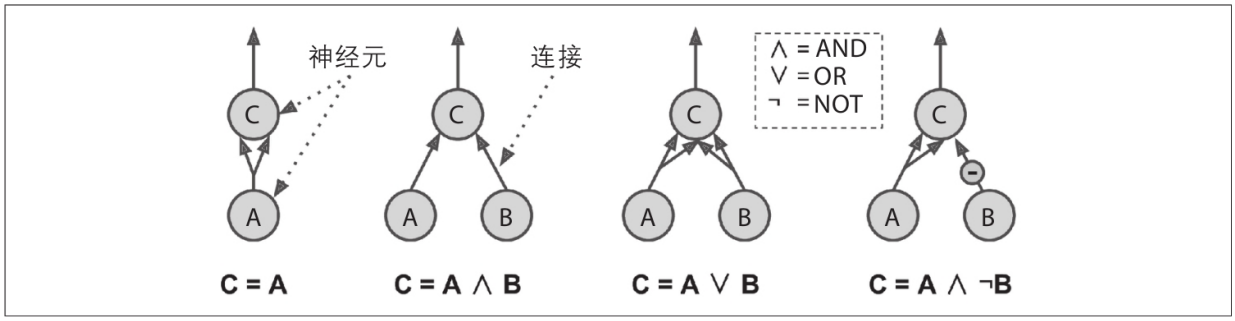


图10-3：ANN执行简单的逻辑运算

让我们看看这些网络的作用：

• 左边的第一个网络是恒等函数：如果神经元A被激活，那么神经元C也被激活（因为它从神经元A接收到两个输入信号）；如果神经元A关闭，那么神经元C也关闭。

• 第二个网络执行逻辑AND：仅当神经元A和B都被激活（单个输入信号不足以激活神经元C）时，神经元C才被激活。

• 第三个网络执行逻辑OR：如果神经元A或神经元B被激活（或两者都激活），则神经元C被激活。

• 最后，如果我们假设输入连接可以抑制神经元的活动（生物神经元就是这种情况），则第四个网络计算出一个稍微复杂的逻辑命题：只有在神经元A处于活动状态和神经元B关闭时，神经元C才被激活。如果

神经元A一直处于活动状态，那么你会得到逻辑NOT：神经元B关闭时神经元C处于活动状态，反之亦然。

你可以想象如何将这些网络组合起来以计算复杂的逻辑表达式（有关示例，请参见本章末的练习题）。

10.1.3 感知器

感知器是最简单的ANN架构之一，由Frank Rosenblatt于1957年发明。它基于稍微不同的人工神经元（见图10-4），称为阈值逻辑单元（TLU），有时也称为线性阈值单元（LTU）。输入和输出是数字（而不是二进制开/关值），并且每个输入连接都与权重相关联。TLU计算其输入的加权和（$z=w_1x_1+w_2x_2+\cdots+w_nx_n=x^Tw$），然后将阶跃函数应用于该和并输出结果：$h_w(x)=\text{step}(z)$，其中$z=x^Tw$。

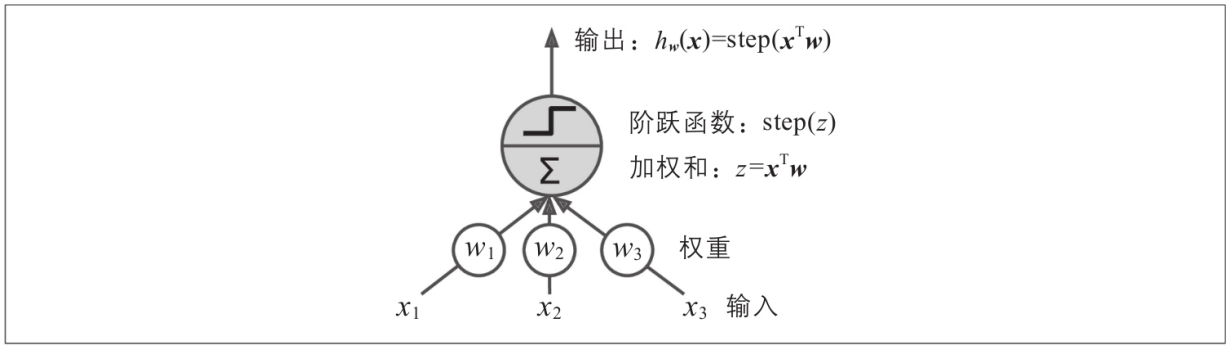


图10-4：阈值逻辑单元：人工神经元，计算其输入的加权和，然后应用阶跃函数

感知器中最常用的阶跃函数是Heaviside阶跃函数（见公式10-1）。有时使用符号函数代替。

公式10-1：感知器中使用的常见阶跃函数（假设阈值=0）

$$\text{heaviside}(z) = \begin{cases} 0, & \text{如果 } z < 0 \\ 1, & \text{如果 } z \geqslant 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1, & \text{如果 } z < 0 \\ 0, & \text{如果 } z = 0 \\ +1, & \text{如果 } z > 0 \end{cases}$$

单个TLU可用于简单的线性二进制分类。它计算输入的线性组合，如果结果超过阈值，则输出正类；否则，它将输出负类（就像逻辑回归或线性SVM分类器一样）。例如，你可以使用单个TLU根据花瓣的长度和宽度对鸢尾花进行分类（就像我们在前面的章节中所做的那样，还添加了额外的偏置特征$x_0=1$）。在这种情况下，训练TLU意味着找到w_0、w_1和w_2的正确值（稍后将讨论训练算法）。

感知器仅由单层TLU[6]组成，每个TLU连接到所有的输入。当一层中的所有神经元都连接到上一层中的每个神经元（即其输入神经元）时，该层称为全连接层或密集层。感知器的输入被送到称为输入神经元的特殊直通神经元：它们输出被送入的任何输入。所有输入神经元形成输入层。此外，通常会添加一个额外的偏置特征（$x_0=1$）：通常使用一种称为偏置神经元的特殊类型的神经元来表示该特征，该神经元始终输出1。具有两个输入和三个输出的感知器如图10-5所示。该感知器可以将实例同时分为三个不同的二进制类，这使其成为多输出分类器。

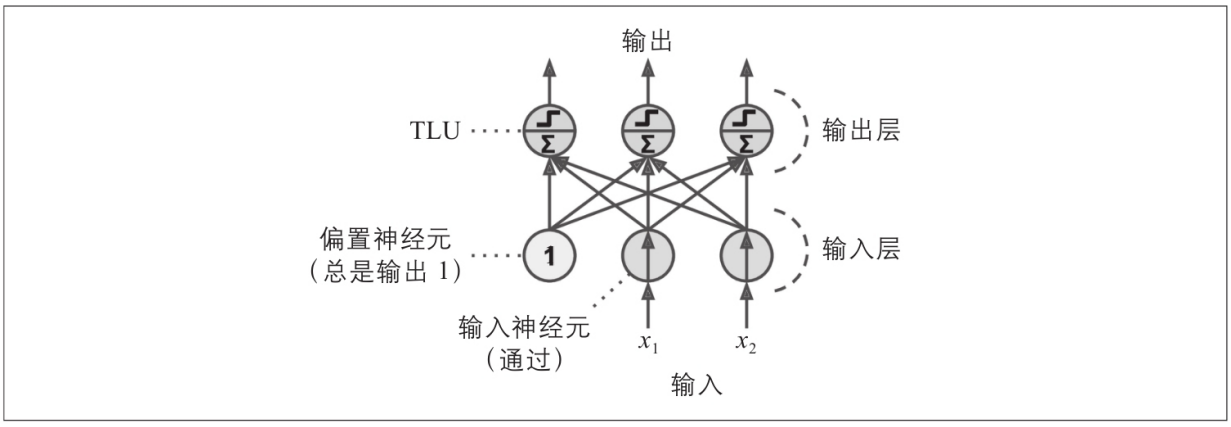


图10-5：具有两个输入神经元、一个偏置神经元和三个输出神经元的感知器架构

借助线性代数的魔力，公式10-2使得可以同时为多个实例高效地计算出一层人工神经元的输出。

公式10-2：计算全连接层的输出

$$h_{W,b}(X)=\phi(XW+b)$$

在此等式中：

• X代表输入特征的矩阵。每个实例一行，每个特征一列。

• 权重矩阵W包含除偏置神经元外的所有连接权重。在该层中，每个输入神经元一行，每个人工神经元一列。

• 偏置向量b包含偏置神经元和人工神经元之间的所有连接权重。每个人工神经元有一个偏置项。

• 函数Φ称为激活函数：当人工神经元是TLU时，它是阶跃函数（但我们在后面会讨论其他激活函数）。

那么，感知器如何训练？Rosenblatt提出的感知器训练算法在很大程度上受Hebb规则启发。Donald Hebb在其1949年的The Organization of Behavior（Wiley）中提出，当一个生物神经元经常触发另一个神经元时，这两个神经元之间的联系就会增强。后来，Siegrid Löwel用有名的措辞概括了Hebb的思想，即“触发的细胞，连接在一起”。也就是说，两个神经元同时触发时，它们之间的连接权重会增加。该规则后来被称为Hebb规则（或Hebb学习）。使用此规则的变体训练感知器，该变体考虑了网络进行预测时所犯的错误。感知器学习规则加强了有助于减少错误的连接。更具体地说，感知器一次被送入一个训练实例，并且针对每个实例进行预测。对于产生错误预测的每个输出神经元，它会增强

来自输入的连接权重，这些权重将有助于正确的预测。该规则如公式10-3所示。

公式10-3：感知器学习规则（权重更新）

$$w_{i,j}^{(下一步)} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

在此等式中：

- $w_{i,j}$是第i个输入神经元和第j个输出神经元之间的连接权重。
- x_i是当前训练实例的第i个输入值。
- $\hat{y}_j$是当前训练实例的第j个输出神经元的输出。
- y_j是当前训练实例的第j个输出神经元的目标输出。
- η是学习率。

每个输出神经元的决策边界都是线性的，因此感知器无法学习复杂的模式（就像逻辑回归分类器一样）。但是，如果训练实例是线性可分的，Rosenblatt证明了该算法将收敛到一个解[7]。这被称为感知器收敛定理。

Scikit-Learn提供了一个Perceptron类，该类实现了单个TLU网络。它可以像你期望的那样使用，例如，在鸢尾植物数据集上（在第4章中介绍）：

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
```

```
iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)  # Iris setosa?

per_clf = Perceptron()
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

你可能已经注意到，感知器学习算法非常类似于随机梯度下降。实际上，Scikit-Learn的Perceptron类等效于使用具有以下超参数的SGDClassifier：loss="perceptron"，learning_rate="constant"，eta0=1（学习率）和penalty=None（无正则化））。请注意，与逻辑回归分类器相反，感知器不输出分类概率；相反，它们基于硬阈值进行预测。这是逻辑回归胜过感知器的原因。

Marvin Minsky和Seymour Papert在1969年的专著Perceptron中，特别指出了感知器的一些严重缺陷，即它们无法解决一些琐碎的问题（例如，异或（XOR）分类问题，参见图10-6的左侧）。任何其他线性分类模型（例如逻辑回归分类器）都是如此，但是研究人员对感知器的期望更高，有些人感到失望，他们完全放弃了神经网络，转而支持更高层次的问题，例如逻辑、问题求解和搜索。

事实证明，可以通过堆叠多个感知器来消除感知器的某些局限性。所得的ANN称为多层感知器（MLP）。MLP可以解决XOR问题，你可以通过计算图10-6右侧所示的MLP的输出来验证：输入（0，0）或（1，1），网络输出0，输入（0，1）或（1，0）则输出1。所有连接的权重等于1，但显示权重的四个连接除外。尝试验证该网络确实解决了XOR问题！

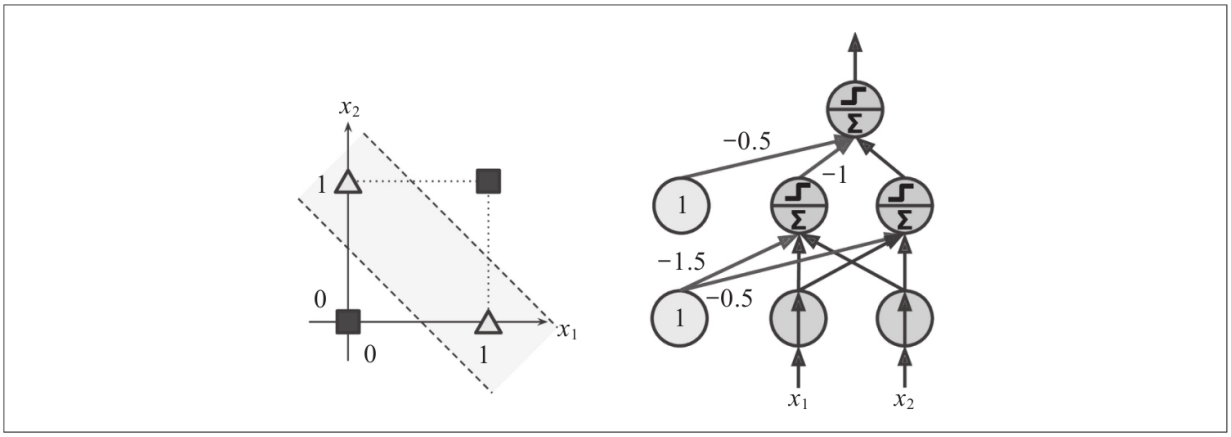


图10-6：XOR分类问题和解决该问题的MLP

10.1.4　多层感知器和反向传播

MLP由一层（直通）输入层、一层或多层TLU（称为隐藏层）和一个TLU的最后一层（称为输出层）组成（见图10-7）。靠近输入层的层通常称为较低层，靠近输出层的层通常称为较高层。除输出层外的每一层都包含一个偏置神经元，并完全连接到下一层。

 信号仅沿一个方向（从输入到输出）流动，因此该架构是前馈神经网络（FNN）的示例。

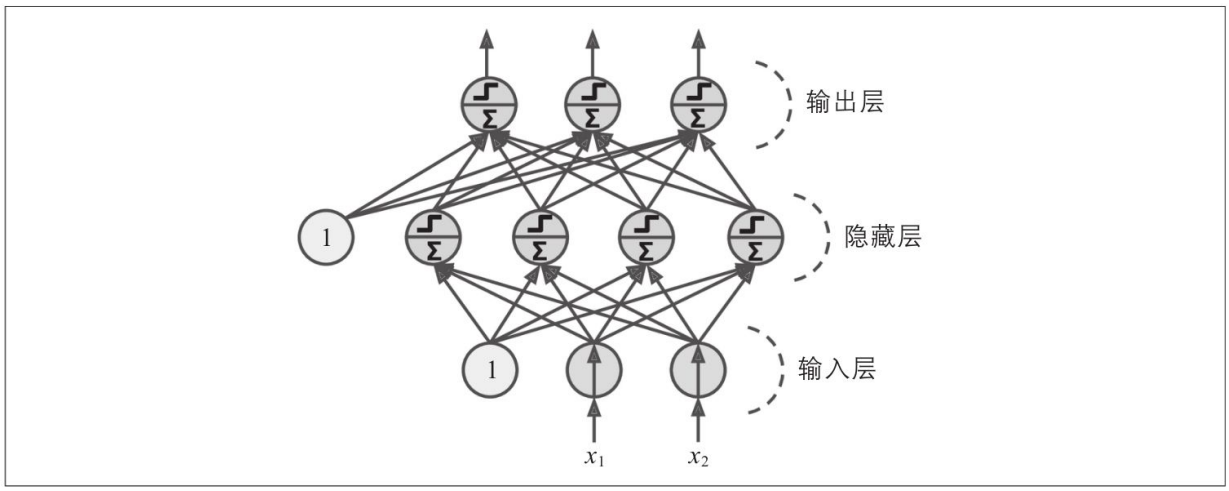


图10-7：具有两个输入、一个含四个神经元的隐藏层和三个输出神经元的多层感知器架构（此处显示了偏置神经元，但通常是隐含的）

当一个ANN包含一个深层的隐藏层[8]时，它称为深度神经网络（DNN）。深度学习领域研究DNN，更广泛地讲包含深度计算堆栈的模型。即便如此，只要涉及神经网络（甚至是浅层的神经网络），许多人就会谈论深度学习。

多年来，研究人员一直在努力寻找一种训练MLP的方法，但没有成功。但在1986年，David Rumelhart、Geoffrey Hinton和Ronald Williams发表的开创性论文[9]介绍了反向传播训练算法，该算法至今仍在使用。简而言之，它是使用有效的技术自动计算梯度下降（在第4章中介绍）[10]：在仅两次通过网络的过程中（一次前向，一次反向），反向传播算法能够针对每个模型参数计算网络误差的梯度。换句话说，它可以找出应如何调整每个连接权重和每个偏置项以减少误差。一旦获得了这些梯度，它便会执行常规的梯度下降步骤，然后重复整个过程，直到网络收敛到解。

 自动计算梯度称为自动微分或者autodiff。有各种autodiff技术，各有优缺点。反向传播使用的一种称为反向模式autodiff。它快速而精确，并且非常适用于微分函数具有多个变量（例如，连接权重）和少量输出（例如，一个损失）的情况。如果你想了解有关autodiff的更多信息，请查看附录D。

让我们更详细地介绍一下该算法：

- 它一次处理一个小批量（例如，每次包含32个实例），并且多次遍历整个训练集。每次遍历都称为一个轮次。

• 每个小批量都传递到网络的输入层，然后将其送到第一个隐藏层。然后该算法将计算该层中所有神经元的输出（对于小批量中的每个实例）。结果传递到下一层，计算其输出并传递到下一层，以此类推，直到获得最后一层（即输出层）的输出。这就是前向通路：就像进行预测一样，只是保留了所有中间结果，因为反向遍历需要它们。

• 接下来，该算法测量网络的输出误差（该算法使用一种损失函数，该函数将网络的期望输出与实际输出进行比较，并返回一些误差测量值）。

• 然后，它计算每个输出连接对错误的贡献程度。通过应用链式法则（可能是微积分中最基本的规则）来进行分析，从而使此步骤变得快速而精确。

• 然后，算法再次使用链式法则来测量这些错误贡献中有多少是来自下面层中每个连接的错误贡献，算法一直进行，到达输入层为止。如前所述，这种反向传递通过在网络中向后传播误差梯度，从而有效地测量了网络中所有连接权重上的误差梯度（因此称为算法）。

• 最终，该算法执行梯度下降步骤，使用刚刚计算出的误差梯度来调整网络中的所有连接权重。

该算法非常重要，值得再次总结：对于每个训练实例，反向传播算法首先进行预测（正向传递）并测量误差，然后反向经过每个层以测量来自每个连接的误差贡献（反向传递），最后调整连接权重以减少错误（梯度下降步骤）。

 随机初始化所有隐藏层的连接权重很重要，否则训练将失败。例如，如果将所有权重和偏置初始化为零，则给定层中的所有神经元将完全相同，从而反向传播将以完全相同的方式影响它们，因此它们将保持相同。换句话说，尽管每层有数百个神经元，但是模型会像每层只有一个神经元一样工作：不会太聪明。相反，如果随机初始化权重，则会破坏对称性，并允许反向传播来训练各种各样的神经元。

为了使该算法正常工作，作者对MLP的架构进行了重要更改，将阶跃函数替换为逻辑（s型）函数：σ（z）=1/（1+exp（-z））。这一点很重要，因为阶跃函数仅包含平坦段，所以没有梯度可使用（梯度下降不能在平面上移动），而逻辑函数在各处均具有定义明确的非零导数，从而使梯度下降在每一步都可以有所进展。实际上，反向传播算法可以与许多其他激活函数（不仅是逻辑函数）一起很好地工作。这是另外两个受欢迎的选择：

双曲正切函数：tanh（z）=2σ（2z）-1

与逻辑函数一样，该激活函数为S形、连续且可微，但其输出值范围为-1到1（而不是逻辑函数的从0到1）。在训练开始时，该范围倾向于使每一层的输出或多或少地以0为中心，这通常有助于加快收敛速度。

线性整流单位函数：ReLU（z）=max（0，z）

ReLU函数是连续的，但不幸的是，在z=0时，该函数不可微分（斜率会突然变化，这可能使梯度下降反弹），如果z<0则其导数为0。但是，实际上它运行良好并且具有计算快速的优点，因此它已成为默认值[11]。最重要的是，它没有最大输出值这一事实有助于减少梯度下降期间的某些问题（我们将在第11章中对此进行讨论）。

这些流行的激活函数及其派生函数如图10-8所示。为什么我们首先需要激活函数？如果连接多个线性变换，那么得到的只是一个线性变换。例如，如果f（x）=2x+3且g（x）=5x-1，则连接这两个线性函数可以得到另一个线性函数：f（g（x））=2（5x-1）+3=10x+1。因此，如果层之间没有非线性，那么即使是很深的层堆叠也等同于单个层，这样你无法解决非常复杂的问题。相反，具有非线性激活函数的足够大的DNN理论上可以近似任何连续函数。

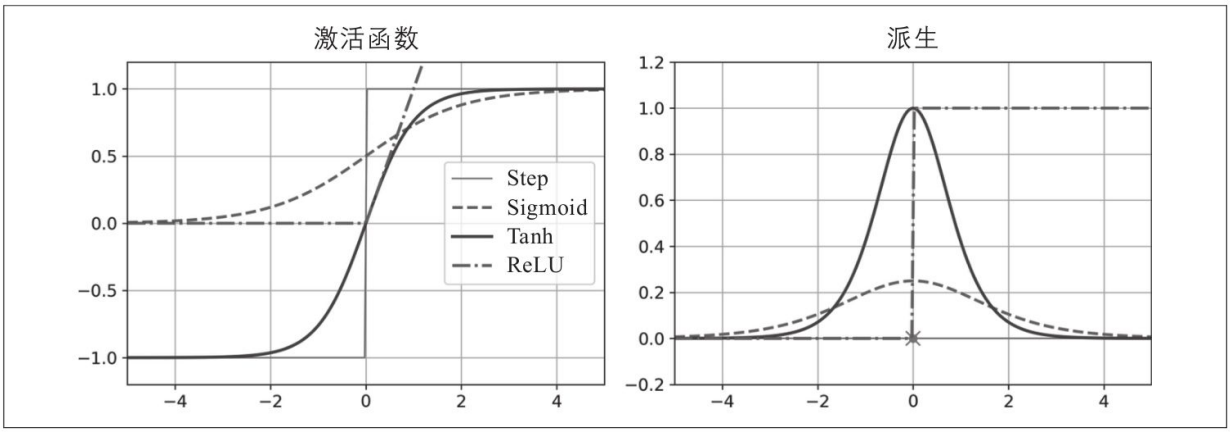


图10-8：激活函数及其派生

好！你知道神经网络来自何处、其结构是什么以及如何计算其输出。你还了解了反向传播算法。但是，你到底可以使用它们做什么呢？

10.1.5 回归MLP

首先，MLP可用于回归任务。如果要预测单个值（房屋的价格，给定其许多特征），则只需要单个输出神经元：其输出就是预测值。对于多元回归（即一次预测多个值），每个输出维度需要一个输出神经元。例如，要在图像中定位物体的中心，你需要预测2D坐标，因此需要两个输出神经元。如果你还想在物体周围放置边框，则还需要两个数字：

物体的宽度和高度。因此，你得到了四个输出神经元。

通常，在构建用于回归的MLP时，你不想对输出神经元使用任何激活函数，因此它们可以输出任何范围的值。如果要保证输出始终为正，则可以在输出层中使用ReLU激活函数。另外，你可以使用softplus激活函数，它是ReLU的平滑变体：softplus（z）=log（1+exp（z））。当z为负时，它接近于0，而当z为正时，它接近于z。最后，如果要保证预测值落在给定的值范围内，则可以使用逻辑函数或双曲正切，然后将标签缩放到适当的范围：逻辑函数的范围为0到1，双曲正切为-1到1。

训练期间要使用的损失函数通常是均方误差，但是如果训练集中有很多离群值，则你可能更愿意使用平均绝对误差。或者，你可以使用Huber损失，这是两者的组合。

 当误差小于阈值δ（通常为1）时，Huber损失为二次方，而当误差大于δ时，Huber损失为线性。线性部分使它对离群值的敏感性低于均方误差，而二次方部分使它比平均绝对误差更收敛并且更精确。

表10-1总结了回归MLP的典型架构。

表10-1：典型的回归MLP架构

超参数	典型值
输入神经元数量	每个输入特征一个（例如，MNIST 为 28×28=784）
隐藏层数量	取决于问题，但通常为 1 到 5
每个隐藏层的神经元数量	取决于问题，但通常为 10 到 100
输出神经元数量	每个预测维度输出 1 个神经元
隐藏的激活	ReLU（或 SELU，见第 11 章）
输出激活	无，或 ReLU / softplus（如果为正输出）或逻辑 / tanh（如果为有界输出）
损失函数	MSE 或 MAE / Huber（如果存在离群值）

10.1.6 分类MLP

MLP也可以用于分类任务。对于二进制分类问题，你只需要使用逻辑激活函数的单个输出神经元：输出将是0到1之间的数字，你可以将其解释为正类的估计概率。负类别的估计概率等于一减去该数字。

MLP还可以轻松处理多标签二进制分类任务（第3章）。例如，你可能有一个电子邮件分类系统，该系统可以预测每个收到的电子邮件是正常邮件还是垃圾邮件，并同时预测它是紧急电子邮件还是非紧急电子邮件。在这种情况下，你需要两个输出神经元，两个都使用逻辑激活函数：第一个输出电子邮件为垃圾邮件的可能性，第二个输出紧急邮件的可能性。更一般地，你为每个正类别用一个输出神经元。请注意，输出概率不一定要加起来为1。这可以使模型输出任何组合的标签：你可以包含非紧急正常邮件、紧急正常邮件、非紧急垃圾邮件，甚至可能是紧急垃圾邮件（尽管可能是一个错误）。

如果每个实例只能属于三个或更多可能的类中的一个类（例如，用于数字图像分类的类0到9），则每个类需要一个输出神经元，并且应该使用softmax激活函数整个输出层（见图10-9）。softmax函数（在第4章中介绍）将确保所有估计的概率在0到1之间，并且它们加起来等于1（如果类是互斥的，则是必需的）。这称为多类分类。

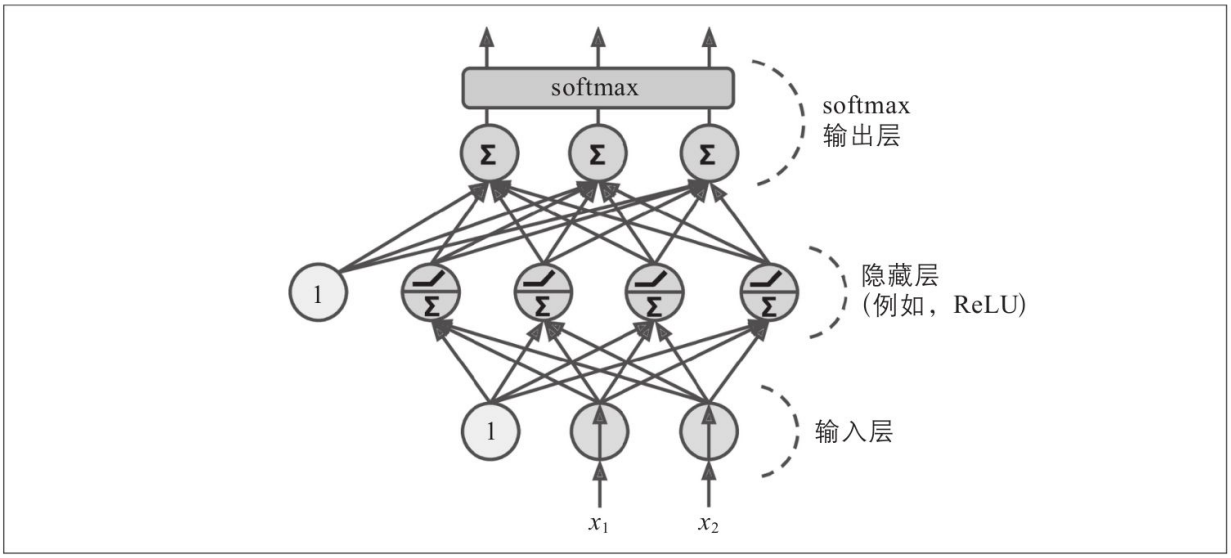


图10-9：用于分类的现代MLP（包括ReLU和softmax）

关于损失函数，由于我们正在预测概率分布，因此交叉熵损失（也称为对数损失，见第4章）通常是一个不错的选择。

表10-2总结了分类MLP的典型架构。

表10-2：典型的MLP架构

超参数	二进制分类	多标签二进制分类	多类分类
输入层和隐藏层	与回归相同	与回归相同	与回归相同
输出神经元数量	1	每个标签 1	每个类 1
输出层激活	逻辑	逻辑	softmax
损失函数	交叉熵	交叉熵	交叉熵

 在继续之前，建议你完成本章末尾的练习1。你将体验到各种神经网络架构，并使用TensorFlow Playground可视化输出。这对于更好地理解MLP所有超参数（层和神经元数、激活函数等）的影响非常有用。

现在，你了解了使用Keras实现MLP所需的所有概念！

[1] Warren S.McCulloch and Walter Pitts，“A Logical Calculus of the Ideas Immanent in Nervous Activity”，The Bulletin of Mathematical Biology 5，no.4（1943）：115-113.
[2] 它们实际上并没有连接，只是非常接近以至于它们可以非常快速地交换化学信号。
[3] 图片由Bruce Blaus提供（Creative Commons 3.0）。转载自https://en.wikipedia.org/wiki/Neuron。
[4] 在机器学习的上下文中，“神经网络”一词通常是指ANN，而不是BNN。
[5] S.Ramon y Cajal绘制的皮质层压板（公共领域）。转载自https://en.wikipedia.org/wiki/Cerebral_cortex。

[6] 感知器这个名字有时用来表示一个只有单个TLU的小型网络。
[7] 请注意，这个解不是唯一的：当数据点线性可分时，存在无限个可以将它们分离的超平面。
[8] 在20世纪90年代，具有两个以上隐藏层的人工神经网络被认为很深。如今，具有数十层甚至数百层的人工神经网络是很常见的，因此“深度”的定义非常模糊。
[9] David Rumelhart等人，“Learning Internal Representations by Error Propagation”，（Defense Technical Information Center technical report，September 1985）。
[10] 实际上，这项技术是由不同领域的不同研究人员多次独立发明的，从1974年的Paul Werbos开始。
[11] 生物神经元似乎实现了大致的sigmoid（S型）激活功能，因此研究人员在很长一段时间内一直停留在S型函数上。但是事实证明，ReLU通常在人工神经网络中效果更好。这是生物学类被误导的情况之一。

10.2 使用Keras实现MLP

Keras是高级深度学习API，可让你轻松构建、训练、评估和执行各种神经网络。其文档（或规范）可从https://keras.io/获得。这个参考实现也称为Keras，由Fransois Chollet开发，是一个研究项目[1]的一部分，并于2015年3月作为开源项目发布。由于其易用性、灵活性和精巧设计，它迅速流行。为了执行神经网络所需的繁重计算，此参考实现依赖于计算后端。目前你可以从三种流行的开源深度学习库中进行选择：TensorFlow、微软的Cognitive Toolkit（CNTK）和Theano。因此，为避免混淆，我们将此参考实现称为多后端Keras。

自2016年底以来发布了其他的实现。现在，你可以在Apache MXNet、苹果的Core ML、JavaScript或TypeScript（可以在网络浏览器中运行Keras代码）和PlaidML（可以在各种GPU设备上运行，而不仅仅是Nvidia）上运行Keras。而且，TensorFlow本身现在与自己的Keras实现程序tf.keras捆绑在一起。它仅支持TensorFlow作为后端，但具有提供一些有用的额外功能的优势（见图10-10）：例如，它支持TensorFlow的数据API，可轻松高效地加载和预处理数据。因此，我们将在本书中使用tf.keras。但是，在本章中，我们将不会使用任何特定于TensorFlow的功能，因此该代码也应该可以在其他Keras实现上很好地运行（至少在Python中），并且只需进行少量修改即可，例如更改导入。

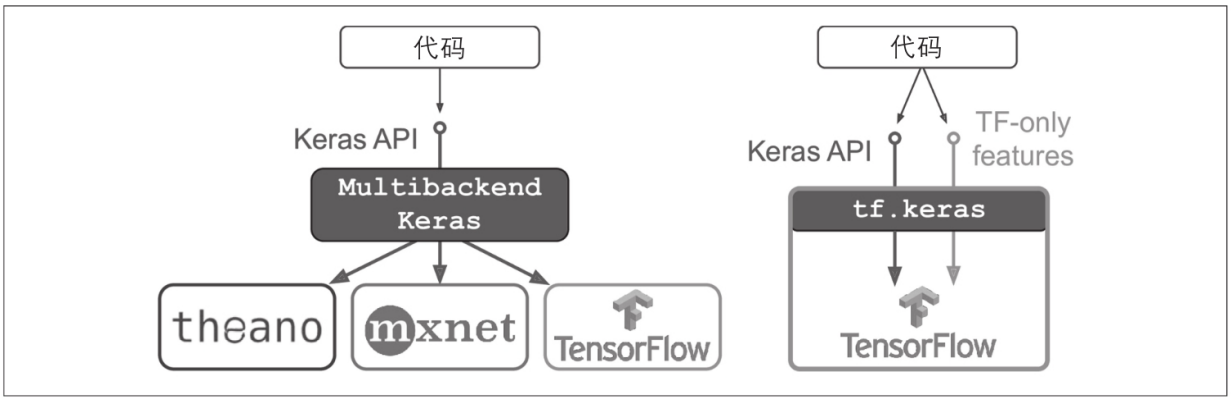


图10-10：Keras API的两种实现：多后端Keras（左）和tf.keras（右）

在Keras和TensorFlow之后，最受欢迎的深度学习库是Facebook的PyTorch库。好消息是它的API与Keras的API十分相似（部分原因是这两个API均受Scikit-Learn和Chainer的启发），因此一旦你了解Keras，便可以轻松切换到PyTorch（如果你想的话）。PyTorch的受欢迎程度在2018年呈指数增长，这主要归功于它的简单性和出色的文档，而这并不是TensorFlow 1.x的主要优势。但是TensorFlow 2可以说与PyTorch一样简单，因为它采用Keras作为其官方高级API，并且其开发人员简化和清理了其余的API。该文档也已被完全重新组织，现在更容易找到所需的内容。同样，PyTorch1.0的主要缺点（例如，有限的可移植性和无计算图分析）已得到解决。健康的竞争对所有人都有利。

好了，该写代码了！由于tf.keras与TensorFlow捆绑在一起，让我们从安装TensorFlow开始。

10.2.1 安装TensorFlow 2

假设你已按照第2章中的安装说明安装了Jupyter和Scikit-Learn，请使用pip安装TensorFlow。如果使用virtualenv创建了隔离环境，则首先需要激活它：

```
$ cd $ML_PATH                # Your ML working directory (e.g., $HOME/ml)
$ source my_env/bin/activate  # on Linux or macOS
$ .\my_env\Scripts\activate   # on Windows
```

接下来，安装TensorFlow 2（如果你不使用virtualenv，则需要管理员权限或添加--user选项）：

```
$ python3 -m pip install -U tensorflow
```

 为了支持GPU，在撰写本书时，你需要安装te n sorflow-gpu而不是tensorflow，但是TensorFlow团队正在努力开发一个单一库，该库将同时支持CPU和GPU的系统。你仍然需要安装额外的库以支持GPU（更多细节请参阅https://tensorflow.org/install）。我们将在第19章中更深入地介绍GPU。

要测试安装，请打开Python或Jupyter notebook，然后导入TensorFlow和tf.keras并打印其版本：

```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> tf.__version__
'2.0.0'
>>> keras.__version__
'2.2.4-tf'
```

第二个版本是由tf.keras实现的Keras API版本。请注意，它以-tf结尾，突出了以下事实：tf.keras实现了Keras API以及一些额外的TensorFlow特定功能。

现在让我们使用tf.keras！我们从构建一个简单的图像分类器开始。

10.2.2 使用顺序API构建图像分类器

首先，我们需要加载数据集。在本章中，我们将介绍Fashion MNIST，它是MNIST的直接替代品（在第3章中介绍）。它具有与MNIST完全相同的格式（70 000张灰度图像，每幅28×28像素，有10个类），但是这些图像代表的是时尚物品，而不是手写数字，因此每个类更有多样性，问题比MNIST更具挑战性。例如，简单的线性模型在MNIST上达到约92%的准确率，但在Fashion MNIST上仅达到约83%的准确率。

使用Keras加载数据集

Keras提供了一些实用程序来获取和加载常见数据集，包括MNIST、Fashion MNIST和我们在第2章中使用的加州房屋数据集。让我们加载Fashion MNIST：

```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

当使用Keras而不是Scikit-Learn来加载MNIST或Fashion MNIST时，一个重要的区别是每个图像都表示为28×28阵列，而不是尺寸为784的一维阵列。此外，像素强度表示为整数（从0到255）而不是浮点数（从0.0到255.0）。让我们看一下训练集的形状和数据类型：

```
>>> X_train_full.shape
(60000, 28, 28)
>>> X_train_full.dtype
dtype('uint8')
```

请注意，数据集已经分为训练集和测试集，但是没有验证集，因此我们现在创建一个。另外，由于我们要使用梯度下降训练神经网络，因此必须比例缩放输入特征。为了简单起见，我们将像素强度除以255.0（将它们转换为浮点数），将像素强度降低到0～1范围内：

```
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

对于MNIST，当标签等于5时，说明图像代表手写数字5。但是，对于Fashion MNIST，我们需要一个类名列表来知道我们要处理的内容：

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

例如，训练集中的第一幅图像代表一件外套：

```
>>> class_names[y_train[0]]
'Coat'
```

图10-11显示了来自Fashion MNIST数据集的一些示例。

图10-11：Fashion MNIST的样本

使用顺序API创建模型

现在让我们建立神经网络！这是具有两个隐藏层的分类MLP：

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

让我们逐行浏览以下代码：

• 第一行创建一个Sequential模型。这是用于神经网络的最简单的Keras模型，它仅由顺序连接的单层堆栈组成。这称为顺序API。

• 接下来，我们构建第一层并将其添加到模型中。它是Flatten层，其作用是将每个输入图像转换为一维度组：如果接收到输入数据X，则计算X.reshape（-1，1）。该层没有任何参数。它只是在那里做一些简单的预处理。由于它是模型的第一层，因此应指定input_shape，其中不包括批处理大小，而仅包括实例的形状。或者，你可以添加keras.layers.InputLayer作为第一层，设置input_shape=[28，28]。

• 接下来，我们添加具有300个神经元的Dense隐藏层。它使用ReLU激活函数。每个D e n s e层管理自己的权重矩阵，其中包含神经元及其输入之间的所有连接权重。它还管理偏置项的一个向量（每个神经元一个）。当它接收到一些输入数据时，它计算公式10-2。

• 然后，我们添加第二个有100个神经元的Dense隐藏层，还是使用ReLU激活函数。

• 最后，我们添加一个包含10个神经元的De n se输出层（每个类一个），使用softmax激活函数（因为这些类是排他的）。

 指定activation="relu"等效于指定activation=keras.activations.relu。keras.activations软件包中提供了其他激活函数，我们将在本书中使用其中的许多函数。有关完整列表，请参见https://keras.io/activations/。

可以不用像我们刚才那样逐层添加层，而可以在创建顺序模型时传递一个层列表：

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

使用来自keras.io的代码示例

keras.io上的代码示例可以与tf.keras一起正常工作，但是你需要更改imports。例如，考虑以下keras.io代码：

```
from keras.layers import Dense
output_layer = Dense(10)
```

你必须像这样更改imports：

```
from tensorflow.keras.layers import Dense
output_layer = Dense(10)
```

也可以使用完整路径：

```
from tensorflow import keras
output_layer = keras.layers.Dense(10)
```

这种方法较为冗长，但我在本书中使用它，因此你可以轻松查看要使用的软件包，并避免混淆标准类和自定义类。在生产代码中，我更喜欢前一个方法。许多人也使用from tensorflow.keras imports layer，然后是layers.Dense（10）。

模型的summary（）方法显示模型的所有层[2]，包括每个层的名称（除非在创建层时进行设置，否则会自动生成），其输出形状（None表示批处理大小任意），以及它的参数数量。总结以参数总数结尾，包括可训练参数和不可训练的参数。在这里，我们只有可训练的参数（我们将在第11章中看到不可训练参数的示例）：

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

请注意，密集层通常具有很多参数。例如，第一个隐藏层的连接权重为784×300，外加300个偏置项，总共有235 500个参数！这为模型提供了足够的灵活性来拟合训练数据，但这也意味着模型存在过拟合的风险，尤其是在你没有大量训练数据的情况下。我们稍后会再谈。

你可以轻松获取模型的层列表，按其索引获取层，也可以按名称获取：

```
>>> model.layers
[<tensorflow.python.keras.layers.core.Flatten at 0x132414e48>,
 <tensorflow.python.keras.layers.core.Dense at 0x1324149b0>,
 <tensorflow.python.keras.layers.core.Dense at 0x1356ba8d0>,
 <tensorflow.python.keras.layers.core.Dense at 0x13240d240>]
>>> hidden1 = model.layers[1]
>>> hidden1.name
'dense'
>>> model.get_layer('dense') is hidden1
True
```

可以使用get_weights（）和set_weights（）方法访问层的所有参数。对于密集层，这包括连接权重和偏置项：

```
>>> weights, biases = hidden1.get_weights()
>>> weights
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
         0.03859074, -0.06889391],
       ...,
       [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
         0.00272203, -0.06793761]], dtype=float32)
>>> weights.shape
(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., 0., ...,  0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)
```

注意，密集层随机初始化了连接权重（这是打破对称性所必需的，正如我们前面所讨论的），并且偏置被初始化为零，这是可以的。如果要使用其他初始化方法，则可以在创建层时设置kernel_initializer（内核是连接权重矩阵的另一个名称）或bias_initializer。我们将在第11章中进一步讨论初始化，但是如果需要完整的列表，请参见https://keras.io/initializers/。

 权重矩阵的形状取决于输入的个数。这就是在Sequential模型中创建第一层时建议指定input_shape的原因。但是，如果你不指定输入形状，那也是可以的：Keras会等到知道输入形状后才真正构建模型。当你向其提供实际数据时（例如，在训练期间），或者在调用其build（）方法时，就会发生这种情况。在真正构建模型之前，所有层都没有权重，而且你也无法执行某些操作（例如打印模型总结或保存模型）。因此，如果在创建模型时知道输入形状，则最好指定它。

编译模型

创建模型后，你必须调用compile（）方法来指定损失函数和要使用的优化器。你也可以选择指定在训练和评估期间要计算的其他指标：

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

 使用loss="sparse_categorical_crossentropy"等同于使用loss=keras.losses.sparse_categorical_crossentropy。同样，指定optimizer="sgd"等同于指定optimizer=keras.optimizers.SGD（），而metrics=["acc uracy"]等同于metrics=[kera s.metrics.sparse_categori cal_accuracy]（使用此损失时）。在本书中，我们将使用许多其他的损失、优化器和指标。有关完整列表，请参见https://keras.io/losses、https://keras.io/optimizers和https://keras.io/metrics。

此代码需要一些解释。首先，我们使用"sparse_categorical_crossentropy"损失，因为我们具有稀疏标签（即对于每个实例，只有一个目标类索引，在这种情况下为0到9），并且这些类是互斥的。相反，如果每个实例的每个类都有一个目标概率（例如独热向量，[0.，0.，0.，1.，0.，0.，0.，0.，0.，0]代表类3），则我们需要使用"categorical_crossentropy"损失。如果我们正在执行二进制分类（带有一个或多个二进制标签），则在输出层中使用"sigmoid"（即逻辑）激活函数，而不是"softmax"激活函数，并且使用"binary_crossentropy"损失。

 如果要将稀疏标签（即类索引）转换为独热向量标签，使用keras.utils.to_categorical（）函数。反之则使用np.argmax（）函数和axis=1。

关于优化器，"sgd"表示我们使用简单的随机梯度下降来训练模型。换句话说，Keras将执行先前所述的反向传播算法（即反向模式自动微分加梯度下降）。我们将在第11章中讨论更有效的优化器（它们改进梯度下降部分，而不是自动微分）。

 使用SGD优化器时，调整学习率很重要。因此通常需要使用optimizer=keras.optimizers.SGD（lr=???）来设置学习率，而不是使用optimizer="sgd"（默认值为lr=0.01）来设置学习率。

最后，由于这是一个分类器，因此在训练和评估过程中测量其"accuracy"很有用。

训练和评估模型

现在该模型已准备好进行训练。为此我们只需要调用其fit（）方法即可：

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [======] - 3s 49us/sample - loss: 0.7218     - accuracy: 0.7660
                                       - val_loss: 0.4973 - val_accuracy: 0.8366
Epoch 2/30
55000/55000 [======] - 2s 45us/sample - loss: 0.4840     - accuracy: 0.8327
                                       - val_loss: 0.4456 - val_accuracy: 0.8480
[...]
Epoch 30/30
55000/55000 [======] - 3s 53us/sample - loss: 0.2252     - accuracy: 0.9192
                                       - val_loss: 0.2999 - val_accuracy: 0.8926
```

我们将输入特征（X_train）和目标类（y_train）以及要训练的轮次数传递给它（否则它将默认为1，这绝对不足以收敛为一个好的模型）。我们还传递了一个验证集（这是可选的）。Keras将在每个轮次结束时测量此集合上的损失和其他指标，这对于查看模型的实际效果非

常有用。如果训练集的性能好于验证集，则你的模型可能过拟合训练集（或者存在错误，例如训练集和验证集之间的数据不匹配）。

就是这样！训练了神经网络注[3]。在训练期间的每个轮次，Keras会显示到目前为止已处理的实例数（以及进度条）、每个样本的平均训练时间、损失和精度（或你要求的任何其他额外指标）（针对训练集和验证集）。你可以看到训练损失减少了，这是一个好兆头，经过30个轮次后，验证准确率达到了89.26%。这与训练精确率相差不大，因此似乎并没有发生过拟合现象。

 你可以将validation_split设置为希望Keras用于验证的训练集的比率，而不是使用validation_data参数传递验证集。例如，validation_split=0.1告诉Keras使用数据的最后10%（在乱序之前）进行验证。

如果训练集非常不平衡，其中某些类的代表过多，而其他类的代表不足，那么在调用fit（）方法时设置class_weight参数会很有用，这给代表性不足的类更大的权重，给代表过多的类更小的权重。Keras在计算损失时将使用这些权重。如果你需要每个实例的权重，设置sample_weight参数（如果class_weight和sample_weight都提供了，Keras会把它们相乘）。如果某些实例由专家标记，而另一些实例使用众包平台标记，则按实例权重可能会有用：你可能希望为前者赋予更多权重。你还可以通过将其作为validation_data元组的第三项添加到验证集中来提供样本权重（但不提供类权重）。

fit（）方法返回一个History对象，其中包含训练参数（history.params）、经历的轮次列表（history.epoch），最重要的是包含在训练集和验证集（如果有）上的每个轮次结束时测得的损失和额外指标的字典（history.history）。如果使用此字典创建pandas DataFrame并调用其plot（）方法，则会获得如图10-12所示的学习曲线：

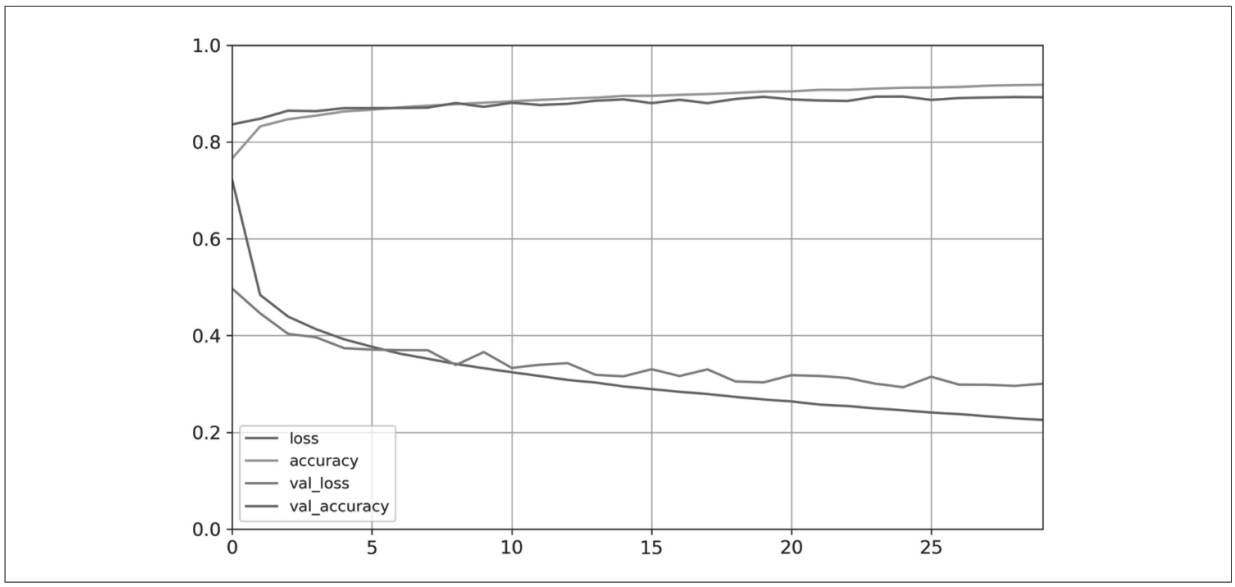


图10-12：学习曲线：每个轮次测得的平均训练损失和准确率，以及每个轮次结束时测得的平均验证损失和准确率

```
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```

你可以看到训练期间训练准确率和验证准确率都在稳步提高，而训练损失和验证损失则在下降。好！而且，验证曲线与训练曲线很接近，这意味着没有太多的过拟合。在这种特殊情况下，该模型看起来在验证集上的表现要好于训练开始时在训练集上的表现。但是事实并非如此：确实，验证误差是在每个轮次结束时计算的，而训练误差是使用每个轮次的运行平均值计算的。因此，训练曲线应向左移动半个轮次。如果这样做，你会看到训练和验证曲线在训练开始时几乎完全重叠。

 绘制训练曲线时，应将其向左移动半个轮次。

训练集的性能最终会超过验证性能，就像通常情况下训练足够长的时间一样。你可以说模型尚未完全收敛，因为验证损失仍在下降，因此你可能应该继续训练。这就像再次调用fit（）方法那样简单，因为Keras只是从它停止的地方继续训练（你应该能够达到接近89%的验证准确率）。

如果你对模型的性能不满意，则应回头调整超参数。首先要检查的是学习率。如果这样做没有帮助，请尝试使用另一个优化器（并在更改任何超参数后始终重新调整学习率）。如果性能仍然不佳，则尝试调整模型超参数（例如层数、每层神经元数以及用于每个隐藏层的激活函数的类型）。你还可以尝试调整其他超参数，例如批处理大小（可以使用batch_size参数在fit（）方法中进行设置，默认为32）。在本章的最后，我们将回到超参数调整。对模型的验证精度感到满意后，应在测试集上对其进行评估泛化误差，然后再将模型部署到生产环境中。你可以使用evaluate（）方法轻松完成此操作（它还支持其他几个参数，例如batch_size和sample_weight，请查看文档以获取更多详细信息）：

```
>>> model.evaluate(X_test, y_test)
10000/10000 [==========] - 0s 29us/sample - loss: 0.3340 - accuracy: 0.8851
[0.3339798209667206, 0.8851]
```

正如我们在第2章中看到的那样，在测试集上获得比在验证集上略低的性能是很常见的，因为超参数是在验证集而不是测试集上进行调优的（但是在本示例中，我们没有做任何超参数调整，因此较低的精度只是运气不好）。切记不要调整测试集上的超参数，否则你对泛化误差的估计将过于乐观。

使用模型进行预测

接下来，我们可以使用模型的predict（）方法对新实例进行预测。由于没有实际的新实例，因此我们将仅使用测试集的前三个实例：

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.03, 0.  , 0.01, 0.  , 0.96],
       [0.  , 0.  , 0.98, 0.  , 0.02, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

如你所见，对于每个实例，模型估计从0类到9类每个类的概率。例如，对于第一个图像，模型估计是第9类（脚踝靴）的概率为96%，第5类的概率（凉鞋）为3%，第7类（运动鞋）的概率为1%，其他类别的概率可忽略不计。换句话说，它“相信”第一个图像是鞋类，最有可能是脚踝靴，但也可能是凉鞋或运动鞋。如果你只关心估计概率最高的类（即使该概率非常低），则可以使用pre dict_classes（）方法：

```
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

在这里，分类器实际上对所有三个图像进行了正确分类（图像如图10-13所示）：

```
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```

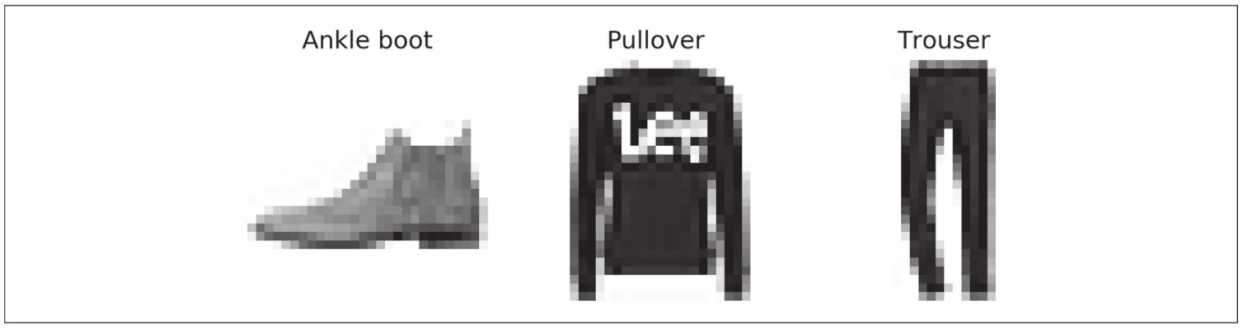


图10-13：正确分类的Fashion MNIST图像

现在，你知道如何使用顺序API来构建、训练、评估和使用分类MLP。但是回归呢？

10.2.3 使用顺序API构建回归MLP

让我们转到加州的住房问题，并使用回归神经网络解决它。为简单起见，我们将使用Scikit-Learn的fetch_california_housing（）函数加载数据。该数据集比我们在第2章中使用的数据集更简单，因为它仅包含数字特征（没有ocean_proximity特征），并且没有缺失值。加载数据后，我们将其分为训练集、验证集和测试集，然后比例缩放所有特征：

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

使用顺序API来构建、训练、评估和使用回归MLP进行预测与我们进行分类非常相似。主要区别在于输出层只有一个神经元（因为我们只预

测一个单值），并且不使用激活函数，而损失函数是均方误差。由于数据集噪声很大，我们只使用比以前少的神经元的单层隐藏层，以避免过拟合：

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer="sgd")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3] # pretend these are new instances
y_pred = model.predict(X_new)
```

如你所见，顺序API非常易于使用。但是尽管顺序模型非常普遍，但有时构建具有更复杂拓扑结构或具有多个输入或输出的神经网络还是常见的。为此，Keras提供了函数式API。

10.2.4 使用函数式API构建复杂模型

非顺序神经网络的一个示例是“宽深”神经网络。这种神经网络架构是由Heng-Tze Cheng等人在2016年发表的论文引入的[4]。它将所有或部分输入直接连接到输出层，如图10-14所示。这种架构使神经网络能够学习深度模式（使用深度路径）和简单规则（通过短路径）[5]。相比之下，常规的MLP迫使所有数据流经整个层的堆栈。

因此，数据的简单模式最终可能会因为顺序被转换而失真。

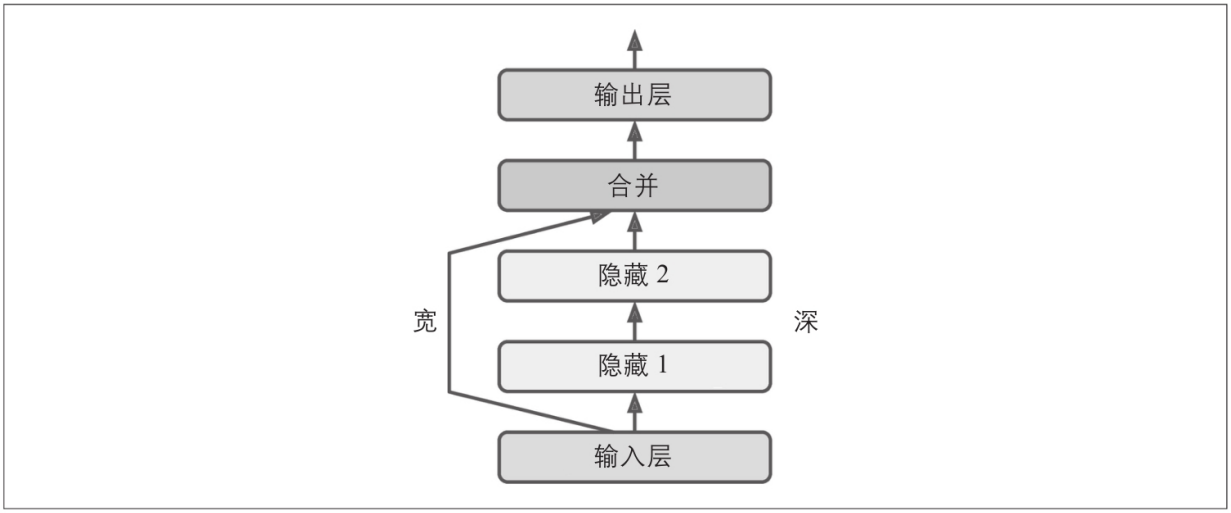


图10-14：宽深神经网络

让我们建立这样一个神经网络来解决加州的住房问题：

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.Concatenate()([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])
```

让我们遍历这段代码的每一行：

• 首先，我们需要创建一个I n p ut对象[6]。这是模型需要的输入类型的规范，包括其shape和dtype。我们很快就会看到，一个模型实际上可能有多个输入。

• 接下来，我们创建一个包含30个神经元的D e n s e层，使用ReLU激活函数。创建它后，请注意，我们像调用函数一样将其传递给输入。这就是将其称为函数式API的原因。注意，我们只是在告诉Keras它应该如何将各层连接在一起。尚未处理任何实际数据。

• 然后，我们创建第二个隐藏层，然后再次将其用作函数。请注意，我们将第一个隐藏层的输出传递给它。

• 接下来，我们创建一个Con cate nate层，再次像函数一样立即使用它来合并输入和第二个隐藏层的输出。你可能更喜欢keras.layers.concatenate（）函数，该函数创建一个Concatenate层并立即使用给定的输入对其进行调用。

• 然后我们创建具有单个神经元且没有激活函数的输出层，然后像函数一样调用它，将合并结果传递给它。

• 最后，我们创建一个Keras Model，指定要使用的输入和输出。

一旦构建了Keras模型，一切都与之前的一样，因此无须在此处重复：你必须编译模型，对其进行训练，评估并使用它来进行预测。

但是如果你想通过宽路径送入特征的子集，而通过深路径送入特征的另一个子集（可能有重合）呢（见图10-15）？在这种情况下，一种解决方案是使用多个输入。例如，假设我们要通过宽路径送入5个特征（特征0到4），并通过深路径送入6个特征（特征2到7）：

```
input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="output")(concat)
model = keras.Model(inputs=[input_A, input_B], outputs=[output])
```

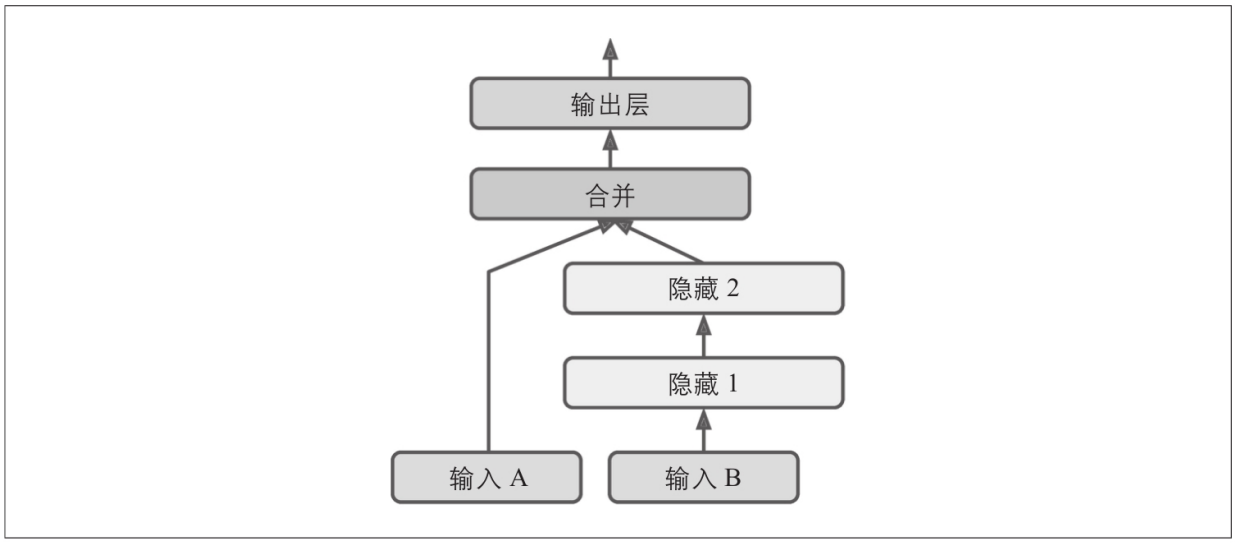


图10-15：处理多输入

该代码是不言自明的。你应该至少命名最重要的层，尤其是当模型变得有点复杂时。请注意，在创建模型时，我们指定了input=[input_A，input_B]。现在我们可以像往常一样编译模型了，但是当我们调用fit（）方法时，必须传递一对矩阵（X_train_A，X_train_B）：各输入一个矩阵[7]，而不是传递单个输入矩阵X_train。当你调用evaluate（）或predict（）时，X_valid、X_test和X_new同样如此：

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))

X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                    validation_data=((X_valid_A, X_valid_B), y_valid))
mse_test = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B))
```

在许多用例中，你可能需要多个输出：

• 这个任务可能会需要它。例如，你可能想在图片中定位和分类主要物体。这既是回归任务（查找物体中心的坐标以及宽度和高度），又是分类任务。

• 同样，你可能有基于同一数据的多个独立任务。当然你可以为每个任务训练一个神经网络，但是在许多情况下，通过训练每个任务一个输出的单个神经网络会在所有任务上获得更好的结果。这是因为神经网络可以学习数据中对任务有用的特征。例如，你可以对面部图片执行多任务分类，使用一个输出对人的面部表情进行分类（微笑、惊讶等），使用另一个输出来识别他们是否戴着眼镜。

• 另一个示例是作为正则化技术（即训练约束，其目的是减少过拟合，从而提高模型的泛化能力）。例如，你可能希望在神经网络结构中添加一些辅助输出（见图10-16），以确保网络的主要部分自己能学习有用的东西，而不依赖于网络的其余部分。

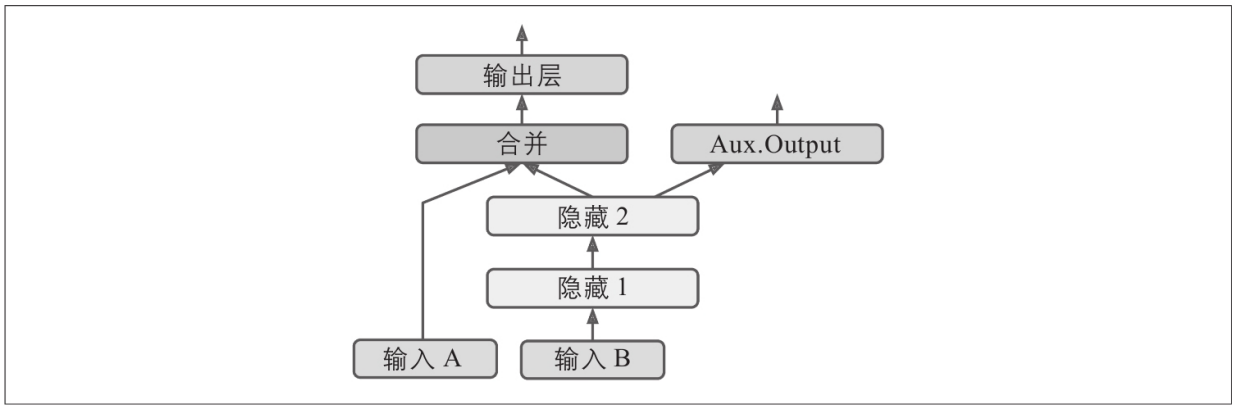


图10-16：处理多个输出，在此示例中，添加辅助输出以进行正则化

添加额外的输出非常容易：只需将它们连接到适当的层，然后将它们添加到模型的输出列表中即可。例如，以下代码构建了如图10-16所示的网络：

```
[...] # Same as above, up to the main output layer
output = keras.layers.Dense(1, name="main_output")(concat)
```

```
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)
model = keras.Model(inputs=[input_A, input_B], outputs=[output, aux_output])
```

每个输出都需要自己的损失函数。因此当我们编译模型时，应该传递一系列损失[8]（如果传递单个损失，Keras将假定所有输出必须使用相同的损失）。默认情况下，Keras将计算所有这些损失，并将它们简单累加即可得到用于训练的最终损失。我们更关心主要输出而不是辅助输出（因为它仅用于正则化），因此我们要给主要输出的损失更大的权重。幸运的是，可以在编译模型时设置所有的损失权重：

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1], optimizer="sgd")
```

现在当训练模型时，需要为每个输出提供标签。在此示例中，主要输出和辅助输出应预测出相同的结果，因此它们应使用相同的标签。除了传递y_train之外，还需要传递（y_train，y_train）（对于y_valid和y_test也是如此）：

```
history = model.fit(
    [X_train_A, X_train_B], [y_train, y_train], epochs=20,
    validation_data=([X_valid_A, X_valid_B], [y_valid, y_valid]))
```

当评估模型时，Keras将返回总损失以及所有单个损失：

```
total_loss, main_loss, aux_loss = model.evaluate(
    [X_test_A, X_test_B], [y_test, y_test])
```

同样，predict（）方法将为每个输出返回预测值：

```
```

```
y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```

如你所见，你可以使用函数式API轻松构建所需的任何网络结构。让我们看一下构建Keras模型的最后一种方法。

10.2.5 使用子类API构建动态模型

顺序API和函数式API都是声明性的：首先声明要使用的层以及应该如何连接它们，然后才能开始向模型提供一些数据进行训练或推断。这具有许多优点：可以轻松地保存、克隆和共享模型；可以显示和分析它的结构；框架可以推断形状和检查类型，因此可以及早发现错误（即在任何数据通过模型之前）。由于整个模型是一个静态图，因此调试起来也相当容易。但另一方面是它是静态的。一些模型涉及循环、变化的形状、条件分支和其他动态行为。对于这种情况，或者只是你喜欢命令式的编程风格，则子类API非常适合你。

只需对Model类进行子类化，在构造函数中创建所需的层，然后在call（）方法中执行所需的计算即可。例如，创建以下WideAndDeepModel类的实例将给我们一个等效于刚刚使用函数式API构建的模型。然后，你可以像我们刚做的那样对其进行编译、评估并使用它进行预测：

```
class WideAndDeepModel(keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs) # handles standard args (e.g., name)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output

model = WideAndDeepModel()
```

这个示例看起来非常类似于函数式API，只是我们不需要创建输入。我们只使用call（）方法的输入参数，就可以将构造函数中层的创建与其在call（）方法中的用法分开[9]。最大的区别是你可以在call（）方法中执行几乎所有你想做的操作：for循环、if语句、底层TensorFlow操作，等等（见第12章）。这使得它成为研究新想法的研究人员的绝佳API。

这种额外的灵活性的确需要付出一定的代价：模型的架构隐藏在call（）方法中，因此Keras无法对其进行检查。它无法保存或克隆。当你调用summary（）方法时，你只会得到一个图层列表，而没有有关它们如何相互连接的信息。而且Keras无法提前检查类型和形状，并且更容易出错。因此，除非你确实需要这种额外的灵活性，否则你应该坚持使用顺序API或函数式API。

 Keras模型可以像常规层一样使用，因此你可以轻松地将它们组合以构建复杂的结构。

现在你知道如何使用Keras构建和训练神经网络了，那么如何保存它们呢？

10.2.6　保存和还原模型

使用顺序API或函数式API时，保存训练好的Keras模型非常简单：

```
model = keras.models.Sequential([...]) # or keras.Model([...])
model.compile([...])
model.fit([...])
model.save("my_keras_model.h5")
```

Keras使用HDF5格式保存模型的结构（包括每一层的超参数）和每一层的所有模型参数值（例如，连接权重和偏置）。它还可以保存优化器（包括其超参数及其可能具有的任何状态）。在第19章，我们可以看到如何使用Tensorflow的SavedModel格式来保存一个tf.keras模型。

通常你有一个训练模型并保存模型的脚本，以及一个或多个加载模型并使用其进行预测的脚本（或Web服务）。加载模型同样简单：

```
model = keras.models.load_model("my_keras_model.h5")
```

 当使用顺序API或函数式API时，这是适用的，但不幸的是，在使用模型子类化时，它将不起作用。你至少可以使用save_weights（）和load_weights（）来保存和还原模型参数，但是你需要自己保存和还原其他所有内容。

但是，如果训练持续几个小时怎么办？这是很常见的，尤其是在大型数据集上进行训练时。在这种情况下，你不仅应该在训练结束时保存模型，还应该在训练过程中定期保存检查点，以免在计算机崩溃时丢失所有内容。但是如何告诉fit（）方法保存检查点呢？使用回调。

10.2.7　使用回调函数

fit（）方法接受一个callbacks参数，该参数使你可以指定Keras在训练开始和结束时，每个轮次的开始和结束时（甚至在处理每个批量之前和之后）将调用的对象列表。例如，在训练期间ModelCheckpoint回调会定期保存模型的检查点，默认情况下，在每个轮次结束时：

```
[...] # build and compile the model
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

此外，如果在训练期间使用验证集，则可以在创建ModelCheckpoint时设置save_best_only=True。在这种情况下，只有在验证集上的模型性能达到目前最好时，它才会保存模型。这样，你就不必担心训练时间太长而过拟合训练集：只需还原训练后保存的最后一个模型，这就是验证集中的最佳模型。以下代码是实现提前停止的简单方法（见第4章）：

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb])
model = keras.models.load_model("my_keras_model.h5") # roll back to best model
```

实现提前停止的另一种方法是使用EarlyStopping回调。如果在多个轮次（由patience参数定义）的验证集上没有任何进展，它将中断训练，并且可以选择回滚到最佳模型。你可以将两个回调结合起来以保存模型的检查点（以防计算机崩溃），并在没有更多进展时尽早中断训练（以避免浪费时间和资源）：

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
```

可以将轮次数设置为较大的值，因为训练将在没有更多进展时自动停止。在这种情况下，无须还原保存的最佳模型，因为EarlyStopping回调将跟踪最佳权重，并在训练结束时为你还原它。

 keras.callbacks包中还有许多其他回调函数。

如果需要额外的控制，则可以轻松编写自己的自定义回调。作为如何执行的示例，以下自定义回调将显示训练过程中验证损失与训练损失之间的比率（例如，检测过拟合）：

```
class PrintValTrainRatioCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print("\nval/train: {:.2f}".format(logs["val_loss"] / logs["loss"]))
```

如你所料，你可以实现on train be__gin（）、on_train_end（）、on_epoch_begin（）、on e_poch_end（）、on_batch_begin（）和on_batch_end（）。如果需要的话，在评估和预测期间也可以使用回调（例如用于调试）。为了进行评估，你应该实现on_test_begin（）、on_test_end（）、on_test_batch_begin（）或on test__batch end_（）（由evaluate（）调用），为了进行预测，你应该实现on_predict_begin（）、on_pre dict_end（）、on_predict_batch_begin（）或on_predict_batch_end（）（由predict（）调用）。

现在让我们看看使用tf.keras时肯定应该在工具箱中有的另一种工具：TensorBoard。

10.2.8　使用TensorBoard进行可视化

TensorBoard是一款出色的交互式可视化工具，可用于在训练期间查看学习曲线；比较多次运行的学习曲线；可视化计算图；分析训练统计数据；查看由模型生成的图像；把复杂的多维数据投影到3D，自动聚类并进行可视化，等等！安装TensorFlow时会自动安装此工具，因此你已经拥有了它。

要使用它，你必须修改程序以便将要可视化的数据输出到名为事件文件的特殊二进制日志文件中。每个二进制数据记录称为摘要。

TensorBoard服务器将监视日志目录，并将自动获取更改并更新可视化效果：这使你可以可视化实时数据（有短暂延迟），例如训练期间的学习曲线。通常你需要把TensorBoard服务器指向根日志目录并配置程序，以使其在每次运行时都写入不同的子目录。这样相同的TensorBoard服务器实例可以使你可视化并比较程序多次运行中的数据，而不会混淆所有内容。

让我们首先定义用于TensorBoard日志的根日志目录，再加上一个将根据当前日期和时间生成一个子目录的函数，以便每次运行时都不同。你可能希望在日志目录中包含其他信息，例如你正在测试的超参数值，以使你更容易知道在TensorBoard中查看的内容：

```
import os
root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():
    import time
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir() # e.g., './my_logs/run_2019_06_07-15_15_22'
```

好消息是Keras提供了一个不错的TensorBoard（）回调：

```
[...] # Build and compile your model
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid),
                    callbacks=[tensorboard_cb])
```

这就是全部！不难使用。如果运行此代码，TensorBoard（）回调将为你创建日志目录（如果需要，还有父目录），并且在训练期间它将创建事件文件并向其写入摘要。第二次运行该程序（也许更改某些超参数值）后，你将得到一个类似于以下内容的目录结构：

```
my_logs/
├── run_2019_06_07-15_15_22
│   ├── train
│   │   ├── events.out.tfevents.1559891732.mycomputer.local.38511.694049.v2
│   │   ├── events.out.tfevents.1559891732.mycomputer.local.profile-empty
│   │   └── plugins/profile/2019-06-07_15-15-32
│   │       └── local.trace
│   └── validation
│       └── events.out.tfevents.1559891733.mycomputer.local.38511.696430.v2
└── run_2019_06_07-15_15_49
    └── [...]
```

每次运行有一个目录，每个目录包含一个用于训练日志的子目录和一个用于验证日志的子目录。两者都包含事件文件，但是训练日志还包含概要分析跟踪：这使TensorBoard可以准确显示模型在所有设备上花费在模型各部分上的时间，对于查找性能瓶颈非常有用。

接下来，你需要启动TensorBoard服务器。一种方法是在终端中运行命令。如果你在虚拟环境中安装了TensorFlow，则应将其激活。接下来在项目的根目录（或从任何其他位置，只要指向适当的日志目录）运行以下命令：

```
$ tensorboard --logdir=./my_logs --port=6006
TensorBoard 2.0.0 at http://mycomputer.local:6006/ (Press CTRL+C to quit)
```

如果你的终端找不到tensorboard脚本，那你必须更新PATH环境变量，以便它包含脚本安装目录（或者你可以在命令行中用python3-m tensorboard.main替换tensorboard）。服务器启动后，你可以打开Web浏览器并转到http://localhost:6006。

或者你可以直接在Jupyter中运行以下命令来使用TensorBoard。第一行加载TensorBoard扩展，第二行在端口6006上启动TensorBoard服务器（除非它已经启动）并连接到它：

```
%load_ext tensorboard
%tensorboard --logdir=./my_logs --port=6006
```

无论用哪种方式，你都应该看到TensorBoard的Web界面。单击“SCALARS”选项卡以查看学习曲线（见图10-17）。在左下方，选择要显示的日志（例如，第一次和第二次运行的训练日志），然后单击epoch_loss标量。请注意两次运行的训练损失都下降得很好，但是第二次下降得更快。实际上，我们使用的学习率为0.05（optimizer=keras.optimizers.SGD（lr=0.05））， 而不是0.001。

你还可以可视化整个图形、学习的权重（投影到3D）或分析跟踪。TensorBoard（）回调也具有记录额外数据的选项，例如嵌入（见第13章）。

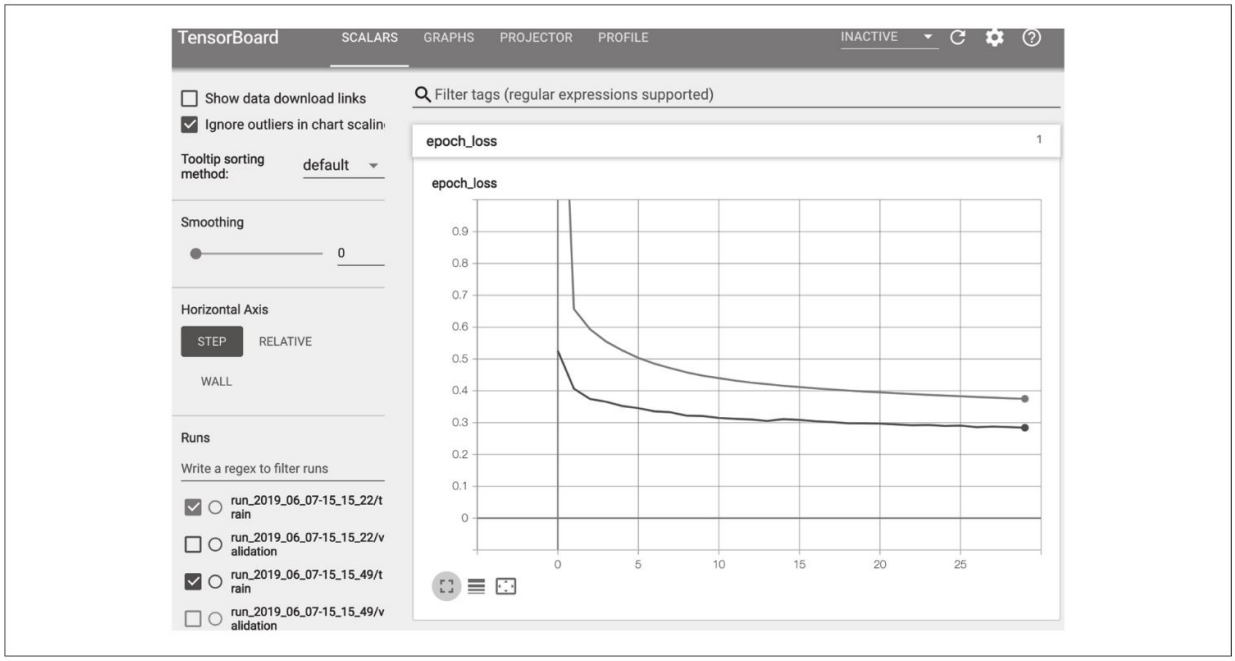


图10-17：用TensorBoard可视化学习曲线

此外，TensorFlow在tf.summary包中提供了一个较底层的API。以下代码使用create_file_writer（）函数创建一个S ummaryWriter，并

将该函数用作上下文来记录标量、直方图、图像、音频和文本，然后可以使用TensorBoard将其可视化（尝试一下！）：

```
test_logdir = get_run_logdir()
writer = tf.summary.create_file_writer(test_logdir)
with writer.as_default():
    for step in range(1, 1000 + 1):
        tf.summary.scalar("my_scalar", np.sin(step / 10), step=step)
        data = (np.random.randn(100) + 2) * step / 100 # some random data
        tf.summary.histogram("my_hist", data, buckets=50, step=step)
        images = np.random.rand(2, 32, 32, 3) # random 32×32 RGB images
        tf.summary.image("my_images", images * step / 1000, step=step)
        texts = ["The step is " + str(step), "Its square is " + str(step**2)]
        tf.summary.text("my_text", texts, step=step)
        sine_wave = tf.math.sin(tf.range(12000) / 48000 * 2 * np.pi * step)
        audio = tf.reshape(tf.cast(sine_wave, tf.float32), [1, -1, 1])
        tf.summary.audio("my_audio", audio, sample_rate=48000, step=step)
```

这实际上是一个有用的可视化工具，甚至超过了TensorFlow或深度学习。

让我们总结一下你到目前为止在本章中学到的知识：我们了解了神经网络的来源，MLP是什么以及如何将其用于分类和回归，如何使用tf.keras的顺序API来构建MLP，以及如何使用函数式API或子类API来构建更复杂的模型架构。你学习了如何保存和还原模型，以及如何使用回调函数来保存检查点，提前停止，等等。最后，你学习了如何使用TensorBoard进行可视化。你已经可以使用神经网络来解决许多问题了！但是，你可能想知道如何选择隐藏层的数量、网络中神经元的数量以及所有其他超参数。让我们现在来看一下。

[1] ONEIROS项目（开放式神经电子智能机器人操作系统）。
[2] 你可以使用keras.utils.plot_model（）来生成模型的图像。
[3] 如果训练数据或验证数据与预期的形状不匹配，则会出现异常。这可能是最常见的错误，因此你应该熟悉错误消息。该消息实际上非常清楚：例如，如果你尝试使用包含一维图像的数组（X_train.reshape（-1，784））训练该模型，则将出现以下异常：“ValueError：Error when checking input：expectedflatten input

to have 3 dimensions, but got array with shape (60000, 784)”。

[4] Heng-Tze Cheng et al., “Wide&Deep Learning for Recommender Systems”, Proceedings of the First Workshop on Deep Learning for Recommender Systems (2016): 7-10.

[5] 短路径还可以用于向神经网络提供手动设计的特征工程。

[6] 名称input_用于避免覆盖Python的内置input()函数。

[7] 另外你可以传递一个将输入名称映射到输入值的字典，例如{"wide_input": X_train_A、"deep_input": X_train_B}。当有很多输入时，这特别有用，可以避免顺序错误。

[8] 或者你可以传递一个字典，该字典将每个输出名称映射到相应的损失。就像输入一样，当有多个输出时，这很有用，可以避免顺序错误。损失权重和指标（稍后讨论）也可以使用字典进行设置。

[9] Keras模型具有一个output属性，因此我们不能将该名称用于主输出层，这就是为什么我们将其重命名为main_output的原因。

10.3 微调神经网络超参数

神经网络的灵活性也是它们的主要缺点之一：有许多需要调整的超参数。你不仅可以使用任何可以想象的网络结构，而且即使在简单的MLP中，你也可以更改层数、每层神经元数、每层要使用的激活函数的类型、权重初始化逻辑，以及更多。你如何知道哪种超参数最适合你的任务？

一种选择是简单地尝试超参数的许多组合，然后查看哪种对验证集最有效（或使用K折交叉验证）。例如我们可以像第2章中一样使用GridSearchCV或RandomizedSearchCV来探索超参数空间。为此我们需要将Keras模型包装在模仿常规Scikit-Learn回归器的对象中。第一步是创建一个函数，该函数将在给定一组超参数的情况下构建并编译Keras模型：

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation="relu"))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(lr=learning_rate)
    model.compile(loss="mse", optimizer=optimizer)
    return model
```

此函数为单变量回归（仅一个输出神经元）创建简单的Sequential模型，使用给定的输入形状以及给定数量的隐藏层和神经元，并使用配置了指定学习率的SGD优化器对其进行编译。像Scikit-Learn一样，最好为尽可能多的超参数提供合理的默认值。

接下来，让我们基于build_model（）函数创建一个KerasRegressor：

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

KerasRegressor对象是使用build_model（）构建的Keras模型的一个包装。由于创建时未指定任何超参数，因此它将使用我们在build_model（）中定义的默认超参数。现在，我们可以像常规Scikit-Learn回归器一样使用该对象：我们可以使用其fit（）方法进行训练，然后使用其score（）方法进行评估，然后使用其predict（）方法进行预测，如以下代码所示：

```
keras_reg.fit(X_train, y_train, epochs=100,
              validation_data=(X_valid, y_valid),
              callbacks=[keras.callbacks.EarlyStopping(patience=10)])
mse_test = keras_reg.score(X_test, y_test)
y_pred = keras_reg.predict(X_new)
```

请注意，传递给fit（）方法的任何其他参数都将传递给内部的Keras模型。还要注意，该分数将与MSE相反，因为Scikit-Learn希望获得分数，而不是损失（即分数越高越好）。

我们不想训练和评估这样的单个模型，尽管我们想训练数百个变体，并查看哪种变体在验证集上表现最佳。由于存在许多超参数，因此最好使用随机搜索而不是网格搜索（见第2章）。让我们尝试探索隐藏层的数量、神经元的数量和学习率：

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distribs = {
    "n_hidden": [0, 1, 2, 3],
    "n_neurons": np.arange(1, 100),
    "learning_rate": reciprocal(3e-4, 3e-2),
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distribs, n_iter=10, cv=3)
rnd_search_cv.fit(X_train, y_train, epochs=100,
                  validation_data=(X_valid, y_valid),
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

这与我们在第2章中所做的相同，只是这里我们将额外的参数传递给fit（）方法，并将它们传递给内部的Keras模型。请注意，RandomizedSearchCV使用K折交叉验证，因此它不使用X_valid和y_valid，它们仅用于提前停止。

探索可能持续数小时，具体时间取决于硬件、数据集的大小、模型的复杂性以及n_iter和cv的值。当结束时，你可以访问找到的最佳参数、最佳分数和经过训练的Keras模型，如下所示：

```
>>> rnd_search_cv.best_params_
{'learning_rate': 0.0033625641252688094, 'n_hidden': 2, 'n_neurons': 42}
>>> rnd_search_cv.best_score_
-0.3189529188278931
>>> model = rnd_search_cv.best_estimator_.model
```

现在你可以保存该模型，在测试集上对其进行评估，如果对它的性能满意，可以将其部署到生产环境中。使用随机搜索并不困难，它可以很好地解决许多相当简单的问题。但是，当训练速度很慢时（对于具有较大数据集的更复杂的问题），此方法将仅探索超参数空间的一小部分。你可以通过手动协助搜索过程来部分缓解此问题：首先使用宽范围的超参数值快速进行随机搜索，然后使用以第一次运行中找到的最佳值为中心，用较小范围的值运行另一个搜索，以此类推。该方法有望放大一组好的超参数。但是，这非常耗时，也不能最好地利用时间。

幸运的是，有许多技术可以比随机方法更有效地探索搜索空间。它们的核心思想很简单：当空间的某个区域被证明是好的时，应该对其进行更多的探索。此类技术可为你解决“缩放”过程，并在更短的时间内提供更好的解决方案。以下是一些可用于优化超参数的Python库：

Hyperopt

一个流行的库，用于优化各种复杂的搜索空间（包括诸如学习率的实数值和诸如层数的离散值）。

Hyperas、kopt或Talos

有用的库，用于优化Keras模型的超参数（前两个基于Hyperopt）。

Keras Tuner

Google针对Keras模型提供的易于使用的超参数优化库，可用于可视化和分析的托管服务。

Scikit-Optimize（skopt）

通用优化库。BayesSearchCV类使用类似于GridSearchCV的接口来进行贝叶斯优化。

Spearmint

贝叶斯优化库。

Hyperband

基于Lisha Li等人的最新Hyperband论文[1]的快速超参数调整库。

Sklearn-Deap

基于进化算法的超参数优化库，具有类似于GridSearchCV的接口。

此外许多公司提供超参数优化服务。我们将在第19章中讨论Google Cloud AI Platform的超参数调整服务。其他选项包括Arimo和SigOpt的服务以及CallDesk的Oscar。

超参数调整仍然是研究的活跃领域，而进化算法正在卷土重来。例如，查看DeepMind的2017年的优秀论文[2]，作者同时优化了整体模型及其超参数。Google还使用了一种进化方法，不仅用于搜索超参数，而且还为该问题寻找最佳神经网络架构。他们的AutoML套件已作为云服务提供。也许手动构建神经网络的时代即将结束？可以查看Google关于该主题的文章。实际上，进化算法已成功用于训练单个神经网络，从而取代了无处不在的梯度下降！请参阅Uber在2017年发表的文章，作者介绍了他们的Deep Neuroevolution技术。

但是尽管取得了这些令人振奋的进步以及这些工具和服务，但是了解每个超参数的合理值，仍然有助于你构建快速原型并限制搜索空间。以下各节为选择MLP中隐藏层和神经元的数量以及为某些主要超参数选择合适的值提供了指导。

10.3.1 隐藏层数量

对于许多问题，你可以从单个隐藏层开始并获得合理的结果。只要具有足够的神经元，只有一个隐藏层的MLP理论上就可以对最复杂的功能进行建模。但是对于复杂的问题，深层网络的参数效率要比浅层网络高得多：与浅层网络相比，深层网络可以使用更少的神经元对复杂的函数进行建模，从而使它们在相同数量的训练数据下可以获得更好的性能。

为了理解原因，假设要求你使用某些绘图软件来绘制森林，但禁止复制和粘贴任何内容。这要花费大量的时间：你必须分别绘制每棵树（逐个分支，逐个叶子）。如果你可以改为绘制一片叶子，将其复制并粘贴以绘制一个分支，然后复制并粘贴该分支以创建一棵树，最后复制并粘贴此树以创建森林，那么你将很快完成。现实世界中的数据通常以这种层次结构进行构造，而深度神经网络会自动利用这一事实：较低的隐藏层对低层结构（例如形状和方向不同的线段）建模，中间的隐藏层组合这些低层结构，对中间层结构（例如正方形、圆形）进行建模，而最高的隐藏层和输出层将这些中间结构组合起来，对高层结构（例如人脸）进行建模。

这种分层架构不仅可以帮助DNN更快地收敛到一个好的解，而且还可以提高DNN泛化到新数据集的能力。例如，如果你已经训练了一个模型来识别图片中的人脸，并且现在想训练一个新的神经网络来识别发型，则可以通过重用第一个网络的较低层来开始训练。你可以将它们初始化为第一个网络较低层的权重和偏置值，而不是随机初始化新神经网络前几层的权重和偏置值。这样，网络就不必从头开始学习大多数图片中出现的所有底层结构。只需学习更高层次的结构（例如发型）。这称为迁移学习。

总而言之，对于许多问题，你可以仅从一两个隐藏层开始，然后神经网络就可以正常工作。例如，仅使用一个具有几百个神经元的隐藏层，就可以轻松地在MNIST数据集上达到97％以上的准确率，而使用具有相同总数的神经元的两个隐藏层，可以在大致相同训练时间上轻松达到98％以上的精度。对于更复杂的问题，你可以增加隐藏层的数量，直到开始过拟合训练集为止。非常复杂的任务（例如图像分类或语音识别）通常需要具有数十层（甚至数百层，但不是全连接的网络，如我们将在第14章中看到的）的网络，并且它们需要大量的训练数据。你几乎不必从头开始训练这样的网络：重用一部分类似任务的经过预训练的最新网络更为普遍。这样，训练就会快得多，所需的数据也要少得多（我们将在第11章中对此进行讨论）。

10.3.2　每个隐藏层的神经元数量

输入层和输出层中神经元的数量取决于任务所需的输入类型和输出类型。例如，MNIST任务需要28×28=784个输入神经元和10个输出神经元。

对于隐藏层，通常将它们调整大小以形成金字塔状，每一层的神经元越来越少，理由是许多低层特征可以合并成更少的高层特征。MNIST的典型神经网络可能具有3个隐藏层，第一层包含300个神经元，第二层包含200个神经元，第三层包含100个神经元。但是，这种做法已被很大程度上放弃了，因为似乎在所有隐藏层中使用相同数量的神经元，在大多数情况下层的表现都一样好，甚至更好；另外，只需要调整一个超参数，而不是每层一个。也就是说，根据数据集，有时使第一个隐藏层大于其他隐藏层是有帮助的。

就像层数一样，你可以尝试逐渐增加神经元的数量，直到网络开始过拟合为止。但是在实践中，选择一个比你实际需要的层和神经元更多的模型，然后使用提前停止和其他正则化技术来防止模型过拟合，通常更简单、更有效。Google的科学家Vincent Vanhoucke称之为“弹力裤”方法：与其浪费时间寻找与自己的尺码完全匹配的裤子，不如使用大尺寸的弹力裤来缩小到合适的尺寸。使用这种方法，一方面，可以避免可能会破坏模型的瓶颈层。另一方面，如果一层的神经元太少，它将没有足够的表征能力来保留来自输入的所有有用信息（例如具有两个神经元的层只能输出2D数据，因此如果它处理3D数据，一些信息将会丢失）。无论网络的其余部分有多强大，这些信息都将永远无法恢复。

 通常通过增加层数而不是每层神经元数，你将获得更多收益。

10.3.3　学习率、批量大小和其他超参数

隐藏层和神经元的数量并不是你可以在MLP中进行调整的唯一超参数。以下是一些最重要的信息，以及有关如何设置它们的提示：

学习率

学习率可以说是最重要的超参数。一般而言，最佳学习率约为最大学习率的一半（即学习率大于算法发散的学习率，如我们在第4章中看到的）。找到一个好的学习率的一种方法是对模型进行数百次迭代训练，从非常低的学习率（例如10-5）开始，然后逐渐将其增加到非常大的值（例如10）。这是通过在每次迭代中将学习率乘以恒定因子来完成的（例如，将exp（log（106）/500）乘以500次迭代中的10-5到10）。如果将损失作为学习率的函数进行绘制（对学习率使用对数坐标），你应该首先看到它在下降。但是过一会儿学习率将过大，因此损失将重新上升：最佳学习率将比损失开始攀升的点低一些（通常比转折点低约10倍）。然后你可以重新初始化模型，并以这种良好的学习率正常训练模型。我们将在第11章中介绍更多的学习率技术。

优化器

选择比普通的小批量梯度下降更好的优化器（并调整其超参数）也很重要。我们将在第11章中了解几个高级优化器。

批量大小

批量的大小可能会对模型的性能和训练时间产生重大影响。使用大批量的主要好处是像GPU这样的硬件加速器可以有效地对其进行处理（见第19章），因此训练算法每秒会看到更多的实例。因此，许多研究人员和从业人员建议使用可容纳在GPU RAM中的最大批量。但是这里有一个陷阱：在实践中，大批量通常会导致训练不稳定，尤其是在训练开始时，结果模型泛化能力可能不如小批量训练的模型。2018年4月，Yann LeCun甚至在推特上写道：“朋友不要让朋友使用大于32的

小批量处理。”并引用了Dominic Masters和Carlo Luschi在2018年发表的一篇论文[3]，得出的结论是首选使用小批量（从2到32），因为小批量可以在更少的训练时间内获得更好的模型。但是，其他论文则提出相反意见。在2017年，Elad Hoffer等人[4]和Priya Goyal等人[5]的论文表明，可以通过各种技术手段使用非常大的批量处理（最多8192），例如提高学习率（即开始学习以较低的学习率进行训练，然后提高学习率，如第11章所述）。这导致了非常短的训练时间，没有泛化能力的差距。因此一种策略是尝试使用大批量处理，慢慢增加学习率，如果训练不稳定或最终表现令人失望，则尝试使用小批量处理。

激活函数

我们在本章前面讨论了如何选择激活函数：通常，ReLU激活函数是所有隐藏层的良好的默认设置。对于输出层，这实际上取决于你的任务。

迭代次数

在大多数情况下，实际上不需要调整训练迭代次数，只需使用提前停止即可。

 最佳学习率取决于其他超参数，尤其是批量大小，因此如果你修改了任何超参数，请确保也更新学习率。

有关调整神经网络超参数的更多最佳实践，请查看Leslie Smith的2018年优秀论文[6]。到此结束我们对人工神经网络及其在Keras中实现的介绍。在接下来的几章中，我们将讨论训练非常深度网络的技术。我们还将探索如何使用TensorFlow的较底层API自定义模型，以及如何使用Data API有效地加载和预处理数据。我们还将深入探讨其他

流行的神经网络架构：用于图像处理的卷积神经网络、用于顺序数据的递归神经网络、用于表征学习的自动编码器以及用于建模和生成数据的生成式对抗网络[7]。

[1] Lisha Li et al.，“Hyperband：A Novel Bandit-Based Approach to Hyperparameter Optimization”，Journal of Machine Learning Research 18（April 2018）：1-52.
[2] Max Jaderberg et al.，“Population Based Training of Neural Networks”，arXiv preprint arXiv：1711.09846（2017）.
[3] Dominic Masters and Carlo Luschi，“Revisiting Small Batch Training for Deep Neural Networks”，arXiv preprint arXiv：1804.07612（2018）.
[4] Elad Hoffer et al.，“Train Longer，Generalize Better：Closing the Generalization Gap in Large Batch Training of Neural Networks”，Proceedings of the 31st International Conference on Neural Information Processing Systems（2017）：1729－1739.
[5] Priya Goyal et al.，“Accurate，Large Minibatch SGD：Training ImageNet in 1 Hour”，arXiv preprint arXiv：1706.02677（2017）.
[6] Leslie N.Smith，“A Disciplined Approach to Neural Network Hyper-Parameters：Part 1—Learning Rate，Batch Size，Momentum，and Weight Decay”，arXiv preprint arXiv：1803.09820（2018）.
[7] 附录E中介绍了一些其他的ANN架构。

10.4 练习题

1. TensorFlow Playground是由TensorFlow团队构建的神经网络模拟器。在本练习中，你只需单击几下即可训练几个二元分类器，并对模型的架构及其超参数进行调整，以直观了解神经网络的工作方式及其超参数的功能。花一些时间来探索以下内容：

a. 由神经网络学习到的模式。单击Run按钮（左上方），尝试训练默认的神经网络。请注意它如何快速找到适合分类任务的最优解。第一隐藏层中的神经元已经学会了简单模式，而第二隐藏层中的神经元已经学会了将第一隐藏层的简单模式组合为更复杂的模式。通常，层数越多，模式越复杂。

b. 激活函数。尝试用ReLU激活函数替换tanh激活函数，然后再次训练网络。注意，它更快找到了最优解，但这次边界是线性的。这是由于ReLU函数的形状引起的。

c. 局部极小值的风险。修改网络架构，使其只有一个具有三个神经元的隐藏层。对其进行多次训练（要重置网络权重，请单击Play按钮旁边的Reset按钮）。请注意，训练时间变化很大，有时甚至会停留在局部最小值中。

d. 当神经网络太小时会发生什么。删除一个神经元，只保留两个。请注意，即使你尝试多次，神经网络现在也无法找到一个好的解决方案。该模型的参数太少，系统欠拟合训练集。

e. 当神经网络足够大时会发生什么。将神经元的数量设置为8个，然后对网络进行几次训练。请注意，它现在始终能快速运行，并且不会卡在某一点。这显示了神经网络理论中的一个重要发现：大型神经

网络几乎永远不会陷入局部极小值，即使陷入了，这些局部最优也几乎与全局最优一样好。但是，它们仍然可能长时间卡在一个平台上。

f.深层网络中梯度消失的风险。选择spiral数据集（Data下的右下数据集），并将网络架构更改为四个隐藏层，每个隐藏层具有8个神经元。请注意，训练花费的时间更长，并且经常长时间停留在一个平台上。还要注意，最高层（右侧）的神经元倾向于比最低层（左侧）的神经元更快地进化。这个问题称为“梯度消失”问题，可以通过更好的权重初始化和其他技术、更好的优化器（例如AdaGrad或Adam）或批量归一化（在第11章中讨论）来缓解。

g.走得更远一点，花一个小时左右的时间来尝试其他参数，并了解它们的作用，以建立对神经网络的直观了解。

2.使用原始人工神经元（如图10-3中的神经元）绘制一个ANN，以计算AB（其中表示XOR操作）。提示：（A∧¬B∨（¬A∧B）

3.为什么通常最好使用逻辑回归分类器而不是经典的感知器（即使用感知器训练算法训练的单层阈值逻辑单元）？如何调整感知器以使其等同于逻辑回归分类器？

4.为什么逻辑激活函数是训练第一个MLP的关键要素？

5.列举三种常用的激活函数。你能画出来吗？

6.假设你有一个MLP，该MLP由一个输入层和10个直通神经元组成，随后是一个包含50个神经元的隐藏层，最后是3个神经元组成的输出层。所有人工神经元都使用ReLU激活函数。

- 输入矩阵X的形状是什么？

• 隐藏层的权重向量Wh及其偏置向量bh的形状是什么？

• 输出层的权重向量Wo及其偏置向量bo的形状是什么？

• 网络的输出矩阵Y的形状是什么？

• 编写等式计算网络的输出矩阵Y（作为X、W_h、b_h、W_o和b_o的函数）。

7.如果要将电子邮件分类为垃圾邮件或正常邮件，你需要在输出层中有多少个神经元？你应该在输出层中使用什么激活函数？相反如果你想解决MNIST，则在输出层中需要多少个神经元，应该使用哪种激活函数？如第2章所述，如何使你的网络预测房价？

8.什么是反向传播，它如何工作？反向传播和反向模式的autodiff有什么区别？

9.你能否列出可以在基本MLP中进行调整的所有超参数？如果MLP过拟合训练数据，你如何调整这些超参数来解决问题？

10.在MNIST数据集上训练一个深度MLP，你可以使用keras.datasets.mnist.load_data（）加载它，看是否可以获得98％以上的精度。请尝试使用本章介绍的方法来寻找最佳学习率（即通过成倍地提高学习率，画出误差，并找出误差发生的点）。尝试一下其他的功能——保存检查点，使用早期停止，并使用TensorBoard绘制学习曲线。

附录A中提供了这些练习题的解答。

第11章　训练深度神经网络

在第10章中，我们介绍了人工神经网络并训练了第一个深度神经网络。但是它们是浅层网络，只有几个隐藏层。如果你需要解决一个复杂的问题，例如检测高分辨率图像中的数百种物体，该怎么办？你可能需要训练更深的DNN，也许10层或更多层，每层包含数百个神经元，有成千上万个连接。训练深度DNN并不是在公园里散步。以下是你可能会遇到的一些问题：

• 你可能会遇到棘手的梯度消失问题或相关的梯度爆炸问题。这是在训练过程中通过DNN反向传播时，梯度变得越来越小或越来越大时发生的。这两个问题都使得较低的层很难训练。

• 对于如此大的网络，你可能没有足够的训练数据，或者做标签的成本太高。

• 训练可能会非常缓慢。

• 具有数百万个参数的模型会有严重过拟合训练集的风险，尤其是在没有足够的训练实例或噪声太大的情况下。

在本章中，我们将研究所有这些问题，并介绍解决这些问题的技术。我们从探索梯度消失和梯度爆炸问题及其一些受欢迎的解决方法开始。接下来，我们将研究迁移学习和无监督预训练，即使在标签数据很少的情况下，它们也可以帮助你解决复杂的任务。然后我们将讨论可以极大加速训练大型模型的各种优化器。最后我们将介绍一些流行的针对大型神经网络的正则化技术。

使用这些工具，你就能够训练非常深的网络。欢迎使用深度学习！

11.1 梯度消失与梯度爆炸问题

正如我们在第10章中讨论的那样，反向传播算法的工作原理是从输出层到输入层，并在此过程中传播误差梯度。一旦算法计算出成本函数相对于网络中每个参数的梯度，就可以使用这些梯度以梯度下降步骤来更新每个参数。

不幸的是，随着算法向下传播到较低层，梯度通常会越来越小。结果梯度下降更新使较低层的连接权重保持不变，训练不能收敛到一个良好的解。我们称其为梯度消失问题。在某些情况下，可能会出现相反的情况：梯度可能会越来越大，各层需要更新很大的权重直到算法发散为止。这是梯度爆炸问题，它出现在递归神经网络中（见第15章）。更笼统地说，深度神经网络很受梯度不稳定的影响，不同的层可能以不同的速度学习。

这些问题很久以前就凭经验观察到了，这也是深度神经网络在2000年代初期被大量抛弃的原因之一。目前尚不清楚是什么原因导致在训练DNN时使梯度如此不稳定，但是Xavier Glorot和Yoshua Bengio在2010年的一篇论文中阐明了一些观点[1]。作者发现了一些疑点，包括流行的逻辑sigmoid激活函数和当时最流行的权重初始化技术（即平均值为0且标准差为1的正态分布）。简而言之，它们表明使用此激活函数和此初始化方案，每层输出的方差远大于其输入的方差。随着网络的延伸，方差在每一层之后都会增加，直到激活函数在顶层达到饱和为止。实际上，由于逻辑函数的平均值为0.5，而不是0（双曲线正切函数的平均值为0，在深度网络中的表现比逻辑函数稍微好一些），因此饱和度实际上变得更差。

查看逻辑激活函数（见图11-1），你可以看到，当输入变大（负数或正数）时，该函数会以0或1饱和，并且导数非常接近0。因此反向传播开始时它几乎没有梯度可以通过网络传播回去。当反向传播通过顶层

向下传播时，存在的小梯度不断被稀释，因此对于底层来说，实际上什么也没有留下。

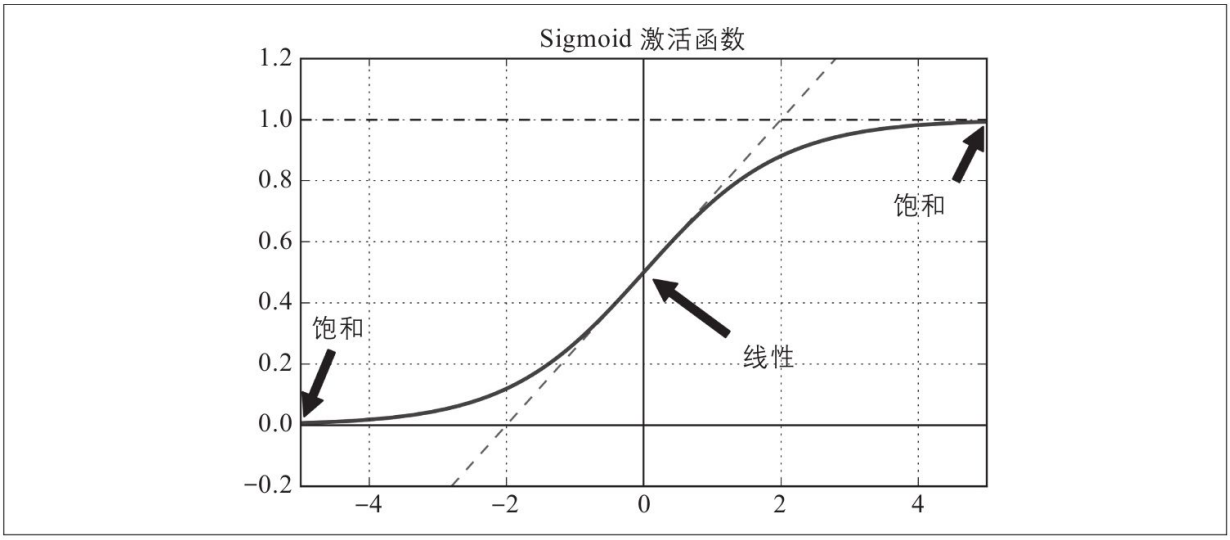


图11-1：逻辑激活函数的饱和

11.1.1 Glorot和He初始化

Glorot和Bengio在它们的论文中提出了一种能显著缓解不稳定梯度问题的方法。他们指出，我们需要信号在两个方向上正确流动：进行预测时，信号为正向；在反向传播梯度时，信号为反向。我们既不希望信号消失，也不希望它爆炸并饱和。为了使信号正确流动，作者认为，我们需要每层输出的方差等于其输入的方差[2]，并且我们需要在反方向时流过某层之前和之后的梯度具有相同的方差（如果你对数学细节感兴趣，请查看本论文）。除非该层具有相等数量的输入和神经元（这些数字称为该层的扇入和扇出），否则实际上不可能同时保证两者，但是Glorot和Bengio提出了一个很好的折中方案，在实践中证明很好地发挥作用：必须按照公式11-1中所述的随机初始化每层的连接权重，其中$fan_{avg}=(fan_{in}+fan_{out})/2$。这种初始化策略称为Xavier初始化或者Glorot初始化，以论文的第一作者命名。

公式11-1：Glorot初始化（使用逻辑激活函数时）

正态分布，其均值为0，方差为 $\sigma^2 = \frac{1}{fan_{avg}}$

或-r和+r之间的均匀分布，其中 $r = \sqrt{\frac{3}{fan_{avg}}}$

如果在公式11-1中你用fan_{in}替换fan_{avg}，则会得到Yann LeCun在20世纪90年代提出的初始化策略。他称其为LeCun初始化。Genevieve Orr和Klaus-Robert Müller甚至在其1998年出版的Neural Networks: Tricks of the Trade（Springer）一书中进行了推荐。当$fan_{in}=fan_{out}$时，LeCun初始化等效于Glorot初始化。研究人员花了十多年的时间才意识到这一技巧的重要性。使用Glorot初始化可以大大加快训练速度，这是导致深度学习成功的诀窍之一。

一些论文[3]为不同的激活函数提供了类似的策略。这些策略的差异仅在于方差的大小以及它们使用的是fan_{avg}还是fan_{in}，如表11-1所示（对于均匀分布，只需计算 $r = \sqrt{3\sigma^2}$）。ReLU激活函数的初始化策略（及其变体，包括ELU激活函数）有时简称为He初始化。本章稍后将解释SELU激活函数。它应该与LeCun初始化一起使用（最好与正态分布一起使用，如我们所见）。

表11-1：每种激活函数的初始化参数

初始化	激活函数	σ^2（正常）
Glorot	None、tanh、logistic、softmax	$1 / fan_{avg}$
He	ReLU 和变体	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

默认情况下，Keras使用具有均匀分布的Glorot初始化。创建层时，可以通过设置kernel_initializer="he_uniform"或kernel_initializer="he_normal"来将其更改为He初始化：

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

如果你要使用均匀分布但基于fanavg而不是fanin进行He初始化，则可以使用Variance Scaling初始化，如下所示：

```
he_avg_init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg',
                                                 distribution='uniform')
keras.layers.Dense(10, activation="sigmoid", kernel_initializer=he_avg_init)
```

11.1.2 非饱和激活函数

Glorot和Bengio在2010年的论文中提出的一项见解是，梯度不稳定的问题部分是由于激活函数选择不当所致。在此之前，大多数人都认为，如果大自然母亲选择在生物神经元中使用类似sigmoid的激活函数，那么它们必定是一个好选择。但是事实证明，其他激活函数在深度神经网络中的表现要好得多，尤其是ReLU激活函数，这主要是因为它对正值不饱和（并且计算速度很快）。

不幸的是，ReLU激活函数并不完美。它有一个被称为“濒死的ReLU”的问题：在训练过程中，某些神经元实际上“死亡”了，这意味着它们停止输出除0以外的任何值。在某些情况下，你可能会发现网络中一半的神经元都死了，特别是如果你使用较大的学习率。当神经元的权重进行调整时，其输入的加权和对于训练集中的所有实例均为负数，神经元会死亡。发生这种情况时，它只会继续输出零，梯度下降不会再影响它，因为ReLU函数的输入为负时其梯度为零[4]。

要解决此问题，你可能需要使用ReLU函数的变体，例如leaky ReLU。该函数定义为$\text{LeakyReLU}_{\alpha}(z)=\max(\alpha z, z)$（见图11-2）。超参数α定义函数“泄漏”的程度：它是$z<0$时函数的斜率，通常设置为0.01。这个小的斜率确保了leaky ReLU永远不会死亡。

它们可能会陷入长时间的昏迷，但是有机会最后醒来。2015年的一篇论文比较了ReLU激活函数的几种变体[5]，其结论之一是泄漏的变体要好于严格的ReLU激活函数。实际上，设置α=0.2（大泄漏）似乎比α=0.01（小泄漏）会产生更好的性能。本论文还对随机的leaky ReLU（RReLU）进行了评估，在训练过程中在给定范围内随机选择α，在测试过程中将其固定为平均值。RReLU的表现也相当不错，似乎可以充当正则化函数（减少了过拟合训练集的风险）。最后，本文评估了参数化leaky ReLU（PReLU），其中α可以在训练期间学习（不是超参数，它像其他任何参数一样，可以通过反向传播进行修改）。据报道，PReLU在大型图像数据集上的性能明显优于ReLU，但是在较小的数据集上，它存在过拟合训练集的风险。

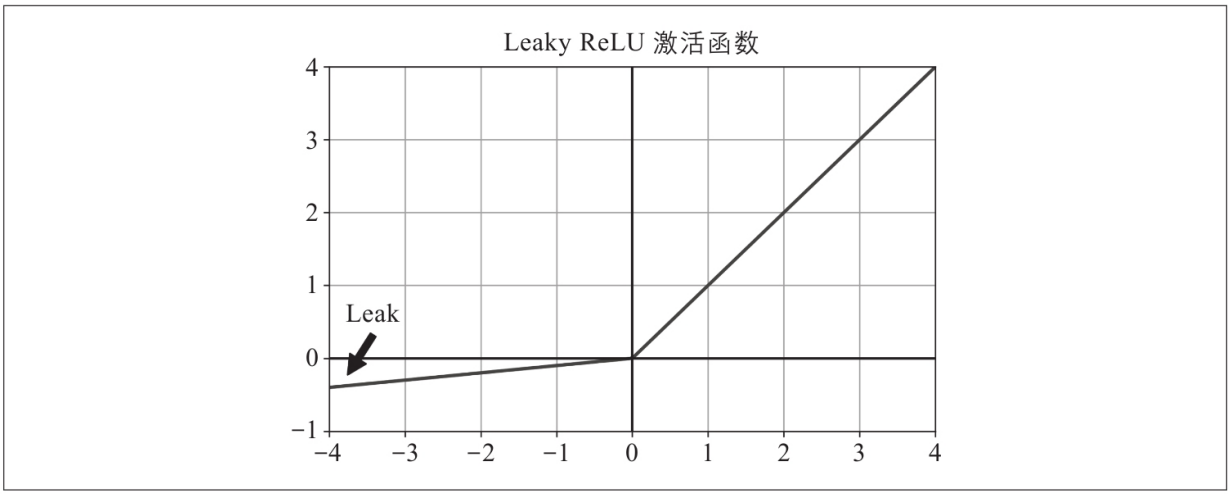


图11-2：Leaky ReLU：与ReLU类似，对负值有一个小斜率

最后但并非最不重要的一点是，Djork-ArnéClevert等人在2015年发表的论文提出了一种新的激活函数[6]，称为指数线性单位（Exponential Linear Unit，ELU），该函数在作者的实验中胜过所有ReLU变体：减少训练时间，神经网络在测试集上表现更好。图11-3绘制了函数图，公式11-2给出了其定义。

公式11-2：ELU激活函数

$$\mathrm{ELU}_{\alpha}(z)=\begin{cases}\alpha\left(\exp(z)-1\right) & \text{如果 } z<0\\ z & \text{如果 } z\geqslant 0\end{cases}$$

ELU激活函数与ReLU函数非常相似，但有一些主要区别：

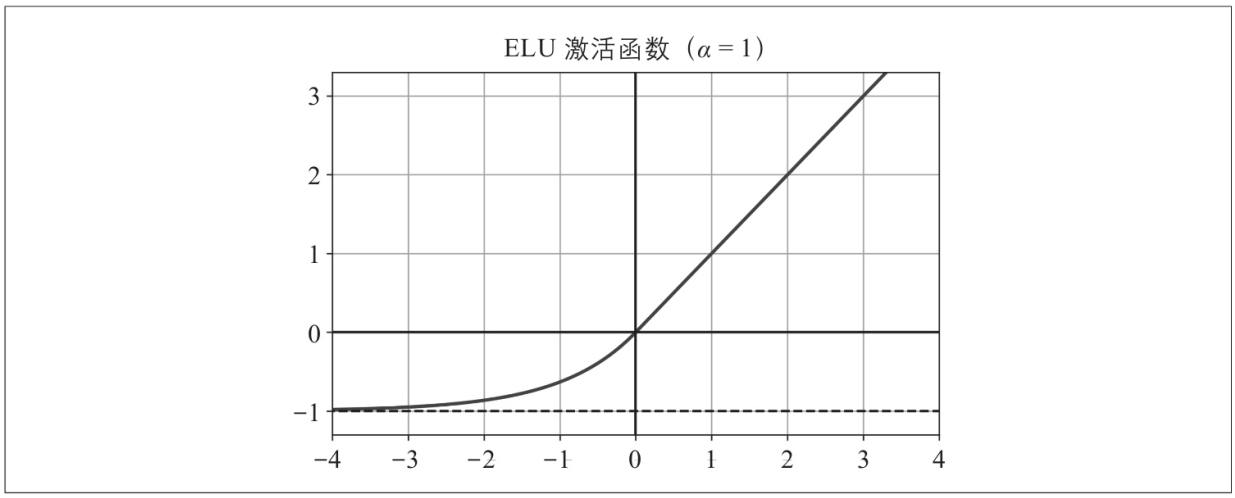


图11-3：ELU激活函数

• 当z<0时，它取负值，这使该单元的平均输出接近于0，有助于缓解梯度消失的问题。超参数α定义一个值，该值为当z为较大负数时ELU函数逼近的值。通常将其设置为1，但是你可以像其他任何超参数一样对其进行调整。

• 对于z<0，它具有非零梯度，从而避免了神经元死亡的问题。

• 如果α等于1，则该函数在所有位置（包括z=0左右）都是平滑的，这有助于加速梯度下降，因为它在z=0的左右两侧弹跳不大。

ELU激活函数的主要缺点是它的计算比ReLU函数及其变体要慢（由于使用了指数函数）。它在训练过程中更快的收敛速度弥补了这种缓慢的计算，但是在测试时，ELU网络将比ReLU网络慢。

然后，Günter Klambauer等人在2017年发表的论文[7]提出了可扩展的ELU（Scaled ELU，SEIU）激活函数：顾名思义，它是ELU激活函数的可扩展变体。作者表明，如果你构建一个仅由密集层堆叠组成的神经网络，并且如果所有隐藏层都使用SELU激活函数，则该网络是自归一化的：每层的输出倾向于在训练过程中保留平均值0和标准差1，从而解决了梯度消失/梯度爆炸的问题。结果，SELU激活函数通常大大优于这些神经网络（尤其是深层神经网络）的其他激活函数。但是，有一些产生自归一化的条件（有关数学证明，请参见论文）：

• 输入特征必须是标准化的（平均值为0，标准差为1）。

• 每个隐藏层的权重必须使用LeCun正态初始化。在Keras中，这意味着设置kernel_initializer="lecun_normal"。

• 网络的架构必须是顺序的。不幸的是，如果你尝试在非顺序架构（例如循环网络）中使用SELU（见第15章）或具有跳过连接的网络（即在Wide&Deep网络中跳过层的连接），将无法保证自归一化，因此SELU不一定会胜过其他激活函数。

• 本论文仅在所有层都是密集层的情况下保证自归一化，但一些研究人员指出SELU激活函数也可以改善卷积神经网络的性能（见第14章）。

 那么，你应该对深度神经网络的隐藏层使用哪个激活函数呢？尽管你的目标会有所不同，但通常SELU>ELU>leaky ReLU（及其变体）>ReLU>tanh>logistic。如果网络的架构不能自归一化，那么ELU的性能可能会优于SELU（因为SELU在z=0时不平滑）。如果你非常关心运行时延迟，那么你可能更喜欢leaky ReLU。如果你不想调整其他超参数，则可以使用Keras使用的默认α值（例如，leaky ReLU为0.3）。如果你有空闲时间和计算能力，则可以使用交叉验证来评估其他激活函数，例如，如果网络过拟合，则为RReLU；如果你的训练集很大，则为PReLU。也就是说，由于ReLU是迄今为止最常用的激活函数，因此许多库和硬件加速器都提供了ReLU特定的优化。因此，如果你将速度放在首位，那么ReLU可能仍然是最佳选择。

要使用leaky ReLU激活函数，创建一个LeakyReLU层，并将其添加到你想要应用它的层之后的模型中：

```
model = keras.models.Sequential([
    [...]
    keras.layers.Dense(10, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(alpha=0.2),
    [...]
])
```

对于PReLU，将LeakyRelu（alpha=0.2）替换为PReLU（）。Keras当前没有RReLU的官方实现，但是你可以轻松地实现自己的（要了解如何实现，请参阅第12章末尾的练习）。

对于SELU激活，在创建层时设置activation="selu"和kernel_initializer="lecun_normal"：

```
layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")
```

11.1.3 批量归一化

尽管将He初始化与ELU（或ReLU的任何变体）一起使用可以显著减少在训练开始时的梯度消失/梯度爆炸问题的危险，但这并不能保证它们在训练期间不会再出现。

在2015年的一篇论文中[8]，Sergey Ioffe和Christian Szegedy提出了一种称为批量归一化（BN）的技术来解决这些问题。该技术包括在模型中的每个隐藏层的激活函数之前或之后添加一个操作。该操作对每个输入零中心并归一化，然后每层使用两个新的参数向量缩放和偏移其结果：一个用于缩放，另一个用于偏移。换句话说，该操作可以使模型学习各层输入的最佳缩放和均值。在许多情况下，如果你将BN层添加为神经网络的第一层，则无须归一化训练集（例如，使用StandardScaler）；BN层会为你完成此操作（因为它一次只能查看一个批次，它还可以重新缩放和偏移每个输入特征）。

为了使输入零中心并归一化，该算法需要估计每个输入的均值和标准差。通过评估当前小批次上的输入的均值和标准差（因此称为“批量归一化”）来做到这一点。公式11-3逐步总结了整个操作。

公式11-3：批量归一化算法

1. $$\boldsymbol{\mu}_{\mathrm{B}}=\frac{1}{m_{\mathrm{B}}}\sum_{i=1}^{m_{\mathrm{B}}}\boldsymbol{x}^{(i)}$$

2. $$\boldsymbol{\sigma}_{\mathrm{B}}^{2}=\frac{1}{m_{\mathrm{B}}}\sum_{i=1}^{m_{\mathrm{B}}}\left(\boldsymbol{x}^{(i)}-\boldsymbol{\mu}_{\mathrm{B}}\right)^{2}$$

3. $$\hat{\boldsymbol{x}}^{(i)}=\frac{\boldsymbol{x}^{(i)}-\boldsymbol{\mu}_{\mathrm{B}}}{\sqrt{\boldsymbol{\sigma}_{\mathrm{B}}^{2}+\boldsymbol{\varepsilon}}}$$

4. $$\boldsymbol{z}^{(i)}=\boldsymbol{\gamma}\otimes\hat{\boldsymbol{x}}^{(i)}+\boldsymbol{\beta}$$

在此算法中：

• μ_B是输入均值的向量，在整个小批量B上评估（每个输入包含一个均值）。

• σ_B是输入标准差的向量，也在整个小批量中进行评估（每个输入包含一个标准差）。

• m_B是小批量中的实例数量。

• $x_{(i)}$是实例i的零中心和归一化输入的向量。

• γ是该层的输出缩放参数向量（每个输入包含一个缩放参数）。

• ⊗表示逐元素乘法（每个输入乘以其相应的输出缩放参数）。

• β是层的输出移动（偏移）参数向量（每个输入包含一个偏移参数）。每个输入都通过其相应的移动参数进行偏移。

• ε是一个很小的数字以避免被零除（通常为10^{-5}）。这称为平滑项。

• $z^{(i)}$是BN操作的输出。它是输入的缩放和偏移版本。

因此在训练期间，BN会归一化其输入，然后重新缩放并偏移它们。好！那在测试期间呢？这不那么简单。确实，我们可能需要对单个实例而不是成批次的实例做出预测：在这种情况下，我们无法计算每个输入的均值和标准差。而且，即使我们确实有一批次实例，它也可能太小，或者这些实例可能不是独立的和相同分布的，因此在这批实例上计算统计信息将是不可靠的。一种解决方法是等到训练结束，然后通过神经网络运行整个训练集，计算BN层每个输入的均值和标准差。然后，在进行预测时，可以使用这些“最终”的输入均值和标准差，而不是一个批次的输入均值和标准差。然而，大多数批量归一化的实现都是通过使用该层输入的均值和标准差的移动平均值来估计训练期间的最终统计信息。这是Keras在使用BatchNormalization层时自动执行的操作。综上所述，在每个批归一化层中学习了四个参数向量：通过常规反向传播学习γ（输出缩放向量）和β（输出偏移向量），和使用指数移动平均值估计的μ（最终的输入均值向量）和σ（最终输入标准差向量）。请注意，μ和σ是在训练期间估算的，但仅在训练后使用（以替换方程式11-3中的批量输入均值和标准差）。

Ioffe和Szegedy证明，批量归一化极大地改善了他们试验过的所有深度神经网络，从而极大地提高了ImageNet分类任务的性能（ImageNet是将图像分类为许多类的大型图像数据库，通常用于评估计算机视觉系统）。消失梯度的问题已大大减少，以至于它们可以使用饱和的激活函数，例如tanh甚至逻辑激活函数。网络对权重初始化也不太敏感。作者可以使用更大的学习率，大大加快了学习过程。它们特别指出：

批量归一化应用于最先进的图像分类模型，以少14倍的训练步骤即可达到相同的精度，在很大程度上击败了原始模型……使用批量归一化网络的集成，我们在ImageNet分类中改进了已发布的最好结果：前5位

的验证错误达到了4.9%（和4.8%的测试错误），超过了人工评分者的准确性。

最后，就像不断赠送的礼物一样，批量归一化的作用就像正则化一样，减少了对其他正则化技术（如dropout，本章稍后将介绍）的需求。

但是，批量归一化确实增加了模型的复杂性（尽管它可以消除对输入数据进行归一化的需求，正如我们前面所讨论的）。此外，还有运行时间的损失：由于每一层都需要额外的计算，因此神经网络的预测速度较慢。幸运的是，经常可以在训练后将BN层与上一层融合，从而避免了运行时的损失。这是通过更新前一层的权重和偏置来完成的，以便它直接产生适当缩放和偏移的输出。例如，如果前一层计算X W+b，则BN层将计算 $\gamma \otimes (XW+b - \mu) / \sigma + \beta$ （忽略分母中的平滑项 ε ）。如果我们定义 $W' = \gamma \otimes W / \sigma$ 和 $b' = \gamma \otimes (b - \mu) / \sigma + \beta$ ，则方程式可简化为 $XW' + b'$ 。因此，如果我们用更新后的权重和偏置（W′ 和b′ ）替换前一层的权重和偏置（W和b），就可以去掉BN层（TFLite的优化器会自动执行此操作，请参阅第19章）。

 你可能会发现训练相当慢，因为当你使用批量归一化时每个轮次要花费更多时间。通常情况下，这被BN的收敛速度要快得多的事实而抵消，因此达到相同性能所需的轮次更少，总而言之，墙上的时间通常会更短（这是墙上的时钟所测量的时间）。

用Keras实现批量归一化

与使用Keras进行的大多数操作一样，实现批量归一化既简单又直观。只需在每个隐藏层的激活函数之前或之后添加一个BatchNormalization层，然后可选地在模型的第一层后添加一个BN层。例如，此模型在每个隐藏层之后作为模型的第一层（展平输入图像之后）应用BN：

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

就这样！在这个只有两个隐藏层的小示例中，批量归一化不可能产生非常积极的影响。但是对于更深层的网络，它可以带来巨大的改变。

显示模型摘要：

```
>>> model.summary()
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization_v2 (Batc (None, 784)               3136
_________________________________________________________________
dense_50 (Dense)             (None, 300)               235500
_________________________________________________________________
batch_normalization_v2_1 (Ba (None, 300)               1200
_________________________________________________________________
dense_51 (Dense)             (None, 100)               30100
_________________________________________________________________
batch_normalization_v2_2 (Ba (None, 100)               400
_________________________________________________________________
dense_52 (Dense)             (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

如你所见，每个BN层的每个输入添加了四个参数：γ、β、μ和σ（例如，第一个BN层添加了3136个参数，即4×784）。最后两个参数μ和σ是移动平均值。它们不受反向传播的影响，因此Keras称其为“不可训练”[9]（如果你计算BN参数的总数3136+1200+400，然后除以2，则得到2368，即此模型中不可训练参数的总数）。

让我们看一下第一个BN层的参数。两个是可训练的（通过反向传播），两个不是：

```
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
[('batch_normalization_v2/gamma:0', True),
 ('batch_normalization_v2/beta:0', True),
 ('batch_normalization_v2/moving_mean:0', False),
 ('batch_normalization_v2/moving_variance:0', False)]
```

现在，当你在Keras中创建BN层时，它还会创建两个操作，在训练期间的每次迭代中，Keras都会调用这两个操作。这些操作会更新移动平均值。由于我们使用的是TensorFlow后端，因此这些操作是TensorFlow操作（我们将在第12章中讨论TF操作）：

```
>>> model.layers[1].updates
[<tf.Operation 'cond_2/Identity' type=Identity>,
 <tf.Operation 'cond_3/Identity' type=Identity>]
```

BN论文的作者主张在激活函数之前（而不是之后）添加BN层（就像我们刚才所做的那样）。关于此问题，存在一些争论，哪个更好取决于你的任务——你也可以对此进行试验，看看哪个选择最适合你的数据集。要在激活函数之前添加BN层，必须从隐藏层中删除激活函数，并将其作为单独的层添加到BN层之后。此外，由于批量归一化层的每个输入都包含一个偏移参数，因此你可以从上一层中删除偏置项（创建时只需传递use_bias=False即可）：

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(100, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
```

```
    keras.layers.Dense(10, activation="softmax")
])
```

BatchNormalization类具有许多可以调整的超参数。默认值通常可以，但是你偶尔可能需要调整omentum。BatchNormalization层在更新指数移动平均值时使用此超参数。给定一个新值v（即在当前批次中计算的输入均值或标准差的新向量），该层使用以下公式来更新运行时平均$\hat{v}$：

$$\hat{v} \leftarrow \hat{v} \times momentum + v \times (1 - momentum)$$

一个良好的动量值通常接近1；例如0.9、0.99或0.999（对于较大的数据集和较小的批处理，你需要更多的9）。

另一个重要的超参数是axis：它确定哪个轴应该被归一化。默认为-1，这意味着默认情况下它将对最后一个轴进行归一化（使用跨其他轴计算得到的均值和标准差）。当输入批次为2D（即批次形状为[批次大小，特征]）时，这意味着将基于在批次中所有实例上计算得到的均值和标准差对每个输入特征进行归一化。例如，先前代码示例中的第一个BN层将独立地归一化（重新缩放和偏移）784个输入特征中的每一个。如果将第一个BN层移动到Flatten层之前，则输入批次将为3D，形状为[批次大小，高度，宽度]。因此，BN层将计算28个均值和28个标准差（每列像素1个，在批次中的所有实例以及在列中所有行之间计算），它将使用相同的均值和标准差对给定列中的所有像素进行归一化。也是只有28个缩放参数和28个偏移参数。相反，你如果仍然要独立的处理784个像素中的每一个，则应设置axis=[1，2]。

请注意，BN层在训练期间和训练后不会执行相同的计算：它在训练期间使用批处理统计信息，在训练后使用“最终”的统计信息（即移动平均的最终值）。让我们看一下这个类的源代码，看看如何处理：

```
class BatchNormalization(keras.layers.Layer):
    [...]
    def call(self, inputs, training=None):
        [...]
```

如你所见，call（）方法是执行计算的方法，它有一个额外的训练参数，默认情况下将其设置为None，但是在训练过程中fit（）方法将其设置为1。如果你需要编写自定义层，它的行为在训练期间和测试期间必须有所不同，把一个训练参数添加到call（）方法中，并在该方法中使用这个参数来决定要计算的内容[10]（我们将在第12章中讨论自定义层）。

BatchNormalization已成为深度神经网络中最常用的层之一，以至于在图表中通常将其省略，因为假定在每层之后都添加了BN。但是Hongyi Zhang等人最近的论文[11]可能会改变这一假设：通过使用一种新颖的fixed-update（fixup）权重初始化技术，作者设法训练了一个非常深的神经网络（10 000层！），没有使用BN，在复杂的图像分类任务上实现了最先进的性能。但是，由于这是一项前沿研究，因此你在放弃批量归一化之前，可能需要等待其他研究来确认此发现。

11.1.4 梯度裁剪

缓解梯度爆炸问题的另一种流行技术是在反向传播期间裁剪梯度，使它们永远不会超过某个阈值。这称为梯度裁剪[12]。这种技术最常用于循环神经网络，因为在RNN中难以使用批量归一化，正如我们将在第15章中看到的那样。对于其他类型的网络，BN通常就足够了。

在Keras中，实现梯度裁剪仅仅是一个在创建优化器时设置clipvalue或clipnorm参数的问题，例如：

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

该优化器会将梯度向量的每个分量都裁剪为-1.0和1.0之间的值。这意味着所有损失的偏导数（相对于每个可训练的参数）将限制在-1.0和1.0之间。阈值是你可以调整的超参数。注意，它可能会改变梯度向量的方向。例如，如果原始梯度向量为[0.9，100.0]，则其大部分指向第二个轴的方向。但是按值裁剪后，将得到[0.9，1.0]，该点大致指向两个轴之间的对角线。实际上，这种方法行之有效。如果要确保“梯度裁剪”不更改梯度向量的方向，你应该通过设置clipnorm而不是clipvalue按照范数来裁剪。如果2范数大于你选择的阈值，则会裁剪整个梯度。例如，如果你设置clipnorm=1.0，则向量[0.9，100.0]将被裁剪为[0.00899964，0.9999595]，保留其方向，但几乎消除了第一个分量。如果你观察到了在训练过程中梯度爆炸（可以使用TensorBoard跟踪梯度的大小），可能要尝试使用两种方法（按值裁剪和按范数裁剪），看看哪个选择在验证集上表现更好。

[1] Xavier Glorot and Yoshua Bengio，“Understanding the Difficulty of Training Deep Feedforward Neural Networks，”Proceedings of the 13th International Conference on Artificial Intelligence and Statistics（2010）：249-256.

[2] 这是一个类比：如果将麦克风放大器的旋钮设置得太接近零，则人们不会听到你的声音，但是如果设置得太接近最大值，你的声音会变得饱和，别人无法理解你在说什么。现在想象一下这样一个放大器链条：它们都需要正确设置，以使你的声音在链条的末尾响亮而清晰。你的声音必须以相同的振幅从每个放大器发出。

[3] E.g.，Kaiming He et al.，“Delving Deep into Rectifiers：Surpassing Human-Level Performance on ImageNet Classification，”Proceedings of the 2015 IEEE International Conference on Computer Vision（2015）：1026－1034。

[4] 除非它是第一个隐藏层的一部分，否则死亡的神经元有时可能会复活：梯度下降可能确实会调整下面各层中的神经元，使得死亡神经元输入的加权和再次为正。

[5] Bing Xu et al.，“Empirical Evaluation of Rectified Activations in Convolutional Network，”arXiv preprint arXiv：1505.00853（2015）.
[6] Djork-ArnéClevert et al.，“Fast and Accurate Deep Network Learning by Exponential Linear Units（ELUs），”Proceedings of the International Conference on Learning Representations（2016）.
[7] Günter Klambauer et al.，“Self-Normalizing Neural Networks，”Proceedings of the 31st International Conference on Neural Information Processing Systems（2017）：972－981.
[8] Sergey Ioffe and Christian Szegedy，“Batch Normalization：Accelerating Deep Network Training by Reducing Internal Covariate Shift，”Proceedings of the 32nd International Conference on Machine Learning（2015）：448-456.
[9] 但是，它们是在训练期间根据训练数据估算的，因此可以争论说它们是可以训练的。在Keras中，“不可训练”的意思是“不受反向传播的影响”。
[10] Keras API还指定了keras.backend.learning_phase（）函数，该函数应在训练期间返回1，其他为0。
[11] Hongyi Zhang et al.，“Fixup Initialization：Residual Learning Without Normalization，”arXiv preprint arXiv：1901.09321（2019）.
[12] Razvan Pascanu et al.，“On the Difficulty of Training Recurrent Neural Networks，”Proceedings of the 30th International Conference on Machine Learning（2013）：1310－1318.

11.2 重用预训练层

从头开始训练非常大的DNN通常不是一个好主意：相反，你应该总是试图找到一个现有的与你要解决的任务相似的神经网络（我们将在第14章讨论如何找到它们），然后重用该网络的较低层。此技术称为迁移学习。它不仅会大大加快训练速度，而且会大大减少训练数据。

假设你可以访问一个经过训练的DNN，将图片分成100个不同的类别，其中包括动物、植物、车辆和日常物品。你现在想训练DNN来对特定类型的车辆进行分类。这些任务非常相似，甚至部分有重叠，因此你应该尝试重用第一个网络的一部分（见图11-4）。

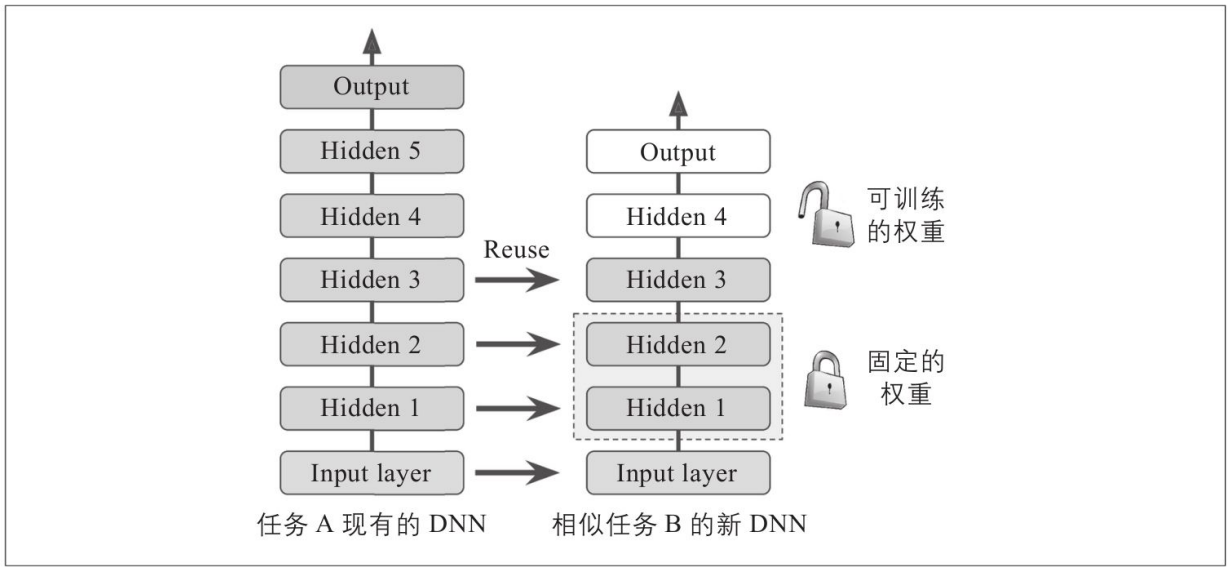


图11-4：重用预训练层

 如果新任务的输入图片的大小与原始任务中使用的图片不同，通常必须添加预处理步骤将其调整为原始模型所需的大小。一般而言，当输入具有类似的低级特征时，迁移学习最有效。

通常应该替换掉原始模型的输出层，因为它对于新任务很有可能根本没有用，甚至对于新任务而言，可能没有正确数量的输出。

类似地，原始模型的上部分隐藏层不太可能像下部分那样有用，因为对新任务最有用的高级特征可能与对原始任务最有用的特征有很大的不同。你需要找到正确的层数来重用。

 任务越相似，可重用的层越多（从较低的层开始）。对于非常相似的任务，请尝试保留所有的隐藏层和只是替换掉输出层。

首先尝试冻结所有可重复使用的层（使其权重不可训练，这样梯度下降就不会对其进行修改），训练模型并查看其表现。然后尝试解冻上部隐藏层中的一两层，使反向传播可以对其进行调整，再查看性能是否有所提高。你拥有的训练数据越多，可以解冻的层就越多。当解冻重用层时，降低学习率也很有用，可以避免破坏其已经调整好的权重。

如果你仍然无法获得良好的性能，并且你的训练数据很少，试着去掉顶部的隐藏层，然后再次冻结所有其余的隐藏层。你可以进行迭代，直到找到合适的可以重复使用的层数。如果你有大量的训练数据，则可以尝试替换顶部的隐藏层而不是去掉它们，你甚至可以添加更多的隐藏层。

11.2.1　用Keras进行迁移学习

让我们看一个示例。假设Fashion MNIST数据集仅包含8个类别，例如，除凉鞋和衬衫之外的所有类别。有人在该数据集上建立并训练了Keras模型，并获得了相当不错的性能（精度>90%）。我们将此模型称为A。你现在要处理另一项任务：你有凉鞋和衬衫的图像，想要训练一个二元分类器（正=衬衫，负=凉鞋）。你的数据集非常小，只有200张带标签的图像。当你使用与模型A相同的架构训练一个新模型（称为模型B）时，它的性能相当好（97.2%的精度）。但是由于这是一项容易

得多的任务（只有两个类），所以你希望有更多。喝早茶时，你意识到你的任务与任务A非常相似，也许通过迁移学习可以有所帮助？让我们找出答案！

首先，你需要加载模型A并基于该模型的层创建一个新模型。让我们重用除输出层之外的所有层：

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

请注意，model_A和model_B_on_A现在共享一些层。当训练model_B_on_A时，也会影响model_A。如果想避免这种情况，需要在重用model_A的层之前对其进行克隆。为此，请使用clone_model（）来克隆模型A的架构，然后复制其权重（因为clone_model（）不会克隆权重）：

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

现在你可以为任务B训练model_B_on_A，但是由于新的输出层是随机初始化的，它会产生较大的错误（至少在前几个轮次内），因此将存在较大的错误梯度，这可能会破坏重用的权重。为了避免这种情况，一种方法是在前几个轮次时冻结重用的层，给新层一些时间来学习合理的权重。为此，请将每一层的可训练属性设置为False并编译模型：

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                     metrics=["accuracy"])
```

 冻结或解冻层后，你必须总是要编译模型。

现在，你可以训练模型几个轮次，然后解冻重用的层（这需要再次编译模型），并继续进行训练来微调任务B的重用层。解冻重用层之后，降低学习率通常是个好主意，可以再次避免损坏重用的权重：

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=1e-4) # the default lr is 1e-2
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                     metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))
```

那么，最终结果是什么？好了，该模型的测试精度为99.25%，这意味着迁移学习将错误率从2.8%降低到了几乎0.7%！那是四倍的差距！

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.06887910133600235, 0.9925]
```

你被说服了吗？你不应该：我作弊了！我尝试了许多配置，直到发现一个有明显改进的配置。如果你试着更改类别或随机种子，会发现改进通常会下降，甚至消失或反转。我所做的就是“折磨数据直到信服为止”。当论文看起来过于优秀时，你应该要怀疑：也许这个浮华的新技术实际上并没有多大帮助（事实上，它甚至可能降低性能），但作者尝试了许多变体，仅报告了最好的结果（这可能是由于运气所致），而没

有提及他们在途中遇到了多少次失败。在大多数情况下，这根本不是恶意的，但这是造成如此多的科学结果永远无法复现的部分原因。

我为什么要作弊？事实证明，迁移学习在小型密集型网络中不能很好地工作，大概是因为小型网络学习的模式很少，密集网络学习的是非常特定的模式，这在其他任务中不是很有用。迁移学习最适合使用深度卷积神经网络，该神经网络倾向于学习更为通用的特征检测器（尤其是在较低层）。我们将在第14章中使用刚刚讨论的技术重新审视迁移学习（我保证，这一次不会作弊！）。

11.2.2 无监督预训练

假设你要处理一个没有太多标签训练数据的复杂任务，但是不幸的是，你找不到在类似任务上训练的模型。不要失去希望！首先，你应该尝试收集更多带有标签的训练数据，但如果做不到，你仍然可以执行无监督预训练（见图11-5）。确实，收集未标记的训练实例通常很便宜，但标记它们却很昂贵。如果你可以收集大量未标记的训练数据，则可以尝试使用它们来训练无监督模型，例如自动编码器或GAN（见第17章）。然后，你可以重用自动编码器的较低层或GAN判别器的较低层，在顶部为你的任务添加输出层，并使用有监督学习（即带有标记的训练实例）来微调最终的网络。

Geoffrey Hinton和他的团队在2006年使用的正是这种技术，它导致了神经网络的复兴以及深度学习的成功。直到2010年，无监督预训练——通常使用受限的Boltzmann机器（RBM，见附录E）——是深度网络的标准，只有在梯度消失问题得到解决后，使用有监督学习来训练DNN才变得更加普遍。当你要解决的任务很复杂，没有可重复使用的相似模型，标签的训练数据很少，但是无标签的训练数据很多时，无监督预训练（今天通常使用自动编码器或GAN而不是RBM）仍然是一个不错的选择。

请注意，在深度学习的早期很难训练深度模型，因此人们会使用一种称为贪婪逐层预训练（greedy layer-wise pretraining）的技术（如图11-5所示）。它们首先训练一个单层的无监督模型，通常是RBM，然后冻结该层并在其之上添加另一个层，然后再次训练模型（实际上只是训练新层），然后冻结新层并在其上添加另一层，再次训练模型，以此类推。如今，事情变得简单多了：人们通常只用一次就可以训练完整的无监督模型（即在图11-5中，直接从第三步开始），并使用自动编码器或GAN（不是RBM）。

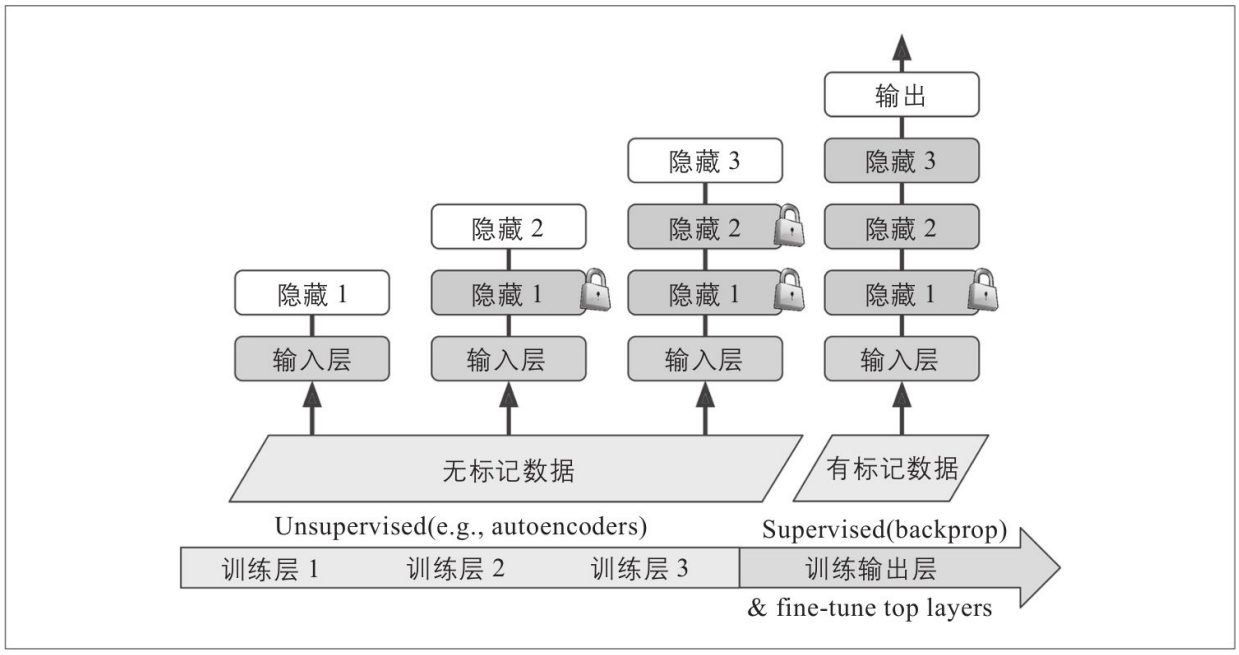


图11-5：在无监督训练中，使用一个无监督学习技术对无标记数据（或所有数据）进行训练，然后使用一个有监督学习技术对有标记数据进行最后任务的微调，无监督部分可以一次训练一层，如图所示，也可以直接训练整个模型。

11.2.3 辅助任务的预训练

如果你没有太多标记的训练数据，最后一个选择是在辅助任务上训练第一个神经网络，你可以轻松地为其获得或生成标记的训练数据，然后对实际任务重用该网络的较低层。第一个神经网络的较低层将学习特征检测器，第二个神经网络可能会重用这些特征检测器。

例如，如果你要构建一个识别人脸的系统，每个人可能只有几张照片，显然不足以训练一个好的分类器。收集每个人的数百张图片不切实际。但是，你可以在网络上收集很多随机人物的图片，然后训练第一个神经网络来检测两个不同的图片是否是同一个人。这样的网络将会学习到很好的人脸特征检测器，因此重用其较低层可以使你用很少的训练数据来训练一个好的人脸分类器。

对于自然语言处理（NLP）应用，你可以下载数百万个文本文档的语料库并从中自动生成带标签的数据。例如，你可以随机屏蔽一些单词并训练模型来预测缺失的单词（例如，它应该能预测句子“What___you saying”中的缺失单词可能是“are”或者“were”）。如果你可以训练模型在此任务上达到良好的性能，那么它已经对语言有相当多的了解，你当然可以在实际任务中重用它并在带标签的数据上进行微调（我们将在第15章中讨论更多的预训练任务）。

 自我监督学习是指你从数据本身自动生成标签，然后使用有监督学习技术在所得到的“标签”数据集上训练模型。由于此方法不需要任何人工标记，因此最好将其分类为无监督学习的一种形式。

11.3 更快的优化器

训练一个非常大的深度神经网络可能会非常缓慢。到目前为止，我们已经知道了四种加快训练速度（并获得了更好的解决方法）的方法：对连接权重应用一个良好的初始化策略，使用良好的激活函数，使用批量归一化，以及重用预训练网络的某些部分（可能建立在辅助任务上或使用无监督学习）。与常规的梯度下降优化器相比，使用更快的优化器也可以带来巨大的速度提升。在本节中，我们将介绍最受欢迎的算法：动量优化、Nesterov加速梯度、AdaGrad、RMSProp，最后是Adam和Nadam优化。

11.3.1 动量优化

想象一下，一个保龄球在光滑的表面上沿着平缓的坡度滚动：它开始时速度很慢，但很快会获得动量，直到最终达到终极速度（如果有摩擦或空气阻力）。这是Boris Polyak在1964年提出的动量优化背后的非常简单的想法[1]。相比之下，常规的梯度下降法只是在斜坡上采取小的、常规的步骤，因此算法将花费更多的时间到达底部。

回想一下，梯度下降通过直接减去权重的成本函数$J(\theta)$的梯度乘以学习率η（$\eta \nabla_\theta J(\theta)$）来更新权重$\theta$。等式是：$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$。它不关心较早的梯度是什么。如果局部梯度很小，则它会走得非常缓慢。

动量优化非常关心先前的梯度是什么：在每次迭代时，它都会从动量向量m（乘以学习率η）中减去局部梯度，并通过添加该动量向量来更新权重（见公式11-4）。换句话说，梯度是用于加速度而不是速度。为了模拟某种摩擦机制并防止动量变得过大，该算法引入了一个新的超参数β，称为动量，必须将其设置为0（高摩擦）和1（无摩擦）之间。典型的动量值为0.9。

公式11-4：动量算法

1. $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta)$

2. $\theta \leftarrow \theta + \mathbf{m}$

你可以轻松地验证，如果梯度保持恒定，则最终速度（即权重更新的最大大小）等于该梯度乘以学习率η再乘以1/（1 - β）（忽略符号）。例如，如果β=0.9，则最终速度等于梯度乘以学习率的10倍，因此动量优化最终比梯度下降快10倍！这使动量优化比梯度下降要更快得地从平台逃脱。我们在第4章中看到，当输入的比例非常不同时，成本函数将看起来像一个拉长的碗（见图4-7）。梯度下降相当快地沿着陡峭的斜坡下降，但是沿着山谷下降需要很长时间。相反，动量优化将沿着山谷滚动得越来越快，直到达到谷底（最优解）。在不使用批量归一化的深层神经网络中，较高的层通常会得到比例不同的输入，因此使用动量优化会有所帮助。它还可以帮助绕过局部优化问题。

 由于这种动量势头，优化器可能会稍微过调，然后又回来，再次过调，在稳定于最小点之前会多次振荡。这是在系统中有一些摩擦力的原因之一：它消除了这些振荡，从而加快了收敛速度。

在Keras中实现动量优化不费吹灰之力：只需要使用SGD优化器并设置其超参数momentum，然后就可躺下来看效果！

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

动量优化的一个缺点是它增加了另一个超参数来调整。但是，动量值为0.9通常在实践中效果很好，几乎总是比常规的“梯度下降”更快。

11.3.2 Nesterov加速梯度

Yurii Nesterov于1983年[2]提出的动量优化的一个小变体几乎总是比原始动量优化要快。Nesterov加速梯度（Nesterov Accelerated Gradient，NAG）方法也称为Nesterov动量优化，它不是在局部位置 θ，而是在 $\theta+\beta m$处沿动量方向稍微提前处测量成本函数的梯度（见公式11-5）。

公式11-5：Nesterov加速梯度算法

1. $m \leftarrow \beta m - \eta \Delta_\theta J(\theta+\beta m)$

2. $\theta \leftarrow \theta + m$

这种小的调整有效是因为通常动量向量会指向正确的方向（即朝向最优解），因此使用在该方向上测得的更远的梯度而不是原始位置上的梯度会稍微准确一些。如图11-6所示（其中∇_1代表在起点 θ 处测量的成本函数的梯度，∇_2代表在位于 $\theta+\beta m$点的梯度）。

正如你所看到的，Nesterov更新会最终稍微接近最优解。一段时间后，这些小的改进累加起来，NAG就比常规的动量优化要快得多。此外，请注意，当动量推动权重跨越谷底时，∇_1继续推动越过谷底，而∇_2推回谷底。这有助于减少振荡，因此NAG收敛更快。

NAG通常比常规动量优化更快。要使用它，只需在创建SGD优化器时设置nesterov=True即可：

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

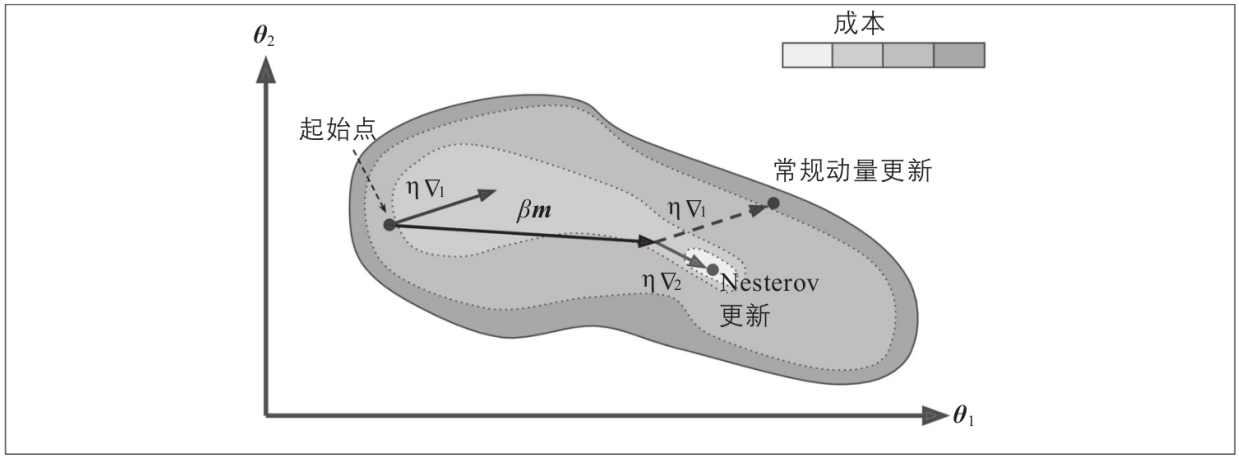


图11-6：常规与Nesterov动量优化：前者使用在动量步骤之前计算的梯度，而后者使用在动量步骤之后计算的梯度

11.3.3 AdaGrad

再次考虑拉长的碗状问题：梯度下降从快速沿最陡的坡度下降开始，该坡度没有直接指向全局最优解，然后非常缓慢地下降到谷底。如果算法可以更早地纠正其方向，使它更多地指向全局最优解，那将是很好的。AdaGrad算法[3]通过沿最陡峭的维度按比例缩小梯度向量（见公式11-6）来实现此校正。

公式11-6：AdaGrad算法

$$1.\quad \boldsymbol{s} \leftarrow \boldsymbol{s} + \nabla_{\theta} J(\boldsymbol{\theta}) \otimes \nabla_{\theta} J(\boldsymbol{\theta})$$

$$2.\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \, \nabla_{\theta} J(\boldsymbol{\theta}) \oslash \sqrt{\boldsymbol{s} + \varepsilon}$$

第一步将梯度的平方累加到向量s中（请记住，⊗符号表示逐元素相乘）。此向量化形式等效于针对向量s中的每个元素si计算$s_i \leftarrow s_i +$（$\partial J(\theta)/\partial \theta_i$）^{2}。换句话说，每个$s_i$累加关于参数$\theta_i$的成本函数偏导数的平方。如果成本函数沿第i个维度陡峭，则s_i将在每次迭代中变得越来越大。

第二步几乎与“梯度下降”相同，但有一个很大的区别：梯度向量按比例因子$\sqrt{\boldsymbol{s}+\varepsilon}$缩小了（⊘符号代表逐元素相除，而 ε 是避免除以零的平滑项，通常设置为10^{-10}）。此向量化形式等效于对所有参数 θ i同时计算$\theta_i \leftarrow \theta_i - \eta\, \partial J(\boldsymbol{\theta}) / \partial \theta_i / \sqrt{\boldsymbol{s}_i+\varepsilon}$。

简而言之，该算法会降低学习率，但是对于陡峭的维度，它的执行速度要比对缓慢下降的维度的执行速度要快。这称为自适应学习率。它有助于将结果更新更直接地指向全局最优解（见图11-7）。另一个好处是，它几乎不需要调整学习率超参数 η 。

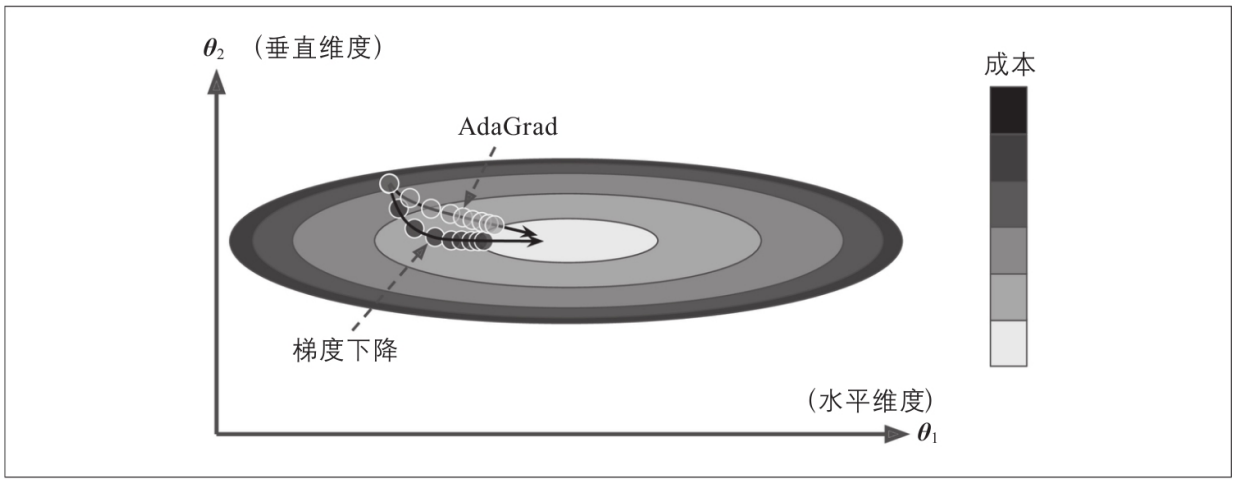


图11-7：AdaGrad与梯度下降法：前者可以较早地修正其方向，使之指向最优解

对于简单的二次问题，AdaGrad经常表现良好，但是在训练神经网络时，它往往停止得太早。学习率被按比例缩小，以至于算法在最终达到全局最优解之前完全停止了。因此，即使Keras有Adagrad优化器，你也不应使用它来训练深度神经网络（不过，它对于诸如线性回归之类的简单任务可能是有效的）。尽管如此，了解AdaGrad仍有助于掌握其他自适应学习率优化器。

11.3.4 RMSProp

正如我们所看到的，AdaGrad有下降太快，永远不会收敛到全局最优解的风险。RMSProp算法[4]通过只是累加最近迭代中的梯度（而不是自训练开始以来的所有梯度）来解决这个问题。它通过在第一步中使用指数衰减来实现（见公式11-7）。

公式11-7：RMSProp算法

$$1.\quad \mathbf{s} \leftarrow \beta \mathbf{s} + (1-\beta)\ \nabla_\theta J(\boldsymbol{\theta}) \otimes \nabla_\theta J(\boldsymbol{\theta})$$

$$2.\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\ \nabla_\theta J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$$

衰减率β通常设置为0.9。是的，它又是一个新的超参数，但是此默认值通常效果很好，因此你可能根本不需要调整它。

如你所料，Keras有RMSprop优化器：

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

注意rho参数对应于公式11-7中的β，除了非常简单的问题外，该优化器几乎总是比AdaGrad表现更好。实际上，直到Adam优化出现之前，它一直是许多研究人员首选的优化算法。

11.3.5　Adam和Nadam优化

Adam[5]代表自适应矩估计，结合了动量优化和RMSProp的思想：就像动量优化一样，它跟踪过去梯度的指数衰减平均值。就像RMSProp一样，它跟踪过去平方梯度的指数衰减平均值（见公式11-8）[6]。

公式11-8：Adam算法

1. $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} - (1-\beta_1)\ \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1-\beta_2)\ \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\hat{\boldsymbol{m}} \leftarrow \frac{m}{1-\beta_1^t}$
4. $\hat{\boldsymbol{s}} \leftarrow \frac{s}{1-\beta_2^t}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \hat{\boldsymbol{m}} \oslash \sqrt{\hat{\boldsymbol{s}} + \varepsilon}$

在此等式中，t表示迭代次数（从1开始）。

如果只看步骤1、2和5，你会发现Adam与动量优化和RMSProp非常相似。唯一的区别是步骤1计算的是指数衰减的平均值，而不是指数衰减的总和，但除了常数因子（衰减平均值是衰减总和的1 - β_1倍）外，它们实际上是等效的。第3步和第4步在技术上有些细节：由于m和s初始化为0，因此在训练开始时它们会偏向0，这两个步骤将有助于在训练开始时提高m和s。

动量衰减超参数β_1通常被初始化为0.9，而缩放衰减超参数β_2通常被初始化为0.999。如前所述，平滑项ε通常会初始化为一个很小的数字，例如10^{-7}。这些是Adam类的默认值（准确地说，epsilon的默认值为None，它告诉Keras使用keras.backend.epsilon（），默认值为10 - 7。你可以使用keras.backend.set_epsilon（）来改变）。这是使用Keras来创建Adam优化器的方法：

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

由于Adam是一种自适应学习率算法（如AdaGrad和RMSProp），因此对学习率超参数η需要较少的调整。你通常可以使用默认值η=0.001，这使得Adam甚至比梯度下降更易于使用。

 如果你开始对所有这些不同的技术感到不知所措，并且想知道如何为自己的任务选择合适的一个，请不要担心，本章末提供了一些实用的准则。

最后，Adam的两个变体值得一提：

AdaMax

请注意，在公式11-8的步骤2中，Adam累加了s中的梯度平方（对于最近的梯度，权重更大）。在第5步中，如果忽略 ε 和第3步和第4步（无论如何都是技术细节），Adam将以s的平方根按比例缩小参数更新。简而言之，Adam按时间衰减梯度的l_2范数按比例缩小参数更新（请注意，l_2范数是平方和的平方根）。与Adam在同一篇论文中介绍的AdaMax将l_2范数替换为l_∞范数（一种表达最大值的新颖方法）。具体来说，它用$s \leftarrow \max(\beta_2 s, \nabla_\theta J(\theta))$替换公式11-8中的步骤2，它删除第4步，在第5步中，将梯度更新按比例s缩小，这是时间衰减梯度的最大值。实际上，这可以使AdaMax比Adam更稳定，但这确实取决于数据集，通常Adam的表现更好。因此，如果你用Adam在某些任务上遇到问题，这是你可以尝试的另一种优化器。

Nadam

Nadam优化是Adam优化加上Nesterov技巧，因此其收敛速度通常比Adam稍快。在他介绍这种技术的报告中[7]，研究人员Timothy Dozat在各种任务上对许多不同的优化器进行了比较，发现Nadam总体上胜过Adam，但有时不如RMSProp。

 自适应优化方法（包括RMSProp、Adam和Nadam优化）通常很棒，可以快速收敛到一个好的解决方案。然而，Ashia C.Wilson等人在2017年发表的论文[8]，表明它们可以导致在某些数据集上泛化不佳的

解决方案。因此，当你对模型的性能感到失望时，尝试改用普通的Nesterov加速梯度：你的数据集可能对自适应梯度过敏。还要检查最新的研究，因为它发展很快。

到目前为止讨论的所有优化技术都仅仅依赖于一阶偏导数（Jacobians）。一些优化文献还包含了基于二阶偏导数（Hessian，这是Jacobian的偏导数）的一些算法。不幸的是，这些算法很难应用于深度神经网络，因为每个输出有n^2个Hessian（其中n是参数的数量），而不是每个输出只有n个Jacobian。由于DNN通常具有成千上万的参数，因此二阶优化算法甚至不适合存储在内存中，即使可以的话，计算Hessians也太慢了。

训练稀疏模型

所有的优化算法都产生了密集模型，这意味着大多数参数都是非零的。如果你在运行时需要一个非常快的模型，或者需要占用更少的内存，那么你可能更喜欢使用一个稀疏模型。

实现这一点的一个简单方法是像往常一样训练模型，然后去掉很小的权重（将它们设置为零）。注意，这通常不会导致非常稀疏的模型，而且可能会降低模型的性能。一个更好的选择是在训练时使用强l正规化（我们在本章后面将看到怎么做），因为它会迫使优化器产生尽可能多的为零的权重（详见4.5.2节）。

如果这些技术仍然不够，请查看TensorFlow Model Optimization Toolkit（TF-MOT），它提供了一个剪枝API，能够根据连接的大小在训练期间迭代地删除连接。

表11-2比较了我们目前讨论的所有优化器（*不好，**平均，***好）。

表11-2：优化器比较

类别	收敛速度	收敛质量
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	***
Adagrad	***	*（停止太早）
RMSprop	***	** 或者 ***
Adam	***	** 或者 ***
Nadam	***	** 或者 ***
AdaMax	***	** 或者 ***

11.3.6 学习率调度

找到一个好的学习率非常重要。如果你设置得太高，训练可能会发散（如4.2节所述）。如果你设置得太低，训练最终会收敛到最优解，但是这将花费很长时间。如果将它设置得稍微有点高，它一开始会很快，但是最终会围绕最优解震荡，不会真正稳定下来。如果你的计算力预算有限，则可能必须先中断训练，然后才能正确收敛，从而产生次优解决（见图11-8）。

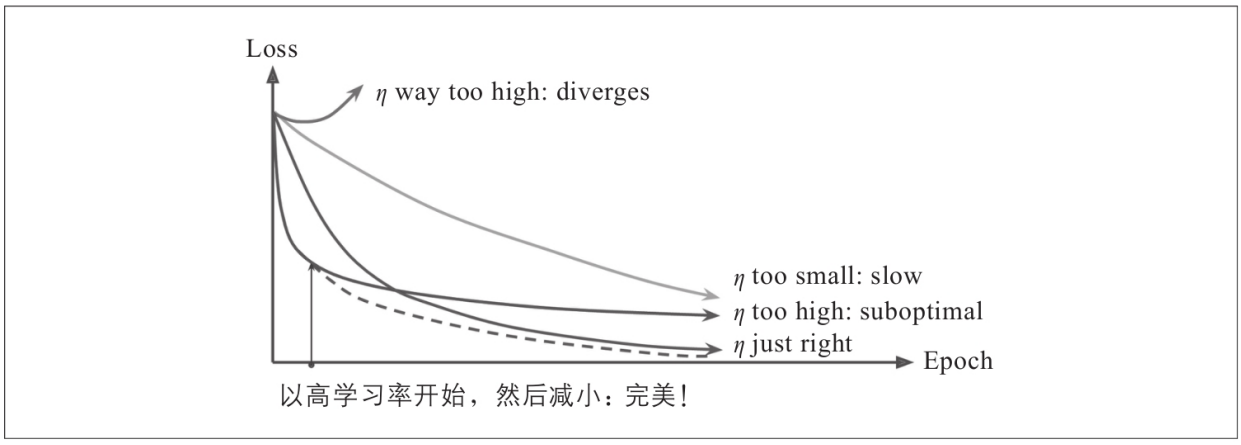


图11-8：各种学习率 η 的学习曲线

正如我们在第10章中讨论的那样，你可以对模型进行数百次的迭代训练以找到一个良好的学习率，然后将学习率从很小的值呈指数级地增加到很大的值，再查看学习曲线并选择一个学习率略低于学习曲线开始

回升的时候。然后，你可以重新初始化模型，并以该学习率对其进行训练。

但是，你可以做得比恒定学习率更好：如果你从一个较大的学习率开始，一旦训练没有取得进展后就降低它，那与恒定学习率相比，你就可以更快地找到一个最优解。有许多不同的策略可以降低训练期间的学习率。从低学习率开始，增加它，然后再降低它也是有好处的。这些策略称为学习率调度（我们在第4章中简要介绍了此概念）。以下这些是最常用的学习率调度：

幂调度

将学习率设置为迭代次数t的函数：$\eta(t) = \eta_0 / (1+t/s)^c$。初始学习率$\eta_0$、幂c（通常设置为1）和步骤s是超参数。学习率在每一步都会下降。在s个步骤之后，它下降到$\eta_0/2$。再在s个步骤之后，它下降到$\eta_0/3$，然后下降到$\eta_0/4$，然后是$\eta_0/5$，以此类推。如你所见，此调度开始迅速下降，然后越来越慢。当然，幂调度需要调整η_0和s（可能还有c）。

指数调度

将学习速率设置为$\eta(t) = \eta_0 \ 0.1^{t/s}$。学习率每s步将逐渐下降10倍。幂调度越来越缓慢地降低学习率，而指数调度则使学习率每s步降低10倍。

分段恒定调度

对一些轮次使用恒定的学习率（例如，对于5个轮次，$\eta_0=0.1$），对于另外一些轮次使用较小的学习率（例如，对于50个轮次，$\eta_0=0.001$），以此类推。尽管这个方法可以很好地工作，但仍需要仔细研究以找出正确的学习率顺序以及使用它们的轮次。

性能调度

每N步测量一次验证误差（就像提前停止一样），并且当误差停止下降时，将学习率降低 λ 倍。

1周期调度1

与其他方法相反，1周期调度（在Leslie Smith于2018年的论文中引入[9]）从提高初始学习率 η_0 开始，在训练中途线性增长至 η_1。然后，它在训练的后半部分将学习率再次线性降低到 η_0，通过将学习率降低几个数量级（仍然是线性的）来完成最后几个轮次。使用与找到最优学习率相同的方法来选择最大学习率 η_1，而初始学习率 η_0 大约要低10倍。当使用动量时，我们首先从高动量开始（例如0.95），然后在训练的前半部分将其降低到较低的动量（例如线性降低到0.85），然后在训练的后半部分将其恢复到最大值（例如0.95），并以该最大值结束最后几个轮次。史密斯做了很多实验，表明这种方法通常能够大大加快训练速度并达到更好的性能。例如，在流行的CIFAR10图像数据集上，此方法仅在100个轮次内就达到了91.9％的验证准确率，而不是通过标准方法（具有相同的神经网络架构）在800个轮次内达到的90.3％的准确率。

2013年，Andrew Senior等人撰写的论文[10]比较了使用动量优化来训练深度神经网络进行语音识别时一些最受欢迎的学习率调度的性能。作者得出的结论是，在这种情况下，性能调度和指数调度都表现良好。他们喜欢指数调度，因为它易于微调，比最优解决方法收敛速度更快（他们还提到实现起来比性能调度更容易，但在Keras中，这两个选项都很简单）。也就是说，1周期方法的性能似乎更好。

在Keras中实现幂调度是最简单的选择：只需在创建优化器时设置超参数decay即可：

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

decay是s（学习率除以多个数字单位所需要的步数）的倒数，Keras假定c等于1。

指数调度和分段调度也非常简单。你首先需要定义一个函数，该函数采用当前轮次并返回学习率。例如，让我们实现指数调度：

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

如果不想对η_0和s进行硬编码，则可以创建一个返回配置函数的函数：

```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1**(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

接下来，创建一个LearningRateScheduler回调函数，为其提供调度函数，然后将此回调函数传递给fit（）方法：

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

LearningRateScheduler将在每个轮次开始时更新优化器的learning_rate属性。通常每个轮次更新一次学习率就足够了，但是如果你希望更频繁地更新学习率（例如在每个步骤中），则可以编写自己

的回调函数（请参阅notebook的“Exponential Scheduling”部分示例）。如果每个轮次有很多个步骤，则每一步都更新学习率是有意义的。另外，你可以使用keras.optimizers.schedules方法，这将在稍后进行介绍。

调度函数可以选择将当前学习率作为第二个参数。例如，以下调度函数将以前的学习率乘以$0.1^{1/20}$，这将导致相同的指数衰减（但现在衰减从轮次0开始而不是1开始）：

```
def exponential_decay_fn(epoch, lr):
    return lr * 0.1**(1 / 20)
```

此实现依赖于优化器的初始学习率（与先前的实现相反），因此请确保对其进行适当的设置。

保存模型时，优化器及其学习率也会随之保存。这意味着有了这个新的调度函数，你只需加载经过训练的模型，从中断的地方继续进行训练。但是，如果你的调度函数使用epoch参数，事情就变得不那么简单了：epoch不会被保存，并且每次你调用fit（）方法时都会将其重置为0。如果你要继续训练一个中断的模型，则可能会导致一个很高的学习率，这可能会损害模型的权重。一种解决方法是手动设置fit（）方法的initial_epoch参数，使epoch从正确的值开始。

对于分段恒定调度，可以使用以下调度函数（如前所述，可以根据需要定义一个更通用的函数。有关示例，请参见notebook的“Diecewise Lonstant Scheduling”部分），然后创建带有此函数的LearningRateScheduler回调函数，并将其传递给fit（）方法，就像我们对指数调度所做的那样：

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
```

```
        return 0.01
    elif epoch < 15:
        return 0.005
    else:
        return 0.001
```

对于性能调度，请使用ReduceLROnPlateau回调函数。例如，如果将以下回调函数传递给fit（）方法，则每当连续5个轮次的最好验证损失都没有改善时，它将使学习率乘以0.5（有其他选项可用，请查看文档以获取更多详细信息）：

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
```

最后，tf.keras提供了另一种实现学习率调度的方法：使用keras.optimizers.schedule中可以使用的调度之一来定义学习率，然后将该学习率传递给任意优化器。这种方法在每个步骤更新学习率而不是每个轮次。例如，以下是实现与我们先前定义的exponential_decay_fn（）函数相同的指数调度的方法：

```
s = 20 * len(X_train) // 32 # number of steps in 20 epochs (batch size = 32)
learning_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
```

这很简单，而且当你保存模型时，学习率及其调度（包括其状态）也将被保存。但是，这种方法不是Keras API的一部分，它只适用于tf.keras。

对于1周期方法，实现起来没有特别的困难：只需创建一个每次迭代都能修改学习率的自定义回调函数即可（你可以通过更改self.model.optimizer.lr来更新优化器的学习率）。有关示例请参见notebook的“lycle scheduling”部分。

总而言之，指数衰减、性能调度和1周期都可以大大加快收敛速度，因此请尝试一下！

[1] Boris T.Polyak，“Some Methods of Speeding Up the Convergence of Iteration Methods，”USSR Computational Mathematics and Mathematical Physics 4，no.5（1964）：1-17.
[2] Yurii Nesterov，“A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence O（1/k2），”Doklady AN USSR 269（1983）：543-547.
[3] John Duchi et al.，“Adaptive Subgradient Methods for Online Learning and Stochastic Optimization，”Journal of Machine Learning Research 12（2011）：2121－2159.
[4] 该算法由Geoffrey Hinton和Tijmen Tieleman于2012年创建，并由Geoffrey Hinton在他的Coursera神经网络课程中（幻灯片：https://homl.info/57；视频：https://homl.info/58）介绍。有趣的是，由于作者没有写论文来描述该算法，因此研究人员经常在它们论文中引用为“slide 29 in lecture 6”。
[5] Diederik P.Kingma and Jimmy Ba，“Adam：A Method for Stochastic Optimization，”arXiv preprint arXiv：1412.6980（2014）.
[6] 这些是对梯度的均值和（无中心）方差的估计。均值通常称为一阶矩，而方差通常称为二阶矩，这是算法名字的由来。
[7] Timothy Dozat，“Incorporating Nesterov Momentum into Adam”（2016）.
[8] Ashia C.Wilson et al.，“The Marginal Value of Adaptive Gradient Methods in Machine Learning，”Advances in Neural Information Processing Systems 30（2017）：4148-4158.
[9] Leslie N.Smith，“A Disciplined Approach to Neural Network Hyper-Parameters：Part 1—Learning Rate，Batch Size，Momentum，and Weight Decay，”arXiv preprint arXiv：1803.09820（2018）.

[10] Andrew Senior et al.，“An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition，”Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing（2013）：6724－6728.

11.4　通过正则化避免过拟合

有了4个参数，我可以拟合出一头大象，有了5个参数，我可以让它摆动鼻子。

——John von Neumann，Enrico Fermi在Nature 427中引用

拥有数千个参数，你可以拟合出整个动物园。深度神经网络通常具有数万个参数，有时甚至有数百万个。这给它们带来了难以置信的自由度，意味着它们可以拟合各种各样的复杂数据集。但是，这种巨大的灵活性也使网络易于过拟合训练集。我们需要正则化。我们已经在第10章中实现了最好的正则化技术之一：提前停止。而且，即使“批量归一化”被设计用来解决不稳定的梯度问题，它的作用也像一个很好的正则化。在本节中，我们将研究其他流行的神经网络正则化技术：l_1和l_2正则化、dropout和最大范数正则化。

11.4.1　l_1和l_2正则化

就像在第4章中对简单线性模型所做的一样，可以使用l_2正则化来约束神经网络的连接权重，如果想要稀疏模型（许多权重等于0）则可以使用l_1正则化。以下是使用0.01的正则化因子将l_2正则化应用于Keras层的连接权重的方法：

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

12（）函数返回一个正则化函数，在训练过程中的每个步骤都将调用该正则化函数来计算正则化损失。然后将其添加到最终损失中。如你所料，如果需要l_1正则化，可以只使用keras.regularizers.l1（）。

如果你同时需要l_1和l_2正则化，请使用keras.regularizers.l1_l2（）（同时指定两个正则化因子）。

由于你通常希望将相同的正则化函数应用于网络中的所有层，并在所有隐藏层中使用相同的激活函数和相同的初始化策略，因此你可能会发现自己重复了相同的参数。这使代码很难看且容易出错。为了避免这种情况，你可以尝试使用循环来重构代码。另一种选择是使用Python的functools.partial（）函数，该函数可以使你为带有一些默认参数值的任何可调用对象创建一个小的包装函数：

```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

11.4.2 dropout

对于深度神经网络，dropout是最受欢迎的正则化技术之一。它是由Geoffrey Hinton在2012年的论文中提出的[1]，并且在Nitish Srivastava等人2014年的论文中得到了进一步的详述[2]，已被证明是非常成功的：只需要增加dropout，即使最先进的神经网络也能获得1～2%的准确率提升。这听起来可能不算很多，但是当模型已经具有95%的准确率时，将准确率提高2%意味着将错误率降低了近40%（从5%的错误率降低到大约3%）。

这是一个非常简单的算法：在每个训练步骤中，每个神经元（包括输入神经元，但始终不包括输出神经元）都有暂时“删除”的概率p，

这意味着在这个训练步骤中它被完全忽略，但在下一步中可能处于活动状态（见图11-9）。超参数p称为dropout率，通常设置为10％到50％：在循环神经网络中接近20～30％（见第15章），在卷积神经网络中接近40～50％（见第14章）。训练后，神经元不再被删除。这就是全部内容（除了一个技术细节，我们马上进行讨论）。

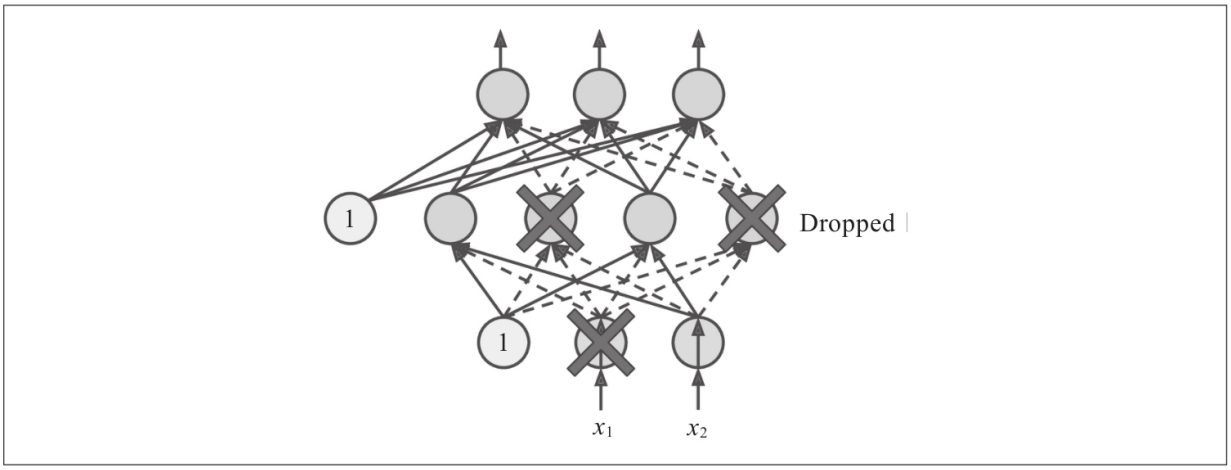


图11-9：使用dropout正则化，在每个训练迭代中，一个或多个层中的所有神经元的随机子集（输出层除外）被“删除”：这些神经元在这个迭代中输出0（用虚线箭头表示）

首先令人惊讶的是，这种破坏性技术完全有效。如果公司告诉员工每天早上扔硬币来决定是否去上班，公司的业绩会更好吗？好吧，谁知道呢？也许会！该公司将被迫调整其组织结构。它不能依靠任何一个人来操作咖啡机或执行任何其他关键任务，因此必须将这种专业知识分散到多个人中。员工必须学会与许多同事合作，而不仅仅是少数几个。该公司将变得更具弹性。如果一个人辞职，不会有太大的区别。尚不清楚这种想法是否真的适用于公司，但它确实适用于神经网络。经过dropout训练的神经元不能与其相邻的神经元相互适应，它们必须自己发挥最大的作用。它们也不能过分依赖少数输入神经元，它们必须注意每个输入神经元。它们最终对输入的微小变化不太敏感。最后，你将获得一个更有鲁棒性的网络，该网络有更好的泛化能力。

了解dropout能力的另一种方法是认识到在每个训练步骤中都会生成一个独特的神经网络。由于每个神经元都可以存在或不存在，因此共有2^N个可能的网络（其中N是可以dropout神经元的总数）。这是一个巨大的数字，几乎不可能对同一神经网络进行两次采样。一旦你运行了10 000个训练步骤，你实质上就已经训练了10 000个不同的神经网络（每个神经网络只有一个训练实例）。这些神经网络显然不是独立的，因为它们共享许多权重，但是它们却是完全不同的。所得的神经网络可以看作是所有这些较小的神经网络的平均集成。

 在实践中，你通常只可以对第一至第三层（不包括输出层）中的神经元应用dropout。

有一个很小但很重要的技术细节。假设p=50%，在这种情况下，在测试过程中，一个神训练深度神经网络丨323经元被连接到输入神经元的数量是训练期间（平均）的两倍。为了弥补这一事实，我们需要在训练后将每个神经元的输入连接权重乘以0.5。如果不这样做，每个神经元得到的总输入信号大约是网络训练时的两倍，表现可能不会很好。更一般而言，训练后我们需要将每个输入连接权重乘以保留概率（1-p）。或者，我们可以在训练过程中将每个神经元的输出除以保留概率（这些替代方法不是完全等效的，但效果一样好）。

要使用Keras实现dropout，可以使用keras.layers.Dropout层。在训练期间，它会随机丢弃一些输入（将它们设置为0），然后将其余输入除以保留概率。训练之后，它什么都不做，只是将输入传递到下一层。以下代码使用0.2的dropout率在每个Dense层之前应用dropout正则化：

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
```

```
        keras.layers.Dropout(rate=0.2),
        keras.layers.Dense(10, activation="softmax")
    ])
```

 由于dropout仅在训练期间激活，因此比较训练损失和验证损失可能会产生误导。具体而言，模型可能会过拟合训练集，但仍具有相似的训练损失和验证损失。因此，请确保没有使用dropout来评估训练损失（例如，训练之后）。如果你发现模型过拟合，则可以提高dropout率。相反，如果模型欠拟合训练集，则应尝试降低dropout率。它还可以帮助增加较大层的dropout率，并减小较小层的dropout率。此外，许多最新的架构仅在最后一个隐藏层之后才使用dropout，因此如果完全dropout太强，你可以尝试一下此方法。

dropout确实会明显减慢收敛速度，但是如果正确微调，通常会导致更好的模型。因此，花额外的时间和精力是值得的。

 如果要基于SELU激活函数（如前所述）对自归一化网络进行正则化，则应使用alpha dropout：这是dropout的一种变体，它保留了其输入的均值和标准差（在与SELU相同的论文中介绍，因为常规的dropout会破坏自归一化）。

11.4.3 蒙特卡罗（MC）Dropout

2016年，Yarin Gal和Zoubin Ghahramani的论文[3]增加了一些更好的理由来使用dropout：

• 首先，论文建立了dropout网络（即在每个权重层之前都包含一个dropout层的神经网络）与近似贝叶斯推理之间的深刻联系[4]，为dropout提供了坚实的数学依据。

• 其次，作者介绍了一种称为MC Dropout的强大技术，该技术可以提高任何训练后的dropout模型的性能，而无须重新训练甚至根本不用修改它，它可以更好地测量模型的不确定性，并且实现起来非常简单。

如果这一切听起来像是“一个奇怪的小窍门”广告，那么请看下面的代码。它是MC Dropout的完整实现，可提升我们先前训练的dropout模型，而无须重新训练它：

```
y_probas = np.stack([model(X_test_scaled, training=True)
                     for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

我们仅对测试集进行100个预测，设置training=True来确保Dropout层处于激活状态，然后把预测堆叠起来。由于dropout处于激活状态，因此所有预测都将有所不同。回想一下，predict（）返回一个矩阵，其中每个实例一行，每个类一列。因为测试集中有10 000个实例和10个类，所以这是一个形状为[10000，10]的矩阵。我们堆叠了100个这样的矩阵，因此y_probas是一个形状为[100，10000，10]的数组。一旦我们对第一个维度进行平均（axis=0），我们得到y_proba，它是形状[10000，10]的数组，就像我们通过单个预测得到的一样。就是这样！对具有dropout功能的多个预测进行平均，这使蒙特卡罗估计通常比关闭dropout的单个预测的结果更可靠。例如，让我们看一下模型对Fashion MNIST测试集中第一个实例的预测，关闭dropout：

```
>>> np.round(model.predict(X_test_scaled[:1]), 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.99]],
      dtype=float32)
```

该模型看起来似乎可以确定该图像属于第9类（踝靴）。你应该相信吗？真的有那么小的怀疑吗？将其与dropout激活时所做的预测进行

比较：

```
>>> np.round(y_probas[:, :1], 2)
array([[[0.  , 0.  , 0.  , 0.  , 0.  , 0.14, 0.  , 0.17, 0.  , 0.68]],
       [[0.  , 0.  , 0.  , 0.  , 0.  , 0.16, 0.  , 0.2 , 0.  , 0.64]],
       [[0.  , 0.  , 0.  , 0.  , 0.  , 0.02, 0.  , 0.01, 0.  , 0.97]],
       [...]
```

这说明了一个截然不同的故事：显然，当我们激活dropout时，该模型不能确定了。它似乎仍然偏爱第9类，但有点不确定第5类（凉鞋）和第7类（运动鞋），考虑到它们都是鞋类，这是有道理的。一旦我们对第一个维度进行平均，我们将获得以下MC Dropout预测：

```
>>> np.round(y_proba[:1], 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.22, 0.  , 0.16, 0.  , 0.62]],
      dtype=float32)
```

模型仍然认为此图像属于第9类，但只有62％的置信度，这比99％的置信度要合理得多。另外，准确了解它认为可能的其他类也很有用。你也可以看看概率估计的标准差：

```
>>> y_std = y_probas.std(axis=0)
>>> np.round(y_std[:1], 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.28, 0.  , 0.21, 0.02, 0.32]],
      dtype=float32)
```

显然，概率估计值存在很大差异：如果你要建立风险敏感的系统（例如医疗或金融系统），应该格外小心地对待这种不确定的预测。你绝对不会将其视为99％的自信预测。此外，模型的准确率从86.8提高到86.9：

```
>>> accuracy = np.sum(y_pred == y_test) / len(y_test)
>>> accuracy
0.8694
```

 你使用的蒙特卡罗样本数量（在此示例中为100）是可以调整的超参数。数值越高，预测及其不确定性估计的精度就越高。但是，如果将其加倍，则推理时间也将加倍。此外，多于一定数量的样本，你几乎看不到任何改善。因此，你的工作就是在延迟和准确率之间找到适当的权衡，具体取决于你的应用。

如果你的模型包含在训练过程中以特殊方式运行的其他层（例如BatchNormalization层），则你不应像我们刚才那样强制训练模式。相反，你应该使用以下MCDropout类[5]来替换Dropout层：

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

在这里，我们只是继承了Dropout层，并覆盖call（）方法来强制设置training参数为True（见第12章）。同样，你可以通过继承AlphaDropout来定义MCAlpha Dropout类。如果你要从头开始创建模型，则只需使用MCDropout而不是Dropout。但是，如果你有一个已经使用Dropout训练过的模型，则需要创建一个与现有模型相同的新模型，不同之处在于它用MCDropout替换了Dropout层，然后将现有模型的权重复制到你的新模型中。

简而言之，MC Dropout是一种出色的技术，可以提升dropout模型并提供更好的不确定性估计。当然，由于这只是训练期间的常规dropout，所以它也像正则化函数。

11.4.4 最大范数正则化

对于神经网络而言，另一种流行的正则化技术称为最大范数正则化：对于每个神经元，它会限制传入连接的权重w，使得$||w||_2 \leqslant r$，其中r是最大范数超参数，$||\cdot||_2$是ℓ_2范数。

最大范数正则化不会把正则化损失项添加到总体损失函数中。取而代之的是，通常在每个训练步骤后通过计算$||w||_2$来实现，如有需要，请重新缩放$w\left(w \leftarrow w\frac{r}{||w||_2}\right)$。

减小r会增加正则化的数量，并有助于减少过拟合。最大范数正则化还可以帮助缓解不稳定的梯度问题（如果你未使用“批量归一化”）。

要在Keras中实现最大范数正则化，请将每个隐藏层的kernel_constraint参数设置为具有适当最大值的max_norm（）约束，如下所示：

```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",
                   kernel_constraint=keras.constraints.max_norm(1.))
```

每次训练迭代后，模型的fit（）方法会调用由max_norm（）返回的对象，将层的权重传递给该对象，并获得返回的缩放权重，然后替换该层的权重。如第12章所述，如果需要，你可以定义自己的自定义约束函数，并将其用作kernel_constraint。你还可以通过设置bias_constraint参数来约束偏置项。

max_norm（）函数的参数axis默认为0。Dense层通常具有形状为[输入数量，神经元数量]的权重，因此使用axis=0意味着最大范数约束将独立应用于每个神经元的权重向量。如果你要将最大范数与卷积层一

起使用（见第14章），请确保正确设置max_norm（）约束的axis参数（通常axis=[0，1，2]）。

[1] Geoffrey E.Hinton et al.，“Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors，”arXiv preprint arXiv：1207.0580（2012）.
[2] Nitish Srivastava et al.，“Dropout：A Simple Way to Prevent Neural Networks from Overfitting，”Journal of Machine Learning Research 15（2014）：1929－1958.
[3] Yarin Gal and Zoubin Ghahramani，“Dropout as a Bayesian Approximation：Representing Model Uncertainty in Deep Learning，”Proceedings of the 33rd International Conference on Machine Learning（2016）：1050－1059.
[4] 具体来说，它们表明，在称为深度高斯过程的特定类型的概率模型中，训练dropout网络在数学上等同于近似贝叶斯推理。
[5] 此MCDropout类可以与所有Keras API（包括顺序API）一起使用。如果你只关心函数API或子类API，则不必创建MCDropout类。你可以创建常规的Dropout层，使用training=True来进行调用。

11.5 总结和实用指南

在本章中，我们涵盖了广泛的技术，你可能想知道应该使用哪种技术。这取决于任务，还没有明确的共识，但是我发现表11-3中的配置在大多数情况下可以很好地工作，而无须进行大量的超参数调整。也就是说，请不要将这些默认值视为硬性规定！

表11-3：默认的DNN配置

超参数	默认值
内核初始化	He 初始化
激活函数	ELU

表11-3：默认的DNN配置（续）

超参数	默认值
归一化	浅层网络：不需要；深度网络：批量归一化
正则化	提前停止（如果需要，可加 ℓ_2）
优化器	动量优化（或 RMSProp 或 Nadam）
学习率调度	1 周期

如果网络是密集层的简单堆叠，则它可以自归一化，你应该使用表11-4中的配置。

表11-4：用于自归一化网络的DNN配置

超参数	默认值
内核初始化	LeCun 初始化
激活函数	SELU
归一化	不需要（自规一化）
正则化	如果需要：Alpha dropout
优化器	动量优化（或 RMSProp 或 Nadam）
学习率调度	1 周期

不要忘了归一化输入特征！你如果可以找到解决类似问题的神经网络，那应该尝试重用部分神经网络；如果有大量未标记的数据，则应使用无监督预训练；如果有相似任务的大量标记的数据，则应该在辅助任务上使用预训练。

虽然先前的指南应该能涵盖大多数情况，但以下是一些例外情况：

• 如果你需要稀疏模型，则可以使用1正则化（可以选择在训练后将很小的权重归零）。如果你需要更稀疏的模型，则可以使用TensorFlow模型优化工具包。这会破坏自归一化，因此在这种情况下，你应使用默认配置。

• 如果你需要低延迟的模型（执行闪电般快速预测的模型），则可能需要使用更少的层，将批量归一化层融合到先前的层中，并使用更快的激活函数，例如leaky ReLU或仅仅使用ReLU。拥有稀疏模型也将有所帮助。最后，你可能想把浮点精度从32位降低到16位甚至8位（见19.2节）。再一次检查TFMOT。

• 如果你要构建风险敏感的应用，或者推理延迟在你的应用中不是很重要，则可以使用MC Dropout来提高性能并获得更可靠的概率估计以及不确定性估计。

有了这些准则，你现在就可以训练非常深的网络了！我希望你现在相信使用Keras可以走很长一段路了。但是，有时候你可能需要更多的

控制。例如，编写自定义损失函数或调整训练算法。对于这种情况，你需要使用TensorFlow的较低级API，如第12章中所述。

11.6 练习题

1.将所有权重初始化为相同的值是否可以（只要该值是使用He初始化随机选择的）？2.将偏置项初始化为0可以吗？

3.列举SELU激活函数相比ReLU的三个优点。

4.在哪种情况下，你想使用以下每个激活函数：SELU、leaky ReLU（及其变体）、ReLU、tanh、logistic和softmax？

5.如果在使用SGD优化器时将超参数momentum设置得太接近1（例如0.99999），会发生什么情况？

6.列举三种能产生稀疏模型的方法。

7.dropout会减慢训练速度吗？它会减慢推理速度（即对新实例进行预测）吗？MC Dropout呢？

8.在CIFAR10图像数据集上练习训练深度神经网络：

a.构建一个DNN，其中包含20个隐藏的层，每个层包含100个神经元（这太多了，但这是本练习的重点）。使用He初始化和ELU激活函数。

b.使用Nadam优化和提前停止，在CIFAR10数据集上训练网络。你可以使用keras.datasets.cifar10.load_data（）加载它。该数据集由10个类别的60 000个32×32像素的彩色图像（50 000个用于训练，10 000个用于测试）组成，因此你需要一个具有10个神经元的softmax输出层。记住，每次更改模型的架构或超参数时，都要寻找正确的学习率。

c.现在尝试添加批量归一化并比较学习曲线：收敛速度是否比以前快？会产生更好的模型吗？它如何影响训练速度？

d.尝试用SELU替换批量归一化，并进行必要的调整以确保网络是自归一化的（即归一化输入特征，使用LeCun正态初始化，确保DNN仅仅包含一系列的密集层等）。

e.尝试使用Alpha Dropout正则化模型。然后，在不重新训练模型的情况下，看看是否可以使用MC Dropout获得更好的准确率。

f.使用1周期调度来重新训练模型，看看它是否可以提高训练速度和模型准确率。

附录A中提供了这些练习题的解答。

第12章　使用TensorFlow自定义模型和训练

到目前为止，我们仅仅使用了TensorFlow的高层API tf.keras，但它已经使我们走得很远了：我们构建了各种神经网络架构，包括回归和分类网络宽深网络以及自归一化网络，使用了各种技术，例如批归一化、dropout和学习率调度。实际上，你遇到的情况中有95%不需要tf.keras（和tf.data，见第13章）以外的任何内容。但是现在是时候深入研究TensorFlow并了解其底层Python API了。当你需要额外的控制来编写自定义损失函数、自定义指标、层、模型、初始化程序、正则化函数、权重约束等时，这将非常有用。你甚至可能需要完全控制训练循环本身，例如对梯度使用特殊的变换或约束（不仅仅对它们进行裁剪），或者对网络的不同部分使用多个优化器。我们将在本章介绍所有这些情况，还将探讨如何使用TensorFlow的自动图形生成功能来增强自定义模型和训练算法。但是首先，让我们快速浏览一下TensorFlow。

 TensorFlow 2.0（beta）于2019年6月发布，使TensorFlow更加易于使用。本书的第一版使用TF 1，而此版使用TF 2。

12.1 TensorFlow快速浏览

如你所知，TensorFlow是一个强大的用于数值计算的库，特别适合大规模机器学习并对其进行了微调（但你可以将其用于需要大量计算的任何其他操作）。它由Google Brain团队开发，并为许多Google的大规模服务提供了支持，例如Google Cloud Speech、Google Photos和Google Search。它于2015年11月开源，现在是最受欢迎的深度学习库（就论文引用、公司采用率、GitHub上的星星数量等而言）。无数项目将TensorFlow用于各种机器学习任务，例如图像分类、自然语言处理、推荐系统和时间序列预测。

那么TensorFlow提供什么？总结如下：

- 它的核心与NumPy非常相似，但具有GPU支持。

- 它支持分布式计算（跨多个设备和服务器）。

- 它包含一种即时（JIT）编译器，可使其针对速度和内存使用情况来优化计算。它的工作方式是从Python函数中提取计算图，然后进行优化（通过修剪未使用的节点），最后有效地运行它（通过自动并行运行相互独立的操作）。

- 计算图可以导出为可移植格式，因此你可以在一个环境中（例如在Linux上使用Python）训练TensorFlow模型，然后在另一个环境中（例如在Android设备上使用Java）运行TensorFlow模型。

- 它实现了自动微分（autodiff）（见第10章和附录D），并提供了一些优秀的优化器，例如RMSProp和Nadam（见第11章），因此你可以轻松地最小化各种损失函数。

TensorFlow在这些核心功能的基础上提供了更多功能：最重要的当然是tf.keras[1]，但它还具有数据加载和预处理操作（tf.data、tf.io等）、图像处理操作（tf.image）、信号处理操作（tf.signal）等（有关TensorFlow的Python API的概述见图12-1）。

我们将介绍TensorFlow API的许多包和功能，但是不可能全部覆盖它们，因此你应该花一些时间浏览API。你会发现它非常丰富并且有很好的文档。

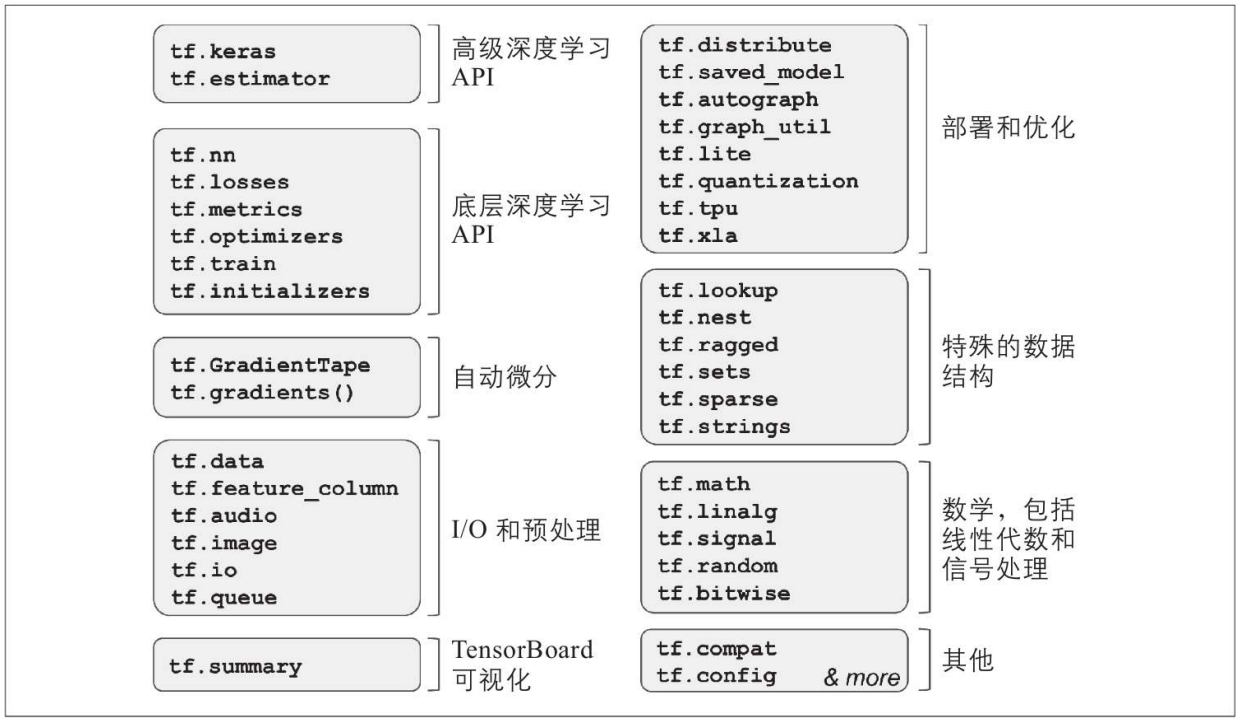


图12-1：TensorFlow的Python API

在最底层，每个TensorFlow操作（以下简称op）都是使用高效的C++代码[2]实现的。许多操作都有称为内核的多种实现：每个内核专用于特定的设备类型，例如CPU、GPU甚至TPU（张量处理单元）。如你所知，GPU可以通过将GPU分成许多较小的块并在多个GPU线程中并行运行它们来极大地加快计算速度。TPU甚至更快：它们是专门为深度学习操

作[3]而构建的定制ASIC芯片（我们将在第19章中讨论如何利用GPU或TPU来使用TensorFlow）。

TensorFlow的架构如图12-2所示。大多数时候，你的代码使用高级API（尤其是tf.keras和tf.data）。但是当你需要更大的灵活性时，可以使用较低级别的Python API直接处理张量。注意也可以使用其他语言的API。无论如何，TensorFlow的执行引擎都会有效地运行操作，如果你告诉它，它也可以跨多个设备和机器运行。

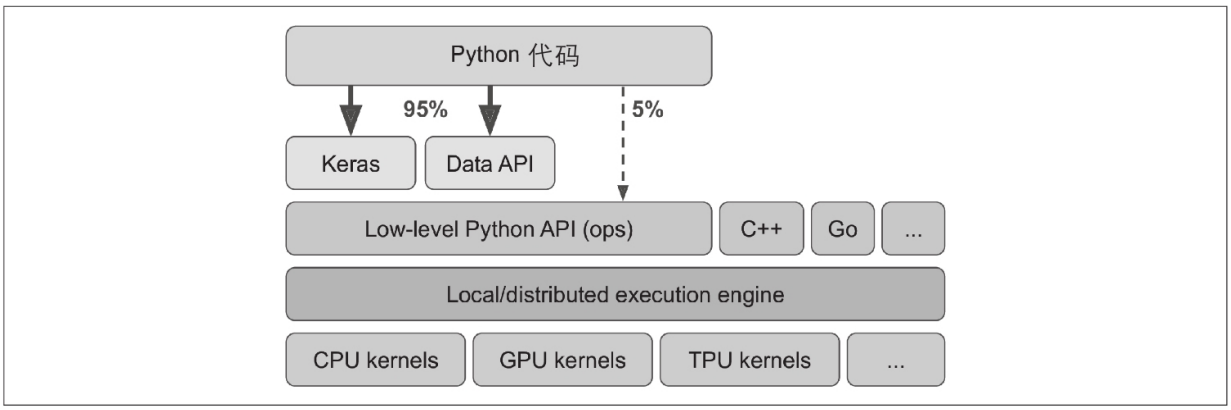


图12-2：TensorFlow的架构

TensorFlow不仅可以在Windows、Linux和macOS上运行，而且可以在移动设备（使用TensorFlow Lite）上运行，包括iOS和Android（见第19章）。如果你不想使用Python API，则可以使用C++、Java、Go和Swift API。甚至还有一个名为TensorFlow.js的JavaScript实现，你可以直接在浏览器中运行模型。

TensorFlow不仅仅是函数库，更是广泛的生态系统的核心。首先，有TensorBoard可以进行可视化（见第10章）。接下来是TensorFlow Extended（TFX），它是Google为TensorFlow项目进行生产环境而构建的一组库：它包括用于数据验证、预处理、模型分析和服务的工具（使用TF Serving，见第19章）。Google的TensorFlow Hub提供了一种轻松下载和重用预训练的神经网络的方法。你还可以在TensorFlow的模型花园中获得许多神经网络架构，其中一些已经过预先训练。查看

TensorFlow资源和https://github.com/jtoy/awesome-tensorflow，了解更多基于TensorFlow的项目。你可以在GitHub上找到数百个TensorFlow项目，通常很容易找到你想要的代码。

 越来越多的ML论文连同实现一起发布，有时甚至带有预训练的模型。查看https://paperswithcode.com/，可以轻松找到它们。

最后但并非不重要的一点是，TensorFlow拥有一支热情而乐于助人的开发人员组成的团队以及一个大型社区，致力于对其进行改进。要问技术问题，你应该使用http://stackoverflow.com/并用tensorflow和python标记你的问题。你可以通过GitHub提交错误和功能请求。有关一般讨论，请加入Google讨论组。

好的，该开始编码了！

[1] TensorFlow包含另一个名为Estimators API的深度学习API，但TensorFlow团队建议使用tf.keras。
[2] 如果需要（但可能不需要），则可以使用C++API编写自己的操作。
[3] 要了解有关TPU及其工作方式的更多信息，请访问https://homl.info/tpus。

12.2 像NumPy一样使用TensorFlow

TensorFlow的API一切都围绕张量，张量从一个操作流向另一个操作，因此命名为TensorFlow。张量非常类似NumPy的ndarray，它通常是一个多维度组，但它也可以保存标量（简单值，例如42）。当我们创建自定义成本函数、自定义指标、自定义层等时，这些张量将非常重要，因此让我们来看看如何创建和操作它们。

12.2.1 张量和操作

你可以使用tf.constant（）创建张量。例如，这是一个张量，表示具有两行三列浮点数的矩阵：

```
>>> tf.constant([[1., 2., 3.], [4., 5., 6.]]) # matrix
<tf.Tensor: id=0, shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
>>> tf.constant(42) # scalar
<tf.Tensor: id=1, shape=(), dtype=int32, numpy=42>
```

就像ndarray一样，tf.Tensor具有形状和数据类型（dtype）：

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32
```

索引的工作方式非常类似于NumPy：

```
>>> t[:, 1:]
<tf.Tensor: id=5, shape=(2, 2), dtype=float32, numpy=
```

```
array([[2., 3.],
       [5., 6.]], dtype=float32)>
>>> t[..., 1, tf.newaxis]
<tf.Tensor: id=15, shape=(2, 1), dtype=float32, numpy=
array([[2.],
       [5.]], dtype=float32)>
```

最重要的是，可以使用各种张量操作：

```
>>> t + 10
<tf.Tensor: id=18, shape=(2, 3), dtype=float32, numpy=
array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>
>>> tf.square(t)
<tf.Tensor: id=20, shape=(2, 3), dtype=float32, numpy=
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)>
>>> t @ tf.transpose(t)
<tf.Tensor: id=24, shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

请注意，t+10等效于调用tf.add（t，10）（实际上，Python调用了方法t.__add__（10），该方法仅调用tf.add（t，10））。还支持其他运算符，例如-和*。@运算符是在Python 3.5中添加的，用于矩阵乘法，等效于调用tf.matmul（）函数。

你可以找到所需的所有基本数学运算（tf.add（）、tf.multiply（）、tf.square（）、tf.exp（）、tf.sqrt（）等）以及在NumPy找到的大多数运算（例如tf.reshape（）、tf.squeeze（）、tf.tile（））。某些函数的名称与NumPy中的名称不同。例如，tf.reduce_mean（）、tf.reduce_sum（）、tf.reduce_max（）和tf.math.log（）等效于np.mean（）、np.sum（）、np.max（）和np.log（）。名称不同时，通常有充分的理由。例如，在TensorFlow中，你必须编写tf.transpose（t），不能就像在NumPy中一样只是写t.T。原因是tf.transpose（）函数与NumPy的T属性没有完全相同的功能：在TensorFlow中，使用自己的转置数据副本创建一个新的张量，而在NumPy中，t.T只是相同数据的转置视

图。类似地，tf.reduce_sum（）操作之所以这样命名，是因为其GPU内核（即GPU实现）使用的reduce算法不能保证元素添加的顺序：因为32位浮点数的精度有限，因此每次你调用此操作时，结果可能会稍有不同。tf.reduce_mean（）也是如此（当然tf.reduce_max（）是确定性的）。

 许多函数和类都有别名。例如，tf.add（）和tf.math.add（）是同一函数。这使得TensorFlow可以为最常见的操作使用简洁的名称[1]，同时保留组织良好的软件包。

Keras的底层API

Keras API在keras.backend中有自己的底层API。它包含诸如square（）、exp（）和sqrt（）等函数。在tf.keras中，这些函数通常只调用相应的TensorFlow操作。如果要编写可移植到其他Keras实现中的代码，则应使用这些Keras函数。但是它们仅涵盖TensorFlow中所有可用函数的子集，因此在本书中，我们直接使用TensorFlow操作。这是使用keras.backend的简单示例，它通常简称为K：

```
>>> from tensorflow import keras
>>> K = keras.backend
>>> K.square(K.transpose(t)) + 10
<tf.Tensor: id=39, shape=(3, 2), dtype=float32, numpy=
array([[11., 26.],
       [14., 35.],
       [19., 46.]], dtype=float32)>
```

12.2.2 张量和NumPy

张量可以与NumPy配合使用：你可以用NumPy数组创建张量，反之亦然。你甚至可以将TensorFlow操作应用于NumPy数组，将NumPy操作应用于张量：

```
>>> a = np.array([2., 4., 5.])
>>> tf.constant(a)
<tf.Tensor: id=111, shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>
>>> t.numpy() # or np.array(t)
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
>>> tf.square(a)
<tf.Tensor: id=116, shape=(3,), dtype=float64, numpy=array([4., 16., 25.])>
>>> np.square(t)
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)
```

 请注意，默认情况下NumPy使用64位精度，而TensorFlow使用32位精度。这是因为32位精度通常对于神经网络来说绰绰有余，而且运行速度更快且使用的RAM更少。因此，当你从NumPy数组创建张量时，请确保设置dtype=tf.float32。

12.2.3 类型转换

类型转换会严重影响性能，并且自动完成转换很容易被忽视。为了避免这种情况，TensorFlow不会自动执行任何类型转换：如果你对不兼容类型的张量执行操作，会引发异常。例如，你不能把浮点张量和整数张量相加，甚至不能相加32位浮点和64位浮点：

```
>>> tf.constant(2.) + tf.constant(40)
Traceback[...]InvalidArgumentError[...]expected to be a float[...]
>>> tf.constant(2.) + tf.constant(40., dtype=tf.float64)
Traceback[...]InvalidArgumentError[...]expected to be a double[...]
```

起初这可能有点烦人，但这是有充分理由的！当然当你确实需要转换类型时，可以使用tf.cast（）：

```
>>> t2 = tf.constant(40., dtype=tf.float64)
>>> tf.constant(2.0) + tf.cast(t2, tf.float32)
<tf.Tensor: id=136, shape=(), dtype=float32, numpy=42.0>
```

12.2.4 变量

到目前为止，我们看到的tf.Tensor值是不变的，无法修改它们。这意味着我们不能使用常规张量在神经网络中实现权重，因为它们需要通过反向传播进行调整。另外还可能需要随时间改变其他参数（例如动量优化器跟踪过去的梯度）。我们需要的是tf.Variable：

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
>>> v
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

tf.Variable的行为与tf.Tensor的行为非常相似：你可以使用它执行相同的操作，它在NumPy中也可以很好地发挥作用，并且对类型也很挑剔。但是也可以使用assign（）方法（或assign_add（）或assign_sub（），给变量增加或减少给定值）进行修改。你还可以通过使用单元（或切片）的assign（）方法（直接指定将不起作用）或使用scatter_update（）或scatter_nd_update（）方法来修改单个单元（或切片）：

```
v.assign(2 * v)           # => [[2., 4., 6.], [8., 10., 12.]]
v[0, 1].assign(42)        # => [[2., 42., 6.], [8., 10., 12.]]
v[:, 2].assign([0., 1.])  # => [[2., 42., 0.], [8., 10., 1.]]
v.scatter_nd_update(indices=[[0, 0], [1, 2]], updates=[100., 200.])
                          # => [[100., 42., 0.], [8., 10., 200.]]
```

 实际上，你几乎不需要手动创建变量，因为Keras提供了add_weight（）方法，我们将看到该方法会为你解决这个问题。而且模型参数通常由优化器直接更新，因此你几乎不需要手动更新变量。

12.2.5 其他数据结构

TensorFlow支持其他几种数据结构，包括以下内容（请参阅notebook中的“Tensors and Operations”或附录F以获取更多详细信息）：

稀疏张量（tf.SparseTensor）

有效地表示主要包含零的张量。tf.sparse程序包包含稀疏张量的操作。

张量数组（tf.TensorArray）

张量的列表。默认情况下，它们的大小是固定的，但可以选择动态设置。它们包含的所有张量必须具有相同的形状和数据类型。

不规则张量（tf.RaggedTensor）

表示张量列表的静态列表，其中每个张量具有相同的形状和数据类型。tf.ragged程序包包含用于不规则的张量的操作。

字符串张量

tf.string类型的常规张量。它们表示字节字符串，而不是Unicode字符串，因此如果使用Unicode字符串（常规的Python 3字符串，例如"café"）创建字符串张量，则它将自动被编码为UTF-8（例如，b"caf\xc3\xa9"）。或者，你可以使用类型为tf.int32的张量来表示Unicode字符串，其中每个项都表示一个Unicode代码点（例如[99、97、102、233]）。tf.strings包（带有s）包含用于字节字符串和Unicode字符串的操作（并将它们转换为另一个）。重要的是要注意，tf.string是原子级的，这意味着它的长度不会出现在张量的形状

中。一旦你将其转换为Unicode张量（即包含Unicode代码点的tf.int32类型的张量）后，长度就会显示在形状中。

集合

表示为常规张量（或稀疏张量）。例如，tf.constant（[[[1，2]，[3，4]]）代表两个集合{1，2}和{3，4}。通常，每个集合由张量的最后一个轴上的向量表示。

你可以使用tf.sets包中的操作来操作集。

队列

跨多个步骤存储的张量。TensorFlow提供了各种队列：简单的先进先出（FIFO）队列（FIFOQueue），可以区分某些元素优先级的队列（PriorityQueue），将其元素（RandomShuffleQueue）随机排序，通过填充（PaddingFIFOQueue）批处理具有不同形状的元素。这些类都在tf.queue包中。

有了张量-运算-变量和各种数据结构，你现在就可以自定义模型和训练算法了！

[1] 一个值得注意的例外是tf.math.log（），它很常用但没有tf.log（）别名（因为它可能与日志记录混淆了）。

12.3 定制模型和训练算法

让我们从创建一个自定义损失函数开始，这是一个简单而常见的用例。

12.3.1 自定义损失函数

假设你想训练回归模型，但是训练集有点噪声。当然，你首先尝试通过删除或修复异常值来清理数据集，但事实证明这还不够。数据集仍然有噪声。你应该使用哪个损失函数？均方误差可能会对大误差惩罚太多而导致模型不精确。平均绝对误差不会对异常值惩罚太多，但是训练可能需要一段时间才能收敛，并且训练后的模型可能不太精确。这可能是使用Huber损失（在第10章中介绍）而不是旧的MSE的好时机。目前，Huber损失不是官方Keras API的一部分，但可以在tf.keras中使用（使用keras.losses.Huber类的实例）。但是，让我们假装它不存在：实现它就是小菜一碟！只需创建一个将标签和预测作为参数的函数，然后使用TensorFlow操作来计算每个实例的损失：

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) < 1
    squared_loss = tf.square(error) / 2
    linear_loss  = tf.abs(error) - 0.5
    return tf.where(is_small_error, squared_loss, linear_loss)
```

 为了获得更好的性能，你应使用向量化实现，如本例所示。此外，如果你想从TensorFlow的图形功能中受益，则应仅使用TensorFlow操作。

最好返回每个实例包含一个损失的张量，而不是返回实例的平均损失。这样，Keras可以根据要求使用类别权重或样本权重（见第10

章）。

现在，你可以在编译Keras模型时使用此损失，然后训练模型：

```
model.compile(loss=huber_fn, optimizer="nadam")
model.fit(X_train, y_train, [...])
```

就是这样！对于训练期间的每个批次，Keras调用huber_fn（）函数来计算损失并使用它执行“梯度下降”步骤。此外，它跟踪从轮次开始以来的总损失，并显示平均损失。但是当你保存模型时，这种自定义损失会怎样？

12.3.2 保存和加载包含自定义组件的模型

保存包含自定义损失函数的模型效果很好，因为Keras会保存函数的名称。每次加载时，都需要提供一个字典，将函数名称映射到实际函数。一般而言，当加载包含自定义对象的模型时，需要将名称映射到对象：

```
model = keras.models.load_model("my_model_with_a_custom_loss.h5",
                                custom_objects={"huber_fn": huber_fn})
```

在当前的实现中，在-1和1之间的任何误差都被认为是“很小”。但是，如果你想要一个不同的阈值怎么办？一种解决方案是创建一个函数，该函数创建已配置的损失函数：

```
def create_huber(threshold=1.0):
    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss  = threshold * tf.abs(error) - threshold**2 / 2
```

```
        return tf.where(is_small_error, squared_loss, linear_loss)
    return huber_fn

model.compile(loss=create_huber(2.0), optimizer="nadam")
```

不幸的是，当你保存模型时，阈值不会被保存。这意味着在加载模型时必须指定阈值（请注意，使用的名称是"huber_fn"，这是你为Keras命名的函数的名称，而不是创建函数时的名称）：

```
model = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
                                custom_objects={"huber_fn": create_huber(2.0)})
```

你可以通过创建keras.losses.Loss类的子类，然后实现其get_config（）方法来解决此问题：

```
class HuberLoss(keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)
    def call(self, y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < self.threshold
        squared_loss = tf.square(error) / 2
        linear_loss  = self.threshold * tf.abs(error) - self.threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

 Keras API当前仅指定如何使用子类定义层、模型、回调和正则化。如果使用子类构建其他组件（例如损失、性能度量、初始化或约束），则它们可能无法移植到其他Keras实现中。Keras API可能也会更新来指定所有这些组件的子类。

让我们看一下这段代码：

• 构造函数接受**kwargs并将它们传递给父类构造函数，该父类构造函数处理标准超参数：损失的name和用于聚合单个实例损失的reduction算法。默认情况下，它是"s um_over_batch_size"，这意味着损失将是实例损失的总和，由样本权重（如果有）加权，再除以批量大小（而不是权重之和，因此不是权重的平均）[1]。其他可能的值为"sum"和"none"。

• call（）方法获取标签和预测，计算所有实例损失，然后将其返回。

• get_config（）方法返回一个字典，将每个超参数名称映射到其值。它首先调用父类的get_config（）方法，然后将新的超参数添加到此字典中（请注意，在Python 3.5中添加了方便的{**x}语法）。

然后，你可以在编译模型时使用此类的任何实例：

```
model.compile(loss=HuberLoss(2.), optimizer="nadam")
```

当你保存模型时，阈值会同时一起保存。在加载模型时，只需要将类名映射到类本身即可：

```
model = keras.models.load_model("my_model_with_a_custom_loss_class.h5",
                                custom_objects={"HuberLoss": HuberLoss})
```

当你保存模型时，Keras会调用损失实例的get_config（）方法，并将配置以JSON格式保存到HDF5文件中。加载模型时，它在HuberLoss类上调用from_config（）类方法：此方法由基类（Loss）实现，并创建该类的实例，并将**config传递给构造函数。

这就是你需要为定制损失做的！不是很难，不是吗？定制激活函数、初始化、正则化和约束也一样简单。让我们现在来看这些。

12.3.3 自定义激活函数、初始化、正则化和约束

大多数Keras功能，例如损失、正则化、约束、初始化、度量、激活函数、层甚至完整模型，都可以以几乎相同的方式进行自定义。在大多数情况下，你只需要编写带有适当输入和输出的简单函数即可。以下是自定义激活函数（等同于keras.activations.softplus（）或tf.nn.softplus（））、自定义Glorot初始化（等同于kera s.initializers.glorot_normal（））、自定义l1正则化（等同于keras.regularizers.l1（0.01）），以及确保权重均为正的自定义约束（等同于kera s.contraints.nonneg（）或tf.nn.relu（））的例子：

```
def my_softplus(z): # return value is just tf.nn.softplus(z)
    return tf.math.log(tf.exp(z) + 1.0)

def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)

def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))

def my_positive_weights(weights): # return value is just tf.nn.relu(weights)
    return tf.where(weights < 0., tf.zeros_like(weights), weights)
```

如你所见，参数取决于自定义函数的类型。然后可以正常使用这些自定义函数。例如：

```
layer = keras.layers.Dense(30, activation=my_softplus,
                           kernel_initializer=my_glorot_initializer,
                           kernel_regularizer=my_l1_regularizer,
                           kernel_constraint=my_positive_weights)
```

激活函数将应用于此Dense层的输出，其结果将传递到下一层。层的权重将使用初始化程序返回的值进行初始化。在每个训练步骤中，权重将传递给正则化函数以计算正则化损失，并将其添加到主要损失中以得到用于训练的最终损失。最后在每个训练步骤之后将调用约束函数，并将层的权重替换为约束权重。

如果函数具有需要与模型一起保存的超参数，你需要继承适当的类，例如keras.regularizers.Regularizer，keras.constraints.Constraint，keras.initializers.Initializer或keras.layers.Layer（适用于任何层，包括激活函数）。就像我们为自定义损失所做的一样，这是一个用于1正则化的简单类，它保存了其factor超参数（这一次我们不需要调用父类构造函数或get_config（）方法，因为它们不是由父类定义的）：

```
class MyL1Regularizer(keras.regularizers.Regularizer):
    def __init__(self, factor):
        self.factor = factor
    def __call__(self, weights):
        return tf.reduce_sum(tf.abs(self.factor * weights))
    def get_config(self):
        return {"factor": self.factor}
```

请注意，你必须为损失、层（包括激活函数）和模型实现call（）方法，或者为正则化、初始化和约束实现__call__（）方法。对于指标，情况有些不同，我们现在来看看。

12.3.4 自定义指标

损失和指标在概念上不是一回事：损失（例如交叉熵）被梯度下降用来训练模型，因此它们必须是可微的（至少是在求值的地方），并且梯度在任何地方都不应为0。另外，如果人类不容易解释它们也没有问题。相反，指标（例如准确率）用于评估模型，它们必须更容易被解释，并且可以是不可微的或在各处具有0梯度。

也就是说，在大多数情况下，定义一个自定义指标函数与定义一个自定义损失函数完全相同。实际上，我们甚至可以将之前创建的Huber损失函数用作指标[2]。它可以很好地工作（持久性也可以以相同的方式工作，在这种情况下，仅保存函数名"huber_fn"）：

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

对于训练期间的每一批次，Keras都会计算该指标并跟踪自轮次开始以来的均值。大多数时候，这是你想要的，但不总是！例如，考虑一个二元分类器的精度。正如我们在第3章中看到的那样，精度是真正的数量除以正预测的数量（包括真正和假正）。假设该模型在第一批次中做出了5个正预测，其中4个是正确的，即80%的精度。然后假设该模型在第二批次中做出了3个正预测，但它们都是不正确的，即第二批次的精度为0%。如果仅计算这两个精度的均值，则可以得到40%。但是请稍等一下，这不是模型在这两个批次上的精度！实际上，在8个正预测（5+3）中，总共有4个真正（4+0），因此总体精度为50%，而不是40%。我们需要的是一个对象，该对象可以跟踪真正的数量和假正的数量，并且可以在请求时计算其比率。这正是keras.metrics.Precision类要做的事情：

```
>>> precision = keras.metrics.Precision()
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])
<tf.Tensor: id=581729, shape=(), dtype=float32, numpy=0.8>
>>> precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])
<tf.Tensor: id=581780, shape=(), dtype=float32, numpy=0.5>
```

在此示例中，我们创建了一个Precision对象，然后将其用作函数，将第一个批次的标签和预测以及第二个批次的标签和预测传递给它（请注意，我们也可以传递样本权重）。我们使用与刚才讨论的示例相同的真正和假正判断次数。第一批次后，它返回80%的精度；然后在第

二批次之后返回50%（这是到目前为止的总体精度，而不是第二批次的精度）。这被称为流式指标（或状态指标），因为它是逐批次更新的。

在任何时候，我们都可以调用result（）方法来获取指标的当前值。我们还可以使用variables属性查看其变量（跟踪真正和假正的数量），并可以使用reset_states（）方法重置这些变量：

```
>>> precision.result()
<tf.Tensor: id=581794, shape=(), dtype=float32, numpy=0.5>
>>> precision.variables
[<tf.Variable 'true_positives:0' [...] numpy=array([4.], dtype=float32)>,
 <tf.Variable 'false_positives:0' [...] numpy=array([4.], dtype=float32)>]
>>> precision.reset_states() # both variables get reset to 0.0
```

如果需要创建这样的流式度量，创建keras.metrics.Metric类的子类。这是一个跟踪Huber总损失的简单示例以及到目前为止看到的实例数量。当要求得到结果时，它返回比率，这就是平均Huber损失：

```
class HuberMetric(keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs) # handles base args (e.g., dtype)
        self.threshold = threshold
        self.huber_fn = create_huber(threshold)
        self.total = self.add_weight("total", initializer="zeros")
        self.count = self.add_weight("count", initializer="zeros")
    def update_state(self, y_true, y_pred, sample_weight=None):
        metric = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(metric))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))
    def result(self):
        return self.total / self.count
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

让我们看一下这段代码[3]：

• 构造函数使用add wei_ght（）方法创建用于跟踪多个批次的度量状态所需的变量，在这种情况下，该变量包括所有Huber损失的总和

（总计）以及到目前为止看到的实例数（计数）。如果愿意，你可以手动创建变量。Keras会跟踪任何设置为属性的tf.Variable（更一般而言，是任何“可跟踪”的对象，例如层或模型）。

• 当你使用此类的实例作为函数时（像我们对Precision对象所做的那样），将调用update_state（）方法。给定一个批次的标签和预测值（以及采样权重，但这个示例中我们忽略它们），它会更新变量。

• result（）方法计算并返回最终结果，在这种情况下为所有实例的平均Huber度量。当你使用度量作为函数时，首先调用update_state（）方法，然后调用result（）方法，并返回其输出。

• 我们还实现了get_config（）方法来确保threshold与模型一起被保存。

• reset_states（）方法的默认实现将所有变量重置为0.0（但是你可以根据需要覆盖它）。

 Keras会处理变量的持久性，无须采取任何措施。

当你使用简单的函数定义指标时，Keras会自动为每个批次调用该指标，它会跟踪每个轮次的均值，就像我们手动进行的那样。因此HuberMetric类的唯一好处是保存threshdd。但是，某些指标（如精度）不能简单地按批次平均：在这些情况下，除了实现流式指标之外，别无选择。

现在我们已经建立了流式指标，那建立自定义层就像在公园里散步一样简单！

12.3.5 自定义层

你可能偶尔会想要构建一个包含独特层的架构，而TensorFlow没有为其提供默认实现。在这种情况下，你将需要创建一个自定义层。或者你可能只是想构建一个重复的架构，其中包含重复多次的相同层块，因此将每个层块视为一个层会很方便。例如，如果模型是A，B，C，A，B，C，A，B，C层的序列，则你可能想定义一个包含A，B，C层的自定义层D你的模型将简化为D，D，D。让我们看看如何构建自定义层。

首先，某些层没有权重，例如keras.layers.Flatten或keras.layers.ReLU。如果要创建不带任何权重的自定义层，最简单的选择是编写一个函数并将其包装在keras.layers.Lambda层中。例如，以下层将对它的输入应用指数函数：

```
exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
```

然后，可以像使用顺序API、函数式API或子类API等其他任何层一样使用此自定义层。你也可以将其用作激活函数（或者可以使用activation=tf.exp、activation=keras.activations.exponential或仅使用activation="exponential"）。当要预测的值具有非常不同的标度（例如0.001、10、1000）时，有时会在回归模型的输出层中使用指数层。

正如你现在可能已经猜到的，要构建自定义的有状态层（即具有权重的层），你需要创建keras.layers.Layer类的子类。例如以下类实现了Dense层的简化版本：

```
class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
            name="kernel", shape=[batch_input_shape[-1], self.units],
```

```
            initializer="glorot_normal")
        self.bias = self.add_weight(
            name="bias", shape=[self.units], initializer="zeros")
        super().build(batch_input_shape) # must be at the end

    def call(self, X):
        return self.activation(X @ self.kernel + self.bias)

    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1] + [self.units])

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units,
                "activation": keras.activations.serialize(self.activation)}
```

让我们看一下这段代码：

• 构造函数将所有超参数用作参数（在此示例中为units和activation），重要的是它还接受**kwargs参数。它调用父类构造函数，并将其传递给kwargs：这负责处理标准参数，例如input_shape、trainable和name。然后，它将超参数保存为属性，使用keras.activations.get（）函数将激活参数转换为适当的激活函数（它接受函数、标准字符串（如"relu"或"selu"，或None））[4]。

• build（）方法的作用是通过为每个权重调用add_weight（）方法来创建层的变量。首次使用该层时，将调用build（）方法。在这一点上，Keras知道该层的输入形状，并将其传递给build（）方法[5]，这对于创建某些权重通常是必需的。例如，我们需要知道上一层中神经元的数量，以便创建连接权重矩阵（即"Kernel"）：这对应于输入的最后维度的大小。在bu ild（）方法的最后（并且仅在最后），你必须调用父类的build（）方法：这告诉Keras这一层被构建了（它只是设置了s elf.built=True）。

• call（）方法执行所需的操作。在这种情况下，我们计算输入X与层内核的矩阵乘积，添加偏置向量，并对结果应用激活函数，从而获得层的输出。

• compute_output_sha pe（）方法仅返回该层输出的形状。在这种情况下，它的形状与输入的形状相同，只是最后一个维度被替换为该层中神经元的数量。请注意，在tf.keras中，形状是tf.TensorShape类的实例，你可以使用as_list（）将其转换为Python列表。

• get_config（）方法就像以前的自定义类中一样。请注意我们通过调用kera s.activations.serialize（）保存激活函数的完整配置。

现在，你可以像其他任何层一样使用MyDense层！

 通常你可以省略compute_output_sha pe（）方法，因为tf.keras会自动推断出输出形状，除非层是动态的（我们将很快看到）。在其他Keras实现中，此方法是必需的或者其默认实现是假设输出形状与输入形状相同。

要创建一个具有多个输入（例如Concatenate）的层，call（）方法的参数应该是包含所有输入的元组，而同样，compute_output_shape（）方法的参数应该是一个包含每个输入的批处理形状的元组。要创建具有多个输出的层，call（）方法应返回输出列表，而compute_output_shape（）应返回批处理输出形状的列表（每个输出一个）。例如以下这个层需要两个输入并返回三个输出：

```
class MyMultiLayer(keras.layers.Layer):
    def call(self, X):
        X1, X2 = X
        return [X1 + X2, X1 * X2, X1 / X2]

    def compute_output_shape(self, batch_input_shape):
        b1, b2 = batch_input_shape
        return [b1, b1, b1] # should probably handle broadcasting rules
```

现在可以像其他任何层一样使用此层，但是当然只能使用函数式和子类API，而不能使用顺序API（仅接受具有一个输入和一个输出的

层）。

如果你的层在训练期间和测试期间需要具有不同的行为（例如如果使用Dropout或BatchNormalization层），则必须将训练参数添加到call（）方法并使用此参数来决定要做什么。让我们创建一个在训练期间（用于正则化）添加高斯噪声但在测试期间不执行任何操作的层（Keras具有相同功能的层：kera s.layers.GaussianNoise）：

```
class MyGaussianNoise(keras.layers.Layer):
    def __init__(self, stddev, **kwargs):
        super().__init__(**kwargs)
        self.stddev = stddev

    def call(self, X, training=None):
        if training:
            noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
            return X + noise
        else:
            return X

    def compute_output_shape(self, batch_input_shape):
        return batch_input_shape
```

这样，你现在就可以构建所需的任何自定义层了！现在让我们创建自定义模型。

12.3.6 自定义模型

我们已经在第10章中讨论过创建自定义模型类，在讨论子类API[6]时。它很简单：继承keras.Model类，在构造函数中创建层和变量，并实现call（）方法来执行你希望模型执行的任何操作。假设你要构建如图12-3所示的模型。

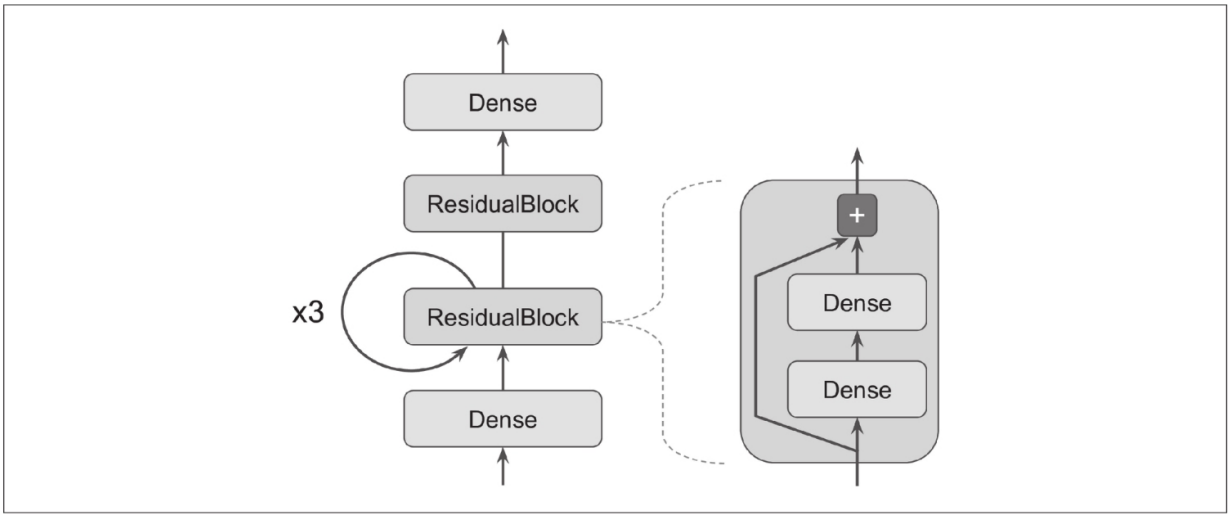


图12-3：自定义模型示例：具有自定义ResidualBlock层（包含跳过连接）的任意模型

输入经过第一个密集层，然后经过由两个密集层组成的残差块并执行加法运算（正如我们将在第14章中看到的那样，残差块将其输入加到其输出中），然后经过相同的残差块3次或者更多次，然后通过第二个残差块，最终结果通过密集输出层。注意，该模型没有多大意义，这只是一个示例，它说明了你可以轻松构建所需的任何模型，即使是包含循环和跳过连接的模型。要实现此模型，最好首先创建一个ResidualBlock层，因为我们将创建几个相同的块（并且我们可能想在另一个模型中重用它）：

```
class ResidualBlock(keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(n_neurons, activation="elu",
                                          kernel_initializer="he_normal")
                       for _ in range(n_layers)]

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        return inputs + Z
```

该层有点特殊，因为它包含其他层。这由Keras透明地处理：它会自动检测到隐藏的属性，该属性包含可跟踪的对象（在这个示例中是层），因此它们的变量会自动添加到该层的变量列表中。这个类的其余部分不言自明。接下来让我们使用子类API定义模型本身：

```
class ResidualRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(30, activation="elu",
                                          kernel_initializer="he_normal")
        self.block1 = ResidualBlock(2, 30)
        self.block2 = ResidualBlock(2, 30)
        self.out = keras.layers.Dense(output_dim)

    def call(self, inputs):
        Z = self.hidden1(inputs)
        for _ in range(1 + 3):
            Z = self.block1(Z)
        Z = self.block2(Z)
        return self.out(Z)
```

我们在构造函数中创建层，并在call（）方法中使用它们。然后就可以像使用任何其他模型一样使用此模型（对其进行编译、拟合、评估和预测）。如果你还希望能够使用save（）方法保存模型并使用keras.models.load_model（）函数加载模型，则必须在两个ResidualBlock类和ResidualRegressor类中都实现get_config（）方法（就像我们之前所做的那样）。另外，你可以使用save_weights（）和load_weights（）方法保存和加载权重。

Model类是Layer类的子类，因此可以像定义层一样定义和使用模型。但是模型具有一些额外的功能，包括其compile（）、fit（）、evaluate（）和predict（）方法（以及一些变体）以及get_layers（）方法（可以按名称或按索引返回任何模型的层）和save（）方法（支持keras.models.load_model（）和keras.models.clone_model（））。如果模型提供的功能比层更多，为什么不将每个层都定义为模型？从技术上讲可以，但是通常可以轻松地将模型的内部组件（即层或可重复使用的层块）与模型本身（即要训练的对象）区分开来。前者应继承Layer类，而后者应继承Model类。

 这样你可以自然而简洁地使用顺序API、函数式API、子类API或是它们的组合，来构建几乎所有你在论文中能找到的模型。“几乎”任何型号吗？是的，我们仍然需要考虑一些事情：首先如何基于模型内部定义损失或指标；其次，如何构建自定义训练循环。

12.3.7 基于模型内部的损失和指标

我们之前定义的自定义损失和指标均基于标签和预测（以及可选的样本权重）。有时候，你可能要根据模型的其他部分来定义损失，例如权重或隐藏层的激活。这对于进行正则化或监视模型的某些内部方面可能很有用。

要基于模型内部定义自定义损失，根据所需模型的任何部分进行计算，然后将结果传递给add_loss（）方法。例如，让我们构建一个自定义回归MLP模型，该模型由5个隐藏层和一个输出层的堆栈组成。此自定义模型还将在上部隐藏层的顶部有辅助输出。与该辅助输出相关的损失称为重建损失（见第17章）：它是重建与输入之间的均方差。通过将这种重建损失添加到主要损失中，我们鼓励模型通过隐藏层保留尽可能多的信息，即使对回归任务本身没有直接用处的信息。实际中，这种损失有时会提高泛化性（这是正则化损失）。以下是带有自定义重建损失的自定义模型的代码：

```
class ReconstructingRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
                                          kernel_initializer="lecun_normal")
                       for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)

    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layers.Dense(n_inputs)
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
```

```
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        return self.out(Z)
```

让我们看一下这段代码：

• 构造函数创建具有5个密集隐藏层和一个密集输出层的DNN。

• build（）方法创建一个额外的密集层，该层用于重建模型的输入。必须在此处创建它，因为它的单元数必须等于输入数，并且在调用build（）方法之前，此数是未知的。

• call（）方法处理所有5个隐藏层的输入，然后将结果传递到重建层，从而产生重构。

• 然后call（）方法计算重建损失（重建与输入之间的均方差），并使用add_loss（）方法[7]将其添加到模型的损失列表中。请注意，我们通过将其乘以0.05（这是你可以调整的超参数）按比例缩小了重建。这确保了重建损失不会在主要损失中占大部分。

• 最后，call（）方法将隐藏层的输出传递到输出层并返回其输出。

同样，你可以通过以所需的任何方式计算来添加基于模型内部的自定义指标，只要结果是指标对象的输出即可。例如，你可以在构造函数中创建keras.metrics.Mean对象，然后在call（）方法中调用它，将recon_loss传递给它，最后通过调用模型的add_metric（）方法将其添加到模型中。这样当你训练模型时，Keras会同时显示每个轮次的平均损失（损失是主要损失加上0.05倍的重建损失）和每个轮次的平均重建误差。两者在训练期间都会下降：

```
Epoch 1/5
11610/11610 [==============] [...] loss: 4.3092 - reconstruction_error: 1.7360
Epoch 2/5
11610/11610 [==============] [...] loss: 1.1232 - reconstruction_error: 0.8964
[...]
```

在超过99%的情况下，到目前为止，我们讨论的所有内容都足以实现你要构建的任何模型，即使具有复杂的架构、损失和指标。但是在极少数情况下，你可能需要自定义训练循环，在那之前，我们需要研究如何在TensorFlow中自动计算梯度。

12.3.8 使用自动微分计算梯度

为了理解如何使用自动微分（见第10章和附录D）来自动计算梯度，让我们考虑一个简单的函数：

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2
```

如果你了解微积分，则可以通过分析发现该函数关于w1的偏导数为6*w1+2*w2。你还可以发现其关于w2的偏导数是2*w1。例如，在点（w1，w2）=（5，3）处，这些偏导数分别等于36和10，因此此点的梯度向量为（36，10）。但是如果这是一个神经网络，则该函数将更加复杂，通常具有数以万计的参数，并且用手工分析找到偏导数将几乎是不可能的任务。一种解决方案是通过在调整相应参数时测量函数输出的变化来计算每个偏导数的近似值：

```
>>> w1, w2 = 5, 3
>>> eps = 1e-6
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps
36.000003007075065
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps
10.000000003174137
```

看起来不错！这工作得很好并且易于实现，但这只是一个近似值，重要的是每个参数至少要调用一次f（）（不是两次，因为我们只计算一次f（w1，w2））。每个参数至少需要调用f（）一次，这种方法对于大型神经网络来说很棘手。因此，我们应该使用自动微分。TensorFlow使这个变得非常简单：

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

我们首先定义两个变量w1和w2，然后创建一个tf.GradientTape上下文，该上下文将自动记录涉及变量的每个操作，最后我们要求该tape针对两个变量[w1，w2]计算结果z的梯度。让我们看一下TensorFlow计算的梯度：

```
>>> gradients
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: id=828229, shape=(), dtype=float32, numpy=10.0>]
```

完美！结果不仅是准确的（精度仅受浮点误差限制），而且无论有多少变量，gradient（）方法都只经历一次已经记录的计算（以相反的顺序），因此它非常有效。就像魔术一样！

 为了节省内存，仅将严格的最小值放入tf.Gra dientTape（）块中。或者通过在tf.GradientTape（）块内使用tape.stop_recording（）块来暂停记录。

调用tape的gradient（）方法后，tape会立即被自动擦除，因此如果你尝试两次调用gradient（），则会出现异常：

```
with tf.GradientTape() as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
dz_dw2 = tape.gradient(z, w2) # RuntimeError!
```

如果你需要多次调用gradient（），则必须使该tape具有持久性，并在每次使用完该tape后将其删除以释放资源[8]：

```
with tf.GradientTape(persistent=True) as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
dz_dw2 = tape.gradient(z, w2) # => tensor 10.0, works fine now!
del tape
```

默认情况下，tape仅跟踪涉及变量的操作，因此如果你尝试针对变量以外的任何其他变量计算z的梯度，则结果将为None：

```
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # returns [None, None]
```

但是，你可以强制tape观察你喜欢的任何张量，来记录涉及它们的所有操作。然后你可以针对这些张量计算梯度，就好像它们是变量一样：

```
with tf.GradientTape() as tape:
    tape.watch(c1)
    tape.watch(c2)
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # returns [tensor 36., tensor 10.]
```

在某些情况下，这可能很有用，如果你要实现正则化损失，从而在输入变化不大时惩罚那些变化很大的激活：损失将基于激活相对于输入的梯度而定。由于输入不是变量，因此你需要告诉tape观察它们。

大多数情况下，一个梯度tape是用来计算单个值（通常是损失）相对于一组值（通常是模型参数）的梯度。这就是反向模式自动微分有用的地方，因为它只需执行一次正向传播和一次反向传播即可一次获得所有梯度。如果你尝试计算向量的梯度，例如包含多个损失的向量，那么TensorFlow将计算向量和的梯度。因此如果你需要获取单独的梯度（例如每种损失相对于模型参数的梯度），则必须调用tape的jacobian（）方法：它对向量中的每个损失执行一次反向模式自动微分（默认情况下全部并行）。它甚至可以计算二阶偏导数（Hessian，即偏导数的偏导数），但实际上很少需要（请参阅notebook的“Computing Gradients with Autodiff”部分）。

在某些情况下，你可能希望阻止梯度在神经网络的某些部分反向传播。为此必须使用tf.stop_gradient（）函数。该函数在前向传递过程中返回其输入（如tf.identity（）），但在反向传播期间不让梯度通过（它的作用类似于常量）：

```
def f(w1, w2):
    return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)

with tf.GradientTape() as tape:
    z = f(w1, w2) # same result as without stop_gradient()

gradients = tape.gradient(z, [w1, w2]) # => returns [tensor 30., None]
```

最后在计算梯度时，你有时可能会遇到一些数值问题。例如，如果你用大数值输入来计算my_softplus（）函数的梯度，则结果为NaN：

```
>>> x = tf.Variable([100.])
>>> with tf.GradientTape() as tape:
...     z = my_softplus(x)
...
>>> tape.gradient(z, [x])
<tf.Tensor: [...] numpy=array([nan], dtype=float32)>
```

这是因为使用自动微分计算此函数的梯度会导致一些数值上的困难：由于浮点精度误差，自动微分最终导致计算无穷除以无穷（返回NaN）。幸运的是，我们可以分析发现softplus函数的导数为1/（1+1/exp（x）），在数值上是稳定的。接下来我们可以告诉TensorFlow在计算my_softplus（）函数的梯度时使用@tf.custom_gradient来修饰它并使它既返回其正常输出又返回计算导数的函数（注意，它将接收到目前为止反向传播的梯度，直到softplus函数。根据链式规则，我们应该将它们乘以该函数的梯度）：

```
@tf.custom_gradient
def my_better_softplus(z):
    exp = tf.exp(z)
    def my_softplus_gradients(grad):
        return grad / (1 + 1 / exp)
    return tf.math.log(exp + 1), my_softplus_gradients
```

现在当我们计算my_better_softplus（）函数的梯度时，即使对于较大的输入值，我们也可以获得正确的结果（但是由于指数的关系，主要输出仍然会爆炸。一种解决方法是使用tf.where（）在输入较大时返回输入）。

恭喜你！现在你可以计算任何函数的梯度（前提是你想计算的那个点是可微的），甚至在需要时阻止反向传播，并编写自己的梯度函数！这可能比你需要的更有灵活性，即使你构建自己的自定义训练循环，正如我们下面要看到的。

12.3.9　自定义训练循环

在极少数情况下，fit（）方法可能不够灵活而无法满足你的需要。例如我们在第10章中讨论的Wide&Deep论文使用了两种不同的优化器：一种用于宽路径，另一种用于深路径。由于fit（）方法只使用一个优化器（编译模型时指定的优化器），因此实现该论文需要编写你自己的自定义循环。

你可能还想编写自定义训练循环，只是为了让自己更有信心，确信它们按照你的意图进行操作（也许你不确定fit（）方法的某些细节）。有时将所有内容都明确显示出来会更安全。但是请记住，编写自定义训练循环会使你的代码更长、更容易出错并且更难以维护。

 除非你真的需要额外的灵活性，否则应该更倾向使用fit（）方法，而不是实现你自己的训练循环，尤其是在团队合作中。

首先，让我们建立一个简单的模型。无须编译它，因为我们将手动处理训练循环：

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="elu", kernel_initializer="he_normal",
                       kernel_regularizer=l2_reg),
    keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

接下来，让我们创建一个小函数，从训练集中随机采样一批实例（在第13章中，我们将讨论Data API，它提供了更好的选择）：

```
def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]
```

让我们定义一个函数，显示训练状态，包括步数、步总数、从轮次开始以来的平均损失（即我们将使用Mean指标来计算），和其他指标：

```
def print_status_bar(iteration, total, loss, metrics=None):
    metrics = " - ".join(["{}: {:.4f}".format(m.name, m.result())
                          for m in [loss] + (metrics or [])])
    end = "" if iteration < total else "\n"
    print("\r{}/{} - ".format(iteration, total) + metrics,
          end=end)
```

除非你不熟悉Python字符串的格式设置，否则这段代码是不言自明的：{：.4f}会格式化小数点后四位数字的浮点数，并使用\r（回车）和end=""确保状态栏始终打印在同一行上。在notebook中，print_status_bar（）函数包括一个进度条，但是你也可以使用方便的tqdm库。

首先我们需要定义一些超参数，然后选择优化器、损失函数和指标（在此示例中是MAE）：

```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]
```

准备构建自定义循环！

```
for epoch in range(1, n_epochs + 1):
    print("Epoch {}/{}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
```

```
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        mean_loss(loss)
        for metric in metrics:
            metric(y_batch, y_pred)
        print_status_bar(step * batch_size, len(y_train), mean_loss, metrics)
    print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
    for metric in [mean_loss] + metrics:
        metric.reset_states()
```

这段代码中做了很多事情，所以让我们来看一下：

• 我们创建了两个嵌套循环：一个用于轮次，另一个用于轮次内的批处理。

• 然后从训练集中抽取一个随机批次。

• 在tf.GradientTape（）块中，我们对一个批次进行了预测（使用模型作为函数），并计算了损失：它等于主损失加其他损失（在此模型中，每层都有一个正则化损失）。由于mean_squared_error（）函数每个实例返回一个损失，因此我们使用tf.reduce_mean（）计算批次中的均值（如果要对每个实例应用不同的权重，则可以在这里进行操作）。正则化损失已经归约到单个标量，因此我们只需要对它们进行求和（使用tf.add_n（）即可对具有相同形状和数据类型的多个张量求和）。

• 接下来，我们要求tape针对每个可训练变量（不是所有变量！）计算损失的梯度，然后用优化器来执行“梯度下降”步骤。

• 然后，我们更新平均损失和指标（在当前轮次内），并显示状态栏。

• 在每个轮次结束时，我们再次显示状态栏以使其看起来完整[9]并打印换行符，然后重置平均损失和指标的状态。

如果你设置优化器的超参数clipnorm或clipvalue，它会为你解决这一问题。如果要对梯度应用任何其他变换，只需在调用apply_gradients（）方法之前进行即可。如果你要对模型添加权重约束（例如在创建层时设置kernel_constraint或bias_constraint），则应更新训练循环以在apply_gradients（）之后应用这些约束：

```
for variable in model.variables:
    if variable.constraint is not None:
        variable.assign(variable.constraint(variable))
```

最重要的是，此训练循环不会处理在训练期间和测试期间行为不同的层（例如Batch Normalization或Dropout）。要处理这些问题，你需要使用training=True调用模型，确保将其传播到需要它的每个层。

如你所见，有很多事情需要自己做，这很容易出错。但好处你可以完全控制，这也是你自己的要求。

既然你知道如何自定义模型的任何部分[10]和训练算法，那么我们来看看如何使用TensorFlow的自动图生成功能：它可以大大加快自定义代码的速度，并且还可以移植到TensorFlow支持的任何平台上（见第19章）。

[1] 使用加权平均值不是一个好主意：如果你这样做了，那么根据每个批次的总权重，两个权重相同但批次不同的实例会对训练产生不同的影响。
[2] 但是，很少使用Huber损失作为指标（最好使用MAE或MSE）。
[3] 这个类仅仅用来说明。一个更简单、更好的实现是继承keras.metrics.Mean类。有关示例，请参阅notebook的“Streaming metrics”部分。
[4] 此函数特定于tf.keras。你可以改用keras.layers.Activation。

[5] Keras API调用此参数input_shape，但由于它还包含批处理维度，因此我更喜欢将其称为batch_input_shape。compute_output_shape（）也相同。

[6] 名称“Subclassing API”通常仅指通过子类创建自定义模型，如本章所述，尽管可以通过子类创建许多其他内容。

[7] 你还可以在模型内部的任何层上调用add_loss（），因为模型以递归方式收集所有层的损失。

[8] 如果tape超出了范围，例如当使用它的函数返回时，Python的垃圾收集器将为你删除它。

[9] 事实是，我们没有处理训练集中的每个实例，因为我们是随机抽样的：一些实例被多次处理，而另一些则根本没有被处理。同样，如果训练集大小不是批处理大小的倍数，我们会错过一些实例。在实践中这是可以的。

[10] 除了优化器，很少有人定制其他部分。有关示例，请参见notebook中的“Lustom Optimizers”部分。

12.4 TensorFlow函数和图

在TensorFlow 1中，图是不可避免的（随之而来的是复杂性），因为它们是TensorFlow API的核心部分。在TensorFlow 2中，它们仍在那儿，但是不那么重要了，而且使用起来要简单得多。为了说明其简单性，让我们从计算其输入的立方的简单函数开始：

```
def cube(x):
    return x ** 3
```

我们显然可以使用Python值（例如一个整数或浮点数）来调用此函数，也可以使用张量来调用它：

```
>>> cube(2)
8
>>> cube(tf.constant(2.0))
<tf.Tensor: id=18634148, shape=(), dtype=float32, numpy=8.0>
```

现在，让我们使用tf.function（）将此Python函数转换为TensorFlow函数：

```
>>> tf_cube = tf.function(cube)
>>> tf_cube
<tensorflow.python.eager.def_function.Function at 0x1546fc080>
```

然后可以完全像原Python函数一样使用此TF函数，并且它会返回相同的结果（但作为张量）：

```
>>> tf_cube(2)
<tf.Tensor: id=18634201, shape=(), dtype=int32, numpy=8>
>>> tf_cube(tf.constant(2.0))
<tf.Tensor: id=18634211, shape=(), dtype=float32, numpy=8.0>
```

在后台，tf.function（）分析了cube（）函数执行的计算，并生成等效的计算图！如你所见，它相当容易（我们将很快看到它的工作原理）。另外我们可以使用tf.function作为修饰器，这在实际中更常见：

```
@tf.function
def tf_cube(x):
    return x ** 3
```

如果需要，可以通过TF函数的python_function属性使用原Python函数：

```
>>> tf_cube.python_function(2)
8
```

TensorFlow可以优化计算图，修剪未使用的节点，简化表达式（例如1+2将替换为3），等等。准备好优化的图后，TF函数会以适当的顺序（并在可能时并行执行）有效地执行图中的操作。因此TF函数通常比原始的Python函数运行得更快，尤其是在执行复杂计算[1]的情况下。大多数时候，你并不需要真正了解很多：当你想增强Python函数时，只需将其转换为TF函数即可。

此外，当你编写自定义损失函数、自定义指标、自定义层或任何其他自定义函数，并在Keras模型中使用它时（如本章所述），Keras会自动将你的函数转换为TF函数——不需要使用tf.function（）。因此大多数时候，所有这些处理都是100％透明的。

 你可以通过在创建自定义层或自定义模型时设置dynamic=True来告诉Keras不要将Python函数转换为TF函数。或者，可以在调用模型的compile（）方法时设置run_eagerly=True。

默认情况下，TF函数会为每个不同的输入形状和数据类型集生成一个新图形，并将其缓存以供后续调用。例如，如果调用tf_cube（tf.constant（10）），将为形状为[]的int32张量生成图形。如果调用tf_cube（tf.constant（20）），则会重用相同的图。但是如果你随后调用tf_cube（tf.constant（[10，20]）），则会为形状为[2]的int32张量生成一个新图。这就是TF函数处理多态（即变化的参数类型和形状）的方式。但是这仅适用于张量参数：如果将Python数值传递给TF函数，则将为每个不同的值生成一个新图：例如，调用tf_cube（10）和tf_cube（20）将生成两个图。

 如果用不同的Python数值多次调用TF函数，则会生成许多图，这会降低程序的运行速度并消耗大量RAM（必须删除TF函数才能释放它）。Python值应保留给很少有唯一值的参数，例如像每层神经元的数量那样的超参数，这使TensorFlow可以更好地优化模型的每个变体。

12.4.1　自动图和跟踪

那么TensorFlow如何生成图呢？它首先分析Python函数的源代码来捕获所有控制流语句，例如for循环、while循环和if语句，以及break、continue和return语句。第一步称为自动图（AutoGraph）。TensorFlow必须分析源代码的原因是Python没有提供任何其他方法来捕获控制流语句：它提供了诸如__add__（）和__mul__（）之类的方法来捕获+和*之类的运算符，但没有__while__（）或__if__（）方法。在分析了函数的代码之后，自动图输出该函数的升级版本，其中所有控制流语句都被相应的TensorFlow操作替换，例如tf.while_loop（）用于循环，tf.cond（）用于if语句。例如，在图12-4中，自动图分析

sum_squares（）Python函数的源代码，并生成tf__sum_squares（）函数。在此函数中，用loop_body（）函数的定义（包含原始for循环的主体）替换了for循环，然后再调用for_stmt（）函数。该调用将在计算图中构建适当的tf.while_loop（）操作。

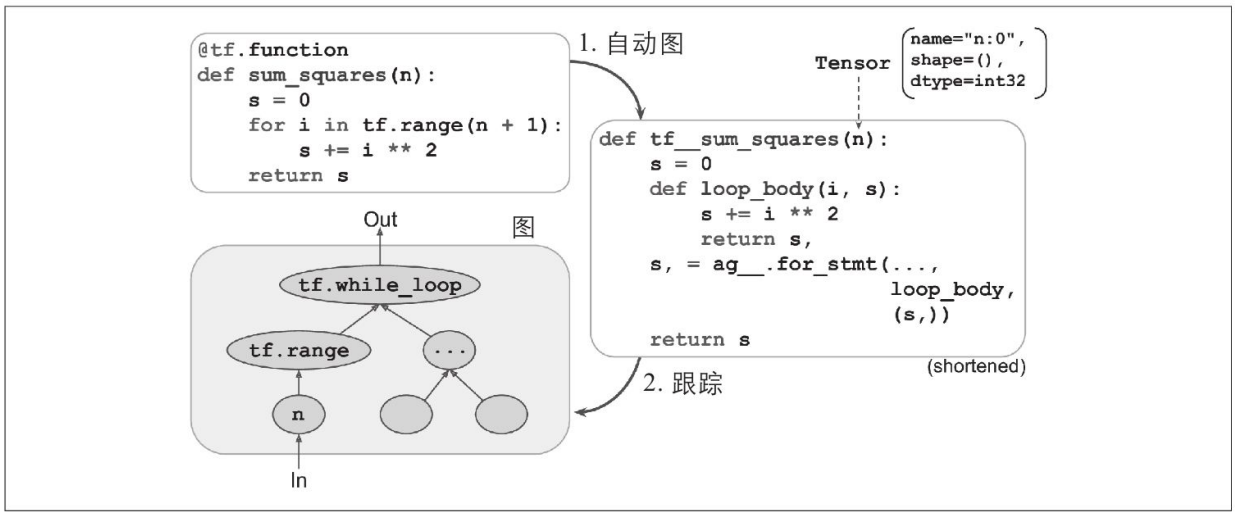


图12-4：TensorFlow如何使用自动图和跟踪生成图

接下来，TensorFlow调用此“升级”函数，但不传递参数，而是传递符号张量——没有任何实际值的张量，仅包含名称、数据类型和形状。例如，如果调用sum_squares（tf.constant（10）），则使用参数为类型int32且形状为[]的符号张量来调用tf__sum_squares（）函数。该函数将在图模式下运行，这意味着每个TensorFlow操作都会在图形中添加一个节点来表示自身及其输出张量（与常规模式相对，称为eager执行或eager模式）。在图形模式下，TF操作不执行任何计算。如果你知道TensorFlow 1，这应该很熟悉，因为图形模式是默认模式。在图12-4中，你可以看到使用符号张量作为参数调用tf__sum_squares（）函数（在本例中为形状[]的int32张量），并且在跟踪过程中生成了最终图形。节点表示操作，箭头表示张量（生成的函数和图形均已简化）。

 要查看生成函数的源代码，可以调用tf.autograph.to_code（sum_squares.python_function）。该代码不是很漂亮，但有时可以帮助调试。

12.4.2 TF函数规则

在大多数情况下，将执行TensorFlow操作的Python函数转换为TF函数很简单：用@tf.function修饰它或让Keras替你处理。但是有一些规则需要遵守：

• 如果调用任何外部库，包括NumPy甚至标准库，此调用将仅在跟踪过程中运行。它不会成为图表的一部分。实际上，TensorFlow图只能包含TensorFlow构造（张量、运算、变量、数据集等）。因此请使用tf.reduce_sum（）代替np.sum（），使用tf.sort（）代替内置的sorted（）函数，以此类推（除非你真的希望代码仅在跟踪过程中运行）。这还有一些其他含义：

• 如果你定义了一个返回np.random.rand（）的TF函数f（x），则仅在跟踪该函数时才会生成随机数，因此f（tf.con stant（2.））和f（tf.constant（3.））将返回相同的随机数，但f（tf.consta nt（[2.，3.]））将返回不同的随机数。如果把np.random.rand（）替换为tf.random.uniform（[]），则每次操作都会生成一个新的随机数，因为该操作将成为图形的一部分。

• 如果你的非TensorFlow代码具有副作用（例如记录某些内容或更新Python计数器），那么你不应期望每次调用TF函数时都会发生这些副作用，因为它们只会在跟踪该函数时发生。

• 你可以在tf.py_function（）操作中包装任何Python代码，但这样做会降低性能，因为TensorFlow无法对此代码进行任何图优化。这也

会降低可移植性，因为该图仅可在安装了Python（并且安装了正确的库）的平台上运行。

• 你可以调用其他Python函数或TF函数，但它们应遵循相同的规则，因为TensorFlow会在计算图中捕获它们的操作。请注意这些其他函数不需要用@tf.function修饰。

• 如果该函数创建了一个TensorFlow变量（或任何其他有状态的TensorFlow对象，例如数据集或队列），则必须在第一次调用时这样做（只有这样做），否则你会得到一个异常。通常最好在TF函数（例如在自定义层的build（）方法中）外部创建变量。如果要为变量分配一个新值，确保调用它的assign（）方法，而不要使用=运算符。

• 你的Python函数的源代码应可用于TensorFlow。如果源代码不可用（例如，如果你在Python shell中定义函数，而该函数不提供对源代码的访问权，或者仅将已编译的*.pyc Python文件部署到生产环境中），则生成图的过程会失败或功能受限。

• TensorFlow只能捕获在张量或数据集上迭代的for循环。因此请确保你使用for i in tf.range（x），而不是使用for i in range（x），否则这个循环不会在图中被捕获。相反它会在跟踪过程中运行（也许这就是你想要的，如果要使用for循环来构建图形，例如在神经网络中创建每一层）。

• 与通常一样，出于性能原因，应尽可能使用向量化实现，而不是使用循环。

是时候总结了！在本章中，我们首先对TensorFlow进行了简要概述，然后介绍了TensorFlow的底层API，包括张量、操作、变量和特殊数据结构。然后我们使用这些工具自定义tf.keras中的几乎每个组件。最后，我们研究了TF函数如何提高性能，如何使用自动图和跟踪来生成

计算图以及编写TF函数时应遵循的规则（如果你想进一步打开黑箱子，例如浏览生成的图形，可以在附录G中找到技术细节）。

在第13章中，我们将研究如何使用TensorFlow有效地加载和预处理数据。

[1] 但是在这个小示例中，计算图是如此之小，以至于根本没有要优化的东西，因此tf_cube（）的运行实际上比cube（）慢得多。

12.5 练习题

1. 如何用一句话形容TensorFlow？它的主要特点是什么？你可以说出其他流行的深度学习库吗？

2. TensorFlow是否可以简单替代NumPy？两者之间的主要区别是什么？

3. 使用tf.range（10）和tf.constant（np.ara nge（10））是否会得到相同的结果？

4. 除了常规张量之外，你还能说出TensorFlow中可用的其他6个数据结构吗？

5. 可以通过编写函数或继承keras.losses.Loss类来自定义损失函数。你何时会使用哪个方法？

6. 同样，自定义指标可以在函数中定义或者在keras.metrics.Metric子类中定义。你何时会使用哪个方法？

7. 什么时候应该自定义图而不是自定义模型？

8. 有哪些示例需要编写你自己的自定义训练循环？

9. 自定义Keras组件可以包含任意Python代码，还是必须转换为TF函数？

10. 如果要将函数转换为TF函数，应遵循哪些主要规则？

11.你何时需要创建动态Keras模型？你是怎样做的？为什么不让所有模型动态化？

12.实现一个执行层规范的自定义层（我们将在第15章中使用这种类型的层）：

a.build（）方法应定义两个可训练的权重α和β，形状均为input_shape[-1：]，数据类型均为tf.float32。α应该用1s初始化，而β必须用0s初始化。

b.call（）方法应计算每个实例特征的均值μ和标准差σ。为此，可以使用tf.nn.moments（inputs，axes=-1，keepdims=True），它返回所有实例的均值μ和方差σ^2（计算方差的平方根以获得标准差）。然后，该函数应计算并返回$\alpha \otimes (X-\mu)/(\sigma+\varepsilon)+\beta$，其中⊗表示逐项相乘（*），而ε是一个平滑项（小常数以避免被零除，例如0.001）。

c.确保你的自定义层产生与keras.layers.LayerNormalization层相同（或几乎相同）的输出。

13.使用自定义训练循环来训练模型从而处理Fashion MNIST数据集（见第10章）。

a.显示每个轮次、迭代、平均训练损失和每个轮次中的平均精度（在每次迭代中更新），以及每个轮次结束时的验证损失和精度。

b.尝试对较高层和较低层使用不同的学习率的不同优化器。

这些练习题的解答在附录A中提供。

第13章　使用TensorFlow加载和预处理数据

到目前为止，我们仅使用了适合放入内存的数据集，但是深度学习系统经常在非常大的数据集上训练，而这些数据集不能完全放入RAM。读取大型数据集并对其进行有效预处理可能对其他深度学习库来说很难实现，但是TensorFlow借助Data API很容易实现：只需创建一个数据集对象，并告诉它从何处获取数据以及如何对其进行转换。TensorFlow负责所有细节的实现，例如多线程、队列、批处理和预取。此外，Data API与tf.keras无缝协同工作！

现成的Data API可以读取文本文件（例如CSV文件）、具有固定大小记录的二进制文件以及使用TensorFlow的TFRecord格式（支持各种大小的记录）的二进制文件。TFRecord是一种包含协议缓冲区的灵活高效的二进制格式（一种开源二进制格式）。Data API还支持从SQL数据库中读取。而且许多开源扩展都可以从各种数据源中读取，例如Google的BigQuery服务。

有效读取大数据集并不是唯一的难点：数据也需要进行预处理，通常是归一化的。而且，它并不总是严格地由数字字段组成，可能存在文本特征、分类特征等。这些需要进行编码、例如使用独热编码、词袋编码或嵌入（如我们将要看到的，嵌入是一种可训练的密集向量，表示类别或令牌）。处理所有这些预处理的一种方法是编写自己的自定义预处理层，也可以使用Keras提供的标准预处理层。

在本章中，我们将介绍Data API、TFRecord格式以及如何创建自定义预处理层，还将介绍如何使用标准Keras层。下面快速浏览TensorFlow生态系统中的一些相关项目：

TF转换（tf.Transform）

使编写单个预处理函数成为可能，可以在完整的训练集中以批处理模式运行该函数，然后再进行训练（以加快速度），之后将其导出为TF函数并集成到训练的模型中，以便一旦部署在生产环境中，它可以即时处理新实例。

TF数据集（TFDS）

提供方便的函数来下载各种类型的常见数据集，包括像ImageNet这样的大型数据集与方便的数据集对象（可以使用Data API对其进行操作）。

因此，让我们开始吧！

13.1 数据API

整个数据API都围绕着数据集的概念：你可能会怀疑，这代表了数据元素的一个序列。通常，你使用的是逐步从磁盘中读取数据的数据集，但为了简单起见，让我们使用tf.data.Dataset.from_tensor_slices（）在RAM中完全创建一个数据集：

```
>>> X = tf.range(10)  # any data tensor
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
<TensorSliceDataset shapes: (), types: tf.int32>
```

from_tensor_slices（）函数采用一个张量并创建一个tf.data.Dataset，其元素都是X的切片（沿第一个维度），因此此数据集包含10个元素：张量0，1，2，…，9。在这种情况下，如果我们使用tf.data.Dataset.range（10），则将获得相同的数据集。

你可以像以下这样简单地遍历数据集的元素：

```
>>> for item in dataset:
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
[...]
tf.Tensor(9, shape=(), dtype=int32)
```

13.1.1 链式转换

有了数据集后，你可以通过调用其转换方法对其应用各种转换。每个方法都返回一个新的数据集，因此你可以像这样进行链式转换（此链

式转换如图13-1所示）：

```
>>> dataset = dataset.repeat(3).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

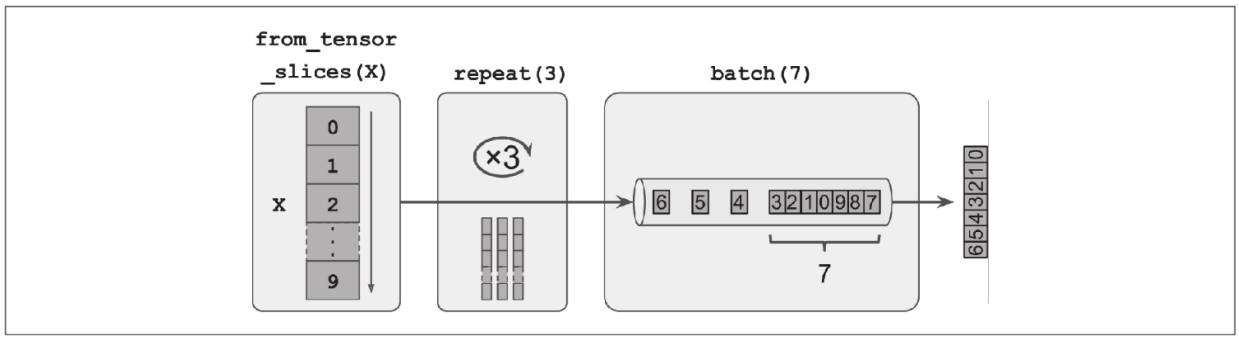


图13-1：链式数据集转换

在此示例中，我们首先在原始数据集上调用repeat（）方法，它返回一个新数据集，该数据集将重复原始数据集的元素三次。当然这不会将内存中的所有数据复制三遍（你如果不带任何参数调用此方法，则新数据集将永远重复源数据集，因此遍历该数据集的代码必须决定何时停止）。然后我们在此新数据集上调用batch（）方法，并再次创建一个新的数据集。这把先前数据集的元素以7个元素为一个批次分组。最后我们遍历此最终数据集的元素。如你所见，batch（）方法最后输出一个大小为2而不是7的最终批次，但是如果你希望它删除最终批次，可以使用drop_remainder=True调用它，使所有批次具有完全相同的大小。

 数据集方法不会修改数据集，而是创建新数据集，因此请保留对这些新数据集的引用（例如，使用dataset=...），否则将不会发生任何事情。

你还可以通过调用map（）方法来变换元素。例如，这将创建一个新数据集，其中所有元素均是原来的两倍：

```
>>> dataset = dataset.map(lambda x: x * 2) # Items: [0,2,4,6,8,10,12]
```

你可以调用此函数把你所需的任何预处理应用于你的数据。有时这可能包括非常密集的计算，例如重构图像或旋转图像，因此你通常会希望产生多个线程来加快处理速度：这就像设置num_parallel_calls参数一样简单。请注意传递给map（）方法的函数必须可转换为TF函数（见第12章）。

虽然map（）方法将转换应用于每个元素，但是apply（）方法将转换应用于整个数据集。例如，以下代码将unbatch（）函数应用于数据集（此函数目前处于实验阶段，但在将来的版本中很可能会移至核心API）。新数据集中的每个元素都是一个整数张量，而不是一个7个整数的批次：

```
>>> dataset = dataset.apply(tf.data.experimental.unbatch()) # Items: 0,2,4,...
```

也可以使用filter（）方法简单地过滤数据集：

```
>>> dataset = dataset.filter(lambda x: x < 10) # Items: 0 2 4 6 8 0 2 4 6...
```

你通常会希望查看数据集中的一些元素，可以使用take（）方法：

```
>>> for item in dataset.take(3):
...     print(item)
```

```
...
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
```

13.1.2 乱序数据

如你所知，当训练集中的实例相互独立且分布均匀时，梯度下降效果最佳（见第4章）。确保这一点的一种简单方法是使用shuffle（）方法对实例进行乱序。它会创建一个新的数据集，该数据集首先将源数据集的第一项元素填充到缓冲区中。然后无论任何时候要求提供一个元素，它都会从缓冲区中随机取出一个元素，并用源数据集中的新元素替换它，直到完全遍历源数据集为止。它将继续从缓冲区中随机抽取元素直到其为空。你必须指定缓冲区的大小，重要的是要使其足够大，否则乱序不会非常有效[1]。不要超出你有的RAM的数量，即使你有足够的RAM，也不需要超出数据集的大小。如果每次运行程序都想要相同的随机顺序，你可以提供随机种子。例如，以下代码创建并显示一个包含整数0到9的数据集，重复3次，使用大小为5的缓冲区和42的随机种子进行乱序，并以7的批次大小进行批处理：

```
>>> dataset = tf.data.Dataset.range(10).repeat(3) # 0 to 9, three times
>>> dataset = dataset.shuffle(buffer_size=5, seed=42).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 2 3 6 7 9 4], shape=(7,), dtype=int64)
tf.Tensor([5 0 1 1 8 6 5], shape=(7,), dtype=int64)
tf.Tensor([4 8 7 1 2 3 0], shape=(7,), dtype=int64)
tf.Tensor([5 4 2 7 8 9 9], shape=(7,), dtype=int64)
tf.Tensor([3 6], shape=(2,), dtype=int64)
```

如果你在经过乱序的数据集上调用repeat（），则默认情况下，它在每次迭代时生成一个新次序。通常这是个好主意，但如果你希望在每次迭代中重用相同的顺序（例如用于测试或调试），则可以设置reshuffle_each_iteration=False。

对于不适合内存的大型数据集，这种简单的缓冲区乱序方法可能不够用，因为与数据集相比，缓冲区很小。一种解决方法是乱序源数据本身（例如在Linux上，可以使用shuf命令改组文本文件）。这肯定会大大改善乱序！即使源数据已经进行了乱序，你通常也希望对其进行更多的乱序，否则在每个轮次都有相同顺序的重复，该模型最终可能会产生偏差（例如，由于源数据顺序中偶然出现了一些虚假模式）。为了进一步乱序实例，一种常见的方法是将源数据拆分为多个文件，然后在训练过程中以随机顺序读取它们。但是位于同一文件中的实例仍然相互接近。为了避免这种情况，你可以随机选择多个文件并同时读取它们，并且交错它们的记录。然后最重要的是，你可以使用shuffle（）方法添加一个乱序缓冲区。听起来这很麻烦，但是不用担心：Data API只需几行代码就可以实现所有这些功能。让我们看看如何做到这一点。

交织来自多个文件的行

首先，假设你已经加载了加州住房数据集，对其进行乱序（除非已经进行了乱序），然后将其分为训练集、验证集和测试集。之后将每个集合分成许多类似如下的CSV文件（每行包含8个输入特征以及目标房屋中间值）：

```
MedInc,HouseAge,AveRooms,AveBedrms,Popul,AveOccup,Lat,Long,MedianHouseValue
3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442
5.3275,5.0,6.4900,0.9910,3464.0,3.4433,33.69,-117.39,1.687
3.1,29.0,7.5423,1.5915,1328.0,2.2508,38.44,-122.98,1.621
[...]
```

我们还假设train_filepaths包含训练文件路径的列表（并且你还有valid_filepaths和test_filepaths）：

```
>>> train_filepaths
['datasets/housing/my_train_00.csv', 'datasets/housing/my_train_01.csv',...]
```

另外，你可以使用文件模式。例如，train_filepaths="datasets/housing/my_train_*.csv"。现在让我们创建一个仅包含以下文件路径的数据集：

```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

默认情况下，list_files（）函数返回一个乱序的文件路径的数据集。通常这是一件好事，但是如果出于某种原因不希望这样做，则可以设置shuffle=False。

接下来，你可以调用interleave（）方法一次读取5个文件并交织它们的行（使用skip（）方法跳过每个文件的第一行，即标题行）：

```
n_readers = 5
dataset = filepath_dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
    cycle_length=n_readers)
```

interleave（）方法将创建一个数据集，该数据集将从filepath_dataset中拉出5个文件路径，对于每个路径，它将调用你为其提供的函数（在此示例中为lambda）来创建新的数据集（在此示例中为TextLineDataset）。为了清楚起见，在此阶段总共有7个数据集：文件路径数据集、交织数据集和由交织数据集在内部创建的5个TextLineDataset。当我们遍历交织数据集时，它将循环遍历这5个TextLineDatasets，每次读取一行，直到所有数据集都读出为止。然后它将从filepath_dataset获取剩下的5个文件路径，并以相同的方式对它们进行交织，以此类推，直到读完文件路径。

 为了交织效果更好，最好使用具有相同长度的文件。否则最长文件的结尾将不会交织。

默认情况下，interleave（）不使用并行。它只是顺序地从每个文件中一次读取一行。如果你希望它并行读取文件，则可以将num_parallel_calls参数设置为所需的线程数（请注意map（）方法也具有此参数）。你甚至可以将其设置为tf.data.experimental.AUTOTUNE，使TensorFlow根据可用的CPU动态地选择合适的线程数（但是目前这还是实验功能）。让我们看一下数据集现在包含的内容：

```
>>> for line in dataset.take(5):
...     print(line.numpy())
...
b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782'
b'4.1812,52.0,5.7013,0.9965,692.0,2.4027,33.73,-118.31,3.215'
b'3.6875,44.0,4.5244,0.9930,457.0,3.1958,34.04,-118.15,1.625'
b'3.3456,37.0,4.5140,0.9084,458.0,3.2253,36.67,-121.7,2.526'
b'3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442'
```

这些是随机选择的5个CSV文件的第一行（忽略标题行）。看起来不错！但是正如你所看到的，这些只是字节字符串，我们需要解析它们并按比例缩放数据。

13.1.3 预处理数据

让我们实现一个执行预处理的小函数：

```
X_mean, X_std = [...] # mean and scale of each feature in the training set
n_inputs = 8

def preprocess(line):
  defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
  fields = tf.io.decode_csv(line, record_defaults=defs)
  x = tf.stack(fields[:-1])
  y = tf.stack(fields[-1:])
  return (x - X_mean) / X_std, y
```

让我们看一下这段代码：

• 首先，代码假设我们已经预先计算了训练集中每个特征的均值和标准差。X_mean和X_std是一维张量（或NumPy数组），其中包含8个浮点数，每个输入特征一个。

• preprocess（）函数使用一行CSV内容并开始对其进行解析。为此它使用tf.io.decode_csv（）函数，该函数带有两个参数：第一个是要解析的行，第二个是一个包含CSV文件中每一列的默认值的数组。这个数组不仅告诉TensorFlow每列的默认值，而且告诉列数及其类型。在此示例中，我们告诉所有特征列都是浮点数，缺失值应默认为0，我们还提供了一个类型为tf.float32的空数组作为最后一列（目标值）的默认值：该数组告诉TensorFlow该列包含浮点数，但没有默认值，因此如果遇到缺失值，它会引发异常。

• encode_csv（）函数返回标量张量的列表（每列一个），但是我们需要返回一维张量数组。因此我们在除最后一个（目标值）之外的所有张量上调用tf.stack（）：这会将这些张量堆叠到一维度组中。然后我们对目标值执行相同的操作（这使其成为具有单个值的一维张量数组，而不是标量张量）。

• 最后我们通过减去特征均值然后除以特征标准差来缩放输入特征，然后返回一个包含已缩放特征和目标值的元组。

让我们测试一下预处理函数：

```
>>> preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')
(<tf.Tensor: id=6227, shape=(8,), dtype=float32, numpy=
 array([ 0.16579159,  1.216324  , -0.05204564, -0.39215982, -0.5277444 ,
        -0.2633488 ,  0.8543046 , -1.3072058 ], dtype=float32)>,
 <tf.Tensor: [...], numpy=array([2.782], dtype=float32)>)
```

看起来不错！现在，我们可以将该函数应用于数据集。

13.1.4 合并在一起

为了使代码可重用，我们将到目前为止讨论的所有内容放到一个小的辅助函数中：它将创建并返回一个数据集，该数据集有效地从多个CSV文件中加载加州住房数据，对其进行预处理、随机乱序，可以选择重复，并进行批处理（见图13-2）：

```
def csv_reader_dataset(filepaths, repeat=1, n_readers=5,
                       n_read_threads=None, shuffle_buffer_size=10000,
                       n_parse_threads=5, batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths)
    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
        cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
    dataset = dataset.shuffle(shuffle_buffer_size).repeat(repeat)
    return dataset.batch(batch_size).prefetch(1)
```

除了最后一行（prefetch（1））对性能很重要，此代码中的所有内容都很合理。

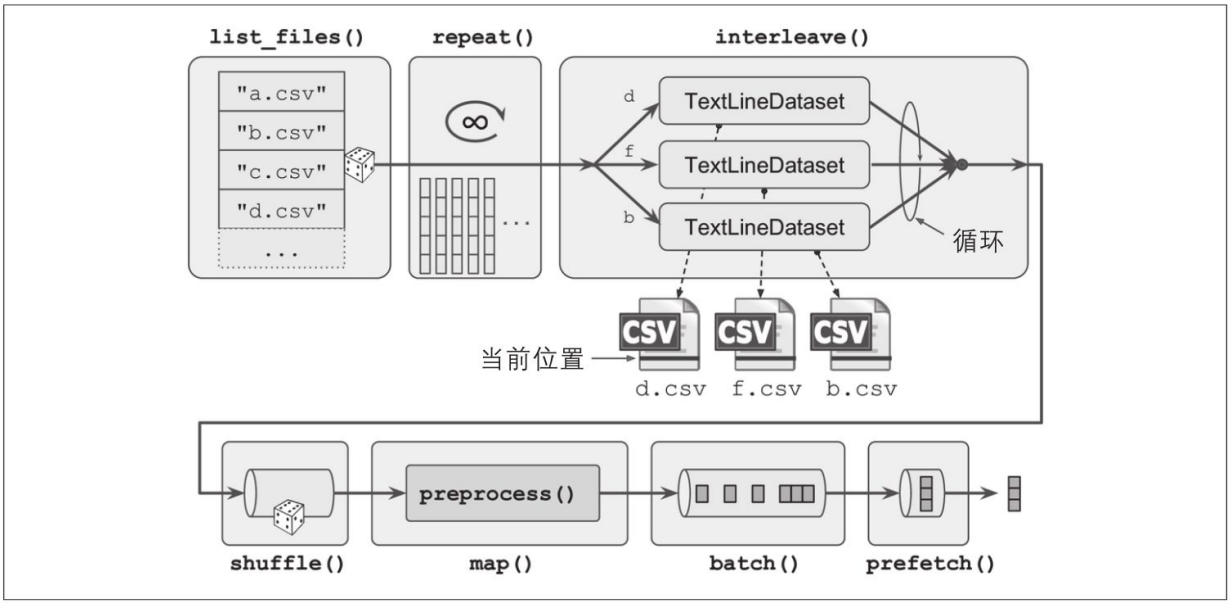


图13-2：从多个CSV文件加载和预处理数据

13.1.5 预取

通过最后调用prefetch（1），我们正在创建一个数据集，该数据集将尽最大可能总是提前准备一个批次[2]。换句话说，当我们的训练算法正处理一个批次时，数据集已经并行工作以准备下一批次了（例如从磁盘中读取数据并对其进行预处理）。如图13-3所示，这可以显著提高性能。如果我们确保加载和预处理是多线程的（通过在调用interleave（）和map（）时设置num_parallel_calls），我们可以在CPU上利用多个内核，希望准备一个批次数据的时间比在GPU上执行一个训练步骤的时间要短一些：这样，GPU将达到几乎100%的利用率（从CPU到GPU的数据传输时间除外[3]），并且训练会运行得更快。

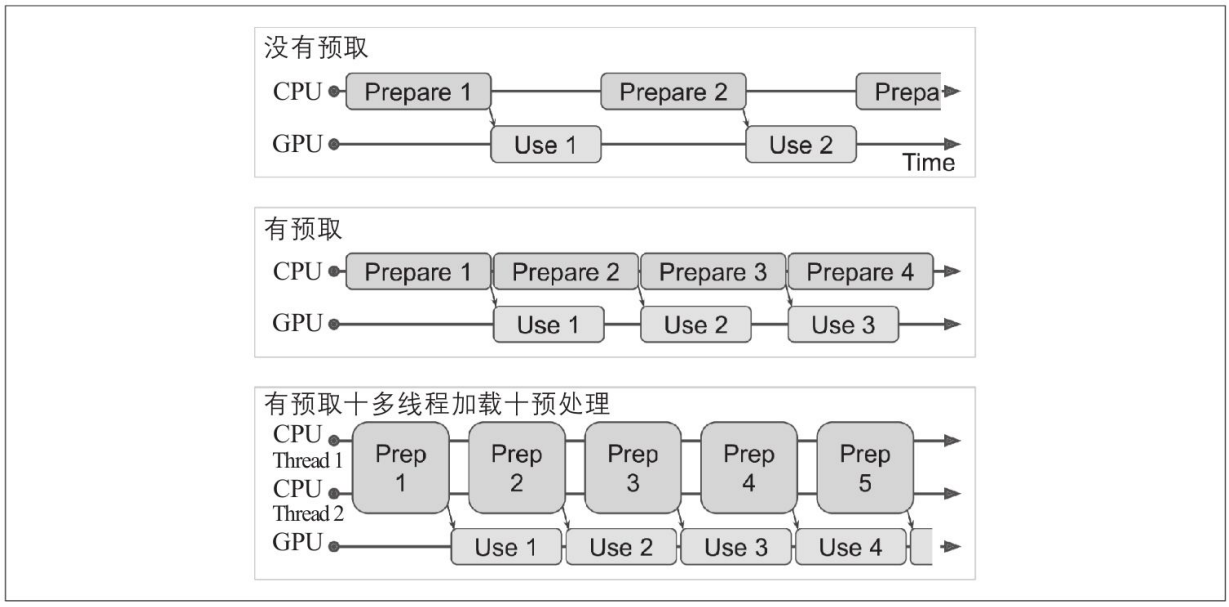


图13-3：通过预取，CPU和GPU并行工作：当GPU处理一个批次时，CPU处理下一批次

如果你打算购买GPU卡，那么它的处理能力和内存大小非常重要（特别是大量的RAM对于计算机视觉至关重要）。要获得良好性能同样很重要的是其内存带宽，它代表每秒可以进出其RAM的GB数据大小。

如果数据集足够小，可以放到内存里，则可以使用数据集的cache（）方法将其内容缓存到RAM中，从而显著加快训练速度。通常应该在加载和预处理数据之后，但在乱序、重复、批处理和预取之前执行此操作。这样，每个实例仅被读取和预处理一次（而不是每个轮次一次），但数据仍会在每个轮次进行不同的乱序，并且仍会提前准备下一批次。

现在你知道如何构建有效的输入流水线来从多个文本文件加载和预处理数据了。我们已经讨论了最常见的数据集方法，但还有更多方法：concatenate（）、zip（）、window（）、reduce（）、shard（）、flat_map（）和padded_batch（）。还有另外两个类方法：from_generator（）和from_tensors（），它们分别从Python生成器或张量列表创建新的数据集。请查看API文档以了解更多详细信息。还要注意，tf.data.experimental中有一些实验性功能，其中许多功能可能会在将来的版本中成为核心API（查看CsvDataset类以及make_csv_dataset（）方法，该方法负责推断每一列的类型）。

13.1.6　和tf.keras一起使用数据集

现在我们可以使用csv_reader_dataset（）函数为训练集创建数据集。请注意我们不需要重复它，因为tf.keras会进行处理。我们还为验证集和测试集创建数据集：

```
train_set = csv_reader_dataset(train_filepaths)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

现在我们可以使用这些数据集[4]简单地构建和训练Keras模型。我们要做的就是将训练数据集和验证数据集传递给fit（）方法，而不是X_train、y_train、X_valid和y_valid5[5]：

```
model = keras.models.Sequential([...])
model.compile([...])
model.fit(train_set, epochs=10, validation_data=valid_set)
```

类似地，我们可以将数据集传递给valuate（）方法和predict（）方法：

```
model.evaluate(test_set)
new_set = test_set.take(3).map(lambda X, y: X) # pretend we have 3 new instances
model.predict(new_set) # a dataset containing new instances
```

与其他集合不同，new_set通常不包含标签（如果包含标签，Keras将忽略它们）。请注意在所有这些情况下，你仍然可以使用NumPy数组而不是你想的数据集（当然它们必须先加载并进行预处理）。

如果你要构建自己的自定义训练循环（如第12章所述），可以非常自然地遍历训练集：

```
for X_batch, y_batch in train_set:
    [...] # perform one Gradient Descent step
```

实际上，甚至可以创建执行整个训练循环的TF函数（见第12章）：

```
@tf.function
def train(model, optimizer, loss_fn, n_epochs, [...]):
    train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs, [...])
    for X_batch, y_batch in train_set:
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

恭喜，你现在知道如何使用数据API建立功能强大的输入流水线了！但是到目前为止，我们使用了CSV文件，这些文件是通用、简单且便捷的，但是效率并不高，并且不能很好地支持大型或复杂的数据结构（例如图像或音频）。因此让我们看看如何使用TFRecords。

 如果你对CSV文件（或使用的任何其他格式）感到满意，就不必使用TFRecords。俗话说，如果没有破裂，就不要修理！当训练过程中的瓶颈是加载和解析数据时，TFRecords很有用。

[1] 想象一下左边有一叠纸牌：假设你只是拿出最上面三张纸牌并对其进行洗牌，然后随机挑选一张纸牌并将其放在右边，另外两张纸牌则留在手中。从你左边拿另一张牌，将三张牌随机洗牌，随机挑选其中一张，放在右边。当你完成所有类似的卡片操作后，你将在右侧看到一叠纸牌：你认为这样洗牌完美吗？

[2] 通常仅预取一个批次就可以了，但在某些情况下你可能需要再预取几个批次。或者你可以通过传递tf.data.experimental.AUTOTUNE让TensorFlow自动决定（这在目前是实验功能）。

[3] 但是请查看tf.data.experimental.prefetch_to_device（）函数，该函数可以将数据直接预取到GPU。

[4] 对数据集的支持特定于tf.keras，这在Keras API的其他实现中不起作用。

[5] fit（）方法负责重复训练数据集。另外你可以在训练数据集上调用repeat（），使其永远重复，也可在调用fit（）方法时指定steps_per_epoch参数。这在极少数情况下可能很有用，例如，如果你想使用跨轮次的乱序缓冲区。

13.2 TFRecord格式

TFRecord格式是TensorFlow首选的格式，用于存储大量数据并有效读取数据。这是一种非常简单的二进制格式，只包含大小不同的二进制记录序列（每个记录由一个长度、一个用于检查长度是否损坏的CRC校验和、实际数据以及最后一个CRC校验和组成）。你可以使用tf.io.TFRecordWriter类轻松创建TFRecord文件：

```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:
    f.write(b"This is the first record")
    f.write(b"And this is the second record")
```

然后你可以使用tf.data.TFRecordDataset读取一个或多个TFRecord文件：

```
filepaths = ["my_data.tfrecord"]
dataset = tf.data.TFRecordDataset(filepaths)
for item in dataset:
    print(item)
```

这将输出：

```
tf.Tensor(b'This is the first record', shape=(), dtype=string)
tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```

 默认情况下，TFRecordDataset将一个接一个地读取文件，但是你可以通过设置num_parallel_reads使其并行读取多个文件并交织记录。另外，你可以使用list_files（）和interleave（）得到与前面读取多个CSV文件相同的结果。

13.2.1　压缩的TFRecord文件

有时压缩TFRecord文件可能很有用，尤其是在需要通过网络连接加载它们的情况下。你可以通过设置options参数来创建压缩的TFRecord文件：

```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
  [...]
```

读取压缩的TFRecord文件时需要指定压缩类型：

```
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                  compression_type="GZIP")
```

13.2.2　协议缓冲区简介

即使每个记录可以使用你想要的任何二进制格式，TFRecord文件通常包含序列化的协议缓冲区（也称为protobufs）。这是一种可移植、可扩展且高效的二进制格式，在2001年由Google开发，并于2008年开源。protobufs现在被广泛使用，尤其是在Google的远程过程调用系统gRPC中。它们使用如下所示的简单语言定义：

```
syntax = "proto3";
message Person {
  string name = 1;
  int32 id = 2;
  repeated string email = 3;
}
```

此定义表示我们使用的是protobuf格式的版本3，它指定每个Person对象[1]（可选）具有string类型的name，类型int32的id和零个或多个email字段（每个都是string类型）。数字1、2和3是字段标识符：它们用于每个记录的二进制表示形式。在.proto文件中定义后，就可以对其进行编译。这要求protoc（protobuf编译器）来生成Python（或其他语言）的访问类。请注意，我们将使用的protobuf定义已经为你编译完成，它们的Python类是TensorFlow的一部分，因此你无须使用protoc。你需要知道的是如何在Python中使用protobuf访问类。为了说明基本用法，让我们看一个简单的示例，该示例使用为Person protobuf生成的访问类（代码在注释中说明）：

```
>>> from person_pb2 import Person  # import the generated access class
>>> person = Person(name="Al", id=123, email=["a@b.com"])  # create a Person
>>> print(person)  # display the Person
name: "Al"
id: 123
email: "a@b.com"
>>> person.name  # read a field
"Al"
>>> person.name = "Alice"  # modify a field
>>> person.email[0]  # repeated fields can be accessed like arrays
"a@b.com"
>>> person.email.append("c@d.com")  # add an email address
>>> s = person.SerializeToString()  # serialize the object to a byte string
>>> s
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person()  # create a new Person
>>> person2.ParseFromString(s)  # parse the byte string (27 bytes long)
27
>>> person == person2  # now they are equal
True
```

简而言之，我们导入由protoc生成的Person类，创建一个实例并使用它，可视化并读写一些字段，然后使用SerializeToString（）方法对其进行序列化。这是准备通过网络保存或传输的二进制数据。在

读取或接收此二进制数据时，我们可以使用ParseFromString（）方法对其进行解析，然后得到被序列化的对象的副本[2]。

我们可以将序列化的Person对象保存到TFRecord文件，然后可以加载并解析它：一切都正常工作。但是，SerializeToString（）和ParseFromString（）不是TensorFlow操作（此代码中的其他操作也不是），因此它们不能包含在TensorFlow函数中（除了将它们包装在tf.py_function（）操作中，这会使代码变慢且可移植性降低，如我们在第12章中所见）。幸运的是，TensorFlow包含特殊的protobuf定义，并为其提供了解析操作。

13.2.3 TensorFlow协议

TFRecord文件中通常使用的主要protobuf是Example protobuf，它表示数据集中的一个实例。它包含一个已命名特征的列表，其中每个特征可以是字节字符串列表、浮点数列表或整数列表。以下是protobuf的定义：

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
message Feature {
    oneof kind {
        BytesList bytes_list = 1;
        FloatList float_list = 2;
        Int64List int64_list = 3;
    }
};
message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };
```

Bytes List、FloatList和Int64List的定义非常简单直接。请注意，[packed=true]用于重复的数字字段以实现更有效的编码。一个Feature包含BytesList或FloatList或Int64List。一个Features（带有s）包含将特征名称映射到相应特征值的字典。最后，一个Example

仅包含Features对象[3]。以下是创建tf.train.Example的方法，该示例表示与先前相同的person并将其写入TFRecord文件：

```
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                                                          b"c@d.com"]))
        }))
```

该代码有点冗长和重复，但相当简单（你可以轻松地将其包装在一个小的辅助函数中）。既然有了Example protobuf，我们可以通过调用其SerializeToString（）方法对其进行序列化，然后将结果数据写入TFRecord文件：

```
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    f.write(person_example.SerializeToString())
```

通常你会编写不止一个Example！你会创建一个转换脚本，该脚本从你当前的格式（例如CSV文件）中读取，为每个实例创建一个Example protobuf，对其进行序列化，然后将其保存到多个TFRecord文件中，最好在处理过程中对其进行乱序。这需要一些工作量，因此请再次确保确实有必要（也许你的流水线可以使用CSV文件正常工作）。现在我们有了一个不错的TFRecord文件，其中包含序列化的Example，让我们尝试加载它。

13.2.4　加载和解析Example

要加载序列化的Example protobufs，我们再次使用tf.data.TFRecordDataset，并使用tf.io.parse_single_example（）解析每个Example。这是一个TensorFlow操作，因此可以包含在TF函数中。它至少需要两个参数：一个包含序列化数据的字符串标量张量，以及每个特征的描述。这个描述是一个字典，将每个特征名称映射到表示特征形状、类型和默认值的tf.io.FixedLenFeature描述符，或者仅表示类型的tf.io.VarLen Feature描述符（如果特征列表的长度有所不同，例如"emails"特征）。

以下代码定义了一个描述字典，然后遍历TFRecord Dataset并解析该数据集包含的序列化的Example protobuf：

```
feature_description = {
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "emails": tf.io.VarLenFeature(tf.string),
}

for serialized_example in tf.data.TFRecordDataset(["my_contacts.tfrecord"]):
    parsed_example = tf.io.parse_single_example(serialized_example,
                                                feature_description)
```

固定长度特征被解析为规则张量，而可变长度特征被解析为稀疏张量。你可以使用tf.sparse.to_dense（）将稀疏张量转换为密集张量，但是在这个示例中，访问它的值更简单：

```
>>> tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
>>> parsed_example["emails"].values
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
```

Bytes List可以包含你想要的任何二进制数据，包括任何序列化的对象。例如，你可以使用tf.io.encode_jpeg（）对JPEG格式的图像进行编码，然后将此二进制数据放入Bytes List。稍后，当你的代码

读取TFRecord时，它将从解析Example开始，然后它需要调用tf.io.decode_jpeg（）来解析数据并获取原始图像（或者你可以使用tf.io.decode_image（）来解码任何BMP、GIF、JPEG或PNG图像）。你还可以通过使用tf.io.serialize_tensor（）来序列化张量，并存储在Bytes List中，然后将生成的字节字符串放入Bytes List特征中。稍后当你解析TFRecord时，可以使用tf.io.parse_tensor（）解析此数据。

与其使用tf.io.parse_single_example（）一个接一个地解析用例，不如使用tf.io.parse_example（）一个批次接一个批次地解析它们：

```
dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(10)
for serialized_examples in dataset:
    parsed_examples = tf.io.parse_example(serialized_examples,
                                          feature_description)
```

如你所见，Example protobuf对于大多数用例可能足够了。但当你处理列表的列表时，使用它可能会有些麻烦。例如，假设你要对文本文档进行分类。每个文档可以被表示为句子的列表，其中每个句子被表示为单词的列表。也许每个文档也都有一个注释列表，其中每个注释都表示为单词的列表。也可能有一些上下文数据，例如文档的作者、标题和出版日期。TensorFlow的SequenceExample protobuf是针对此类用例设计的。

13.2.5 使用SequenceExample Protobuf处理列表的列表

以下是SequenceExample protobuf的定义：

```
message FeatureList { repeated Feature feature = 1; };
message FeatureLists { map<string, FeatureList> feature_list = 1; };
message SequenceExample {
```

```
    Features context = 1;
    FeatureLists feature_lists = 2;
};
```

SequenceExample包含一个用于上下文数据的Feature对象和一个FeatureLists对象（包含一个或多个命名的FeatureList对象（一个名为“content”，另一个名为“comments”））。每个FeatureList包含一个Feature对象的列表，每个对象可以是一个字节字符串列表、一个64位整数列表或一个float列表（在本示例中，每个Feature代表一个句子或一个注释，可能是以字标识符列表的形式）。构建SequenceExample，对其进行序列化和解析的过程与构建Example对其进行序列化和解析相似，但是你必须使用tf.io.parse_single_sequence_example（）解析单个SequenceExample或者使用tf.io.parse_sequence_example（）来解析批处理。这两个函数都返回一个包含上下文特征（作为字典）和特征列表（也作为字典）的元组。如果特征列表包含大小不同的序列（如上例所示），则可能需要使用tf.RaggedTensor.from_sparse（）将它们转换为不规则的张量（有关完整代码，请参见notebook）：

```
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(
    serialized_sequence_example, context_feature_descriptions,
    sequence_feature_descriptions)
parsed_content = tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])
```

既然你知道如何有效地存储、加载和解析数据，下一步就是准备数据，以便可以将其馈送到神经网络。

[1] 由于protobuf对象是要序列化和传输的，因此称为消息。
[2] 本章包含有关使用TFRecords的protobuf所需的最低要求。要了解有关protobuf的更多信息，请访问https://homl.info/protobuf。
[3] 既然其中只包含一个Features对象，为什么要定义Example？好吧，TensorFlow的开发人员可能有一天会决定向其添加更多字段。只

要新的Example定义仍旧包含具有相同ID的features字段，它将向后兼容。这种可扩展性是protobuf的一大特色。

13.3 预处理输入特征

为神经网络准备数据需要将所有特征转换为数值特征，通常将其归一化等。特别是如果你的数据包含分类特征或文本特征，则需要将它们转换为数字。在准备数据文件时，可以使用任何你喜欢的工具（例如NumPy、pandas或Scikit-Learn）提前完成此操作。或者，你可以在使用Data API加载数据时动态地预处理数据（例如使用数据集的map（）方法，如我们之前看到的），也可以在模型中直接包含预处理层。现在让我们来看最后一个选项。

例如，这是使用Lambda层实现标准化层的方法。对于每个特征，它减去均值并除以其标准差（加上一个微小的平滑项，以避免被零除）：

```
means = np.mean(X_train, axis=0, keepdims=True)
stds = np.std(X_train, axis=0, keepdims=True)
eps = keras.backend.epsilon()
model = keras.models.Sequential([
    keras.layers.Lambda(lambda inputs: (inputs - means) / (stds + eps)),
    [...] # other layers
])
```

这不太难！但是你可能更喜欢使用一个很好的自包含自定义层（非常类似于Scikit-Learn的StandardScaler），而不是像means和stds之类的全局变量：

```
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_ = np.mean(data_sample, axis=0, keepdims=True)
        self.stds_ = np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs - self.means_) / (self.stds_ + keras.backend.epsilon())
```

在使用此标准化层之前，你需要通过调用adapt（）方法来使其适应数据集并将其传递给数据样本。这样它就可以为每个特征使用适当的均值和标准差：

```
std_layer = Standardization()
std_layer.adapt(data_sample)
```

该样本必须足够大以代表你的数据集，但不必是完整的训练集：通常，随机选择几百个实例就足够了（但是这取决于你的任务）。接下来你可以像常规层一样使用此预处理层：

```
model = keras.Sequential()
model.add(std_layer)
[...] # create the rest of the model
model.compile([...])
model.fit([...])
```

如果你认为Keras应该包含这样的标准化层，那么以下对你来说是个好消息：当你阅读本书时，keras.layers.Normal标准化层可能已经可以使用了。它的工作方式非常类似于自定义标准化层：首先，创建该层，然后通过将一个数据样本传递给adapt（）方法使其适应你的数据集，最后正常使用该层。

现在让我们看一下分类特征。我们从将它们编码为独热向量开始。

13.3.1　使用独热向量编码分类特征

考虑我们在第2章中探讨的加州住房数据集中的ocean_proximity特征，它是一个具有5个可能值的分类特征："<1H OCEAN""INLAND""NEAR OCEAN""NEAR BAY"和"ISLAND"。再将其提供给神经网络之前我们需要对

该特征进行编码。由于类别很少，我们可以使用独热编码。为此我们首先需要将每个类别映射到其索引（0到4），这可以使用查找表来完成：

```
vocab = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
indices = tf.range(len(vocab), dtype=tf.int64)
table_init = tf.lookup.KeyValueTensorInitializer(vocab, indices)
num_oov_buckets = 2
table = tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)
```

让我们看一下这段代码：

• 我们首先定义词汇表：这是所有可能类别的列表。

• 然后，我们创建带有相应索引（0到4）的张量。

• 接下来，我们为查找表创建一个初始化程序，将类别列表及其对应的索引传递给它。在此示例中，我们已经有此数据，因此我们使用KeyValueTensorInitializer。但是如果类别在文本文件中列出（每行一个类别），我们要使用TextFileInitializer。

• 在最后两行中，我们创建了查找表，为其提供了初始化程序并指定了词汇表外（outof-vocabulary，oov）桶的数量。如果我们查找词汇表中不存在的类别，则查找表将计算该类别的哈希并将这个未知类别分配给oov桶之中的一个。它们的索引从已知类别开始，因此在此示例中，两个oov桶的索引为5和6。

为什么要使用oov桶？如果类别数量很大（例如邮政编码、城市、单词、产品或用户）并且数据集也很大，或者它们一直在变化，那么得到类别的完整列表可能不是很方便。一种解决方法是基于数据样本（而不是整个训练集）定义词汇表，并为不在数据样本中的其他类别添加一些桶。你希望在训练期间找到的类别越多，就应该使用越多的oov桶。

如果没有足够的oov桶，就会发生冲突：不同的类别最终会出现在同一个桶中，因此神经网络将无法区分它们（至少不是基于这个特征）。

现在，让我们使用查找表将一小批分类特征编码为独热向量：

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=514, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> cat_one_hot = tf.one_hot(cat_indices, depth=len(vocab) + num_oov_buckets)
>>> cat_one_hot
<tf.Tensor: id=524, shape=(4, 7), dtype=float32, numpy=
array([[0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.]], dtype=float32)>
```

如你所见，"NEAR BAY"被映射到索引3，未知类别"DESERT"被映射到两个oov桶之一（在索引5），而"INLAND"被映射到索引1，两次。然后我们使用tf.one_hot（）对这些索引进行独热编码。注意我们必须告诉该函数索引的总数，该总数等于词汇表大小加上oov桶的数量。现在你知道如何使用TensorFlow把分类特征编码为独热向量！

就像前面一样，将所有这些逻辑捆绑到一个包含所有的类中并不是很困难。它的adapt（）方法使用数据样本并提取其包含的所有不同类别。它创建一个查找表将每个类别映射到其索引（包括使用oov桶的未知类别）。然后其call（）方法使用查找表将输入类别映射到其索引。还有一个更好的消息：当你阅读本书时，Keras可能会包含一个名为keras.layers.TextVectorization的层，该层能够准确地做到这一点：它的adapt（）方法从数据样本中提取词汇表，而其call（）方法会将每个类别转换为其词汇表中的索引。如果要将这些索引转换为独热向量，则可以在模型的开头添加此层，然后添加将应用tf.one_hot（）函数的Lambda层。

这可能不是最佳解决方法。每个独热向量的大小是词汇表长度加上oov桶数。当只有几个可能的类别时这很可行，但是如果词汇表很大，则使用嵌入对它们进行编码会更加有效。

 根据经验，如果类别数少于10，则通常采用独热编码方式。（但数字可能会有所不同！）如果类别数大于50（通常这种情况需要使用哈希桶），通常最好使用嵌入。在10到50个类别中，你可能需要尝试两种方法，然后看看哪种最适合你。

13.3.2 使用嵌入编码分类特征

嵌入是表示类别的可训练密集向量。默认情况下，嵌入是随机初始化的，例如，"NEAR BAY"类别最初可以由诸如[0.131，0.890]的随机向量表示，而"NEAR OCEAN"类别可以由[0.631，0.791]表示。在此示例中，我们使用2D嵌入，但是维度是可以调整的超参数。由于这些嵌入是可训练的，因此它们在训练过程中会逐步改善。由于它们代表的类别相当相似，“梯度下降”肯定最终会把它们推到接近的位置，而把它们推离"INLAND"类别的嵌入（见图13-4）。实际上，表征越好，神经网络就越容易做出准确的预测，因此训练使嵌入成为类别的有用表征。这称为表征学习（我们将在第17章中看到其他类型的表征学习）。

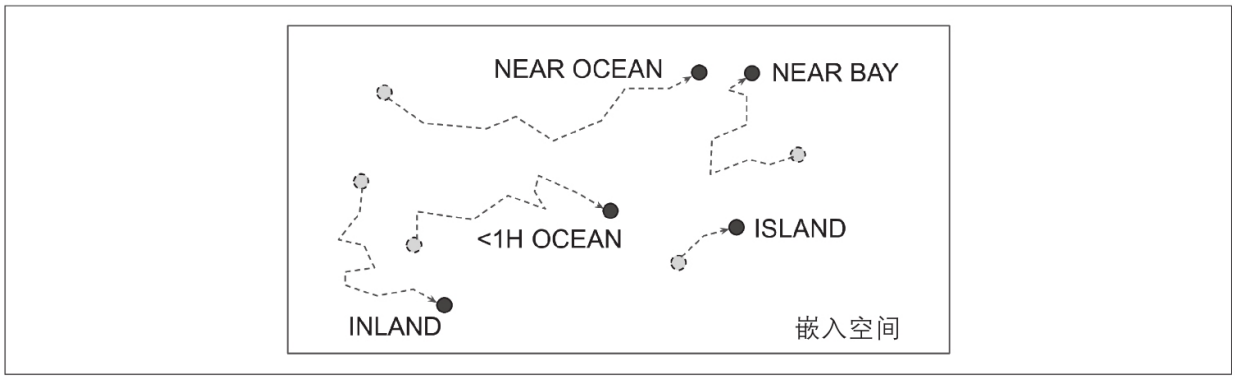


图13-4：训练期间嵌入会逐渐改善

嵌入通常不仅是当前任务的有用表示，而且很多时候这些相同的嵌入可以成功地重用于其他任务。最常见的示例是词嵌入（即单个单词的嵌入）：在执行自然语言处理任务时，与训练自己的词嵌入相比，重用预先训练的词嵌入通常效果会更好。

使用向量来表示词的想法可以追溯到20世纪60年代，许多复杂的技术已被用来生成有用的向量，包括使用神经网络。但是事情真正在2013年取得了成功，当时TomášMikolov和其他Google研究人员发表了一篇论文[1]，描述了一种使用神经网络学习词嵌入的有效技术，大大优于以前的尝试。这使他们能够在非常大的文本语料库上学习嵌入：他们训练了一个神经网络来预测任何给定单词附近的单词，并获得了惊人的词嵌入。例如，同义词具有非常接近的嵌入，法国、西班牙和意大利等与语义相关的词最终聚类在一起。

但是这不仅与邻近性有关：词嵌入还沿着嵌入空间中有意义的轴进行组织。这是一个著名的示例：如果计算King-Man+Woman（添加和减去这些单词的嵌入向量），则结果非常接近Queen单词的嵌入（见图13-5）。换句话说，词嵌入编码了性别的概念！同样，你可以计算Madrid-Spain+France，其结果接近Paris（巴黎），这似乎表明首都的概念也在嵌入中进行了编码。

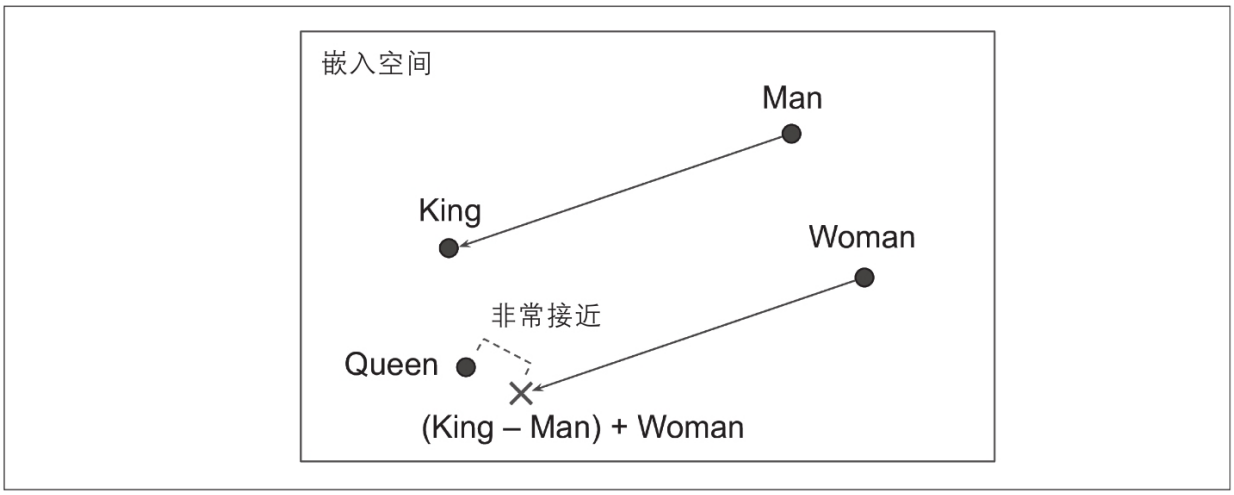


图13-5：相似词的词嵌入趋于接近，某些轴似乎编码了有意义的概念

不幸的是，词嵌入有时会捕获我们最严重的偏见。例如，尽管它们正确地学习到男人是国王，女人是女王，但它们似乎也学习到了男人是医生，而女人是护士：这是一种性别歧视！公平地说，正如Malvina Nissim等人在2019年的论文[2]中指出的那样，这个特定示例可能被夸大了。不管怎样，确保深度学习算法的公平性是重要且活跃的研究课题。

让我们看一下如何手动实现嵌入以了解它们的工作原理（然后我们将使用一个简单的Keras层）。首先我们需要创建一个包含每个类别嵌入的嵌入矩阵，并随机初始化。每个类别和每个oov桶都有一行，每个嵌入维度都有一列：

```
embedding_dim = 2
embed_init = tf.random.uniform([len(vocab) + num_oov_buckets, embedding_dim])
embedding_matrix = tf.Variable(embed_init)
```

在此示例中，我们使用2D嵌入，但是根据经验，嵌入通常有10到300个维度，具体取决于任务和词汇表（你可以调整此超参数）。

该嵌入矩阵是一个随机的6×2矩阵，存储在一个变量中（因此可以在训练过程中通过梯度下降对其进行调整）：

```
>>> embedding_matrix
<tf.Variable 'Variable:0' shape=(6, 2) dtype=float32, numpy=
array([[0.6645621 , 0.44100678],
       [0.3528825 , 0.46448255],
       [0.03366041, 0.68467236],
       [0.74011743, 0.8724445 ],
       [0.22632635, 0.22319686],
       [0.3103881 , 0.7223358 ]], dtype=float32)>
```

让我们对与之前相同的分类特征进行编码，但是这次使用这些嵌入：

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=741, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> tf.nn.embedding_lookup(embedding_matrix, cat_indices)
<tf.Tensor: id=864, shape=(4, 2), dtype=float32, numpy=
array([[0.74011743, 0.8724445 ],
       [0.3103881 , 0.7223358 ],
       [0.3528825 , 0.46448255],
       [0.3528825 , 0.46448255]], dtype=float32)>
```

tf.nn.embedding_lookup（）函数以给定的索引查找在嵌入矩阵中的行，这就是它所做的全部。例如，查找表说"INLAND"类别位于索引1，因此tf.nn.embedding_lookup（）函数返回嵌入矩阵中第1行的嵌入（两次）：[0.3528825，0.46448255]。

Keras提供了一个keras.layers.Embedding层来处理嵌入矩阵（默认情况下是可训练的）。创建层时，它将随机初始化嵌入矩阵，然后使用某些类别索引进行调用时，它将返回嵌入矩阵中这些索引处的行：

```
>>> embedding = keras.layers.Embedding(input_dim=len(vocab) + num_oov_buckets,
...                                    output_dim=embedding_dim)
...
>>> embedding(cat_indices)
<tf.Tensor: id=814, shape=(4, 2), dtype=float32, numpy=
array([[ 0.02401174,  0.03724445],
       [-0.01896119,  0.02223358],
       [-0.01471175, -0.00355174],
       [-0.01471175, -0.00355174]], dtype=float32)>
```

把所有内容放在一起，我们现在可以创建一个Keras模型，该模型可以处理分类特征（以及常规的数值特征）并学习每个类别（以及每个oov桶）的嵌入：

```
regular_inputs = keras.layers.Input(shape=[8])
categories = keras.layers.Input(shape=[], dtype=tf.string)
cat_indices = keras.layers.Lambda(lambda cats: table.lookup(cats))(categories)
cat_embed = keras.layers.Embedding(input_dim=6, output_dim=2)(cat_indices)
encoded_inputs = keras.layers.concatenate([regular_inputs, cat_embed])
```

```
outputs = keras.layers.Dense(1)(encoded_inputs)
model = keras.models.Model(inputs=[regular_inputs, categories],
                           outputs=[outputs])
```

该模型有两个输入：一个常规输入（每个实例包含8个数字特征），以及一个分类输入（每个实例包含一个分类特征）。它使用Lambda层查找每个类别的索引，然后查找这些索引的嵌入。接下来它将嵌入和常规输入合并起来以提供已编码的输入，这些输入准备好被馈送到神经网络。此时我们可以添加任何种类的神经网络，但是我们只是添加了一个密集输出层，而且我们已经创建了Keras模型。

当keras.layers.TextVectorization层可用时，你可以调用其adapt（）方法以使其从一个数据样本中提取词汇表（它会为你创建查找表）。然后你可以将其添加到模型中，它会执行索引查找（替换之前代码示例中的Lambda层）。

 后面有一个Dense层（不具有激活函数和偏置）的独热编码等同于Embedding层。但是，Embedding层使用较少的计算方式（当嵌入矩阵的大小增加时，性能差异会变得明显）。Embedding层的权重矩阵和嵌入矩阵的作用相同。例如，使用大小为20的独热向量和具有10个单位的Embedding层等效于使用input_dim=20和output_dim=10的嵌入层。因此可见，使用比Embedding层后面的层中的单元数更多的嵌入维度是浪费的。

现在让我们更仔细地研究Keras预处理层。

13.3.3　Keras预处理层

TensorFlow团队正在努力提供一组标准的Keras预处理层。在你阅读本书时，可能可以使用它；但是届时API可能会稍有变化，因此如果有任何意外，请参考本章的notebook。这个新的API可能会取代现有的

Feature Columns API，该API较难使用且不直观（如果你想了解有关Feature Columns API的更多信息，请参阅本章的notebook）。

我们已经讨论了其中的两层：keras.layers.Normalization层——执行特征标准化（相当于我们之前定义的Standardization层）；TextVectorization层——能够将输入中的每个单词编码为它在词汇表中的索引。在这两种情况下，你创建层，并使用数据样本调用其adapt（）方法，然后在模型中正常使用该层。其他预处理层遵循相同的模式。

该API还将包括一个keras.layers.Discretization层，它将连续的数据切成不同的离散块，并将每个块编码为一个独热向量。例如，你可以使用它把价格离散化为低、中、高三个类别，分别将其编码为[1，0，0]、[0，1，0]和[0，0，1]。当然这会丢失很多信息，但是在某些情况下，它可以帮助模型检测那些在观察连续值时不那么明显的模式。

 Discretization层不可微分，应该仅在模型开始时使用。实际上，模型的预处理层在训练过程中冻结，因此它们的参数不受梯度下降的影响，因此不需要可微分。这也意味着，如果你希望它可训练的话，则不要在自定义预处理层中直接使用Embedding层：相反，应像前面的代码示例一样，将其单独添加到你的模型中。

你也可以使用PreprocessingStage类链接多个预处理层。例如，以下代码将创建一个预处理流水线，该流水线首先把输入归一化，然后离散化（这可能会使你想起Scikit-Learn流水线）。在使该流水线适应一个数据样本之后，你可以像在模型中的常规层一样使用它（但同样只能在模型的开头使用，因为它包含不可微的预处理层）：

```
normalization = keras.layers.Normalization()
discretization = keras.layers.Discretization([...])
pipeline = keras.layers.PreprocessingStage([normalization, discretization])
pipeline.adapt(data_sample)
```

TextVectorization层还可以选择输出单词计数向量，而不是单词索引。例如，如果词汇表包含三个单词：["and"，"basketball"，"more"]，则文本"more and more"将映射到向量[1、0、2]：单词"and"出现一次，单词“basketball”根本不出现，而单词"more"出现两次。这种文本表示形式称为词袋，因为它完全失去了单词的顺序。大多数情况下，像"and"之类的常用词在大多数文本中都具有很大的值，即使它们通常是最无趣的（例如，在"more and more basketball"文本中，"basketball"一词显然是最重要的，因为它不是一个很常见的词）。因此应该用减少常用单词重要性的方式对单词计数进行归一化。一种常见的方法是将每个单词计数除以出现单词的训练实例总数的对数。此技术称为术语频率×反文档频率（（TermFrequency×Inverse-Document-Frequency（TF-IDF））。例如，让我们假设单词"and""basketball"和"more"分别出现在训练集中的200、10和100个文本实例中：在这种情况下，最终向量将为[1/log（200），0/log（10），2/log（100）]，大约等于[0.19，0.，0.43]。TextVectorization层有（可能）选项可以执行TF-IDF。

 如果标准预处理层不足以完成你的任务，那你仍然可以选择创建自己的自定义预处理层，就像我们之前对Standardization类所做的一样。创建keras.layers.PreprocessingLayer类的子类，该子类有adapt（）方法，还有data_sample参数和一个可选的reset_state参数：如果为True，则adapt（）方法在计算新状态之前重置任何现有的状态，如果为False，则应尝试更新已存在的状态。

如你所见，这些Keras预处理层使预处理更加容易！现在无论你是选择编写自己的预处理层还是使用Keras的预处理层（甚至使用Feature Columns API），所有预处理过程都是即时进行的。但是在训练过程中，最好提前进行预处理。让我们来看看为什么要这样做以及如何去做。

[1] Tomas Mikolov et al.，“Distributed Representations of Words and Phrases and Their Compositionality”，Proceedings of the 26th International Conference on Neural Information Processing Systems 2（2013）：3111-3119.
[2] Malvina Nissim et al.，“Fair Is Better Than Sensational：Man Is to Doctor as Woman Is to Doctor”，arXiv preprint arXiv：1905.09866（2019）.

13.4 TF Transform

如果预处理是计算密集的，那么在训练之前（而不是即时）进行处理会大大提高速度：在训练之前，每个实例对数据进行一次预处理，而不是在训练期间每个实例进行一次数据预处理。如前所述，如果数据集足够小而适合于RAM，则可以使用cache（）方法。但是如果数据集太大，那么Apache Beam或Spark这样的工具会有所帮助。它们可以使你对大量数据（甚至分布在多个服务器上）运行有效的数据处理流水线，因此你可以使用它们在训练之前对所有训练数据进行预处理。

这很好用，确实可以加快训练速度，但是有一个问题：训练完模型后，假设你想把它部署到移动应用程序中。在这种情况下，你需要在应用程序中编写一些代码来预处理数据，然后再将其馈送到模型中。假设你还想把模型部署到TensorFlow.js中，使它可以在Web浏览器中运行，则需要再次编写一些预处理代码。这可能会成为维护方面的噩梦：每当你要修改预处理逻辑时，就需要更新Apache Beam代码、移动应用程序代码和JavaScript代码。这不仅浪费时间，而且容易出错：最终会在训练之前执行的预处理操作与在应用程序或浏览器中执行的预处理操作之间存在细微的差异。这种训练/服务偏差会导致错误或性能下降。

一种改进方法是采用经过训练的模型（采用由Apache Beam或Spark代码预处理过的数据进行训练），并在将其部署到你的应用程序或浏览器之前，添加额外的预处理层来进行实时预处理。那肯定会更好一些，因为现在你有两个版本的预处理代码：Apache Beam代码或Spark代码以及预处理层的代码。

但是如果你只能定义一次预处理操作该怎么办？这就是TF Transform设计的目的。它是TensorFlow Extended（TFX）的一部分，

TFX是用于在生产环境中部署TensorFlow模型的端到端平台。首先要使用诸如TF Transform之类的TFX组件，必须先安装它。它没有与TensorFlow捆绑在一起。通过使用TF Transform函数进行缩放、存储等操作，你只需定义一次预处理函数（在Python中）。你还可以使用所需的任何TensorFlow操作。如果我们只有两个特征，则此预处理函数如下所示：

```
import tensorflow_transform as tft

def preprocess(inputs):  # inputs = a batch of input features
    median_age = inputs["housing_median_age"]
    ocean_proximity = inputs["ocean_proximity"]
    standardized_age = tft.scale_to_z_score(median_age)
    ocean_proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)
    return {
        "standardized_median_age": standardized_age,
        "ocean_proximity_id": ocean_proximity_id
    }
```

接下来，TF Transform允许你使用Apache Beam将preprocess（）函数应用于整个训练集（它提供了AnalyzeAndTransformDataset类，你可以在Apache Beam流水线中使用该类）。在此过程中，它还会计算整个训练集上所有必要的统计信息：在此示例中指housing_median_age特征的均值和标准差以及ocean_proximity特征的词汇表。计算这些统计信息的组件称为分析器。

重要的是，TF Transform还将生成等效的TensorFlow函数，你可以将其插入部署的模型中。该TF函数包含一些常数，这些常数与Apache Beam计算的所有必要的统计信息（均值、标准差和词汇表）相对应。

借助Data API、TFRecords、Keras预处理层和TF Transform，你可以构建高度可扩展的输入流水线来进行训练，并从生产环境中的快速而便携的数据预处理中受益。

但是如果你只想使用标准数据集怎么办？在这种情况下，事情要简单得多：只需使用TFDS！

13.5 TensorFlow数据集项目

TensorFlow数据集（TFDS）项目使下载通用数据集变得非常容易，从小型数据集（如MNIST或Fashion MNIST）到大型数据集（如ImageNet）（你需要大量的磁盘空间）。该列表包括图像数据集、文本数据集（包括翻译数据集）以及音频和视频数据集。你可以访问https://homl.info/tfds来查看完整列表以及每个数据集的描述。

TFDS没有与TensorFlow捆绑在一起，因此你需要安装tensorflow_datasets库（例如使用pip）。然后调用tfds.load（）函数，它会下载你想要的数据（除非之前已经下载过），并将该数据作为数据集的目录返回（通常一个用于训练，另一个用于测试，但这取决于你选择的数据集）。例如，让我们下载MNIST：

```
import tensorflow_datasets as tfds

dataset = tfds.load(name="mnist")
mnist_train, mnist_test = dataset["train"], dataset["test"]
```

然后，你可以应用所需的任何转换（通常是乱序、批处理和预取），并且准备好训练你的模型。以下是一个简单的示例：

```
mnist_train = mnist_train.shuffle(10000).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    [...]
```

load（）函数对下载的每个数据碎片进行乱序（仅针对训练集）。这可能还不够，所以最好对训练数据进行多次乱序。

请注意，数据集中的每个项目都是包含特征和标签的字典。但是Keras希望每个项目都是一个包含两个元素（同样是特征和标签）的元组。你可以使用map（）方法转换数据集，如下所示：

```
mnist_train = mnist_train.shuffle(10000).batch(32)
mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))
mnist_train = mnist_train.prefetch(1)
```

但是通过设置as_supervised=True来使load（）函数执行此操作会更简单（显然这仅适用于带标签的数据集）。你也可以根据需要指定批处理大小。然后你可以将数据集直接传递给tf.keras模型：

```
dataset = tfds.load(name="mnist", batch_size=32, as_supervised=True)
mnist_train = dataset["train"].prefetch(1)
model = keras.models.Sequential([...])
model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd")
model.fit(mnist_train, epochs=5)
```

本章技术性很强，你可能会觉得它与神经网络的抽象美有些不同，但事实是，深度学习通常涉及大量数据，知道如何高效地加载、解析和预处理是一项至关重要的技能。在第14章中，我们将介绍卷积神经网络，它们是用于图像处理和许多其他应用的最成功的神经网络架构。

13.6 练习题

1.为什么要使用Data API？

2.将大数据集拆分为多个文件有什么好处？

3.在训练过程中，如何分辨输入流水线是瓶颈？你可以如何解决？

4.你可以将任何二进制数据保存到TFRecord文件，还是仅保存序列化的协议缓冲区？5.为什么要将所有数据转换为Example protobuf格式？为什么不使用自己的protobuf定义？

6.使用TFRecords时，应何时激活压缩？为什么不系统地进行呢？

7.对数据进行预处理，可以在编写数据文件时，或者在tf.data流水线内，或者在模型内的预处理层中，或使用TF Transform。你能否列出每种方法的一些利弊？

8.列举一些可用于编码分类特征的常用技术。文本呢？

9.加载Fashion MNIST数据集（在第10章中介绍）；将其分为训练集、验证集和测试集；打乱训练集次序；将每个数据集保存到多个TFRecord文件中。每个记录都应该是具有两个特征的序列化Example protobuf：序列化的图像（使用tf.io.serialize_tensor（）序列化每个图像）和标签[1]。然后使用tf.data为每个集合创建一个有效的数据集。最后，使用Keras模型训练这些数据集，包括预处理层来标准化每个输入特征。尝试使输入流水线尽可能高效，使用TensorBoard来可视化数据分析。

10.在本练习中，你将下载一个数据集，对其进行拆分，创建一个tf.data.Dataset来加载并对其进行有效的预处理，然后构建和训练一个包含Embedding层的二进制分类模型：

a.下载大型电影评论数据集，其中包含来自Internet电影数据库的50 000条电影评论。数据有train和test两个目录，每个目录包含一个带有12 500个肯定评论的pos子目录和一个带有12 500个否定评论的neg子目录。每个评论均存储在单独的文本文件中。还有其他文件和文件夹（包括预处理的词袋），但是在本练习中我们将忽略它们。

b.将测试集分为一个验证集（15 000）和一个测试集（10 000）。

c.使用tf.data为每个集合创建一个有效的数据集。

d.创建一个二进制分类模型，使用TextVectorization层对每个评论进行预处理。如果TextVectorization层尚不可用（或者你想挑战），请尝试创建自定义预处理层：你可以使用tf.strings包中的函数，例如，lower（）来使所有内容变为小写，regex_replace（）将标点符号替换为空格，split（）在空格上拆分单词。你应该使用查找表输出单词索引，该索引必须在adapt（）方法中准备。

e.添加一个Embedding层并计算每个评论的平均嵌入值，乘以单词数的平方根（见第16章）。然后将这种重新缩放的均值嵌入传递给模型的其余部分。

f.训练模型，看看可以得到什么准确率。尝试优化你的流水线以使训练速度更快。

g.使用TFDS可以更轻松地加载相同的数据集：tfds.load（"imdb_reviews"）。

这些练习题的解答在附录A中提供。

[1] 对于大图像，你可以改用tf.io.encode_jpeg（）。这样可以节省很多空间，但是会降低图像的质量。

第14章　使用卷积神经网络的深度计算机视觉

虽然早在1996年IMB的深蓝计算机就击败了世界围棋冠军Garry Kasparov，但是直到现在为止，计算机也很难完成一些看似很简单的任务，比如检测出图片中的小狗或者识别语言。为什么这些工作在我们人类看来毫不费力呢？答案是：感知主要发生在我们的意识之外，在大脑专门的视觉、听觉和其他感官模块中。当感知信息到达意识时，它已经被高层次的特征修饰过了。例如，当你看见一张可爱的小狗的照片时，你不能选择不看小狗或者忽视它的可爱。你不能解释你是如何识别一只可爱的小狗，这对你来说只是显而易见的事情。所以，不能相信我们的主观经验：感知根本不是微不足道的事情。要了解它，我们必须着眼于感知模块是如何运作的。

卷积神经网络（CNN）起源于对大脑的视觉皮层的研究，从20世纪80年代起被用于图像识别。在过去几年中，由于计算机计算能力的提高、可用训练数据数量的增加，以及第11章提到的用于深度网络训练技巧的增加，CNN已经在一些复杂的视觉任务中实现了超人性化，广泛用于图片搜索服务、自动驾驶汽车、自动视频分类系统等。此外，不局限于视觉感知，CNN也成功用于其他任务，比如：语音识别或自然语言处理（NLP）。然而，我们现在将专注于视觉应用。

在本章中，我们将探讨CNN的来源、其构建基块以及如何使用TensorFlow和Keras实现它们。然后我们将讨论一些最好的CNN架构以及其他视觉任务，包括物体检测（对图像中的多个物体进行分类并在其周围放置边框）和语义分割（根据像素的类别对每个像素进行分类）。

14.1　视觉皮层的架构

David H.Hubel和Torsten Wiesel在1958年[1]和1959年[2]利用猫做了一系列实验，（在随后的几年又利用猴子做过实验[3]），对视觉皮层的结构提出了重要见解（该成果使作者获得了1981年的诺贝尔生理学或医学奖）。另外，他们指出视觉皮层神经元有一个小的局部接受野，这就意味着它们只对视野的局部区域内的视觉刺激做出反应（见图14-1，其中5个虚线圆圈代表局部接受野）。不同神经元的接受野有可能会重复，它们一起平铺在整个视觉区域中。

此外，他们指出一些神经元作用于图片的水平方向，而另一些神经元作用于其他方向（两个神经元可能有相同的接受野，但是作用于不同方向。）他们也注意到有些神经元有比较大的接受野，它们作用于由多个低阶模式组成的复杂模式。这个发现可以推测出，高阶神经元基于相邻的低阶神经元的输出（见图14-1，每个神经元只跟上一层的少数神经元连接）。这种强大的组织结构可以检测到视觉区域内的所有复杂模式。

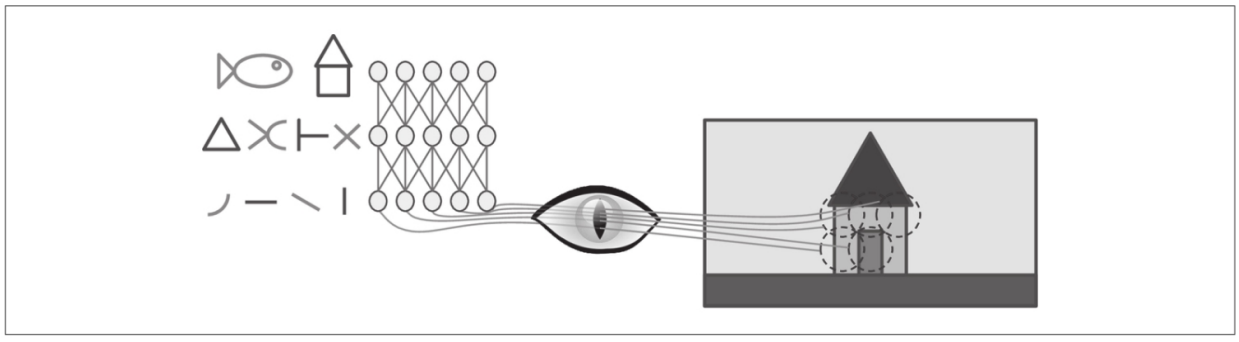


图14-1：视皮层中的生物神经元对视野中称为接受野的小区域中的特定模式作出反应，当视觉信号通过连续的大脑模块进入时，神经元会在更大的接受野中对更复杂的模式做出响应。

这些关于视觉皮层的研究影响了1980年引入的新认知机[4]，然后逐步演变成我们说的卷积神经网络。一个重要的里程碑是Yann LeCun等

人发表于1998年的论文[5]。该论文介绍了广泛用于识别手写支票号码的著名LeNet-5架构。该架构除了一些广为人知的构建块，例如全连接层和S型激活函数，还引入了两个新的构建块：卷积层和池化层。我们现在就开始学习它们吧。

 为什么不简单地使用具有全连接层的深度神经网络来执行图像识别任务呢？不幸的是，尽管这对于较小的图像（例如MNIST）效果很好，但由于需要大量的参数，因此对于较大的图像无能为力。例如，一个100×100像素的图像有10 000个像素，如果第一层只有1000个神经元（已经严重限制了传输到下一层的信息量），则意味着总共有1000万个连接。那只是第一层。CNN使用部分连接层和权重共享解决了此问题。

[1] David H.Hubel，“Single Unit Activity in Striate Cortex of Unrestrained Cats”，The Journal of Physiology 147（1959）：226－238.
[2] David H.Hubel and Torsten N.Wiesel，“Receptive Fields of Single Neurons in the Cat’s Striate Cortex”，The Journal of Physiology 148（1959）：574－591.
[3] David H.Hubel and Torsten N.Wiesel，“Receptive Fields and Functional Architecture of Monkey Striate Cortex”，The Journal of Physiology 195（1968）：215-243.
[4] Kunihiko Fukushima，“Neocognitron：A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position”，Biological Cybernetics 36（1980）：193－202.
[5] Yann LeCun et al.，“Gradient-Based Learning Applied to Document Recognition”，Proceedings of the IEEE 86，no.11（1998）：2278-2324.4

14.2 卷积层

CNN的最重要的构建块是卷积层[1]：第一卷积层的神经元不会连接到输入图像中的每个像素（如前文所述），而只与其接受野内的像素相连接（见图14-2）。反过来，第二卷积层的每个神经元仅连接到位于第一层中的一个小矩阵内的神经元。这种架构允许网络集中在第一个隐藏层的低阶特征中，然后在下一个隐藏层中将它们组装成比较高阶的特征，等等。CNN在图像识别方面效果很好的原因之一是这种层次结构在现实世界的图像中很常见。

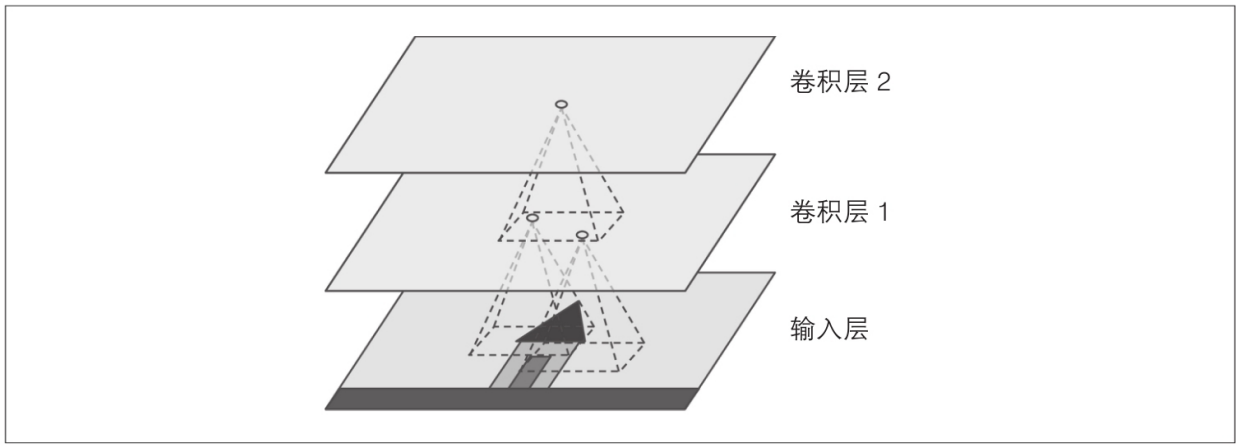


图14-2：具有矩形局部接受野的CNN层

 到目前为止，我们研究的所有多层神经网络都具有由一长列神经元组成的层，我们必须将输入图像展平为一维，然后再将其输入到神经网络中。在CNN中，每一层都以2D表示，这使得将神经元与其相应的输入进行匹配变得更加容易。

位于给定层的第i行第j列的神经元连接到位于第i到$i+f_h-1$行、第j到$j+f_w-1$列的前一层中的神经元的输出，其中f_h和f_w是接受野的高度和宽度（见图14-3）。为了使层的高度和宽度与上一层相同，通常在输入周围添加零，如图所示。这称为零填充。

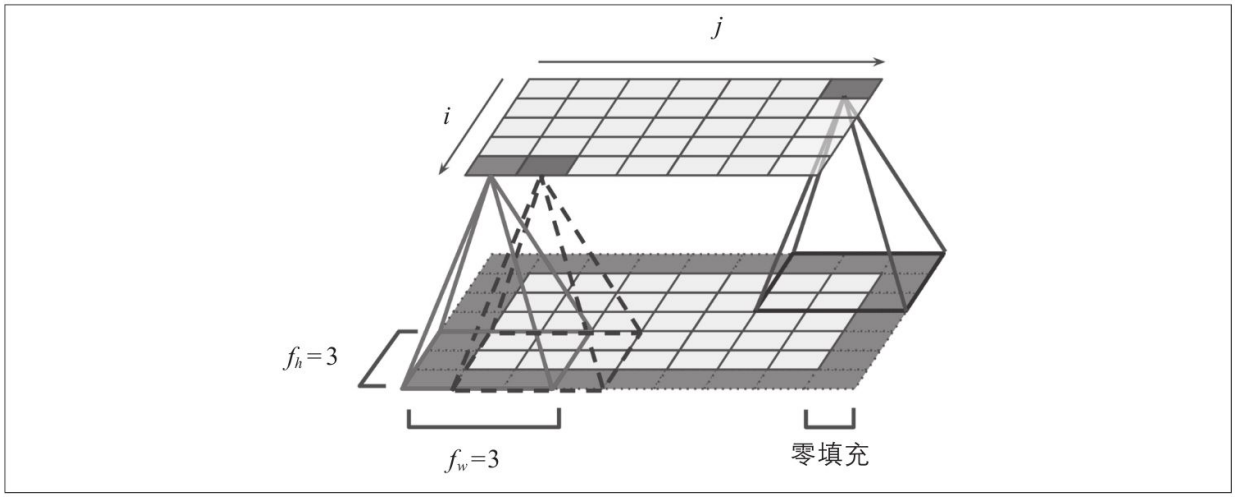


图14-3：层与零填充之间的连接

如图14-4所示，也可以通过隔出接受野的方式来将较大的输入层连接到较小的层。这大大降低了模型的计算复杂度。从一个接受野移到另一个接受野的距离称为步幅。在该图中，使用3×3的接受野和为2的步幅（在此示例中步幅相同，但不一定如此），将5×7的输入层（加上零填充）连接到3×4的层。位于上层第i行、第j列的神经元与位于第$i\times s_h$到$i\times s_h+f_h-1$行、第$j\times s_w$到$j\times s_w+f_w-1$列的上一层神经元的输出连接，其中s_h和s_w是垂直步幅和水平步幅。

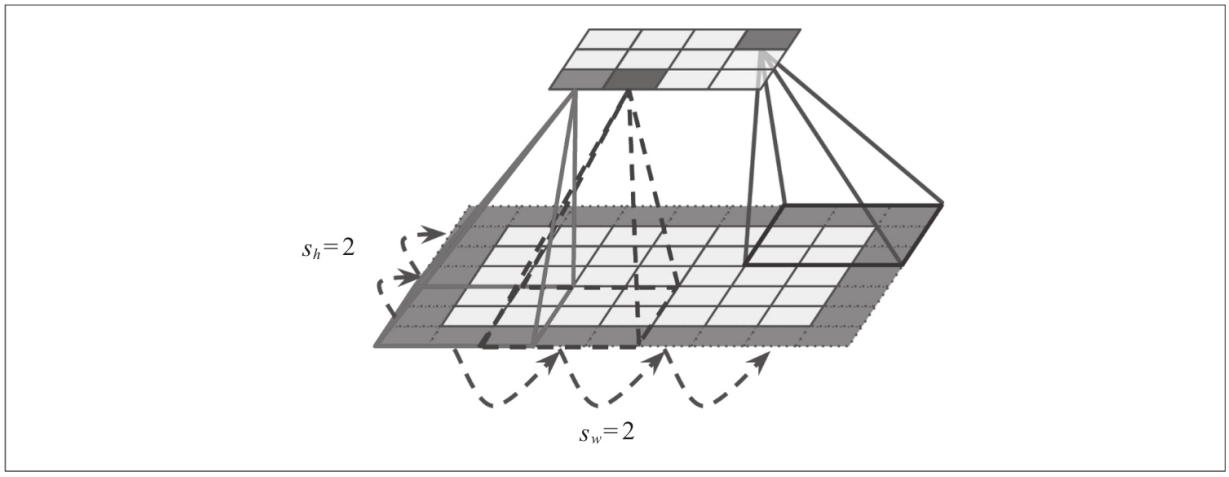


图14-4：使用步幅为2来降低维度

14.2.1 滤波器

神经元的权重可以表示为一幅小图像，其大小相当于接受野的大小。例如，图14-5显示了两个可能的权重集，称为滤波器（或卷积核）。第一个表示一个黑色的正方形，中间有一条垂直的白线（这是一个7×7的矩阵，全为0，除了中间的列（全为1））。使用这些权重的神经元将忽略其接受野中除中心垂直线以外的所有内容（因为所有输入都将乘以0，位于中心垂直线中的输入除外）。第二个滤波器是一个黑色的正方形，中间有一条水平的白线。使用这些权重的神经元将再次忽略其接受野中除中心水平线以外的所有内容。

现在，如果层中的所有神经元都使用相同的垂直线滤波器（和相同的偏置项），并且向网络输入图14-5中所示的输入图像（底部图像），则该层将输出左上方的图像。请注意，垂直白线得到增强，而其余部分变得模糊。类似地，如果所有神经元都使用相同的水平线滤波器，则将获得右上方的图像。水平的白线得到增强，而其余部分被模糊掉。因此，使用相同滤波器的充满神经元的层会输出一个特征图，该图突出显示图像中最激活滤波器的区域。当然，你不必手动定义滤波器：相反，在训练过程中，卷积层将自动学习对其任务最有用的滤波器，而上面的层将学习把它们组合成更复杂的模式。

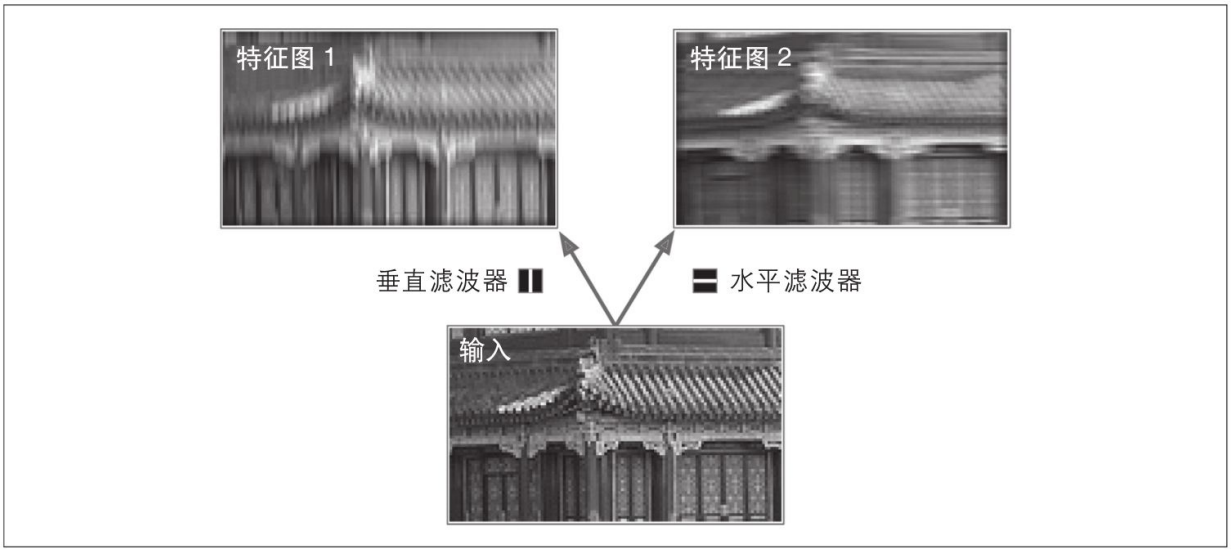


图14-5：应用两个不同的滤波器得到两个特征图

14.2.2　堆叠多个特征图

到目前为止，为简单起见，我将每个卷积层的输出表示为2D层，但实际上卷积层具有多个滤波器（你可以决定多少个）并为每个滤波器输出一个特征图，因此可以更精确地以3D模式显示（见图14-6）。它在每个特征图中每个像素有一个神经元，给定特征图中的所有神经元共享相同的参数（即相同的权重和偏差项）。不同特征图中的神经元使用不同的参数。神经元的接受野与先前描述的相同，但是它扩展到了所有先前层的特征图。简而言之，卷积层将多个可训练的滤波器同时应用于其输入，从而使其能够检测出输入中任何位置的多个特征。

 特征图中所有的神经元共享相同的参数，大大减少了模型中的参数数量。CNN一旦学会了在一个位置识别模式，就可以在其他任何位置识别模式。相反，常规DNN学会了在一个位置识别模式，它就只能在那个特定位置识别它。

输入图像也由多个子层组成：每个颜色通道一个子层。通常有三种：红色、绿色和蓝色（RGB）。灰度图像只有一个通道，但是某些图像可能具有更多通道，例如有额外光频率（例如红外）的卫星图像。

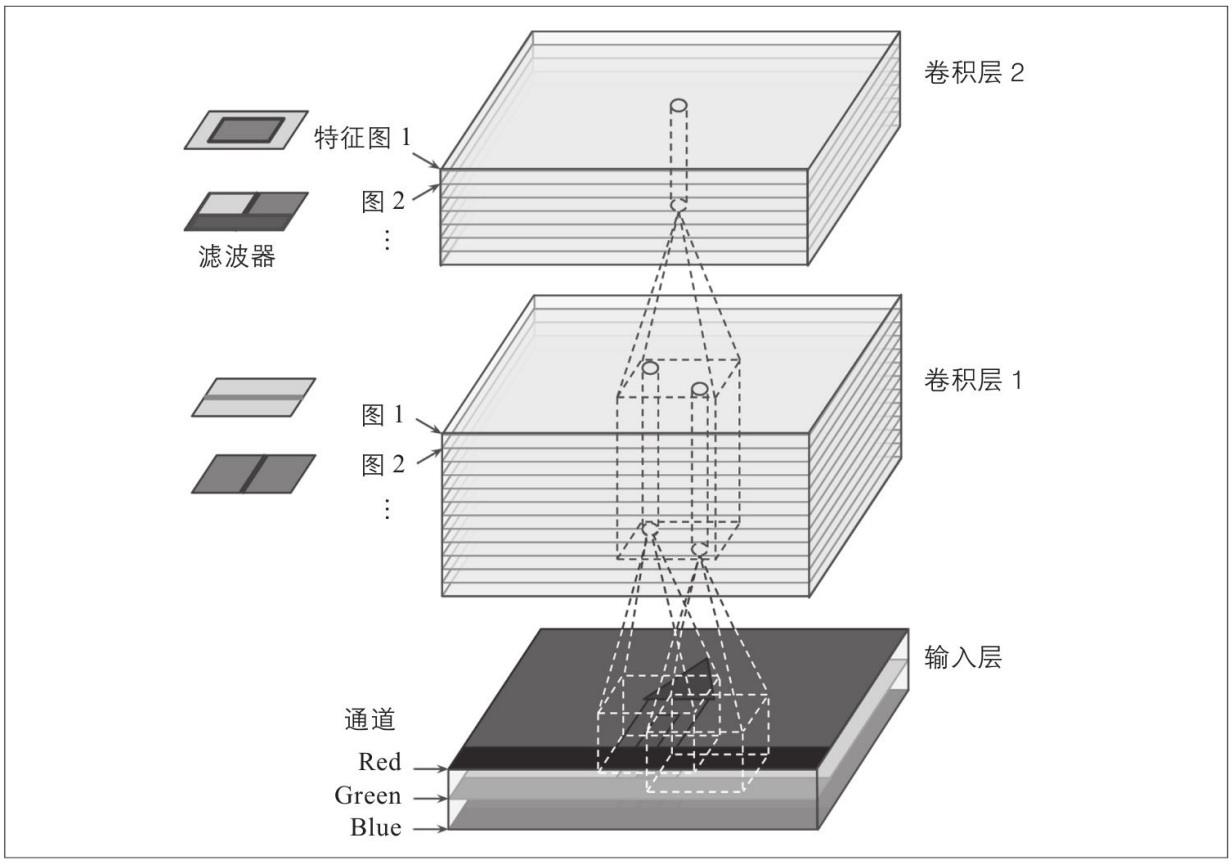


图14-6：具有多个特征图的卷积层以及具有三个颜色通道的图像

具体来说，位于给定卷积层l中位于特征图k的第i行第j列的神经元与位于第$i \times s_h$到$i \times s_h + f_h - 1$行、第$j \times s_w$到$j \times s_w + f_w - 1$列的前一层（l-1）中的神经元的输出相连接，跨越所有的特征图（在l-1层中）。请注意位于同一行i和列j中但位于不同特征图中的所有神经元都连接到上一层中完全相同的神经元的输出。

公式14-1在一个大数学方程式中总结了前面的解释：它显示了在卷积层中如何计算给定神经元的输出。

由于有不同的索引导致有点难看，但是它所做的只是计算所有输入的加权总和，再加上偏置项。

公式14-1：计算卷积层中神经元的输出

$$z_{i,j,k}=b_k+\sum_{u=0}^{f_h-1}\sum_{v=0}^{f_w-1}\sum_{k'=0}^{f_{n'}-1}x_{i',j',k'}\cdot w_{u,v,k',k} \quad \text{其中} \begin{cases} i'=i\times s_h+u \\ j'=j\times s_w+v \end{cases}$$

在此等式中：

• $Z_{i,j,k}$是位于卷积层（第l层）的特征图k中第i行j列中的神经元的输出。

• 如前所述，s_h和s_w是垂直步幅和水平步幅，f_h和f_w是接受野的高度和宽度，$f_{n'}$是上一层（层l-1）中特征图的数量。

• $x_{i',j',k'}$是位于第l-1层、第i'行、第j'列、特征图k'（或通道k'，如果前一层是输入层）的神经元的输出。

• b_k是特征图k（在l层中）的偏置项。你可以将其视为用于调整特征图k整体亮度的旋钮。

• $w_{u,v,k',k}$是层l的特征图k中的任何神经元与其位于u行、v列（相对于神经元的接受野）和特征图k的输入之间的连接权重。

14.2.3 TensorFlow实现

在TensorFlow中，每个输入图像通常表示为形状为[height，width，channels]的3D张量。小批量表示为形状为[mini batch size，height，width，channels]的4D张量。卷积层的权重表示为形状为[f_h，f_w，$f_{n'}$，f_n]的4D张量。卷积层的偏置项简单表示为形状[f_n]的一维张量。

让我们看一个简单的示例。下面的代码使用Scikit-Learn的load_sample_image（）加载两个样本图像（加载两个彩色图像：一个是中国寺庙，另一个是花朵），然后创建两个滤波器并将其应用于两个

图像，最后显示其中一个结果特征图，注意你必须执行pip安装Pillow包来使用load_sample_image（）：

```
from sklearn.datasets import load_sample_image

# Load sample images
china = load_sample_image("china.jpg") / 255
flower = load_sample_image("flower.jpg") / 255
images = np.array([china, flower])
batch_size, height, width, channels = images.shape

# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1  # vertical line
filters[3, :, :, 1] = 1  # horizontal line

outputs = tf.nn.conv2d(images, filters, strides=1, padding="SAME")

plt.imshow(outputs[0, :, :, 1], cmap="gray") # plot 1st image's 2nd feature map
plt.show()
```

让我们看一下这段代码：

• 每个颜色通道的像素强度表示为从0到255的字节，因此我们除以255即可缩放这些特征，得到从0到1的浮点数。

• 然后，我们创建两个7×7的滤波器（一个在中间带有垂直白线，另一个在中间带有水平白线）。

• 我们使用tf.nn.conv2d（）函数应用于两个图像，这是TensorFlow的低层深度学习API的一部分。在此示例中，我们使用零填充（padding="SAME"）和步幅为1。

• 最后，绘制一个结果特征图（类似于图14-5中的右上图）。

tf.nn.conv2d（）行值得更多解释：

• images是输入的小批量（4D张量，如前所述）。

• filter是要应用的一组滤波器（也是4D张量，如前所述）。

• strides等于1，但也可以是包含四个元素的一维度组，其中两个中间元素是垂直步幅和水平步幅（s_h和s_w）。第一个元素和最后一个元素必须等于1。它们可能有一天用于指定批处理步幅（跳过某些实例）和通道步幅（跳过某些上一层的特征图或通道）。

• padding必须为"SAME"或"VALID"：

• 如果设置为"SAME"，则卷积层在必要时使用零填充。将输出大小设置为输入神经元的数量除以步幅（向上取整）所得的值。例如，如果输入大小为13而步幅为5（见图14-7），则输出大小为3（即13/5=2.6，向上舍入为3）。然后根据需要在输入周围尽可能均匀地添加零。当strides=1时，层的输出将具有与其输入相同的空间尺寸（宽度和高度），因此命名为“same”。

• 如果设置为"VA L I D"，则卷积层将不使用零填充，并且可能会忽略输入图像底部和右侧的某些行和列，具体取决于步幅，如图14-7所示（为简单起见，仅这里显示了水平尺寸，当然垂直尺寸也适用相同的逻辑。这意味着每个神经元的接受野都严格位于输入内部的有效位置内（不会超出范围），因此命名为“valid”。

在此示例中，我们手动定义了滤波器，但是在实际的CNN中，你通常将滤波器定义为可训练变量，以便神经网络可以了解哪种滤波器效果最好，如前所述。不用手动创建变量，使用keras.layers.Conv2D层：

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                           padding="same", activation="relu")
```

这段代码使用步幅为1（水平和垂直）和"same"的填充，创建了一个包含32个滤波器（每个3×3）的Conv2D层，并将ReLU激活函数应用于

其输出。如你所见，卷积层具有很多超参数：你必须选择滤波器的数量、其高度和宽度、步幅和填充类型。与往常一样，你可以使用交叉验证来找到正确的超参数值，但这非常耗时。稍后我们将讨论通用的CNN架构，使你了解哪些超参数值在实践中最有效。

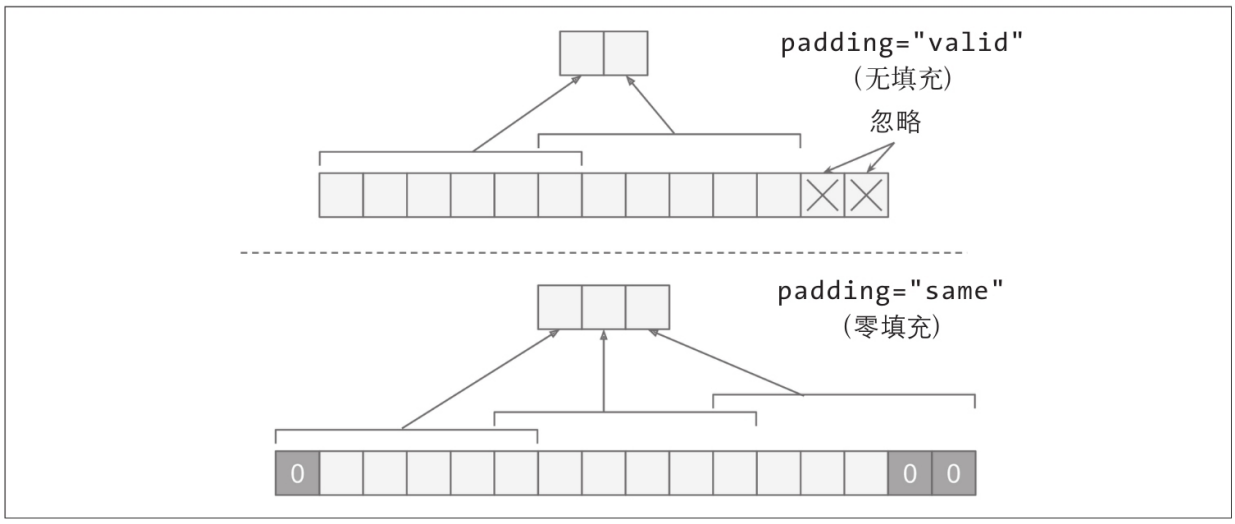


图14-7：Padding为"SAME"或"VALID"（输入宽度为13，滤波器宽度为6，步幅为5）

14.2.4　内存需求

CNN的另一个问题是卷积层需要大量的RAM。在训练期间尤其如此，因为反向传播需要在正向传播过程中计算出的所有中间值。

例如，考虑一个带5×5滤波器的卷积层，输出200个大小为150×100的特征图，步幅为1，填充为"same"。如果输入是150×100 RGB图像（三个通道），则参数的数量为（5×5×3+1）×200=15 200（+1对应于偏置项），与全连接层相比（注：一个具有150×100个神经元的全连接层，每个神经元都连接到所有150×100×3个输入，将有$150^2 \times 100^2 \times 3$=6.75亿个参数！）它很小。但是，200个特征图中的每个特征图都包含150×100个神经元，并且每个神经元都需要计算其5×5×3=75个输入的加权和：总共有2.25亿个浮点乘法。虽然不如全连接层那么糟糕，但是仍然需要大量的计算。此外，如果使用32位浮点数

表示特征图，则卷积层的输出将占用200×150×100×32=96百万位（12 MB）的RAM[2]。这仅用于一个实例——如果训练包含100个实例的批量，则该层将使用1.2 GB的RAM！

在推理期间（即对新实例进行预测时），只要计算了下一层，就可以释放由前一层占用的RAM，因此你只需要两个连续层所需的RAM。但是在训练过程中，需要保留正向传播过程中计算出的所有内容以供反向传播，因此所需的RAM量至少是所有层所需的RAM总量。

 如果由于内存不足而导致训练失败，则可以尝试减小批量的大小。另外你可以尝试使用步幅来减小维度或删除几层。你也可以尝试使用16位浮点数而不是32位浮点数。或者你可以在多个设备上分配CNN。

现在让我们看一下CNN的第二个常见构建块：池化层。

[1] 卷积是一种数学运算，将一个函数在另一个函数上滑动并测量其点乘的积分。它与傅里叶变换和拉普拉斯变换有很深的联系，并在信号处理中大量使用。卷积层实际上使用互相关，这与卷积非常相似（有关更多详细信息，请参见https://homl.info/76）。
[2] 在国际单位制（SI）中，1 MB=1000 KB=1000×1000字节=1000×1000×8位。

14.3 池化层

你一旦了解了卷积层如何工作，池化层就很容易掌握。它们的目标是对输入图像进行下采样（即缩小），以便减少计算量、内存使用量和参数数量（从而降低过拟合的风险）。

就像在卷积层中一样，池化层中的每个神经元都连接到位于一个小的矩形接受野中的上一层中有限数量的神经元的输出。你必须像以前一样定义其大小、步幅和填充类型。但是，池化神经元没有权重。它所做的全部工作就是使用聚合函数（例如最大值或均值）来聚合输入。图14-8显示了最大池化层，它是最常见的池化层类型。在此示例中，我们使用2×2池化内核[1]，步幅为2，没有填充。只有每个接受野中的最大输入值才能进入下一层，而其他输入则被丢弃。例如，在图14-8的左下接受野中，输入值为1、5、3、2，因此只有最大值5传播到下一层。由于步幅为2，因此输出图像的高度为输入图像的一半，宽度为输入图像的一半（由于不使用填充，因此四舍五入）。

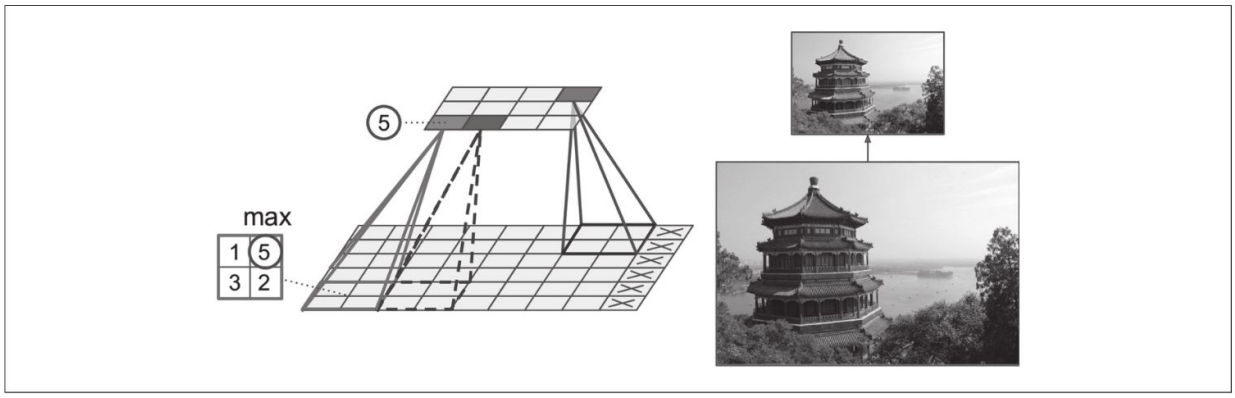


图14-8：最大池化层（2×2池化内核，步幅为2，无填充）


池化层通常独立地作用于每个输入通道，因此输出深度与输入深度相同。

除了减少计算量、内存使用量和参数数量之外，最大池化层还为小变换引入了一定程度的不变性，如图14-9所示。在这里，我们假设亮像素的值比暗像素的值低，并且我们考虑三个图像（A，B，C）经过最大池化层，其内核为2×2，步幅为2。图像B和C与图片A相同，但向右移动一两个像素。如你所见，图像A和B的最大池化层的输出是相同的。这就是变换不变性的含义。对于图像C，输出是不同的：它向右移动了一个像素（但仍然有75%的不变性）。通过在CNN中每隔几层插入一个最大池化层，就可以在更大范围内获得某种程度的变换不变性。而且最大池化提供了少量的旋转不变性和轻微的尺度不变性。在不应该依赖这些细节的预测情况下（例如在分类任务中），这种不变性（即使有局限性）也很有用。

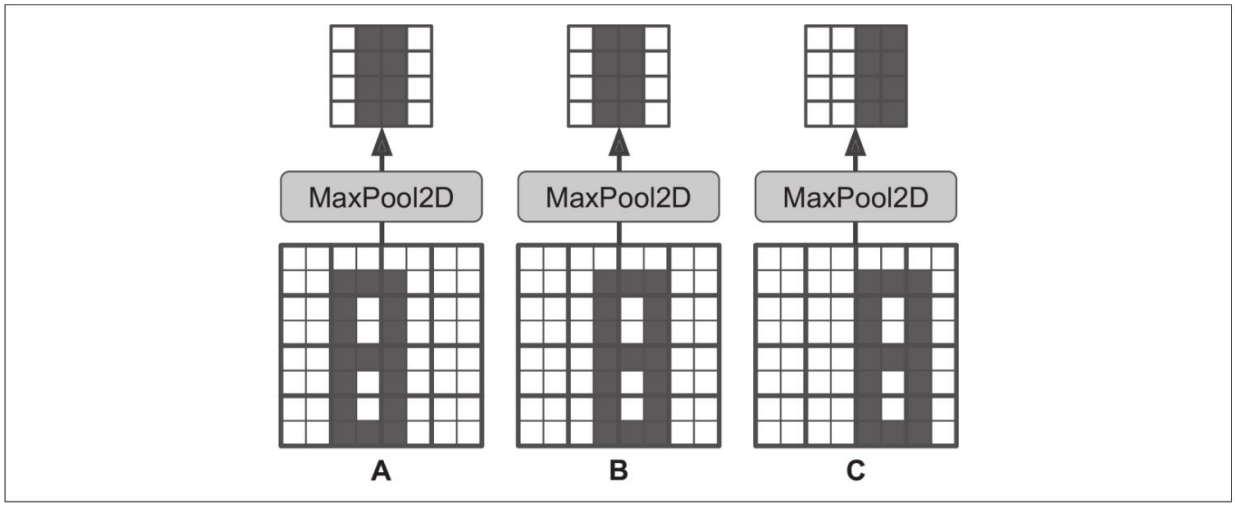


图14-9：小变换的不变性

但是，最大池化也有一些缺点。首先，它显然是非常具有破坏性的，即使使用2×2的小内核并且步幅为2，在两个方向上的输出也将减小到1/2（面积减小到1/4），丢弃了75%输入值。在某些应用中，不变性是不可取的。进行语义分割（根据像素所属的对象对图像中的每个像素进行分类的任务，我们将在本章稍后探讨）：显然，如果输入图像向右平移一个像素，则输出也应向右平移一个像素。在这种情况下，目标是等变性而不是不变性：输入的微小变化应导致输出的相应微小变化。

TensorFlow实现

在TensorFlow中实现最大池化层非常容易。以下代码使用2×2内核创建一个最大池化层。步幅默认为内核大小，因此该层将使用步幅2（水平和垂直）。默认情况下，它使用"valid"填充（即完全不填充）：

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

要创建平均池化层，只需使用AvgPool2D而不是MaxPool2D。如你所料，它的工作原理与最大池化层完全相同，只是它计算的是平均值而不是最大值。平均池化层曾经很受欢迎，但是现在人们通常都使用最大池化层，因为它们通常表现更好。这似乎令人惊讶，因为计算均值通常比计算最大值损失更少的信息。但是另一方面，最大池化仅保留最强的特征，将所有无意义的特征都丢弃掉，因此下一层可获得更清晰的信号来使用。而且与平均池化相比，最大池化提供了更强的变换不变性，并且所需计算量略少于平均池化。

请注意，最大池化和平均池化可以沿深度维度而不是空间维度执行，尽管这并不常见。这可以使CNN学习各种特征的不变性。例如，它可以学习多个滤波器，每个滤波器检测相同模式的不同旋转（例如手写数字，见图14-10），并且深度最大池化层确保输出是相同的，而不考虑旋转。CNN可以类似地学会对其他任何东西的不变性：厚度、亮度、偏斜、颜色，等等。

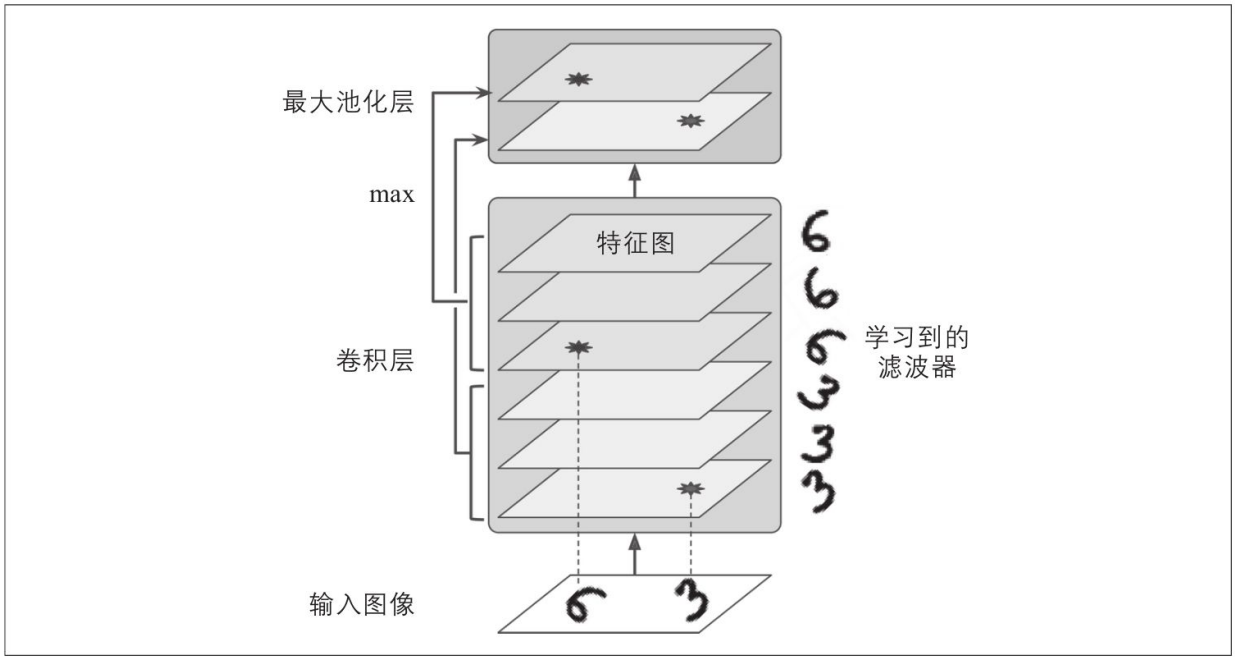


图14-10：深度最大池化可以帮助CNN学习任何不变性

Keras不包括深度最大池化层，但TensorFlow的低层深度学习API包括：只要使用tf.nn.max_pool（）函数，并将内核大小和步幅指定为4元组（即大小为4的元组）。前三个值均为1：即内核大小、步幅和批量处理、高度和宽度应为1。最后一个值应为沿深度维度所需的内核大小和跨度。例如，3（这必须是输入深度的除数。如果上一层输出20个特征图，它将不起作用，因为20不是3的倍数）：

```
output = tf.nn.max_pool(images,
                        ksize=(1, 1, 1, 3),
                        strides=(1, 1, 1, 3),
                        padding="valid")
```

如果要将其作为Keras模型中的一个层包括在内，将其包装在Lambda层中（或创建一个自定义的Keras层）：

```
depth_pool = keras.layers.Lambda(
    lambda X: tf.nn.max_pool(X, ksize=(1, 1, 1, 3), strides=(1, 1, 1, 3),
                             padding="valid"))
```

在现代架构中经常会看到的最后一种类型的池化层是全局平均池化层。它的工作原理非常不同：它所做的是计算整个特征图的均值（这就像使用与输入有相同空间维度的池化内核的平均池化层）。这意味着它每个特征图和每个实例只输出一个单值。尽管这是极具破坏性的（特征图中的大多数信息都丢失了），但它可以用作输出层，正如我们将在本章稍后看到的那样。要创建这样的层，只需使用keras.layers.GlobalAvgPool2D类：

```
global_avg_pool = keras.layers.GlobalAvgPool2D()
```

它等效于此简单的Lambda层，该层计算空间维度（高度和宽度）上的平均值：

```
global_avg_pool = keras.layers.Lambda(lambda X: tf.reduce_mean(X, axis=[1, 2]))
```

现在你知道了创建卷积神经网络的所有构造块。让我们看看如何组装它们。

[1] 到目前为止，我们讨论过的其他内核都有权重，而池化内核没有权重：它们只是无状态的滑动窗口。

14.4 CNN架构

典型的CNN架构堆叠了一些卷积层（通常每个卷积层都跟随一个ReLU层），然后是一个池化层，然后是另外几个卷积层（+ReLU），然后是另一个池化层，以此类推。随着卷积网络的不断发展，图像变得越来越小，但由于卷积层的存在，图像通常也越来越深（即具有更多的特征图）（见图14-11）。在堆栈的顶部添加了一个常规的前馈神经网络，该网络由几个全连接层（+ReLU）组成，最后一层输出预测（例如输出估计类别概率的softmax层）。

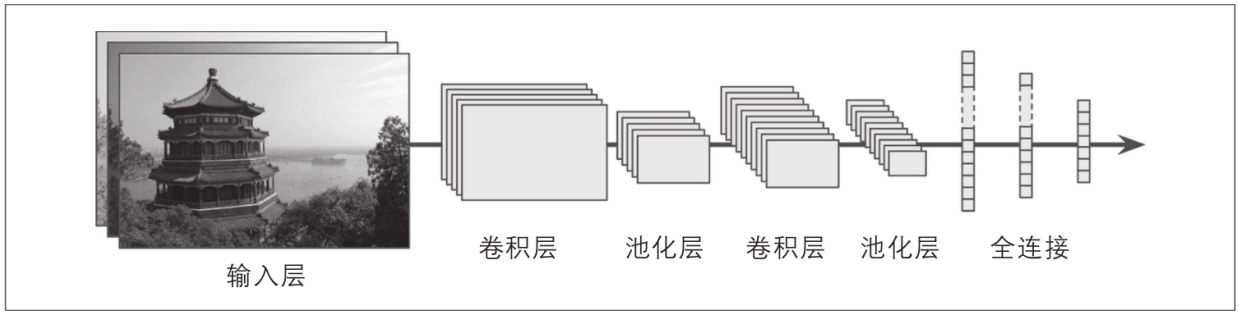


图14-11：典型的CNN架构

 一个常见的错误是使用太大的卷积内核。例如与其使用具有5×5内核的卷积层，不如使用具有3×3内核的两层：它使用较少的参数并需要较少的计算，并且通常性能会更好。第一个卷积层是一个例外：它具有较大的内核（例如5×5），步幅通常为2或更大，这将减小图像的空间维度而不会丢失太多信息，由于输入图像通常只有三个通道，因此不需要太多的计算量。

这是实现简单的CNN来处理Fashion MNIST数据集的方法（见第10章）：

```
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
                        input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")
])
```

让我们看一下这个模型：

• 第一层使用64个相当大的滤波器（7×7），但没有步幅，因为输入图像不是很大。设置了input_shape=[28，28，1]，因为图像是28×28像素，具有单个颜色通道（即灰度）。

• 接下来，我们有一个最大池化层，池化大小为2，因此它将每个空间维度除以2。

• 然后我们重复相同的结构两次：两个卷积层，紧接着是一个最大池化层。对于较大的图像，我们可以重复多次此结构（重复次数是你可以调整的超参数）。

• 请注意随着CNN向输出层延伸，滤波器的数量会增加（最初是64，然后是128，然后是256）：增长是有意义的，因为低层特征的数量通常很少（例如小圆圈、水平线），但是有很多不同的方法可以将它们组合成更高层次的特征。通常的做法是在每个池化层之后将滤波器的数量加倍：由于池化层将每个空间维度除以2，所以我们能负担得起对下一层特征图数量加倍而不必担心参数数量、内存使用量或计算量的暴增。

• 接下来是全连接的网络，该网络由两个隐藏的密集层和一个密集输出层组成。请注意，我们必须将其输入展平，因为密集网络需要每个实例的一维特征阵列。我们还添加了两个dropout层，每层的dropout率均为50%，以减少过拟合的情况。

该CNN在测试集上的精度达到92%以上。它不是最好的，但是也相当不错，显然比我们在第10章中使用密集网络获得的结果要好得多。

多年来，已经开发了这种基本架构的变体，导致该领域取得了惊人的进步。衡量这一进展的一个好方法是在竞赛中（例如ILSVRC ImageNet挑战）的错误率。在这场竞赛中，图像分类的前5名错误率在短短6年内从超过26%降至不到2.3%。前5位的错误率是系统的前5位预测未包含正确答案的测试图像数量。图像很大（高为256像素），共有1000个类别，其中有些确实很微妙（尝试区分120个犬种）。查看获奖作品的演变是了解CNN如何工作的好方法。

我们将首先介绍经典的LeNet-5结构（1998年），然后是ILSVRC挑战的三个获胜者：AlexNet（2012年）、GoogLeNet（2014年）和ResNet（2015年）。

14.4.1 LeNet-5

LeNet-5架构可能是最广为人知的CNN架构[1]。如前所述，它是由Yann LeCun于1998年创建的，已被广泛用于手写数字识别（MNIST）。它由表14-1中所示的层组成。

表14-1：LeNet-5架构

层	类型	特性图	大小	内核大小	步幅	激活函数
Out	全连接	–	10	–	–	RBF
F6	全连接	–	84	–	–	tanh
C5	卷积	120	1×1	5×5	1	tanh
S4	平均池化	16	5×5	2×2	2	tanh
C3	卷积	16	10×10	5×5	1	tanh
S2	平均池化	6	14×14	2×2	2	tanh
C1	卷积	6	28×28	5×5	1	tanh
In	输入	1	32×32	–	–	–

还有一些额外的细节要注意：

• MNIST图像为28×28像素，但是将其零填充为32×32像素并在送入网络之前进行了归一化。网络的其余部分不使用任何填充，这就是图像随着网络延展而尺寸不断缩小的原因。

• 平均池化层比一般的池化层要复杂一些：每个神经元计算其输入的平均值，然后将结果乘以可学习的系数（每个特征图一个），并添加一个可学习的偏置项（同样每个特征图一个），最后应用激活函数。

• C3特征图中的大多数神经元仅连接到了在S2特征图中的三个或四个神经元（而不是S2特征图中的所有6个）。有关详细信息，请参见原始论文中的表1（第8页）[2]。

• 输出层有点特殊：每个神经元输出的是输入向量和权重向量之间的欧几里得距离的平方，而不是计算输入向量和权重向量的矩阵乘法。每个输出测量图像属于特定数字类别的程度。交叉熵成本函数现在是首选，因为它对不良预测的惩罚更大，产生更大的梯度并收敛更快。

Yann LeCun的网站上有LeNet-5分类数字的演示。

14.4.2 AlexNet

AlexNet CNN架构[3]在2012年ImageNet ILSVRC挑战赛中大获全胜：它的前5位错误率是17%，而第2位的错误率为26%！它是由Alex Krizhevsky（因此得名）、Ilya Sutskever和Geoffrey Hinton开发的，与LeNet-5相似，只是更大和更深，它是第一个将卷积层直接堆叠在一起的方法，而不是将池化层堆叠在每个卷积层之上。表14-2给出了其架构。

表14-2：AlexNet架构

层	类型	特征图	大小	内核	步幅	填充	激活
Out	全连接	–	1000	–	–	–	Softmax
F10	全连接	–	4096	–	–	–	ReLU
F9	全连接	–	4096	–	–	–	ReLU
S8	最大池化	256	6×6	3×3	2	valid	–
C7	卷积	256	13×13	3×3	1	same	ReLU
C6	卷积	384	13×13	3×3	1	same	ReLU
C5	卷积	384	13×13	3×3	1	same	ReLU
S4	最大池化	256	13×13	3×3	2	valid	–
C3	卷积	256	27×27	5×5	1	same	ReLU
S2	最大池化	96	27×27	3×3	2	valid	–
C1	卷积	96	55×55	11×11	4	valid	ReLU
In	输入	3(RGB)	227×227	–	–	–	–

为了减少过拟合，作者使用了两种正则化技术。首先，他们在训练期间对F9层和F10层的输出使用了dropout率为50%的dropout技术（见第11章）；其次，他们通过随机变换训练图像的各种偏移量、水平翻转及更改亮度条件来执行数据增强。

数据增强

数据增强通过生成每个训练实例的许多变体来人为地增加训练集的大小。这减少了过拟合，使之成为一种正则化技术。生成的实例应尽可能逼真：理想情况下，给定来自增强训练集的图像，人类应该不能区别

它是否被增强。简单地添加白噪声将无济于事。修改应该是可以学习的（不是白噪声）。

例如，你可以将训练集（训练集中的图片数量各不相同）中的每张图片稍微移动、旋转和调整大小，将生成的图片添加到训练集中（见图14-12）。这迫使模型能更容忍图片中物体的位置、方向和大小的变化。对于更能容忍不同光照条件的模型，你可以类似地生成许多具有各种对比度的图像。通常，你还可以水平翻转图片（文本和其他非对称物体除外）。通过组合这些变换，可以大大增加训练集的大小。

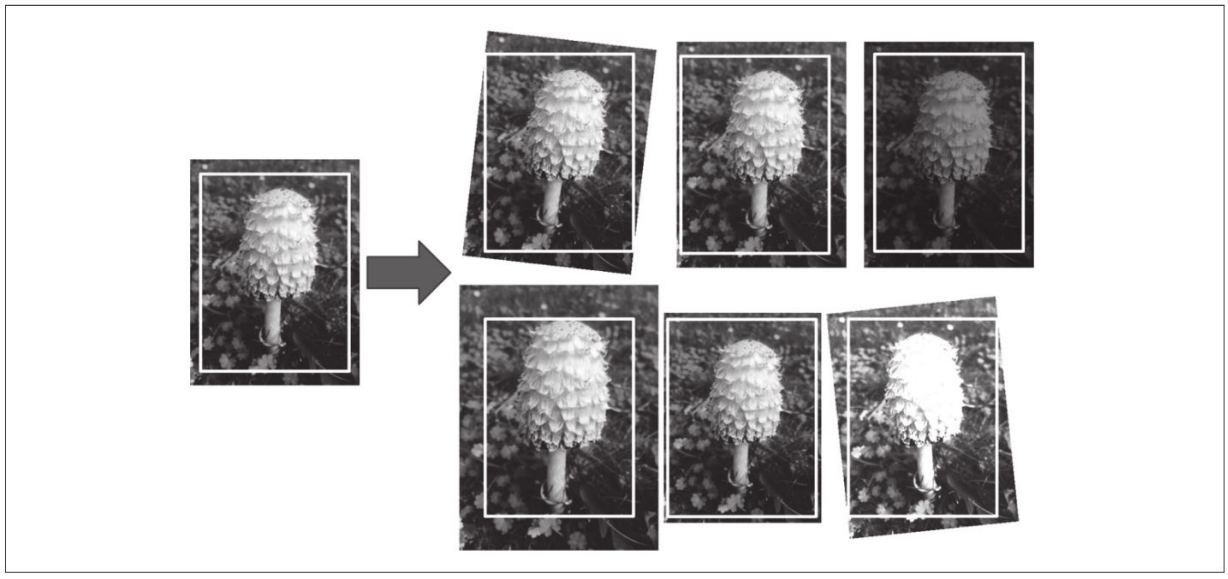


图14-12：从现有实例中生成新的训练实例

AlexNet还在层C1和C3的ReLU之后立即使用归一化步骤，称为局部响应归一化（LRN）：最强激活的神经元会抑制位于相邻特征图中相同位置的其他神经元（在生物神经元中已观察到这种竞争性激活）。这鼓励不同的特征图的专业化，将它们分开，并迫使它们探索更广泛的特征，从而最终改善泛化能力。公式14-2显示了如何应用LRN。

公式14-2：局部响应归一化（LRN）

$$b_i=a_i(k+a\sum_{j=j_{low}}^{j_{high}}a_j^2)^{-\beta} \quad 其中 \begin{cases} j_{high}=\min(i+\frac{r}{2},f_n-1) \\ j_{low}=\max(0,i-\frac{r}{2}) \end{cases}$$

公式中：

• b_i是位于特征图i中位于u行和v列的神经元的归一化输出（请注意在此公式中，我们仅考虑位于此行和列的神经元，因此未显示u和v）。

• a_i是ReLU之后但未归一化之前该神经元的激活。

• k、α、β和r是超参数。k称为偏置，r称为深度半径。

• f_n是特征图的数量。

例如，如果r=2且神经元具有很强的激活作用，它将抑制位于特征图中紧接其自身上方和下方的神经元的激活。

在AlexNet中，超参数的设置如下：r=2，α=0.00002，β=0.75，k=1。这个步骤可以使用tf.nn.local_response_normaliza tion（）函数来实现（如果你想在Keras模型中使用它，请在Lambda层中包装它）。

Matthew Zeiler和Rob Fergus开发了AlexNet的变体ZF Net[4]，赢得了2013年的ILSVRC挑战。它实质上是AlexNet，带有一些经过调整的超参数（特征图的数量、内核大小、步幅等）。

14.4.3 GoogLeNet

GoogLeNet架构由Google研究院[5]的Christian Szegedy等人开发。它将前5名的错误率降低到7%以下而赢得了2014年的ILSVRC挑战。这种出色的性能在很大程度上是由于该网络比以前的CNN更深（如图14-14所示）。称为盗梦空间（inception）模块[6]的子网能使GoogLeNet比以前的架构更有效地使用参数：GoogLeNet实际上具有AlexNet $\frac{1}{2}$ 的参数（大约是600万，而不是6000万）。

图14-13显示了inception模块的架构。符号“3×3+1（S）”表示该层使用3×3内核，步幅为1且填充为"same"。首先将输入信号复制并馈送到四个不同的层。所有卷积层都使用ReLU激活函数。请注意，第二组卷积层使用不同的内核大小（1×1、3×3和5×5），从而可以识别不同比例尺寸的模式。还要注意，每个单层都使用步幅为1和"same"的填充（甚至最大池化层），因此它们的输出都具有与输入相同的高度和宽度。这使得可以在最终的深度合并层中沿着深度维度合并所有的输出（即堆叠来自所有四个顶部卷积层的特征图）。在TensorFlow中可以使用tf.concat（）操作实现此连接层，其中axis=3（该轴是深度）。

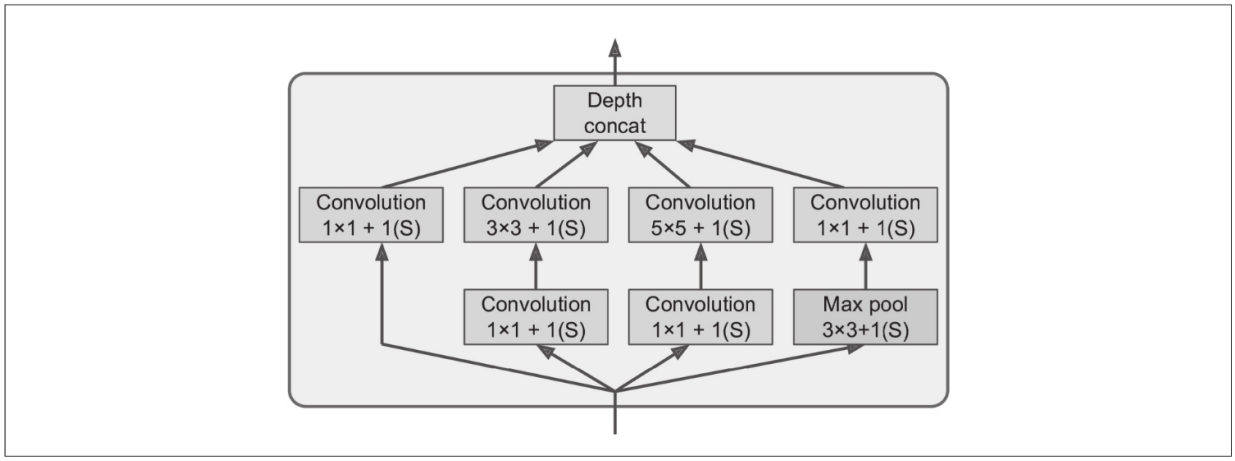


图14-13：Inception模块

你可能想知道为什么inception模块具有带1×1内核的卷积层。这些层肯定不会识别任何特征，因为它们一次只能看到一个像素。实际上，这些层有三个目的：

• 尽管它们无法识别空间特征，但它们可以识别沿深度维度的特征。

• 它们输出比输入更少的特征图，因此它们充当了瓶颈层，这意味着它们降低了维度。这减少了计算量和参数数量，加快了训练速度，并提高了泛化能力。

• 每对卷积层（[1×1，3×3]和[1×1，5×5]）就像一个强大的卷积层，能够识别更复杂的模式。实际上这对卷积层不是在整个图像上扫描简单的线性分类器（就像单个卷积层一样），而是在整个图像上扫描了两层神经网络。

简而言之，你可以将整个inception模块视为类固醇上的卷积层，能够输出以各种比例尺寸识别的复杂模式的特征图。

 每个卷积层的卷积核数是一个超参数。不幸的是，这意味着你需要为添加的每个inception层调整6个超参数。

现在让我们看一下GoogLeNet CNN的架构（见图14-14）。每个卷积层和每个池化层输出的特征图的数量在内核大小之前显示。该架构是如此之深以至于必须以三列表示，但是GoogLeNet实际上是一个高堆叠，包括9个inception模块（带有旋转陀螺的盒子）。inception模块中的6个数字表示模块中每个卷积层输出的特征图的数量（与图14-13中的顺序相同）。请注意，所有卷积层都使用ReLU激活函数。

让我们看看这个网络：

• 前两层将图像的高度和宽度除以4（因此将其面积除以16）以减少计算量。第一层使用较大的内核，因此可以保留很多信息。

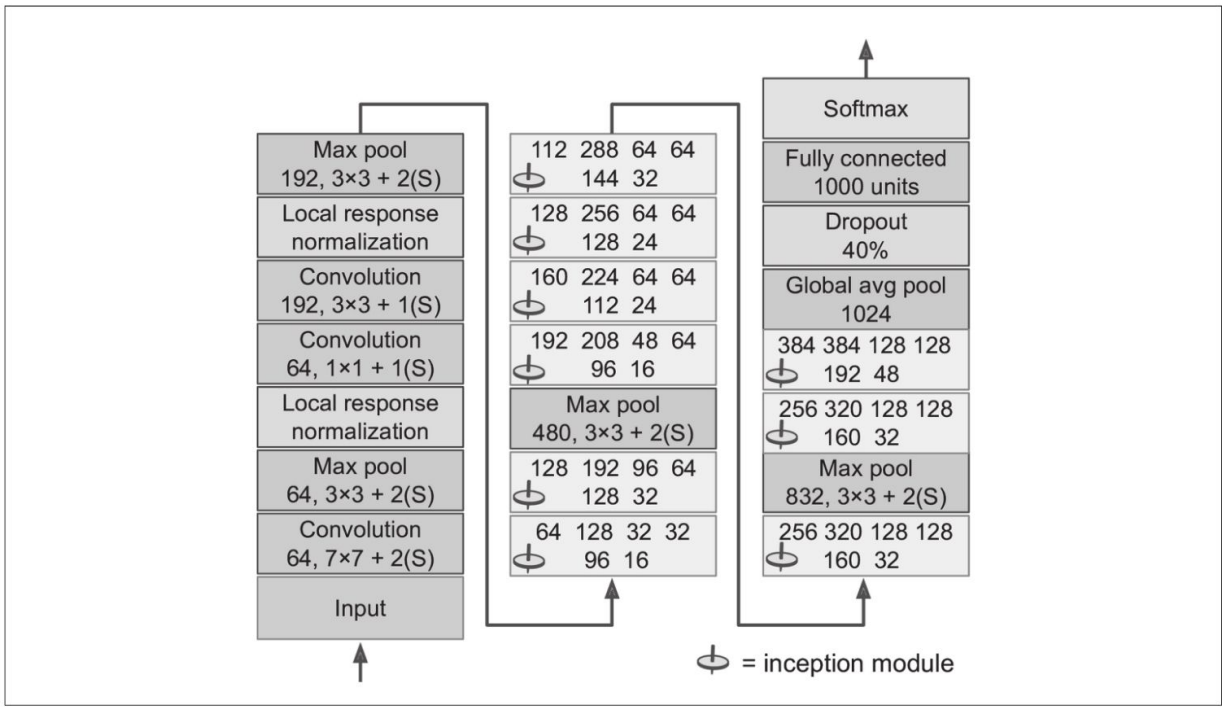


图14-14：GoogLeNet架构

• 然后，局部响应归一化层确保前面的层学习各种各样的特征（如前所述）。

• 接下来是两个卷积层，其中第一层就像一个瓶颈层。如前所述，你可以将这一对层视为一个更智能的卷积层。

• 同样，局部响应归一化层可确保先前的层识别各种模式。

• 接下来，最大池化层将图像的高度和宽度减小到$\frac{1}{2}$，再次加快了计算速度。

• 然后是9个inception模块的高堆叠，与几个最大池化层交错以减少维度并加快网络速度。接下来，全局平均池化层输出每个特征图的均值：这将丢弃所有剩余的空间信息，这是可以的，因为在该点上没有太多的空间信息。实际上，GoogLeNet输入图像通常为224×224像素，因此在经过5个最大池化层（每个高度和宽度除以2）后，特征图将降至

7×7。而且这是分类任务，不是对象定位，因此对象在哪里都没有关系。由于此层带来的降维效果，因此不需要在CNN的顶部（如AlexNet中）有几个全连接层，这大大减少了网络中的参数数量，并降低了过拟合的风险。

• 最后一层是不言自明的：为了进行正则化而dropout，然后是一个具有1000个单元的全连接层（因为有1000个类）和一个softmax激活函数来输出估计的类别概率。

该图略有简化：原始的GoogLeNet架构还包括两个辅助分类器，它们插入在第3和第6个inception模块的顶部。它们都由一个平均池化层、一个卷积层、两个全连接层和一个softmax激活层组成。在训练期间，它们的损失（减少了70%）被添加到总损失中，目的是解决梯度消失的问题并正则化网络。但是，后来证明它们的作用相对较小。

Google研究人员后来提出了GoogLeNet架构的几种变体，包括Inception-v3和Inception-v4，它们使用略微不同的inception模块并获得了更好的性能。

14.4.4 VGGNet

ILSVRC 2014挑战赛的亚军是VGGNet[7]，由牛津大学视觉几何组（VGG）研究实验室的Karen Simonyan和Andrew Zisserman开发。它具有非常简单和经典的架构，具有2或3个卷积层和一个池化层，然后又有2或3个卷积层和一个池化层，以此类推（总共达到16或19个卷积层，具体取决于VGG变体），以及最终的有2个隐藏层的密集网络和输出层。它仅使用3×3滤波器，但使用了许多滤波器。

14.4.5 ResNet

何凯明等使用残差网络（或ResNet）[8]赢得了ILSVRC 2015挑战赛，其前5名的错误率低于3.6%。获胜的变体使用了由152层组成的非

常深的CNN（其他变体具有34、50和101层）。它证实了一个趋势：模型变得越来越深，参数越来越少。能够训练这种深层网络的关键是使用跳过连接（也称为快捷连接）：馈入层的信号也将添加到位于堆栈上方的层的输出中。让我们看看为什么这很有用。

在训练神经网络时，目标是使其成为目标函数h（x）的模型。如果将输入x添加到网络的输出（即添加跳过连接），则网络将被迫建模f（x）=h（x）-x而不是h（x）。这称为残差学习（见图14-15）。

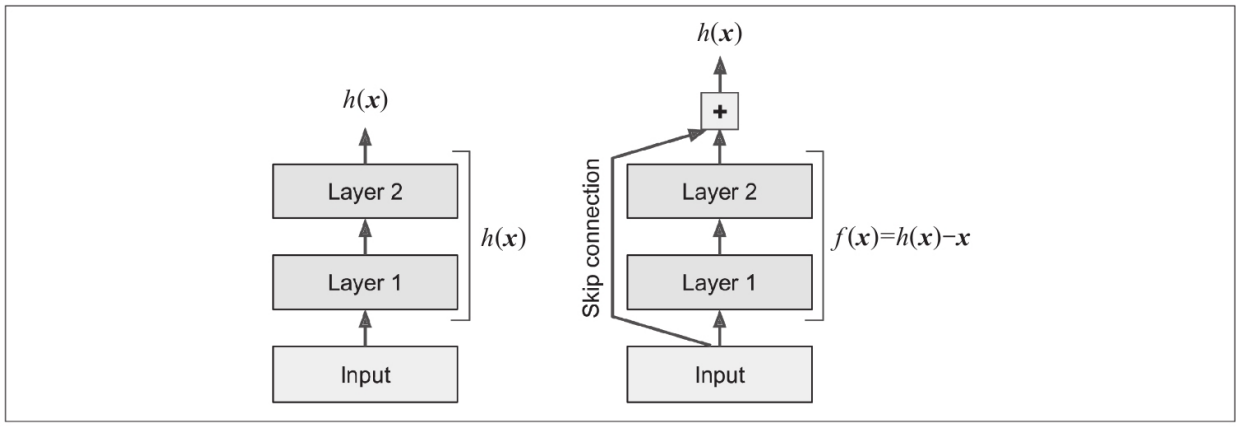


图14-15：残差学习

初始化常规神经网络时，其权重接近零，因此网络仅输出接近零的值。如果添加跳过连接，则生成的网络仅输出其输入的副本。换句话说，它首先对恒等函数建模。如果目标函数与恒等函数相当接近（通常是这种情况），这会大大加快训练速度。

此外，如果添加许多跳过连接，即使几层还没有开始学习，网络也可以开始取得进展（见图14-16）。借助跳过连接，信号可以轻松地在整个网络中传播。深度残差网络可以看作是残差单元（RU）的堆栈，其中每个残差单元都是具有跳过连接的小型神经网络。

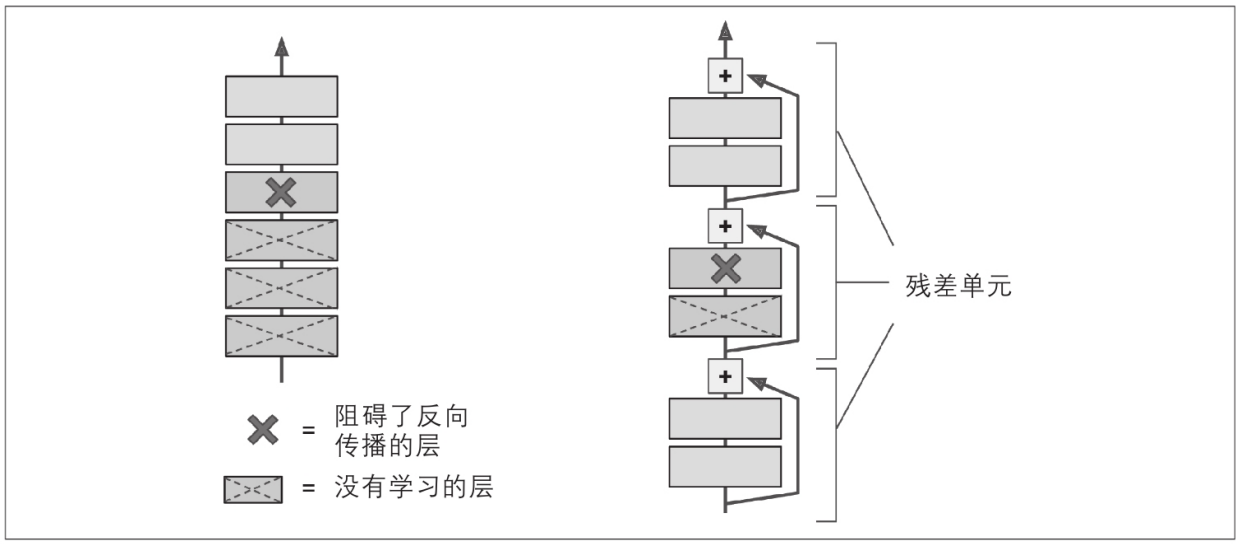


图14-16：常规的深度神经网络（左）和深度残差网络（右）

现在让我们看一下ResNet的架构（见图14-17）。它非常简单。它的开始和结束与GoogLeNet完全相同（除了没有dropout层），在两者之间只是一堆非常简单的残差单元。每个残差单元由两个卷积层（没有池化层）组成，具有批量归一化（BN）和ReLU激活，使用3×3内核并保留空间维度（步幅1，"same"填充）。

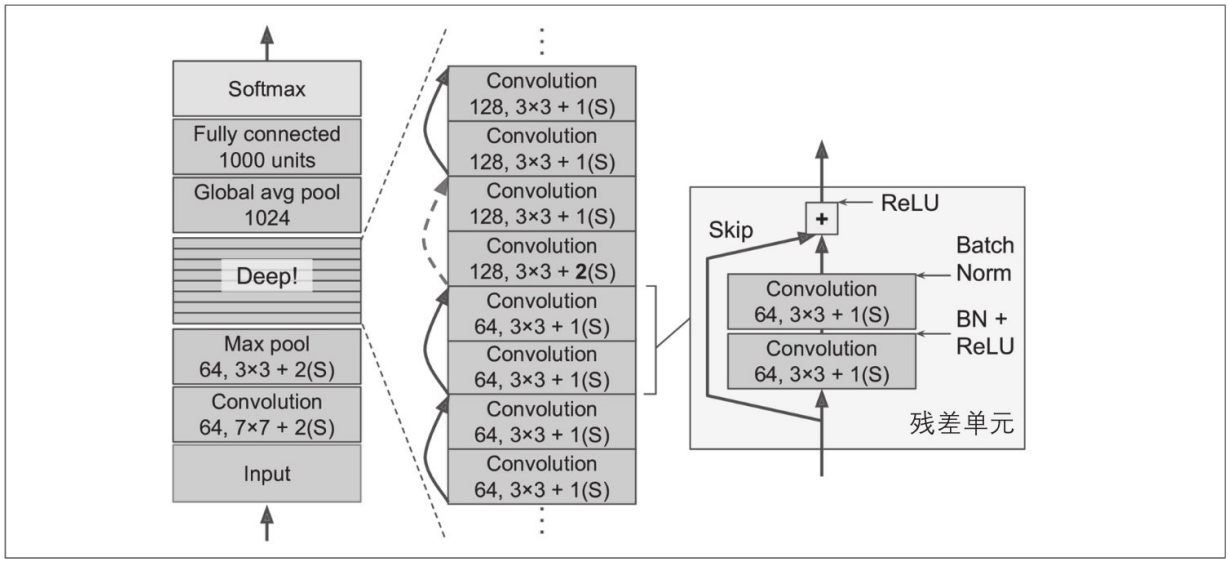


图14-17：ResNet架构

请注意，特征图的数量每隔几个残差单元就增加一倍，同时高度和宽度也减半（使用步幅为2的卷积层）。发生这种情况时，输入不能直接添加到残差单元的输出，因为它们的形状不同（此问题会影响图14-17中虚线箭头所示的跳过连接）。为了解决这个问题，输入将通过步幅为2且具有正确数量的输出特征图的1×1卷积层（请参见图14-18）。

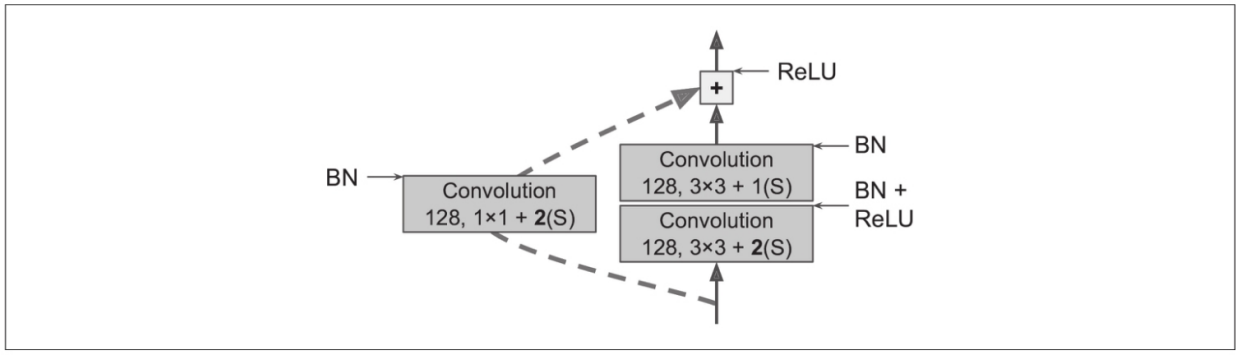


图14-18：改变特征图大小和深度的跳过连接

ResNet-34是具有34层的ResNet（仅计算卷积层和全连接层）[9]，包含具有64个特征图的3个RU、具有128个特征图的4个RU、具有256个特征图的6个RU以及具有512个特征图的3个RU。我们将在本章稍后实现该架构。

更深的ResNet（例如ResNet-152）使用的残差单元略有不同。它们使用三个卷积层代替了两个具有256个特征图的3×3卷积层：第一个1×1卷积层仅具有64个特征图（少4倍），它充当了瓶颈层（如前所述）；然后是一个3×3层，具有64个特征图；最后是另一个1×1卷积层，具有256个特征图（4乘以64）来恢复原始深度。ResNet-152包含3个此类RU，可输出256个特征图，然后是具有512个特征图的8个RU，具有1024个特征图的36个RU，最后是具有2048个特征图的3个RU。

 Google的Inception-v4[10]架构融合了GoogLeNet和ResNet的思想，在ImageNet分类中实现了前5位的错误率接近3％。

14.4.6 Xception

GoogLeNet架构的另一种值得注意的变体：Xception[11]（代表Extreme Inception）由Franois Chollet（Keras的作者）于2016年提出，它在很大的视觉任务上（3.5亿张图像和17 000个类别）明显优于Inception-v3。就像Inception-v4一样，它融合了GoogLeNet和ResNet的思想，但是它用称为深度方向可分离卷积层（或简称为可分离卷积层[12]）的特殊类型替换了inception模块。这些层以前曾在某些CNN架构中使用过，但它们并不像在Xception架构那样重要。虽然常规卷积层使用的滤波器试图同时识别空间模式（例如椭圆形）和跨通道模式（例如，嘴+鼻子+眼睛=脸），但可分离的卷积层的强烈假设是空间模式和跨通道模式可以单独建模（见图14-19）。因此它由两部分组成：第一部分为每个输入特征图应用一个空间滤波器，然后第二部分专门寻找跨通道模式——它是具有1×1滤波器的常规卷积层。

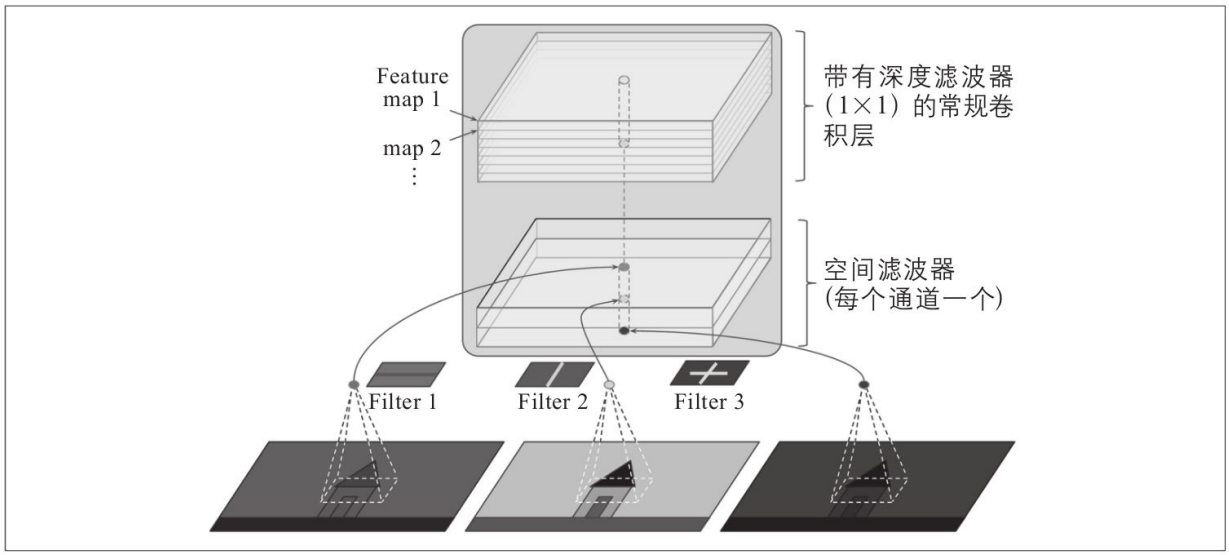


图14-19：深度可分离卷积层

由于可分离的卷积层每个输入通道仅具有一个空间滤波器，因此应避免在通道数量太少的层（例如输入层）之后使用它们（如图14-19所示，该图仅用于插图目的）。因此，Xception架构从2个常规卷积层开

始，但随后的其余部分仅使用可分离卷积（总共34个），外加几个最大池化层和通常的最终层（全局平均池化层和一个密集输出层）。

你可能想知道为什么Xception被视为GoogLeNet的变体，因为它根本不包含任何inception模块。如前所述，inception模块包含带1×1滤波器的卷积层：这些滤波器专门用于识别跨通道模式。但是位于其顶部的卷积层是常规的卷积层，它们既查找空间模式，也查找跨通道模式。因此你可以将inception模块视为常规卷积层（共同考虑空间模式和跨通道模式）和可分离卷积层（分别考虑它们）之间的中间层。在实践中，似乎可分离卷积层通常表现更好。

 与常规卷积层相比，可分离卷积层使用更少的参数、更少的内存和更少的计算，而且通常它们表现更好，因此，你应默认使用它们（除了在通道较少的层之后）。

香港中文大学CUImage团队赢得了ILSVRC 2016挑战赛的冠军。他们使用了许多不同技术的集合，其中包括一个称为GBD-Net[13]的复杂物体检测系统，前5位的错误率低于3%。尽管这一结果令人印象深刻，但该解决方案的复杂性与ResNets的简单性形成了对比。而且一年后，正如我们现在所看到的，另一种相当简单的架构的性能甚至更好。

14.4.7 SENet

ILSVRC 2017挑战赛中获胜的架构是Squeeze-and-Excitation Network（SENet）[14]。该架构扩展了现有架构（例如inception网络和ResNets），并提高了它们的性能。这使SENet以2.25%的前5名错误率赢得了比赛！inception和ResNet的扩展版本分别称为SEInception和SE-ResNet。SENet的增强来自这样一个事实——SENet向原始架构中的每个单元（即每个inception模块或每个残差单元）添加了一个称为SE块的小型神经网络，如图14-20所示。

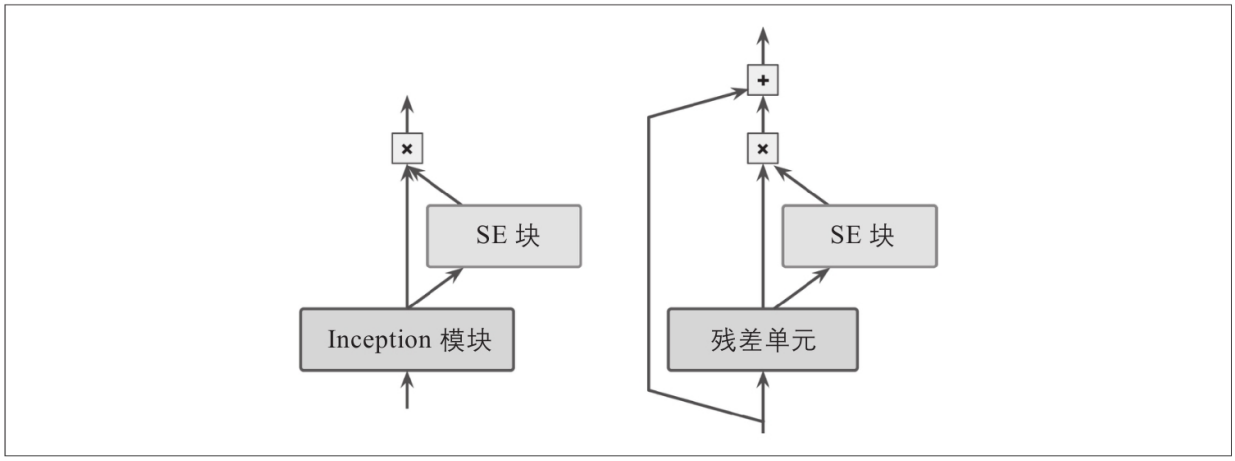


图14-20：SE-Inception模块（左）和SE-ResNet单元（右）

SE块分析其连接的单元的输出，仅专注于深度维度（它不关心任何空间模式），并了解哪一些特征通常最活跃。然后，它使用此信息重新校准特征图，如图14-21所示。例如，一个SE块可能会了解到嘴、鼻子和眼睛通常在图片中同时出现：如果看到嘴巴和鼻子，还应该看到眼睛。因此，如果该块在嘴和鼻子特征图中看到强烈的激活，而在眼睛特征图中只有轻微的激活，则它将增强眼睛特征图（更准确地说，它将降低无关的特征图）。如果眼睛有点与其他东西混淆，则特征图重新校准有助于解决不确定性。

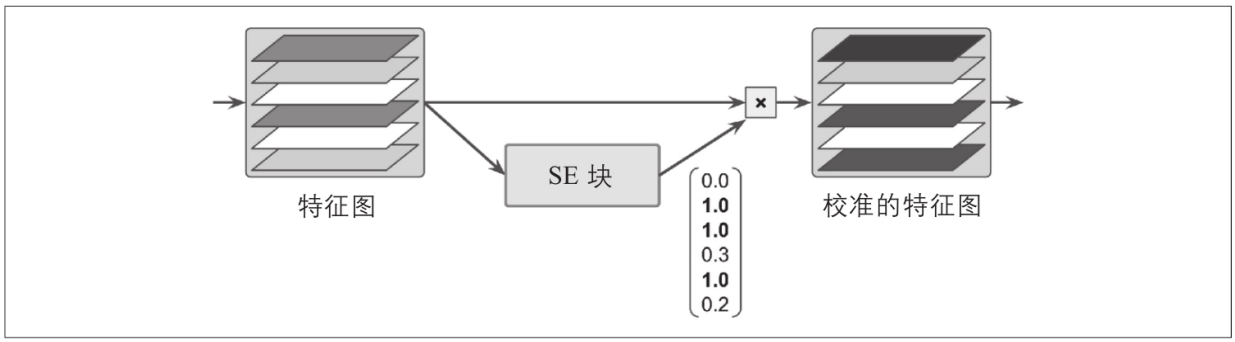


图14-21：SE模块执行特征图重新校准

SE块仅由三层组成：全局平均池化层、使用ReLU激活函数的隐藏密集层和使用sigmoid激活函数的密集输出层（见图14-22）。

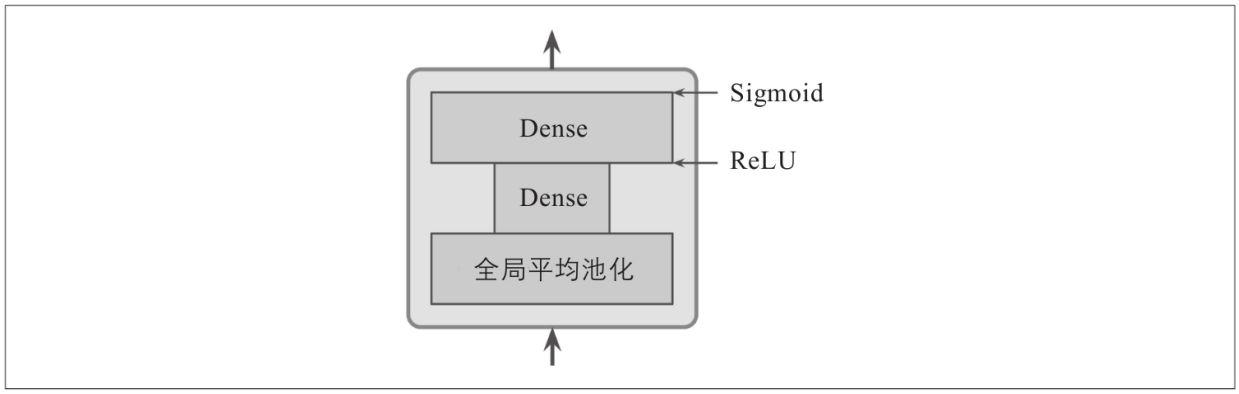


图14-22：SE块架构

如前所述，全局平均池化层为每个特征图计算平均激活：例如，如果其输入包含256个特征图，则它将输出256个数字，代表每个滤波器的总体响应。下一层是“挤压”发生的地方：此层的神经元要明显少于256个，通常比特征图（例如，16个神经元）的数目少16倍，因此将256个压缩为一个小向量（例如，16个维度）。这是特征响应分布的低维向量表示（即嵌入）。这个瓶颈步骤迫使SE块学习特征组合的一般表征形式（当我们在第17章讨论自动编码器时，会再次看到该原理的作用）。最后，输出层进行嵌入，并输出一个重新校准的向量，每个特征图包含一个数字（例如256），每个数字介于0和1之间。然后将特征图与该重新校准的向量相乘，因此不相关的特征（带有一个低重新校准分数）按比例缩小，而相关特征（重新校准分数接近1）则不予考虑。

[1] Yann LeCun et al.，“Gradient-Based Learning Applied to Document Recognition”，Proceedings of the IEEE 86，no.11（1998）：2278－2324.

[2] Yann LeCun et al.,“ Gradient-Based Learning Applied to Document Recognition ”，Proceedings of the IEEE 86, no. 11 (1998): 2278－2324.

[3] Alex Krizhevsky et al.，“ImageNet Classification with Deep Convolutional Neural Networks”，P_roceedings of the 25th International Conference on Neural Information Processing Systems 1（2012）：1097－1105.

[4] Matthew D.Zeiler and Rob Fergus，“Visualizing and Understanding Convolutional Networks”，Proceedings of the European Conference on Computer Vision（2014）：818-833.
[5] Christian Szegedy et al.，“Going Deeper with Convolutions”，Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition（2015）：1-9.
[6] 在2010年的电影《盗梦空间》中，角色不断深入到多层梦境中。因此这些模块的名称来源于电影。
[7] Karen Simonyan and Andrew Zisserman，“Very Deep Convolutional Networks for Large-Scale Image Recognition”，arXiv preprint arXiv：1409.1556（2014）.
[8] Kaiming He et al.，“Deep Residual Learning for Image Recognition”，arXiv preprint arXiv：1512：03385（2015）.
[9] 描述神经网络时仅对具有参数的层进行计数，这是一种常见的做法.
[10] Christian Szegedy et al.，“Inception－v4，Inception-ResNet and the Impact of Residual Connections on Learning”，arXiv preprint arXiv：1602.07261（2016）.
[11] François Chollet，“Xception：Deep Learning with Depthwise Separable Convolutions”，arXiv preprint arXiv：1610.02357（2016）.
[12] 这个名字有时可能有点模棱两可，因为空间上可分离的卷积通常也被称为“可分离卷积”。
[13] Xingyu Zeng et al.，“Crafting GBD-Net for Object Detection”，IEEE Transactions on Pattern Analysis and Machine Intelligence 40，no.9（2018）：2109－2123.
[14] Jie Hu et al.，“Squeeze-and-Excitation Networks”，Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition（2018）：7132－7141.

14.5 使用Keras实现ResNet-34 CNN

到目前为止所描述的大多数CNN架构都非常容易实现（尽管通常你会加载经过预训练的网络，正如我们将看到的那样）。为了说明这一过程，让我们使用Keras从零开始实现ResNet-34。首先创建一个ResidualUnit层：

```
class ResidualUnit(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = keras.activations.get(activation)
        self.main_layers = [
            keras.layers.Conv2D(filters, 3, strides=strides,
                                padding="same", use_bias=False),
            keras.layers.BatchNormalization(),
            self.activation,
            keras.layers.Conv2D(filters, 3, strides=1,
                                padding="same", use_bias=False),
            keras.layers.BatchNormalization()]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                keras.layers.Conv2D(filters, 1, strides=strides,
                                    padding="same", use_bias=False),
                keras.layers.BatchNormalization()]

    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
        for layer in self.skip_layers:
            skip_Z = layer(skip_Z)
        return self.activation(Z + skip_Z)
```

如你所见，此代码非常接近图14-18。在构造函数中，我们创建所需的所有层：主要层在图的右侧，而跳过层在左侧（仅当步幅大于1时才需要）。然后在call（）方法中，使输入经过主要层和跳过层（如果有），然后添加输出层并应用激活函数。

接下来，我们可以使用Sequential模型来构建ResNet-34，因为它实际上只是一个很长的层序列（现在有了ResidualUnit类，我们可以

将每个残差单元视为一个层）：

```
model = keras.models.Sequential()
model.add(keras.layers.Conv2D(64, 7, strides=2, input_shape=[224, 224, 3],
                              padding="same", use_bias=False))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation("relu"))
model.add(keras.layers.MaxPool2D(pool_size=3, strides=2, padding="same"))
prev_filters = 64
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters
model.add(keras.layers.GlobalAvgPool2D())
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation="softmax"))
```

这段代码中唯一有点技巧的部分是将ResidualUnit层添加到模型的循环：如前所述，前3个RU具有64个滤波器，然后余下4个RU具有128个滤波器，以此类推。然后，当滤波器的数量与上一个RU中的相同时，将步幅设置为1，否则将其设置为2。然后添加ResidualUnit，最后更新prev_filters。

令人惊讶的是，我们可以用不到40行代码来构建赢得2015年ILSVRC挑战的模型！这证明了ResNet模型的优雅和Keras API的表达力。实现其他CNN架构并不困难。但是，Keras内置了其中几种架构，那么为什么不使用它们呢？

14.6 使用Keras的预训练模型

通常，你无须手动实现像GoogLeNet或ResNet这样的标准模型，因为在kera s.applications包中只需一行代码即可轻松获得预训练的网络。例如，你可以使用以下代码加载在ImageNet上预训练的ResNet-50模型：

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
```

就这样！这会创建一个ResNet-50模型并下载在ImageNet数据集上预先训练的权重。要使用它，首先需要确保图像尺寸合适。ResNet-50模型需要224×224像素的图像（其他模型可能需要其他尺寸，例如299×299），让我们使用TensorFlow的tf.image.resize（）函数调整我们之前加载的图像的大小：

```
images_resized = tf.image.resize(images, [224, 224])
```

 tf.image.resize（）不会保留宽高比。如果这是一个问题，请在调整大小之前尝试将图像裁剪为适当的宽高比。两种操作可以使用tf.image.crop_and_resize（）一次完成。

预先训练的模型假定以特定方式对图像进行预处理。在某些情况下，它们可能期望输入缩放到从0到1，或从-1到1，等等。每个模型都提供一个preprocess_input（）函数，你可以用来预处理图像。这些函数假定像素值的范围是0到255，因此我们必须将它们乘以255（因为我们之前将其缩放到0-1范围内）：

```
inputs = keras.applications.resnet50.preprocess_input(images_resized * 255)
```

现在我们可以使用预训练的模型进行预测：

```
Y_proba = model.predict(inputs)
```

通常Y_proba输出一个矩阵，每个图像一行，每个类一列（在此例中，共有1000个类）。如果要显示前K个预测（包括类名和每个预测类的估计概率），请使用encode_predictions（）函数。对于每个图像，它返回一个包含前K个预测的数组，其中每个预测都表示为一个包含类标识符[1]、其名称和对应置信度得分的数组：

```
top_K = keras.applications.resnet50.decode_predictions(Y_proba, top=3)
for image_index in range(len(images)):
    print("Image #{}".format(image_index))
    for class_id, name, y_proba in top_K[image_index]:
        print("  {} - {:12s} {:.2f}%".format(class_id, name, y_proba * 100))
    print()
```

输出为：

```
Image #0
  n03877845 - palace       42.87%
  n02825657 - bell_cote    40.57%
  n03781244 - monastery    14.56%
Image #1
  n04522168 - vase         46.83%
  n07930864 - cup          7.78%
  n11939491 - daisy        4.87%
```

两个图像的前三个结果中都显示了正确的类（修道院和雏菊）。考虑到该模型必须从1000个类别中进行选择，这已经相当不错了。

如你所见，使用预先训练的模型来创建一个好的图像分类器非常容易。keras.applications中还提供了其他的视觉模型，包括多个ResNet变体、GoogLeNet变体（例如Inception-v3和Xception）、VGGNet变体以及MobileNet和MobileNetV2（用于移动应用程序的轻量级模型）。

但是，如果要将图像分类器用于不属于ImageNet的图像类，该怎么办？在这种情况下，你仍然可以从预先训练的模型中受益，以进行迁移学习。

[1] 在ImageNet数据集中，每个图像都与WordNet数据集中的一个单词相关联：类ID只是一个WordNet ID。

14.7 迁移学习的预训练模型

如果你想构建图像分类器但没有足够的训练数据，那么重用预训练模型的较低层通常是个好办法，正如我们在第11章中讨论的那样。例如，让我们来训练模型对花的图片进行分类，并使用预先训练的Xception模型。首先，让我们使用TensorFlow Datasets加载数据集（见第13章）：

```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
dataset_size = info.splits["train"].num_examples # 3670
class_names = info.features["label"].names # ["dandelion", "daisy", ...]
n_classes = info.features["label"].num_classes # 5
```

请注意你可以通过设置with_info=True获得有关数据集的信息。在这里我们获得数据集的大小和类的名称。不幸的是，只有一个"train"数据集，没有测试集或验证集，因此我们需要拆分训练集。TF Datasets项目为此提供了一个API。例如，让我们将数据集的前10%用于测试，接下来的15%用于验证，其余的75%用于训练：

```
test_split, valid_split, train_split = tfds.Split.TRAIN.subsplit([10, 15, 75])

test_set = tfds.load("tf_flowers", split=test_split, as_supervised=True)
valid_set = tfds.load("tf_flowers", split=valid_split, as_supervised=True)
train_set = tfds.load("tf_flowers", split=train_split, as_supervised=True)
```

接下来我们必须预处理图像。CNN需要224×224大小的图像，因此我们需要调整它们的大小。我们还需要通过Xception的preprocess_input（）函数来预处理图像：

```
def preprocess(image, label):
    resized_image = tf.image.resize(image, [224, 224])
    final_image = keras.applications.xception.preprocess_input(resized_image)
    return final_image, label
```

用这个预处理函数来处理所有三个数据集，对训练集进行乱序，并对所有数据集添加批量处理和预取：

```
batch_size = 32
train_set = train_set.shuffle(1000)
train_set = train_set.map(preprocess).batch(batch_size).prefetch(1)
valid_set = valid_set.map(preprocess).batch(batch_size).prefetch(1)
test_set = test_set.map(preprocess).batch(batch_size).prefetch(1)
```

如果要执行一些数据增强，可以更改训练集的预处理功能，向训练图像添加一些随机变换。例如，使用tf.image.random_crop（）随机裁切图像，使用tf.image.random_flip_left_right（）随机水平翻转图像，以此类推（有关示例请参阅notebook的“Pretrained Models for Transfer Learning”部分）。

 使用keras.preprocessing.image.ImageDataGeneGenerator类，可以轻松地从磁盘加载图像并以各种方式对其进行扩充：你可以移动、旋转、重新缩放、水平翻转或垂直翻转、剪切或应用任何所需的转换函数。这对于简单项目非常方便。然而，建立tf.data流水线具有许多优点：它可以从任何来源（而不仅仅是本地磁盘）有效地（例如，并行）读取图像；你可以根据需要操作Dataset；如果你编写基于tf.image操作的预处理函数，则可以在tf.data流水线和你要部署到生产环境的模型中使用此函数（请参见第19章）。

接下来，我们加载一个在ImageNet上预训练的Xception模型。我们通过设置include_top=False排除网络的顶部：这排除了全局平均池化层和密集输出层。然后，根据基本模型的输出，添加我们自己的全

局平均池化层，再跟一个每个类一个单位的密集输出层，使用softmax激活函数。最后我们创建Keras模型：

```
base_model = keras.applications.xception.Xception(weights="imagenet",
                                                  include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model = keras.Model(inputs=base_model.input, outputs=output)
```

如第11章所述，至少在训练开始时冻结预训练层的权重通常是一个好主意：

```
for layer in base_model.layers:
    layer.trainable = False
```

 由于我们的模型直接使用基本模型的层，而不是使用base_model对象本身，因此设置base_model.trainable=False无效。

最后，编译模型并开始训练：

```
optimizer = keras.optimizers.SGD(lr=0.2, momentum=0.9, decay=0.01)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, epochs=5, validation_data=valid_set)
```

 除非你有GPU，否则这会非常慢。如果没有，你应该在Colab中使用GPU（免费），运行本章的示例。请参阅https://github.com/ageron/handson-ml2上的说明。

对模型进行几个轮次的训练后，其验证精度应达到75℃～80％，并且不再取得很大进展。这意味着顶层现在已经受过良好的训练，因此我们准备解冻所有层（或者你可以尝试只解冻顶层）并继续进行训练（在冻结或解冻时不要忘记编译模型）。这次我们使用低得多的学习率来避免损坏预训练的权重：

```
for layer in base_model.layers:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.001)
model.compile(...)
history = model.fit(...)
```

这需要一些时间，但是此模型在测试集上应达到约95％的精度。这样你可以开始训练惊人的图像分类器了！但是计算机视觉不仅仅是分类。例如，如果你还想知道花在图片中的位置怎么办？让我们现在来看一下。

14.8 分类和定位

如第10章所述，定位图片中物体可以表示为回归任务：预测物体周围的边界框，一种常见的方法是预测物体中心的水平坐标和垂直坐标，还有其高度和宽度。这意味着我们有四个数字需要预测。它不需要对模型进行太多修改，我们只需要添加具有四个单位的第二个密集输出层（通常在全局平均池化层之上），就可以使用MSE损失对其进行训练：

```
base_model = keras.applications.xception.Xception(weights="imagenet",
                                                  include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output = keras.layers.Dense(n_classes, activation="softmax")(avg)
loc_output = keras.layers.Dense(4)(avg)
model = keras.Model(inputs=base_model.input,
                    outputs=[class_output, loc_output])
model.compile(loss=["sparse_categorical_crossentropy", "mse"],
              loss_weights=[0.8, 0.2], # depends on what you care most about
              optimizer=optimizer, metrics=["accuracy"])
```

但是现在我们有一个问题：花朵数据集在花朵周围没有边界框，因此我们需要自己添加。这通常是机器学习项目中最难、最昂贵的部分之一：获取标签。花时间寻找合适的工具是一个好办法。要使用边界框标注图像，你可能需要使用开源图像标记工具，例如VGG Image Annotator、LabelImg、OpenLabeler或ImgLab，或者使用商业工具（例如LabelBox或Supervisely）。如果你需要标注大量图像，则可能还需要考虑使用众包平台，例如Amazon Mechanical Turk。但是建立众包平台，需要准备发送给工人的表格，对其进行监督并确保他们产生的边界框的质量是很好的，因此要保证这样做是值得的。如果只有几千张图像要标记，并且你不打算经常这样做，那么最好自己动手做。Adriana Kovashka等人写了一篇关于计算机视觉众包的非常实用的论文[1]。我建议你查看一下，即使你不打算使用众包。

假设你已经获得了花朵数据集中每个图像的边界框（我们假设每个图像有一个边界框）。你需要创建一个数据集，其数据项将是经过预处

理的图像的批量处理，以及它们的类标签和边界框。每个数据项都应为以下形式的元组：（images，（class_labels，bounding_boxes））。然后你就可以训练模型了！

 对边界框应该进行归一化，以便水平坐标和垂直坐标以及高度和宽度都在0到1的范围内。而且通常要预测高度和宽度的平方根，而不是直接的高度和宽度值：通过这种方式，对于大边框的10像素错误将不会像对小边框的10像素错误一样受到惩罚。

MSE通常作为成本函数可以很好地训练模型，但是评估模型对边界框的预测能力不是一个很好的指标。最常用的度量指标是“交并比”（Intersection over Union，IoU）：预测边界框和目标边界框之间的重叠面积除以它们的联合面积（见图14-23）。在tf.keras中，它是由tf.keras.metrics.MeanIoU类实现的。

图14-23：边界框的交并比（IoU）度量

对单个物体进行分类和定位是很好的，但是如果图像中包含多个物体（在花朵数据集中通常如此）怎么办？

[1] Adriana Kovashka et al.，“Crowdsourcing in Computer Vision”，Foundations and Trends in Computer Graphics and

Vision 10, no.3（2014）：177－243.

14.9 物体检测

在图像中对多个物体进行分类和定位的任务称为物体检测。直到几年前，一种通用的方法是采用经过训练的CNN来对单个物体进行分类和定位，然后将其在图像上滑动，如图14-24所示。在此示例中，图像被切成6×8的网格，我们显示了CNN（黑色粗矩形）在所有3×3区域中的滑动。当CNN看着图像的左上方时，它检测到最左边的玫瑰的一部分，然后当它第一次向右移动了一步时，又检测到该玫瑰。在下一步中，它开始检测最上面的玫瑰的一部分，然后再向右移动一步，便再次检测到它。然后，你将继续在整个图像中滑动CNN，查看所有3×3区域。此外，由于物体有各种不同的大小，因此你也可以在不同大小的区域上滑动CNN。例如，完成3×3区域后，你可能还希望在所有4×4区域上滑动CNN。

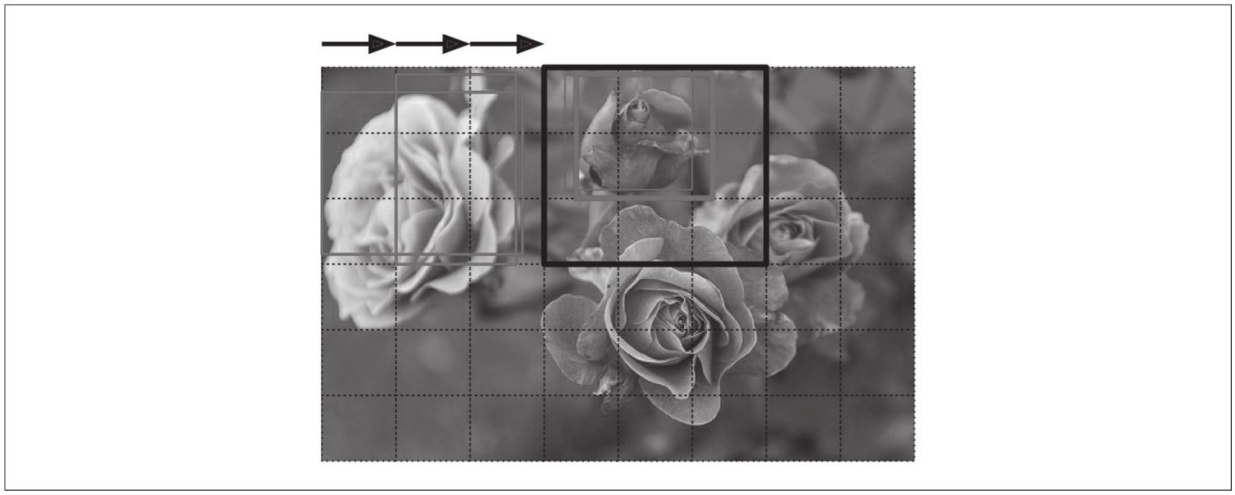


图14-24：通过在图像上滑动CNN来检测多个物体

这项技术非常简单直观，但是如你所见，它将多次检测同一物体，但位置略有不同。然后需要进行一些后期处理，以消除所有不必要的边界框。一种常见的方法称为非极大值抑制。以下是操作方式：

1.首先你需要在CNN中添加一个额外的客观分数（置信度）输出，以估计图像中确实存在花朵的可能性（或者你可以添加“无花朵”类，但这通常不起好的作用）。它必须使用sigmoid激活函数，而且可以使用二元交叉熵损失对其进行训练。然后删除所有置信度得分低于某个阈值的边界框：这将删除所有实际上不包含花的边界框。2.找到具有最高客观分数的边界框，并删除与其重叠很多的所有其他边界框（例如，IoU大于60%）。例如，在图14-24中，具有最大客观分数的边界框是最上面的玫瑰上方的粗边界框（客观分数由边界框的厚度表示）。另一个边界框同一朵玫瑰与最大边界框重叠很多，因此我们将其删除。

3.重复第二步，直到没有更多的边界框可以删除。

这种简单的物体检测方法效果很好，但是它需要多次运行CNN，因此速度很慢。幸运的是，有一种更快的方法可以在图像上滑动CNN：使用全卷积网络（FCN）。

14.9.1　全卷积网络1

FCN的概念由Jonathan Long等人在2015年发表的论文[1]中首次提出，用于语义分割（根据图像所属物体的类别对图像中的每个像素进行分类的任务）。作者指出，你可以用卷积层代替CNN顶部的密集层。为了解这一点，我们来看一个示例：假设一个具有200个神经元的密集层位于一个卷积层的顶部，该卷积层输出100个特征图，每个特征图的大小为7×7（这是特征图的大小，不是内核大小）。每个神经元将计算来自卷积层的所有100×7×7激活的加权和（加上偏差项）。现在让我们看看如果使用200个滤波器（每个滤波器的大小为7×7）并使用"valid"填充将卷积层替换为密集层，会发生什么情况。该层将输出200个特征图，每个特征图为1×1（因为内核正好是输入特征图的大小，并且我们使用的是"valid"填充）。换句话说，它将输出200个数字，就像密集层一样。如果仔细观察卷积层执行的计算，你会发现这些数字与密集层产生的数完全相同。唯一的区别是密集层的输出是形状为

[批量大小，200]的张量，而卷积层将输出形状为[批量大小，1，1，200]的张量。

 要将密集层转换为卷积层，卷积层中的滤波器数必须等于密集层中的单元数，滤波器大小必须等于输入特征图的大小，并且必须使用"valid"填充。步幅可以设置为1或更大，我们在下面很快就会看到。

为什么这很重要？好吧，虽然密集层需要特定的输入大小（因为每个输入特征只有一个权重），但是卷积层可以愉快地处理任何大小的图像[2]（但是，它确实希望其输入具有特定数量的通道，因为每个内核为每个输入通道包含一组不同的权重）。由于FCN仅包含卷积层（以及具有相同属性的池化层），因此可以在任何大小的图像上对其进行训练和执行！12例如，假设我们已经训练了CNN以进行花的分类和定位。它在224×224图像上进行训练，并输出10个数字：通过softmax激活函数输出0到4，这给出了类别概率（每个类别一个）；通过逻辑激活函数输出5，这给出了客观分数；不使用任何激活函数输出6到9，它们表示边界框的中心坐标及其高度和宽度。我们可以将其密集层转换为卷积层。实际上，我们甚至不需要重新培训它。我们可以将权重从密集层复制到卷积层！或者在训练之前将CNN转换为FCN。

现在假设当网络被馈送入一个224×224图像时，输出层之前的最后一个卷积层（也称为bottleneck层）输出7×7特征图（见图14-25的左侧）。如果我们向FCN提供448×448的图像（见图14-25的右侧），则瓶颈层将输出14×14的特征图[3]。由于密集输出层已被卷积层替换，使用了10个大小为7×7的滤波器，填充为"valid"且步幅为1，卷积层输出将由10个特征图组成，每个特征图的大小为8×8（因为14-7+1=8）。换句话说，FCN将只处理一次整个图像，并且将输出一个8×8的网格，其中每个单元包含10个数字（5个类别概率、1个客观分数和4个边界框坐标）。这就像拿一个原始的CNN，以每行8步和每列8步的方式将其在图像上滑动一样。为了可视化，想象一下将原始图像切成14×14的网格，然后在该网格上滑动7×7的窗口。该窗口将有8×8=64个可能的位置，

因此有8×8个预测。但是，由于网络只查看一次图像，因此FCN方法效率更高。实际上，You Only Look Once（YOLO）是一种非常流行的物体检测架构的名称，我们接下来将介绍它。

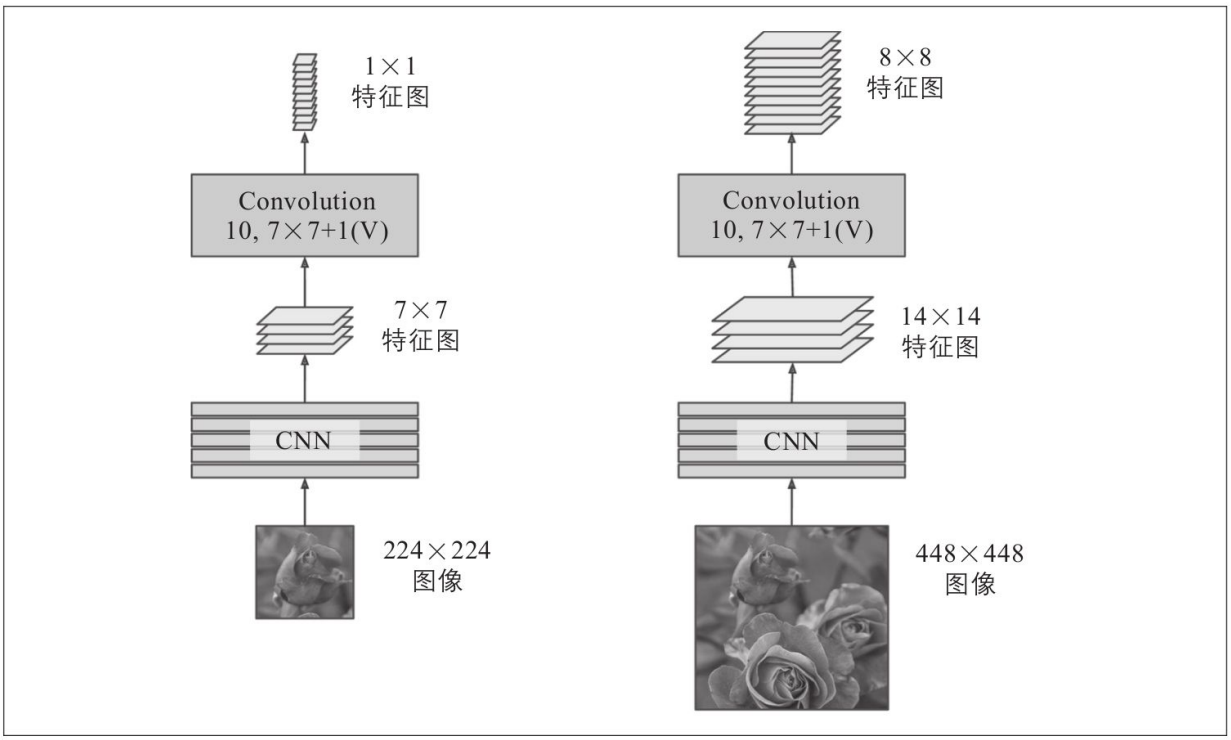


图14-25：相同的全卷积网络处理一个小图像（左）和一个大图像（右）

14.9.2 YOLO

YOLO是Joseph Redmon等人在2015年的论文[4]中提出的一种极其快速、准确的物体检测架构，随后在2016（YOLOv2）[5]和2018（YOLOv3）[6]中进行了改进。如Redmon的演示所示，它是如此之快以至于可以在视频上实时运行。

YOLOv3的架构与我们刚刚讨论的架构非常相似，但有一些重要的区别：

• 它为每个网格单元输出5个边界框（而不是一个），并且每个边界框都带有一个客观分数。由于它在包含20个类的PASCAL VOC数据集上进行了训练，因此每个网格单元还输出20个类概率。每个网格单元共有45个数字：5个边界框，每个边界框具有4个坐标，再加上5个客观分数和20个类概率。

• YOLOv3不预测边界框中心的绝对坐标，而是预测相对于网格单元坐标的偏移，其中（0，0）表示该单元格的左上角，而（1，1）表示右下角。对于每个网格单元，YOLOv3被训练只能预测其中心位于该单元内的边界框（但是边界框本身通常会远远超出网格单元）。YOLOv3将逻辑激活函数应用于边界框坐标，以确保它们保持在0到1的范围内。

• 在训练神经网络之前，YOLOv3会找到5个有代表性的边界框，称为锚框（或先验边界框）。它通过将K-Means算法（见第9章）应用于训练集边界框的高度和宽度来实现。例如，如果训练图像包含许多行人，则锚框之一将可能具有典型的行人的尺寸。然后，当神经网络预测每个网格单元的5个边界框时，它实际上预测了重新缩放每个锚定框的幅度。例如，假设一个锚框的高度为100像素，宽度为50像素，并且网络预测（对于一个网格单元）垂直缩放因子为1.5，水平缩放因子为0.9。那预测的边界框大小为150×45像素。更准确地说，对于每个网格单元和每个锚框，网络都将预测垂直和水平缩放比例的对数。拥有这些先验知识使网络更有可能预测出适当尺寸的边界框，并且由于它能更快地学习到合理边界框的样子，因此也加快了训练速度。

• 该网络使用不同比例的图像进行训练：在训练过程中每隔几批，网络会随机选择一个新的图像尺寸（从330×330到608×608像素）。这使网络能够学习检测不同比例的物体。还可以在不同的比例下使用YOLOv3：较小比例的准确率较低，但速度比较大比例的要快，因此你可以根据你的情况选择合适的折衷方案。

你可能会对更多创新感兴趣，例如使用跳过连接来恢复CNN中丢失的一些空间分辨率（我们将在讨论语义分割时对此进行讨论）。在2016

年的论文中，作者介绍了使用层次分类的YOLO9000模型：该模型预测称为WordTree的视觉层次中每个节点的概率。即使不确定某个特定类型的狗，这也使网络可以很自信地预测图像是否代表狗。我鼓励你继续阅读所有三篇论文：它们阅读起来非常愉悦，提供了如何逐步改善深度学习系统的出色示例。

均值平均精度（mAP）

在物体检测任务中使用的一个常见的指标是均值平均精度（mAP）。“均值平均”听起来有点冗余，不是吗？为了理解该方法，让我们回到第3章中讨论的两个分类指标：精度和召回率。记住这个权衡：召回率越高，精度越低。你可以在精度/召回曲线中将其可视化（见图3-5）。为了将该曲线总结为一个数字，我们可以计算其在曲线下的面积（AUC）。但是请注意，精度/召回曲线可能包含几个部分，某个部分当召回率增加时，精度实际上会提高，尤其是在较低的召回值时（你可以在图3-5的左上方看到它）。这是采用mAP指标的动机之一。

假设分类器在10％的召回率下具有90％的精度，但在20％的召回率下具有96％的精度。这里实际上没有折衷：使用分类器以20％的召回率（而不是10％）更合理，因为你将获得更高的召回率和更高的精度。因此，我们不应该着眼于10％的召回率，而是应该着眼于分类器可以提供至少10％的召回率的最大精度。这是96％，而不是90％。因此要获得关于模型性能的合理概念的一种方法是计算召回率在至少为0％时可以获得的最大精度（然后是10％、20％，以此类推，直至100％），然后计算这些最大精度的平均值。这称为平均精度（AP）指标。当有两个以上类别时，我们可以为每个类别计算AP，然后计算平均AP（mAP）。

在物体检测系统中，存在另外一层的复杂度：如果系统检测到正确的类别但在错误的位置（即边界框完全没有物体）怎么办？当然，我们不应将此视为正的预测。一种方法是定义IOU阈值：例如，我们可以认为只有在IOU大于0.5且预测类别正确时，该预测才是正确的。通常将相应的mAP标记为mAP@0.5（或mAP@50％，或有时为AP50）。在某些比赛中

（例如PASCAL VOC挑战赛）就是这样做的。在其他情况（例如COCO竞争）中，针对不同的IOU阈值（0.50、0.55、0.60，…，0.95）计算mAP，而最终指标是所有这些mAP的平均值（记为AP@[.50：.95]或AP@[.50：0.05：.95]）。是的，这是平均值的平均。

GitHub上提供了使用TensorFlow构建的几种YOLO实现。特别地，请查看Zihao Zang的TensorFlow 2实现。TensorFlow Models项目中还提供了其他物体检测模型，其中许多模型有预先训练的权重。有些甚至已经移植到TF Hub，例如SSD[7]和FasterRCNN[8]，它们都很流行。与YOLO相似，SSD还是“单发（single shot）”检测模型。Faster RCNN也更加复杂：图像首先经过CNN，然后将输出传递到区域提议网络（Region Proposal Network，RPN），该网络在一定区域内提议最有可能包含物体的边界框，并基于CNN的裁剪后的输出对每个边界框运行分类器。

检测系统的选择取决于许多因素：速度、精度、可用的预训练模型、训练时间、复杂性等。论文包含性能度量表，但是测试环境中存在很多可变性，技术发展速度如此之快，以至于很难进行公平的比较，而这种比较对大多数人都有用，并且可以持续有效几个月以上。

因此我们可以通过在物体周围绘制边界框来定位对象。但是也许你想变得更加精确。让我们看看如何来下降到像素级别。

[1] Jonathan Long et al.，“Fully Convolutional Networks for Semantic Segmentation”，Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition（2015）：3431–3440.
[2] 有一个小例外：如果输入大小小于内核大小，则使用"valid"填充的卷积层将报错。
[3] 这假设我们在网络中仅使用"same"填充：实际上，"valid"填充会减小特征图的大小。此外，可以将448除以2几次，直到等于7，而没有任何舍入误差。如果任何层使用了不同于1或2的步幅，则可能存在一些舍入误差，因此特征图最终可能会变小。

[4] Joseph Redmon et al.，“You Only Look Once：Unified，Real-Time Object Detection”，Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition（2016）：779-788.
[5] Joseph Redmon and Ali Farhadi，“YOLO9000：Better，Faster，Stronger”，Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition（2017）：6517-6525.
[6] Joseph Redmon and Ali Farhadi，“YOLOv3：An Incremental Improvement，”arXiv preprint arXiv：1804.02767（2018）.
[7] Wei Liu et al.，“SSD：Single Shot Multibox Detector”，Proceedings of the 14th European Conference on Computer Vision 1（2016）：21-37.
[8] Shaoqing Ren et al.，“Faster R-CNN：Towards Real-Time Object Detection with Region Proposal Networks”，Proceedings of the 28th International Conference on Neural Information Processing Systems 1（2015）：91-99.

14.10　语义分割

在语义分割中，每个像素根据其所属物体的类别（例如，道路、汽车、行人、建筑物等）进行分类，如图14-26所示。请注意，没有区分相同类别的不同物体。例如分割图像右侧的所有自行车都变成了一大块像素。此任务的主要困难在于，当图像通过常规的CNN时，它们会逐渐失去其空间分辨率（由于步幅大于1的层），因此常规的CNN可能最终会知道在图像的左下方某处有一个人，但不会比这更精确了。

就像物体检测一样，有许多种方法可以解决此问题，有些方法非常复杂。但是，Jonathan Long等人在2015年的论文中提出了一个我们之前讨论过的相当简单的解决方法。作者首先采用经过预先训练的CNN，然后将其转换为FCN。CNN对输入图像应用的总步幅为32（如果所有步幅的总和都大于1），则意味着最后一层输出的特征图是输入图像的1/32。这显然太粗糙了，因此他们添加了一个单独的上采样层来把分辨率乘以32。

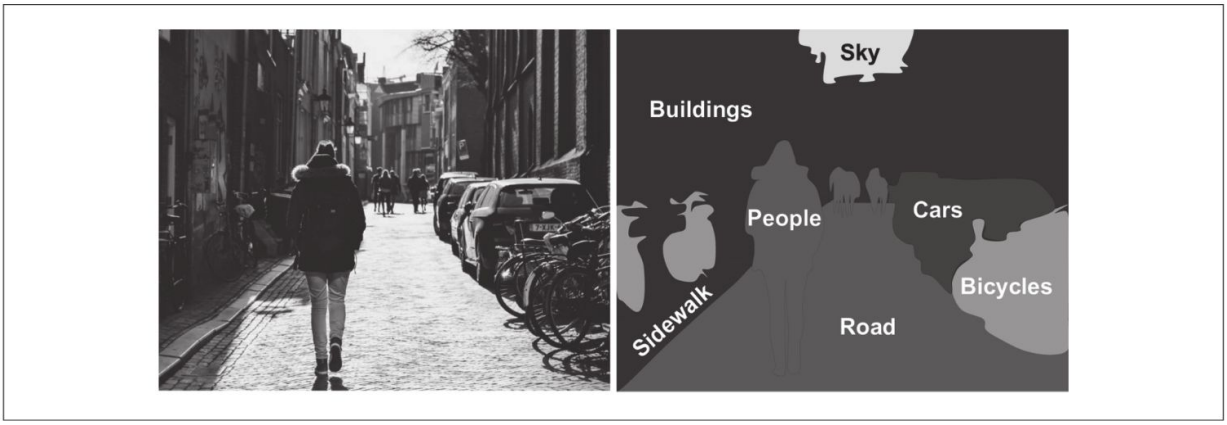


图14-26：语义分割

有几种解决方法可用于上采样（增加图像的大小），例如双线性插值，但仅在×4或×8时才有效。取而代之的是，他们使用转置的卷积层[1]：等效于首先插入空的行和列（充满零）来拉伸图像，然后执行常

规的卷积（见图14-27）。另外，有些人更喜欢将其视为使用分数步幅的常规卷积层（例如，图14-27中的1/2步幅）。可以对转置的卷积层进行初始化以执行接近于线性插值的操作，但是由于它是可训练的层，因此在训练过程中会学习得更好。在tf.keras中，可以使用Conv2DTranspose层。

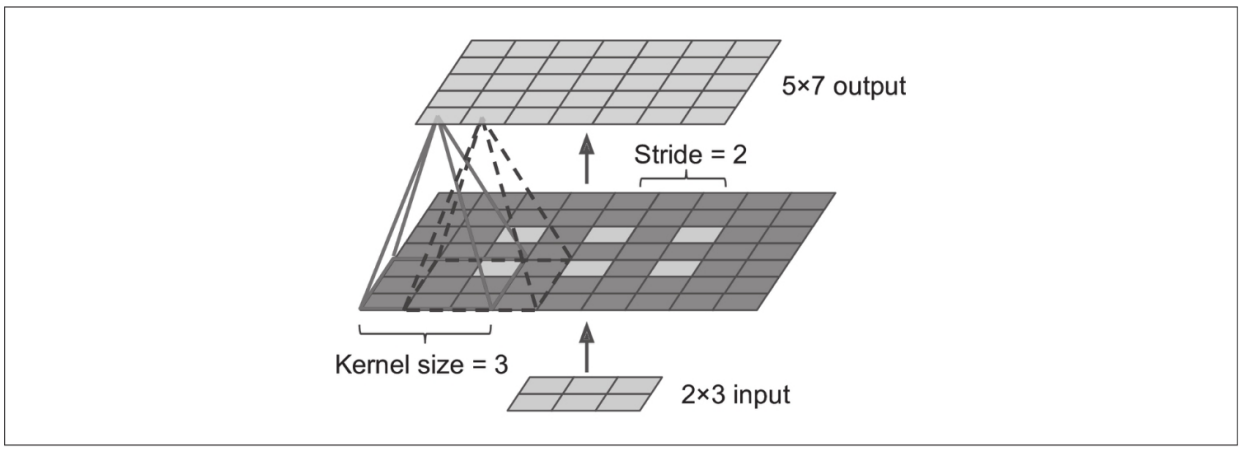


图14-27：使用转置卷积层进行上采样

在转置的卷积层中，步幅定义了输入将被拉伸的程度，而不是滤波步长的大小，因此步幅越大，输出就越大（与卷积层或池化层不同）。

TensorFlow卷积运算

TensorFlow还提供了其他几种卷积层：

keras.layers.Conv1D

为一维输入（例如时间序列或文本（字母或单词的序列））创建卷积层，我们将在第15章中看到。

keras.layers.Conv3D

为3D输入（例如3D PET扫描）创建卷积层。

dilation_rate

将任何卷积层的dilation_rate超参数设置为2或更大的值会创建“à-trous卷积层”（“àtrous”是“带孔”的法语）。这等效于使用常规卷积层，并通过插入为零的行和列（例如，孔）来扩大过滤器。例如，可以以4的扩展率来扩展[[1，2，3]]的1×3滤波器，从而得到[[1，0，0，0，2，0，0，0，3]]扩张过滤器。这使卷积层无须任何计算费用，也无须使用任何额外参数就具有更大的接受野。

tf.nn.depthwise_conv2d（）

可用于创建深度卷积层（但需要自己创建变量）。它将每个滤波器独立地应用于每个单独的输入通道。因此，如果有f_n个滤波器和f_n'个输入通道，则将输出$f_n \times f_n$'特征图。这个解决方法可以，但仍然不够精确。为了做得更好，作者添加了来自较低层的跳过连接：例如，他们将输出图像上采样2倍（而不是32倍），并添加了具有两倍分辨率的较低层的输出。然后，他们对结果进行16倍的上采样，从而得到32倍的总上采样（见图14-28）。这样可以恢复一些早期池化层丢失的空间分辨率。在他们的最佳架构中，他们使用了第二个类似的跳过连接来从更低的层恢复甚至更精细的细节。简而言之，原始CNN的输出经过以下额外步骤：放大×2，添加较低层（适当比例）的输出，放大×2，添加甚至更低层的输出，最后放大×8。甚至有可能扩大到超出原始图像的大小：这可用于提高图像的分辨率，这是一种称为超分辨率的技术。

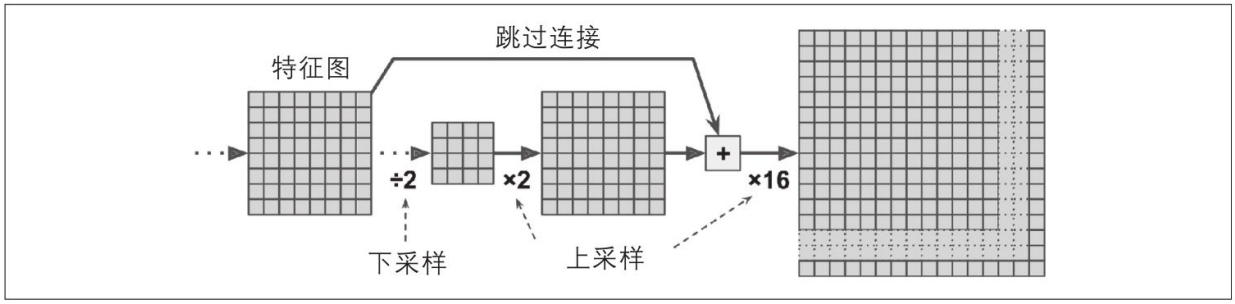


图14-28：跳过层从较低层恢复一些空间分辨率

再次，许多GitHub仓库提供了TensorFlow语义分割的实现（现在为TensorFlow 1），你甚至可以在TensorFlow Models项目中找到经过预训练的实例分割模型。实例分割与语义分割相似，不是合并同一类的所有物体归为一个大块，而是将每个物体与另一个物体区分开（例如，它标识每个自行车）。目前，TensorFlow Models项目中可用的实例分割模型基于Mask R-CNN架构，该架构在2017年的一篇论文中提出[2]：它通过以下方式扩展了Faster R-CNN模型：为每个边界框生成一个像素掩码。因此你不仅可以获得围绕每个物体的边界框，并具有一组估计的类别概率，而且还会得到一个像素掩码，该像素掩码可在边界框中定位属于该物体的像素。

如你所见，深度计算机视觉的领域广阔且发展迅速，每年都涌现出各种基于卷积神经网络的各种架构。在短短几年中取得的进展令人震惊，研究人员现在专注于越来越多的问题，例如对抗性学习（尝试使网络对旨在欺骗其的图像更具抵抗力）、可解释性（理解为什么网络会进行特定的分类），逼真的图像生成（我们将在第17章中介绍）和单次学习（系统只要看一次就可以识别出物体）。一些人甚至探索了全新的架构，例如Geoffrey Hinton的胶囊网络[3]（我在几个视频中展示了它们，并在notebook本中提供了相应的代码）。现在进入第15章，我们将研究如何使用递归神经网络和卷积神经网络处理时序数据。

[1] 这种类型的层有时称为反卷积层，但是它不执行数学家所谓的反卷积，因此应避免使用此名称。

[2] Kaiming He et al.，“Mask R-CNN”，arXiv preprint arXiv:1703.06870（2017）.

[3] Geoffrey Hinton et al.，“Matrix Capsules with EM Routing”，Proceedings of the International Conference on Learning Representations（2018）.

14.11 练习题

1.与用于图像分类的全连接的DNN相比，CNN有什么优势？

2.考虑由三个卷积层组成的CNN，每个卷积层具有3×3内核，步幅为2和"same"填充。最低层输出100个特征图，中间层输出200个特征图，最顶层输出400个特征图。输入图像是200×300像素的RGB图像。

CNN中的参数总数是多少？如果我们使用的是32位浮点数，那么在对单个实例进行预测时，至少该网络需要多少RAM？训练一个包含50个图像的小批量时会怎样？

3.如果训练CNN时GPU内存不足，可以尝试哪5种方法来解决这个问题？

4.为什么要添加最大池化层而不是具有相同步幅的卷积层？

5.你何时要添加局部响应归一化层？

6.与LeNet-5相比，你能说出AlexNet的主要创新之处吗？GoogLeNet、ResNet、SENet和Xception的主要创新是什么呢？

7.什么是全卷积网络？如何将密集层转换为卷积层？

8.语义分割的主要技术困难是什么？

9.从头开始构建自己的CNN，并尝试在MNIST上实现最高的准确性。

10.将迁移学习用于大图像分类，请执行以下步骤：

a. 创建一个训练集，每个类至少包含100个图像。例如，你可以根据位置（海滩、山脉、城市等）对自己的图片进行分类，或者可以使用现有的数据集（例如来自TensorFlow数据集）。

b. 将其分为训练集、验证集和测试集。

c. 构建输入流水线，包括适当的预处理操作，可选择地添加数据扩充。

d. 在此数据集上微调预训练的模型。

11. 阅读TensorFlow的风格转换教程。这是使用深度学习生成艺术的一种有趣方式。

附录A中提供了这些练习题的解答。

第15章　使用RNN和CNN处理序列

击球手将球击出。外野手立即开始奔跑，预测球的轨迹。他追踪它，调整自己的运动，最后抓住球（在一片掌声中）。不管你是在听完朋友的话还是在早餐时期待咖啡的味道，预测都是你一直在做的事情。在本章中，我们将讨论循环神经网络（RNN），这是一类可以预测未来的网络（当然，是一定程度上）。它们可以分析时间序列数据（例如股票价格），并告诉你何时进行买卖。在自动驾驶系统中，它们可以预测汽车的行驶轨迹并帮助避免发生事故。更笼统地说，它们可以处理任意长度的序列，而不是像我们到目前为止考虑的所有网络一样，用于固定大小的输入。例如，它们可以将句子、文档或音频样本作为输入，使其对于自然语言处理应用非常有用，例如自动翻译或语音转文本。

在本章中，我们将首先研究RNN的基本概念，以及如何使用时间反向传播进行训练，然后再使用它们来预测时间序列。之后，我们将探讨RNN面临的两个主要困难：

- 不稳定的梯度（在第11章中讨论过），可以使用多种技术来解决，包括递归dropout和递归层归一化。

- （非常）有限的短期记忆，可以使用LSTM和GRU单元进行扩展。

RNN并不是唯一能够处理顺序数据的神经网络类型：对于较小的序列，常规的密集网络可以解决问题。对于很长的序列，例如音频样本或文本，卷积神经网络实际上也可以很好地工作。我们将讨论这两种可能性，通过实现WaveNet来结束本章：这是一种CNN架构，能够处理成千上万个时间步长的序列。在第16章中，我们将继续探索RNN，看看如何将其用于自然语言处理，以及基于注意力机制的最新架构。让我们开始吧！

15.1　循环神经元和层

到目前为止，我们只关注前馈神经网络，其中激活仅在一个方向上流动，从输入层流向输出层（附录E中讨论了一些例外情况）。循环神经网络看起来非常像前馈神经网络，除了它还具有指向反向的连接。让我们看一下最简单的RNN，它由一个神经元接收输入，产生输出并将该输出反送回自身组成，如图15-1（左）所示。在每个时间步长t（也称为帧），该循环神经元接受输入$x_{(t)}$和前一个时间步长$y_{(t-1)}$的输出。由于在第一个时间步长中没有先前的输出，因此通常将其设置为0。我们可以相对于时间轴来表示这个小网络，如图15-1（右）所示。这被称为随着时间展开网络（它是同一递归神经元，每个时间步长代表一个）。

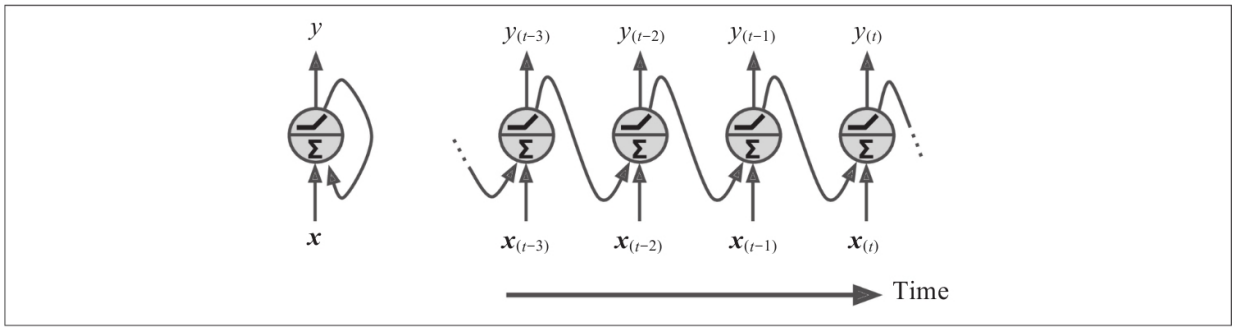


图15-1：循环神经元（左）随着时间展开（右）

你可以轻松地创建一层递归神经元。在每个时间步长t，每个神经元接受输入向量$x_{(t)}$和前一个时间步长的输出向量$y_{(t-1)}$，如图15-2所示。请注意，现在输入和输出都是向量（当只有一个神经元时，输出是标量）。

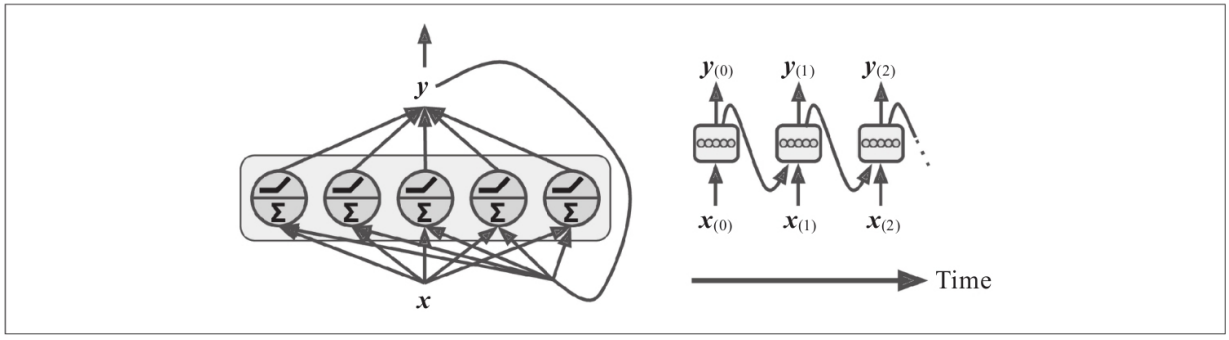


图15-2：一层循环神经元（左）随着时间展开（右）

每个循环神经元都有两组权重：一组用于输入$x_{(t)}$，另一组用于前一个时间步长$y_{(t-1)}$的输出。我们称这些权重向量为w_x和w_y。如果考虑整个循环层而不是仅仅一个循环神经元，则我们可以将所有权重向量放在两个权重矩阵W_x和W_y中。然后可以如预期的那样计算整个循环层的输出向量，如公式15-1所示（b是偏置向量，$\phi(\cdot)$是激活函数（例如ReLU[1]）。

公式15-1：单个实例的循环层的输出

$$\boldsymbol{y}_{(t)}=\phi\left(\boldsymbol{W}_x^{\mathrm{T}}\boldsymbol{x}_{(t)}+\boldsymbol{W}_y^{\mathrm{T}}\boldsymbol{y}_{(t-1)}+\mathbf{b}\right)$$

正如前馈神经网络一样，我们可以通过将时间步长t处的所有输入都放在输入矩阵$X_{(t)}$中，来一次性地计算整个小批量中的递归层的输出（见公式15-2）。

公式15-2：小批量中的所有实例的循环神经元层的输出

$$\boldsymbol{Y}_{(t)}=\phi\left(\boldsymbol{X}_{(t)}\boldsymbol{W}_x+\boldsymbol{Y}_{(t-1)}\boldsymbol{W}_y+\mathbf{b}\right)$$

$$=\phi\left([\boldsymbol{X}_{(t)}\boldsymbol{Y}_{(t-1)}]\boldsymbol{W}+\mathbf{b}\right)\ 其中\ \boldsymbol{W}=\begin{bmatrix}\boldsymbol{W}_x\\ \boldsymbol{W}_y\end{bmatrix}$$

在此等式中：

$Y_{(t)}$是一个$m\times n_{neurons}$矩阵，包含小批量处理中每个实例在时间步长t时该层的输出（m是小批量处理中的实例数量，$n_{neurons}$是神经元数量）。

$X_{(t)}$是一个$m\times n_{inputs}$矩阵，包含所有实例的输入（n_{inputs}是输入特征的数量）。

W_x是一个$n_{inputs}\times n_{neurons}$矩阵，包含当前时间步长的输入的连接权重。

W_y是一个$n_{neurons}\times n_{neurons}$矩阵，包含前一个时间步长的输出的连接权重。

b是大小为$n_{neurons}$的向量，包含每个神经元的偏置项。

• 权重矩阵W_x和W_y经常垂直合并成形状为（$n_{inputs}+n_{neurons}$）$\times n_{neurons}$的单个权重矩阵W（见公式15-2的第二行）。

• 符号$[X_{(t)}\ Y_{(t-1)}]$表示矩阵$X_{(t)}$和$Y_{(t-1)}$的水平合并。

请注意，$Y_{(t)}$是$X_{(t)}$和$Y_{(t-1)}$的函数，$Y_{(t-1)}$是$X_{(t-1)}$和$Y_{(t-2)}$的函数，而$Y_{(t-2)}$是$X_{(t-2)}$和$Y_{(t-3)}$的函数，以此类推。自时间t=0以来，这使$Y_{(t)}$成为所有输入的函数（即$X_{(0)}$，$X_{(1)}$，…，

$X_{(t)}$）。在第一个时间步长t=0时，没有先前的输出，因此通常假定它们均为零。

15.1.1 记忆单元

由于在时间步长t时递归神经元的输出是先前时间步长中所有输入的函数，因此你可以说它具有记忆的形式。在时间步长上保留某些状态的神经网络的一部分称为记忆单元（或简称为单元）。单个循环神经元或一层循环神经元是一个非常基本的单元，它只能学习短模式（通常约为10个步长，但这取决于任务）。在本章的后面，我们将介绍一些能够学习更长模式（大约要长10倍，但这又取决于任务）的更复杂、功能更强大的单元类型。

通常，在时间步长t的单元状态，表示为$h_{(t)}$（“h”代表“隐藏”），是该时间步长的某些输入和其前一个时间步长状态的函数：$h_{(t)}=f(h_{(t-1)}, x_{(t)})$。它在时间步长t处的输出表示为$y_{(t)}$，也是先前状态和当前输入的函数。就目前为止我们讨论的基本单元而言，输出仅等于状态，但在更复杂的单元中，情况并非总是如此，如图15-3所示。

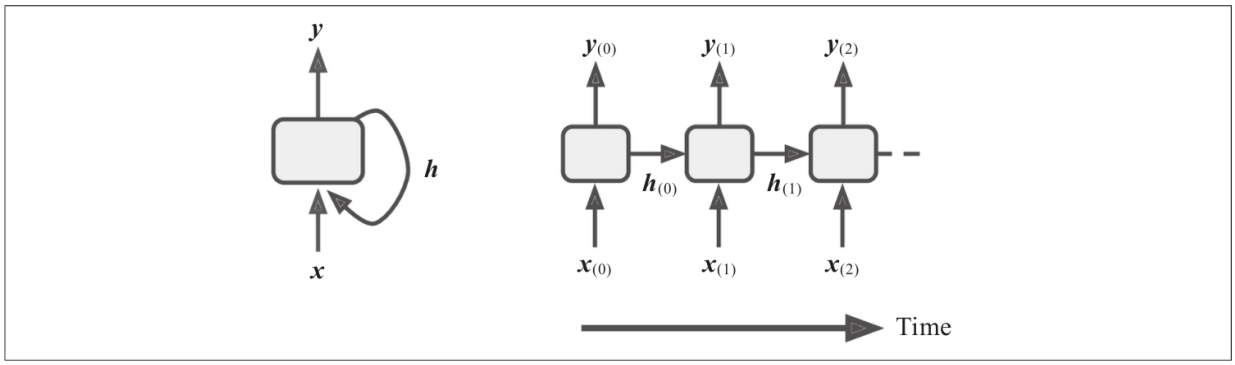


图15-3：单元的隐藏状态和其输出可能不同

15.1.2 输入和输出序列

RNN可以同时接受输入序列并产生输出序列（见图15-4的左上方网络）。这种类型的序列到序列的网络可用于预测诸如股票价格之类的时间序列：你将过去N天的价格作为输入，它必须输出未来偏移一天的价格（即从前N-1天到明天）。

或者，你可以向网络提供一个输入序列，并忽略除了最后一个输出外的所有输出（见图15-4右上角的网络）。换句话说，这是一个序列到向量的网络。例如，你可以向网络提供与电影评论相对应的单词序列，然后网络将输出一个情感得分（例如，从-1[恨]到+1[爱]）。

相反，你可以在每个时间步长中一次又一次地向网络提供相同的输入向量，并让其输出一个序列（见图15-4的左下网络）。这是一个向量到序列的网络。例如，输入可以是图像（或CNN的输出），而输出可以是该图像的描述。

最后，你可能有一个称为编码器的序列到向量的网络，然后是一个称为解码器的向量到序列的网络（见图15-4的右下网络）。例如，这可以用于将句子从一种语言翻译成另一种语言。你向网络提供一种语言的句子，编码器会将其转换为单个向量的表征，然后解码器会将此向量解码为另一种语言的句子。这种称为“编码器-解码器（EncoderDecoder）”的两步模型比使用单个序列到序列的RNN进行即时翻译要好得多（就像图左上角所示）：句子的最后一个单词会影响翻译的第一个单词，因此在翻译之前你需要等待直到看完整个句子。我们将在第16章中了解如何实现编码器-解码器（我们会看到，它比图15-4中所示的复杂得多）。

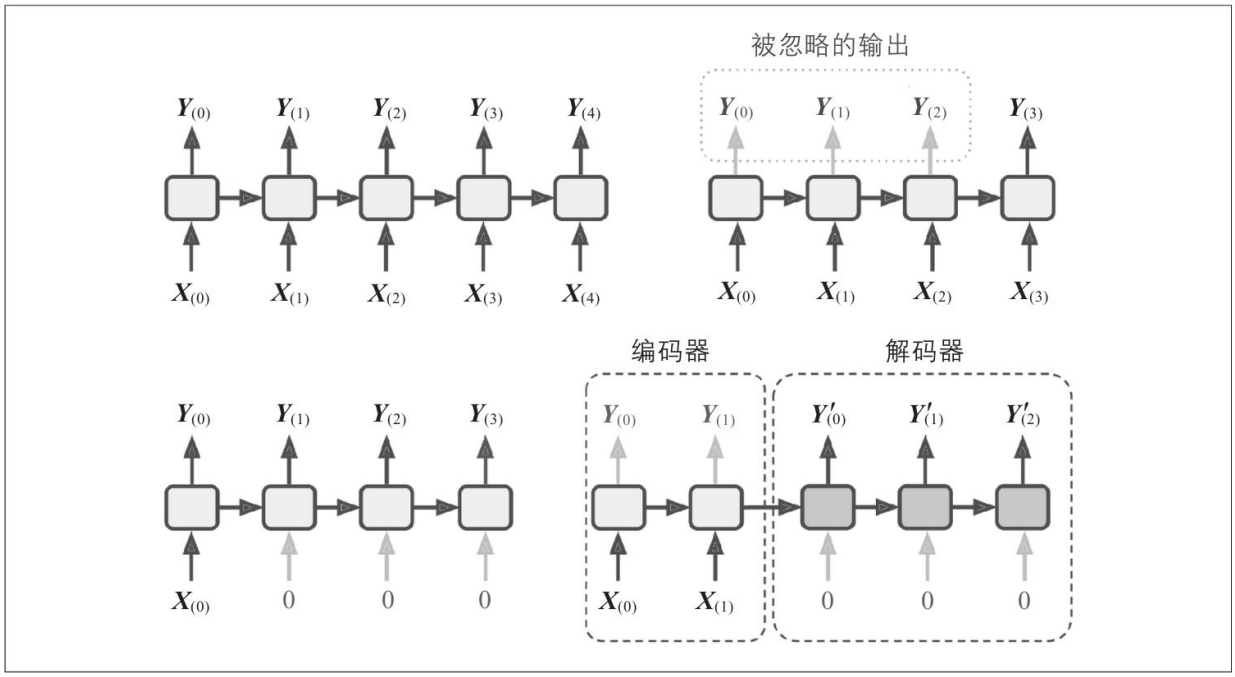


图15-4：序列到序列（左上），序列到向量（右上），向量到序列（左下）和编码器-解码器（右下）网络

听起来很有前途，但是如何训练循环神经网络呢？

[1] 请注意，许多研究人员更喜欢在RNN中使用双曲正切（tanh）激活函数，而不是ReLU激活函数。例如，请看看Vu Pham等人在2013年发表的论文“Dropout Improves Recurrent Neural Networks for Handwriting Recognition”。基于ReLU的RNN也是可能的，如Quoc V.Le等人2015年的论文“A Simple Way to Initialize Recurrent Networks of Rectified Linear Units”所示。

15.2 训练RNN

要训练RNN，诀窍是将其按照时间逐步展开（就像我们刚才所做的那样），然后简单地使用常规的反向传播（见图15-5）。这种策略称为“时间反向传播”（BackPropagation Through Time，BPTT）。

就像在常规反向传播中一样，这里有一个通过展开网络的第一次前向通路（以虚线箭头表示）。然后使用成本函数C（$Y_{(0)}$，$Y_{(1)}$，…，$Y_{(T)}$）（其中T是最大时间步长）来评估输出序列。请注意，此成本函数可能会忽略某些输出，如图15-5所示（例如，在序列到向量RNN中，除最后一个输出外，所有的输出都将被忽略）。然后，该成本函数的梯度通过展开的网络反向传播（由实线箭头表示）。最后，使用在BPTT期间计算的梯度来更新模型参数。请注意，梯度反向通过成本函数使用的所有输出，而不是最终输出（例如，在图15-5中，成本函数是使用网络的最后三个输出$Y_{(2)}$，$Y_{(3)}$和$Y_{(4)}$来计算的，因此梯度流过这三个输出，但不能通过$Y_{(0)}$和$Y_{(1)}$）。而且，由于在每个时间步长使用相同的参数W和b，所以反向传播会做正确的事情，在所有时间步长上求和。

幸运的是，tf.keras为你解决了所有这些复杂问题，让我们开始编码！

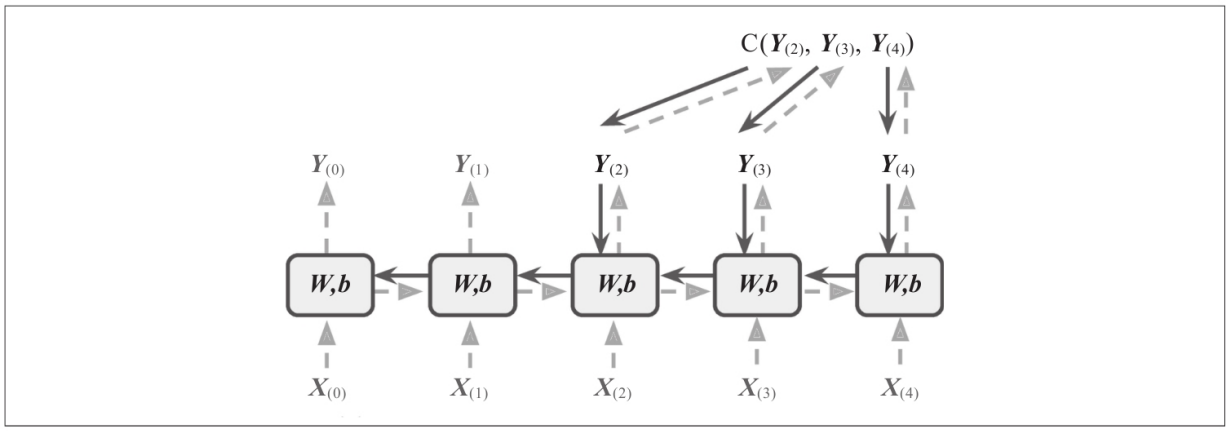


图15-5：时间反向传播

15.3 预测时间序列

假设你正在研究网站上每小时的活跃用户数、城市的每日温度或使用多个指标来评估每个季度公司的财务状况。在所有这些情况下，数据是每个时间步长一个或多个值的序列。这称为时间序列。在前两个示例中，每个时间步长只有一个值，因此它们是单变量时间序列，而在财务示例中，每个时间步长有多个值（例如，公司的收入、债务等），因此它是一个多变量时间序列。典型的任务是预测未来值，这称为预测。另一个常见任务是填补空白：预测过去的缺失值。这称为插补。例如，图15-6显示了3个单变量时间序列，每个时间序列的长度是50个时间步长，此处的目标是预测它们下一个时间步长的值（用X表示）。

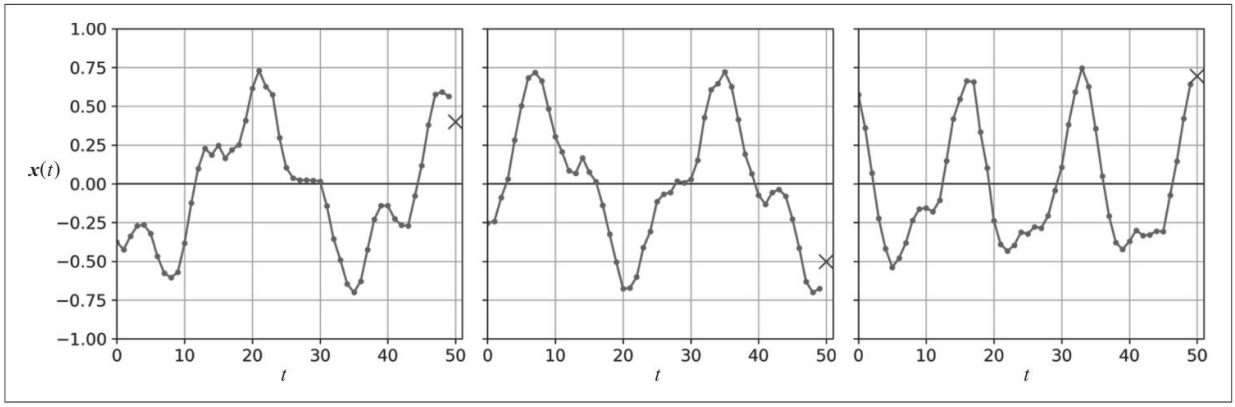


图15-6：时间序列预测

为简单起见，我们使用由generate_time_series（）函数生成的时间序列，如下所示：

```
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10))  #   wave 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) # + wave 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5)   # + noise
    return series[..., np.newaxis].astype(np.float32)
```

此函数根据请求（通过batch_size参数）创建任意数量的时间序列，每个序列的长度为n_steps，并且每个序列中每个时间步长只有一个值（即所有序列都是单变量的）。该函数返回一个形状为[批处理大小，时间步长，1]的NumPy数组，其中每个序列是两个固定振幅但频率和相位随机的正弦波的总和，再加上一点噪声。

 在处理时间序列（以及其他类型的序列，例如句子）时，输入特征通常表示为形状为[批处理大小，时间步长，维度]的3D数组，其中单变量时间序列的维度为1，多变量时间序列的维度更多。

现在，使用此函数创建训练集、验证集和测试集：

```
n_steps = 50
series = generate_time_series(10000, n_steps + 1)
X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]
X_test, y_test = series[9000:, :n_steps], series[9000:, -1]
```

X_train包含7000个时间序列（即其形状为[7000，50，1]），而X_valid包含2000个（从第7000个时间序列到8999个），X_test包含1000个（从9000个时间序列到9999个）。由于我们要为每个序列预测一个值，因此目标是列向量（例如，y_train的形状为[7000，1]）。

15.3.1　基准指标

在开始使用RNN之前，通常最好有一些基准指标，否则我们可能会认为我们的模型工作得很好，但是实际上它的表现要比基本模型差。例如，最简单的方法是预测每个序列中的最后一个值。这被称为单纯预测，有时会令人惊讶地表现不好。在这种情况下，它给了我们约0.020的均方误差：

```
>>> y_pred = X_valid[:, -1]
>>> np.mean(keras.losses.mean_squared_error(y_valid, y_pred))
0.020211367
```

另一种简单的方法是使用全连接的网络。由于它希望每个输入都有一个平的特征列表，因此我们需要添加一个Flatten层。让我们只使用一个简单的线性回归模型，使每个预测是时间序列中值的线性组合：

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])
```

如果我们使用MSE损失和默认的Adam优化器来编译此模型，然后在训练集上训练20个轮次并在验证集上进行评估，得出的MSE约为0.004。这比单纯预测的方法好得多！

15.3.2 实现一个简单的RNN

让我们看看是否可以用简单的RNN击败它：

```
model = keras.models.Sequential([
  keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

那确实是你可以构建的最简单的RNN。如图15-1所示，它仅包含一个有单个神经元的单层。我们不需要指定输入序列的长度（与之前的模型不同），因为循环神经网络可以处理任意数量的时间步长（这就是将第一个输入的维度设置为None的原因）。默认情况下，SimpleRNN层使用双曲正切激活函数。它的工作原理与我们之前看到的完全一样：初始状态$h_{(init)}$设置为0，并将其与第一个时间步长的值$x_{(0)}$一起传递给单个循环神经元。神经元计算这些值的加权总和，并将双曲正切激活函

数应用于其结果，这得出了第一个输出y_0。在简单的RNN中，此输出也是新状态h_0。这个新状态与下一个输入值$x_{(1)}$一起传递给相同的循环神经元，并重复该过程直到最后一个时间步长。然后，该层仅输出最后一个值y_{49}。对于每个时间序列，所有这些都是同时执行的。

 默认情况下，Keras中的循环层仅返回最终输出。要使它们每个时间步长返回一个输出，必须设置return_sequences=True。

如果你编译、拟合和评估该模型（就像之前一样，我们使用Adam训练了20个轮次），你会发现其MSE只有0.014，因此它比单纯方法要好，但它还不能击败简单的线性模型。请注意，对于每个神经元，线性模型的每个输入和每个时间步长都有一个参数，加上一个偏置项（在我们使用的简单线性模型中，共有51个参数）。相反，对于简单RNN中的每个循环神经元，每个输入和每个隐藏状态维度只有一个参数（在简单RNN中，这只是该层中循环神经元的数量），加上一个偏置项。在这个简单的RNN中，总共只有三个参数。

趋势和季节性

还有许多其他模型可以预测时间序列，例如加权移动平均模型或自回归集成移动平均（ARIMA）模型。其中有些要求你首先删除趋势和季节性。例如，如果你正在研究网站上的活动用户数，并且该数字每月以10%的速度增长，你需要从时间序列中消除这种趋势。一旦训练完模型并开始进行预测后，你不得不将趋势添加回去来得到最终的预测。同样，如果你试图预测每个月出售的防晒乳液的数量，你可能会观察到强烈的季节性：因为它在夏季会销售得非常好，而且每年都会重复类似的模式。你必须从时间序列中删除此季节性因素，例如，通过计算每个时间步长的值与一年前的值之间的差（这种技术称为差分）。同样，在对模型进行训练并做出预测之后，你将不得不重新添加季节性模式来得到最终的预测。

使用RNN时，通常不需要执行所有这些操作，但是在某些情况下可能会提高性能，因为该模型不需要学习趋势或季节性。

显然，我们的RNN太简单了以致无法获得良好的性能。因此，让我们尝试添加更多的循环层！

15.3.3 深度RNN

把很多层单元堆叠起来是非常普遍的，如图15-7所示。这给了你深度RNN。

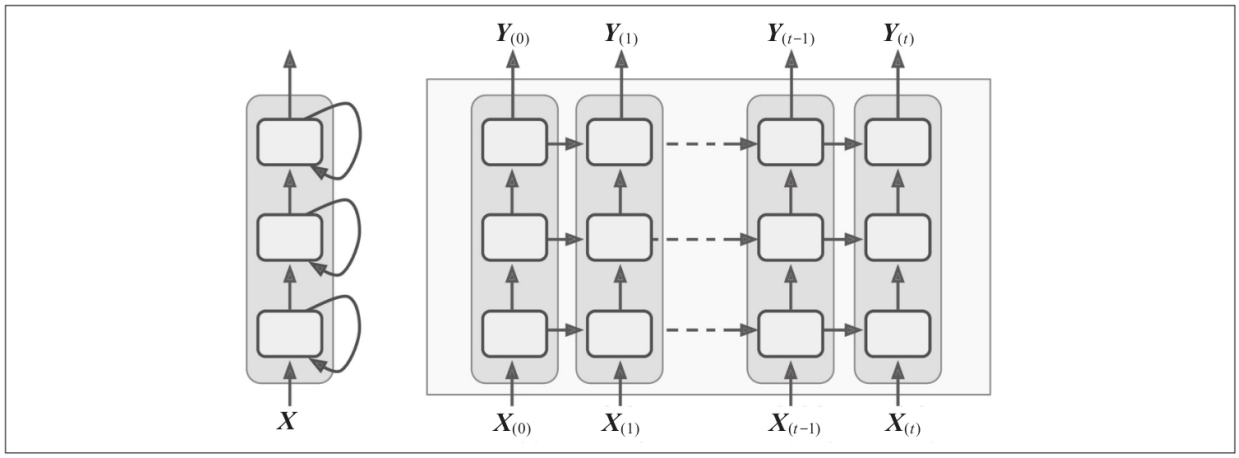


图15-7：深度RNN（左）和随时间展开（右）

使用tf.keras实现深度RNN非常简单：只需堆叠循环层。在此示例中，我们使用了三个SimpleRNN层（但我们可以添加任何其他类型的循环层，例如LSTM层或GRU层，我们将在稍后对此进行讨论）：

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])
```

 确保为所有的循环层设置了return_sequences=True（如果只关心最后一个输出，则最后一个除外）。如果你不这样做，它们会输出一个2D数组（仅包含最后一个时间步长的输出），而不是一个3D数组（包含所有时间步长的输出），并且下一个循环层会抱怨你没有按预期的3D格式提供序列。

如果编译、拟合和评估此模型，你会发现它达到的MSE为0.003。我们终于设法击败了线性模型！

请注意，最后一层不是理想的：它必须有一个单元，因为我们要预测一个单变量时间序列，这意味着我们每个时间步长必须有一个输出值。但是，只有一个单元意味着隐藏状态只是一个数字。那真的不多，可能没什么用。RNN主要使用其他循环层的隐藏状态来传递其所需的所有信息，而不会使用最终层的隐藏状态。此外，由于默认情况下SimpleRNN层使用tanh激活函数，因此预测值必须在-1到1的范围内。但是，如果你要使用其他激活函数怎么办？由于这两个原因，最好把输出层替换为Dense层：运行速度稍快，精度大致相同，并且可以让我们选择所需的任何输出激活函数。如果你做了更改，请确保从第二个（现在是最后一个）循环层中删除return_sequences=True：

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])
```

如果你训练此模型，会发现它收敛得更快并且性能也一样好。另外，你可以根据需要更改输出激活函数。

15.3.4　预测未来几个时间步长

到目前为止，我们仅仅预测了下一个时间步长的值，但是我们可以很容易地通过适当地更改目标来预测未来几步的值（例如，要预测下十步，只需将目标更改为未来10步，而不是未来1步）。但是，如果我们要预测接下来的10个值怎么办？

第一种选择是使用已经训练好的模型，使其预测下一个值，然后将该值加到输入中（就像这个预测值实际上已经有了），然后再次使用该模型预测后面的值，以此类推，如以下代码所示：

```
series = generate_time_series(1, n_steps + 10)
X_new, Y_new = series[:, :n_steps], series[:, n_steps:]
X = X_new
for step_ahead in range(10):
    y_pred_one = model.predict(X[:, step_ahead:])[:, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

Y_pred = X[:, n_steps:]
```

如你所料，由于误差可能会累积（如图15-8所示），因此对下一步长的预测通常会比对未来几个时间步长的预测更为准确。如果你在验证集上评估此方法，会发现MSE约为0.029。这比以前的模型要高得多，但这也是一个更难的任务，因此比较没有太大的意义。把这个性能与单纯预测（仅仅预测这个时间序列在10个时间步长内保持不变）或简单的线性模型进行比较会有更大的意义。单纯方法很糟糕（它的MSE约为0.223），但是线性模型的MSE约为0.0188：这比使用RNN一次预测未来的一步要好得多，而且训练和运行速度也更快。如果你只想预测一些较复杂的任务的未来几个时间步长，这个方法可能会工作得很好。

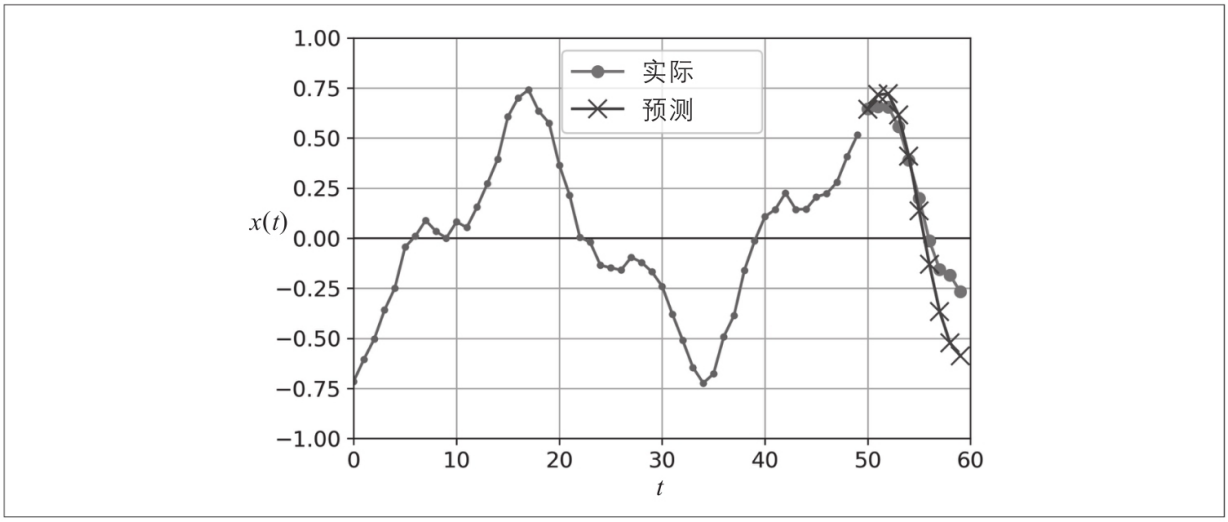


图15-8：预测未来的10步，一次1步

第二种选择是训练RNN一次预测所有10个值。我们仍然可以使用一个序列到向量的模型，但是它输出10个值而不是1个值。但是，我们首先需要将目标更改为包含接下来10个值的向量：

```
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -10:, 0]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]
```

现在我们只需要输出层有10个单元，而不是1个：

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])
```

训练完此模型后，你可以轻松地一次预测接下来的10个值：

```
Y_pred = model.predict(X_new)
```

该模型运行良好：接下来10个时间步长的MSE约为0.008。这比线性模型好得多。但是我们仍然可以做得更好：实际上，与其训练模型在最后一个时间步长预测下一个10个值，不如训练模型在每个时间步长来预测下一个10个值。换句话说，我们可以将这个序列到向量的RNN转换为序列到序列的RNN。这种技术的优势在于，损失将包含每个时间步长的RNN输出项，而不仅仅是最后一个时间步长的输出项。这意味着将有更多的误差梯度流过模型，它们不需要随时间而“流淌”。它们从每个时间步长的输出中流出。这会稳定和加速训练。

为了清楚起见，模型在时间步长0时会输出一个向量，其中包含时间步长1到10的预测，然后在时间步长1时模型会预测时间步长2到11，以此类推。因此，每个目标必须是与输入序列长度相同的序列，在每个步长都必须包含10维向量。让我们准备这些目标序列：

```
Y = np.empty((10000, n_steps, 10)) # each target is a sequence of 10D vectors
for step_ahead in range(1, 10 + 1):
    Y[:, :, step_ahead - 1] = series[:, step_ahead:step_ahead + n_steps, 0]
Y_train = Y[:7000]
Y_valid = Y[7000:9000]
Y_test = Y[9000:]
```

 目标中包含了在输入中出现的值可能会令人惊讶（X_train和Y_train之间有很多重叠）。那不是作弊吗？幸运的是，一点也不是：在每个时间步长上，该模型仅知道过去的时间步长，因此不能向前看。这是一种因果模型。

要将模型转换为序列到序列的模型，我们必须在所有循环层（甚至最后一层）中设置return_sequen ces=True，必须在每个时间步长都应用输出Dense层。Keras为此提供了一个TimeDistributed层：它包装了任何层（例如Dense层），在输入序列的每个时间步长应用这个层。它通过重构输入的形状来有效地做到这一点，以便每个时间步长都被视为

一个单独的实例（将输入的形状从[批量大小，时间步长，输入维度]调整为[批量大小×时间步长，输入维度]。在此示例中，输入维数为20，因为前一个SimpleRNN层有20个单元），然后运行Dense层，最后将输出重构为序列（将输出从[批量大小×时间步长，输出维度]重构为[批量大小，时间步长，输出维度]。在此示例中，由于Dense层有10个单元，因此输出维度的数量为10）[1]。这是更新的模型：

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Dense层实际上支持序列作为输入（甚至是更高维的输入）：它像TimeDistributed（Dense（…））一样处理它们，这意味着它仅仅应用于最后一个输入维度（独立于所有的时间步长）。因此，我们可以用Dense（10）代替最后这一层。但是为了清楚起见，我们将继续使用TimeDistributed（Dense（10））， 因为它清楚地表明在每个时间步长都独立应用了Dense层，并且模型将输出一个序列，而不仅仅是一个向量。

训练期间需要所有的输出，但是只有最后一个时间步长的输出才对预测和评估有用。因此，尽管我们要依赖所有输出的MSE进行训练，但是我们使用自定义指标进行评估，以便在最后一个时间步长来计算输出的MSE：

```
def last_time_step_mse(Y_true, Y_pred):
    return keras.metrics.mean_squared_error(Y_true[:, -1], Y_pred[:, -1])

optimizer = keras.optimizers.Adam(lr=0.01)
model.compile(loss="mse", optimizer=optimizer, metrics=[last_time_step_mse])
```

我们得到的验证MSE约为0.006，比以前的模型好25%。你可以将此方法与第一个方法结合使用：使用这个RNN来预测接下来的10个值，然后将这些值合并到输入时间序列，然后再次使用该模型预测接下来的10个值，并根据需要重复该过程多次。使用这种方法，你可以生成任意长的序列。对于长期预测而言，它可能不太准确，但是如果你的目标是生成原始音乐或文本，则可能会很好，正如我们将在第16章中看到的那样。

 预测时间序列时，将一些误差与预测一起使用通常很有用。为此，第11章介绍了一种有效的技术MC Dropout：在每个记忆单元内添加一个MC Dropout层，删除部分输入和隐藏状态。训练后，为了预测新的时间序列，多次使用模型，并在每个时间步长计算预测的均值和标准差。

简单的RNN可以很好地预测时间序列或处理其他类型的序列，但是在长时间序列上却表现不佳。让我们讨论一下原因，看看有什么对策。

[1] 请注意，TimeDistributed（Dense（n））层等效于Conv1D（n，filter_size=1）层。

15.4 处理长序列

在长序列上训练一个RNN，我们必须运行很多个时间步长，从而使展开的RNN成为一个非常深的网络。就像任何深度神经网络一样，它可能会遇到不稳定的梯度问题，这在第11章中讨论过：它可能永远在训练，或者训练可能会不稳定。此外，当RNN处理一个长序列时，它会逐渐忘记序列中的第一个输入。让我们看看这两个问题，先从不稳定梯度问题开始吧。

15.4.1 应对不稳定梯度问题

我们在深度网络中用于应对不稳定梯度问题的许多技巧也可以用于RNN：良好的参数初始化、更快的优化器、dropout，等等。但是，非饱和激活函数（例如ReLU）在这里可能没有太大的帮助。实际上，它们可能导致RNN在训练过程中变得更加不稳定。为什么？好吧，假设梯度下降以一种在第一个时间步长稍微增加输出的方式来更新权重。由于每个时间步长都使用相同的权重，因此第二个时间步长的输出也可能会略有增加，第三个时间步长的输出也会稍有增加，以此类推，直到输出爆炸为止，而非饱和激活功函数不能阻止这种情况。你可以通过使用较小的学习率来降低这种风险，但也可以使用饱和激活函数（例如双曲正切）（这解释了为什么将其设为默认值）。同样的方式，梯度本身也会爆炸。如果你发现训练不稳定，可能要查看梯度的大小（例如使用TensorBoard），也可以使用梯度裁剪。

此外，批量归一化不能像深度前馈网络那样高效地用于RNN。实际上，你不能在时间步长之间使用它，而只能在递归层之间使用。更准确地说，从技术上讲，可以在记忆单元中添加一个BN层（我们很快会看到），以便将其应用于每个时间步长（在该时间步长的输入和前一个时间步长的隐藏状态上）。但是，在每个时间步长都使用相同的BN层，并使用相同的参数，而不管输入的实际比例和偏移以及隐藏状态如何。在实践中，这不能产生良好的结果，正如César Laurent等人在2015年的

一篇论文[1]中所证明的那样：BN仅在将BN应用于输入而非隐藏状态时才稍微受益。换句话说，当应用于循环层之间（即图15-7中的垂直方向）时，它总比没有好，但不是在递归层中（即水平方向）中。在Keras中，可以简单地通过在每个递归层之前添加一个Batch Normalization层来完成此操作，但是不要期望太多。

归一化的另一种形式通常与RNN一起使用会更好：层归一化。这个想法是Jimmy Lei Ba等人在2016年的一篇论文[2]中介绍的：它与批量归一化非常相似，但是它不是跨批量维度进行归一化，而是在特征维度上进行归一化。它的一个优点是可以在每个时间步长上针对每个实例独立地即时计算所需的统计信息。这也意味着它在训练和测试期间的行为方式相同（与BN相反），并且不需要使用指数移动平均值来估计训练集中所有实例的特征统计信息。像BN一样，层归一化学习每个输入的比例和偏移参数。在RNN中，通常在输入和隐藏状态的线性组合之后立即使用它。

让我们使用tf.keras在一个简单的记忆单元中实现层归一化。为此我们需要定义一个自定义记忆单元。就像常规层一样，不同之处在于其call（）方法采用两个参数：当前时间步长的inputs和上一个时间步长的隐藏states。请注意，states参数是包含一个或多个张量的列表。对于简单的RNN单元，它包含的单个张量等于上一个时间步长的输出，但是其他单元可能具有多个状态张量（例如，LSTMCell具有长期状态和短期状态，我们很快就会看到）。一个单元还必须具有state_size属性和output_size属性。在简单的RNN中，两者都等于单元数量。以下代码实现了一个自定义的记忆单元，该单元的行为类似于SimpleRNNCell，不同之处在于它还在每个时间步长应用“层归一化”：

```
class LNSimpleRNNCell(keras.layers.Layer):
    def __init__(self, units, activation="tanh", **kwargs):
        super().__init__(**kwargs)
        self.state_size = units
        self.output_size = units
        self.simple_rnn_cell = keras.layers.SimpleRNNCell(units,
                                                          activation=None)
        self.layer_norm = keras.layers.LayerNormalization()
```

```
        self.activation = keras.activations.get(activation)
    def call(self, inputs, states):
        outputs, new_states = self.simple_rnn_cell(inputs, states)
        norm_outputs = self.activation(self.layer_norm(outputs))
        return norm_outputs, [norm_outputs]
```

代码非常简单直接[3]。我们的LNSimpleRNNCell类继承自keras.layers.Layer类，就像任何自定义层一样。构造函数采用单元数量和所需的激活函数，并设置state_size和output_size属性，然后创建一个没有激活函数的SimpleRNNCell（因为我们想在线性运算之后但在激活函数之前执行“层归一化”）。然后，构造函数创建LayerNormalization层，最后获取所需的激活函数。Call（）方法通过应用于简单的RNN单元开始，该单元计算当前输入和先前的隐藏状态的线性组合，并返回两个结果（实际上，在SimpleRNNCell中，输出等于隐藏状态：换句话说，new_states[0]等于outputs，因此我们可以在其余call（）方法中安全地忽略new_states）。接下来，call（）方法应用“层归一化”，然后跟随一个激活函数。最后，它返回两个输出（一个作为输出，另一个作为新的隐藏状态）。要使用此自定义单元，我们需要做的就是创建一个keras.layers.RNN层，并向其传递一个单元实例：

```
model = keras.models.Sequential([
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True,
                     input_shape=[None, 1]),
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

同样，你可以创建一个自定义单元在每个时间步长之间应用dropout。但是有一种更简单的方法：所有循环层（除keras.layers.RNN之外）和Keras提供的所有单元都有一个超参数dropout和一个超参数recurrent_dropout：前者定义了应用于输入的dropout率（在每个时间步长），后者定义隐藏状态的dropout率（也在每个时间步长）。

在RNN中每个时间步长不需要创建自定义单元来应用dropout。

使用这些技术，可以缓解不稳定的梯度问题，更有效地训练RNN。现在让我们看一下如何处理短期记忆问题。

15.4.2 解决短期记忆问题

由于数据在遍历RNN时会经过转换，因此在每个时间步长都会丢失一些信息。一段时间后，RNN的状态几乎没有任何最初输入的痕迹。这是一个热门问题。想象一下多莉鱼[4]试图翻译一个长句子。当读完句子后，她不知道如何开始。为了解决这个问题，引入了具有长期记忆的各种类型的单元。它们被证明非常成功，以至于不再使用基本的单元。首先让我们看一下这些长期记忆单元中最流行的：LSTM单元。

LSTM单元

长短期记忆（LSTM）单元由Sepp Hochreiter和Jürgen Schmidhuber于1997年提出[5]，并在随后的几年中由Alex Graves、Hasçim Sak[6]和Wojciech Zaremba[7]等几位研究人员逐步改进。如果你把LSTM单元视为黑匣子，则它可以像基本单元一样使用，组它的性能会更好：训练会收敛得更快，它会检测数据中的长期依赖性。在Keras中，你可以简单地使用LSTM层而不是SimpleRNN层：

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

或者，你可以使用通用的keras.layers.RNN层，为其提供LSTMCell作为参数：

```
model = keras.models.Sequential([
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True,
                     input_shape=[None, 1]),
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

但是，LSTM层在GPU上运行时会使用优化过的实现（见第19章），因此通常最好使用它（如前所做的，当你定义自定义单元时RNN层最有用）。

那么LSTM单元如何工作呢？其架构如图15-9所示。

如果你不查看框内的内容，则LSTM单元看起来与常规单元完全一样，除了它的状态被分为两个向量：$h_{(t)}$和$c_{(t)}$（“c”代表“单元cell”。你可以将$h_{(t)}$视为短期状态，$c_{(t)}$为长期状态。

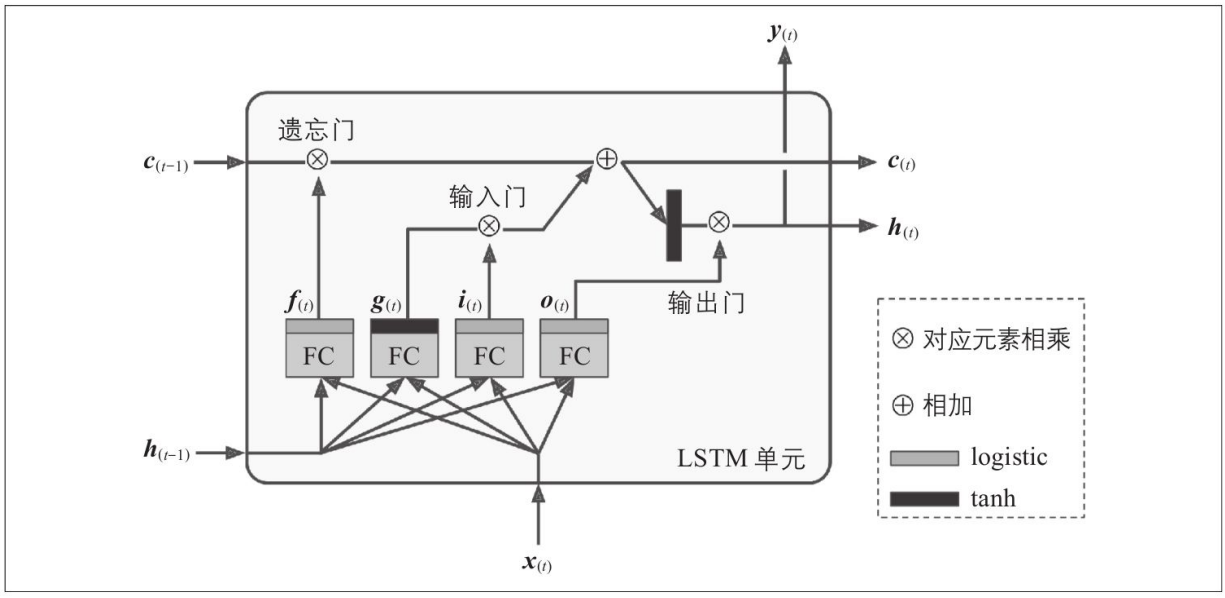


图15-9：LSTM单元

现在打开盒子！关键的思想是网络可以学习长期状态下存储的内容、丢弃的内容以及从中读取的内容。当长期状态$c_{(t-1)}$从左到右遍历网络时，你可以看到它首先经过一个遗忘门，丢掉了一些记忆，然后通过加法操作添加了一些新的记忆（由输入门选择的记忆）。结果

$c_{(t)}$直接送出来，无须任何进一步的转换。因此，在每个时间步长中，都会丢掉一些记忆，并添加一些记忆。此外，在加法运算之后，长期状态被复制并通过tanh函数传输，然后结果被输出门滤波。这将产生短期状态$h_{(t)}$（等于该时间步长的单元输出$y_{(t)}$）。现在，让我们看看新的记忆来自何处以及这些门如何运作。

首先，将当前输入向量$x_{(t)}$和先前的短期状态$h_{(t-1)}$馈入四个不同的全连接层。它们都有不同的目的：

• 主要层是输出$g_{(t)}$的层。它通常的作用是分析当前输入$x_{(t)}$和先前（短期）状态$h_{(t-1)}$。在基本单元中，除了这一层，没有其他东西，它的输出直接到$y_{(t)}$和$h_{(t)}$。相比之下，在LSTM单元中，该层的输出并非直接输出，而是将其最重要的部分存储在长期状态中（其余部分则丢弃）。

• 其他三层是门控制器。由于它们使用逻辑激活函数，因此它们的输出范围是0到1。如你所见，它们的输出被馈送到逐元素乘法运算，因此，如果它们输出0，则关闭门；如果它们输出1，则将门打开。特别地：

• 遗忘门（由$f_{(t)}$控制）控制长期状态的哪些部分应当被删除。

• 输入门（由$i_{(t)}$控制）控制应将$g_{(t)}$的哪些部分添加到长期状态。

• 最后，输出门（由$o_{(t)}$控制）控制应在此时间步长读取长期状态的哪些部分并输出到$h_{(t)}$和$y_{(t)}$。

简而言之，LSTM单元可以学会识别重要的输入（这是输入门的作用），将其存储在长期状态中，只要需要就保留它（即遗忘门的作用），并在需要时将其提取出来。这就解释了为什么这些单元在识别时间序列、长文本、录音等的长期模式方面取得了惊人的成功。公式15-3

总结了如何计算单个实例在每个时间步长的单元的长期状态、短期状态及其输出（与整个小批量的方程非常相似）。

公式15-3：LSTM计算等式

$$
\begin{aligned}
\boldsymbol{i}_{(t)} &= \sigma(\boldsymbol{W}_{xi}^{\mathrm{T}}\boldsymbol{x}_{(t)} + \boldsymbol{W}_{hi}^{\mathrm{T}}\boldsymbol{h}_{(t-1)} + \boldsymbol{b}_i) \\
\boldsymbol{f}_{(t)} &= \sigma(\boldsymbol{W}_{xf}^{\mathrm{T}}\boldsymbol{x}_{(t)} + \boldsymbol{W}_{hf}^{\mathrm{T}}\boldsymbol{h}_{(t-1)} + \boldsymbol{b}_f) \\
\boldsymbol{o}_{(t)} &= \sigma(\boldsymbol{W}_{xo}^{\mathrm{T}}\boldsymbol{x}_{(t)} + \boldsymbol{W}_{ho}^{\mathrm{T}}\boldsymbol{h}_{(t-1)} + \boldsymbol{b}_o) \\
\boldsymbol{g}_{(t)} &= \tanh(\boldsymbol{W}_{xg}^{\mathrm{T}}\boldsymbol{x}_{(t)} + \boldsymbol{W}_{hg}^{\mathrm{T}}\boldsymbol{h}_{(t-1)} + \boldsymbol{b}_g) \\
\boldsymbol{c}_{(t)} &= \boldsymbol{f}_{(t)} \otimes \boldsymbol{c}_{(t-1)} + \boldsymbol{i}_{(t)} \otimes \boldsymbol{g}_{(t)} \\
\boldsymbol{y}_{(t)} &= \boldsymbol{h}_{(t)} = \boldsymbol{o}_{(t)} \otimes \tanh(\boldsymbol{c}_{(t)})
\end{aligned}
$$

在此等式中：

• W_{xi}、W_{xf}、W_{xo}、W_{xg}是四层中的每层与输入向量$x_{(t)}$连接的权重矩阵。

• W_{hi}、W_{hf}、W_{ho}和W_{hg}是四层中的每层与先前的短期状态$h_{(t-1)}$连接的权重矩阵。

• b_i、b_f、b_o和b_g是四层中每层的偏置项。请注意，TensorFlow将b_f初始化为一个全是的1不是0的向量。这样可以防止在训练开始时忘记一切。

窥视孔连接

在常规LSTM单元中，门控制器只能查看输入$x_{(t)}$和先前的短期状态$h_{(t-1)}$。通过让它们也查看长期状态来给它们更多的功能，这可能是一个好主意。Felix Gers和Jérgen Schmidhuber于2000年提出了这个想法[8]。它们提出了一种带有额外连接的LSTM变体，称为窥视孔连接：先前的长期状态$c_{(t-1)}$作为输入添加到遗忘门和输入门的控制器，当前的长期状态$c_{(t)}$作为输入添加到输出门的控制器。这通常会提高性能，但并非总是如此，并且没有明确的模式说明哪个任务更好：你将不得不在你的任务上尝试一下，看看是否有帮助。

在Keras中，LSTM层基于不支持窥视孔的keras.layers.LSTMCell单元。实验性的tf.keras.experimental.PeepholeLSTMCell可以，因此你可以创建keras.layers.RNN层，并将PeepholeLSTM Cell传递给其构造函数。

LSTM单元还有许多其他变体。一种特别流行的变体是GRU单元，我们现在来看一下。

GRU单元

门控循环单元（Gated Recurrent Unit，GRU）（见图15-10）是由Kyunghyun Cho等人在2014年的论文中提出的[9]，该论文还介绍了我们之前讨论的Encoder-Decoder网络。

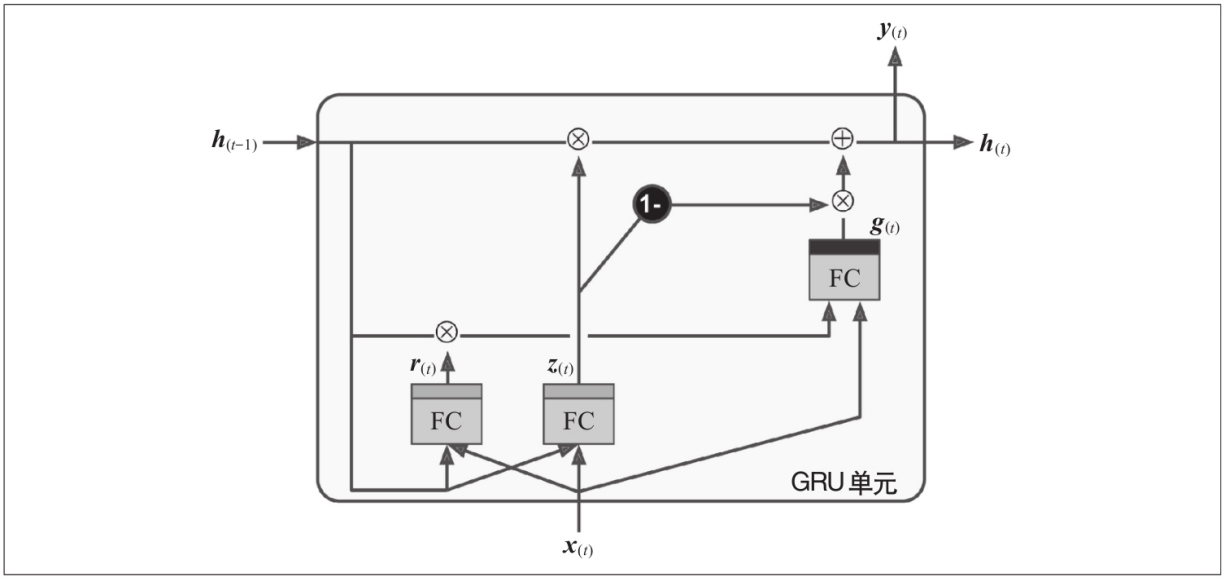


图15-10：GRU单元

GRU单元是LSTM单元的简化版，它的性能似乎也不错[10]（这解释了其日益普及的原因）。以下这些是主要的简化：

• 两个状态向量合并为一个向量$h_{(t)}$。

• 单个门控制器$z_{(t)}$控制遗忘门和输入门。如果门控制器输出1，则遗忘门打开（=1），输入门关闭（1-1=0）。如果输出0，则相反。换句话说，无论何时记忆必须被存储，其存储位置需要首先被删除。实际上，这本身就是LSTM单元的常见变体。

• 没有输出门，在每个时间步长都输出完整的状态向量。但是，有一个新的门控制器$r_{(t)}$控制先前状态的哪一部分将显示给主要层（$g_{(t)}$）。

公式15-4总结了如何为单个实例计算每个时间步长的单元状态。

公式15-4：GRU计算公式

$$\boldsymbol{z}_{(t)}=\sigma(\boldsymbol{W}_{xz}^{\mathrm{T}}\boldsymbol{x}_{(t)}+\boldsymbol{W}_{hz}^{\mathrm{T}}\boldsymbol{h}_{(t-1)}+\boldsymbol{b}_z)$$

$$\boldsymbol{r}_{(t)}=\sigma(\boldsymbol{W}_{xr}^{\mathrm{T}}\boldsymbol{x}_{(t)}+\boldsymbol{W}_{hr}^{\mathrm{T}}\boldsymbol{h}_{(t-1)}+\boldsymbol{b}_r)$$

$$\boldsymbol{g}_{(t)}=\tanh(\boldsymbol{W}_{xg}^{\mathrm{T}}\boldsymbol{x}_{(t)}+\boldsymbol{W}_{hg}^{\mathrm{T}}(\boldsymbol{r}_{(t)}\otimes\boldsymbol{h}_{(t-1)})+\boldsymbol{b}_g)$$

$$\boldsymbol{h}_{(t)}=\boldsymbol{z}_{(t)}\otimes\boldsymbol{h}_{(t-1)}+(1-\boldsymbol{z}_{(t)})\otimes\boldsymbol{g}_{(t)}$$

Keras提供了一个keras.layers.GRU层（基于keras.layers.GRUCell记忆单元），使用它只是用GRU来替换SimpleRNN或LSTM的问题。

LSTM和GRU单元是RNN成功的主要原因之一。尽管它们可以处理比简单RNN更长的序列，但它们的短期记忆仍然非常有限，而且很难学习100个或更多时间步长序列中的长期模式，例如音频样本、长时间序列或长句子。解决这个问题的一种方法是缩短输入序列，例如使用一维卷积层。

使用一维卷积层处理序列

在第14章中，我们看到了2D卷积层的工作原理是在图像上滑动几个相当小的内核（或过滤器），生成多个2D特征图（每个内核一个）。类似地，一维卷积层在一个序列上滑动多个内核，为每个内核生成一维特征图。每个内核将学习检测单个非常短的顺序模式（不长于内核大小）。如果你使用10个内核，则该层的输出将由10个一维的序列（所有长度相同）组成，或者等效地，你可以将此输出视为单个10维的序列。这意味着你可以构建由循环层和一维卷积层（甚至一维池化层）混合而成的神经网络。如果你使用步幅为1且填充为"same"的一维卷积层，则输出序列的长度与输入序列的长度相同。但是，如果你使用填充为"valid"或步幅大于1，则输出序列比输入序列短，因此请确保相应地调整目标。例如，以下模型与前面的模型相同，不同之处在于它从一维

卷积层开始，将输入序列进行2倍下采样，使用步幅为2。内核大小大于步幅，因此所有的输入将用来计算层的输出，因此该模型可以学习保留有用的信息，而只删除不重要的细节。通过缩短序列，卷积层可以帮助GRU层检测更长的模式。请注意，我们还必须裁剪掉目标中的前三个时间步长（因为内核的大小为4，所以卷积层的第一个输出将基于0至3的输入时间步长），并对目标进行2倍的下采样：

```
model = keras.models.Sequential([
    keras.layers.Conv1D(filters=20, kernel_size=4, strides=2, padding="valid",
                        input_shape=[None, 1]),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])

model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train[:, 3::2], epochs=20,
                    validation_data=(X_valid, Y_valid[:, 3::2]))
```

如果你训练和评估该模型，会发现它是迄今为止最好的模型。卷积层确实有帮助，实际上，完全可以只使用一维卷积层，删除整个循环层！

WaveNet

在2016年的一篇论文中[11]，Aaron van den Oord和其他DeepMind研究人员介绍了一种称为WaveNet的架构。它们堆叠了一维卷积层，使每一层的扩散（dilation）率（每个神经元输入的分散程度）加倍：第一个卷积层一次只看到两个时间步长，而下一个卷积层一次看到四个时间步长（其接受野为四个时间步长），下一个看到八个时间步长，以此类推（见图15-11）。这样，较低的层学习短期模式，而较高的层学习长期模式。由于扩散率的加倍，网络可以非常有效地处理非常大的序列。

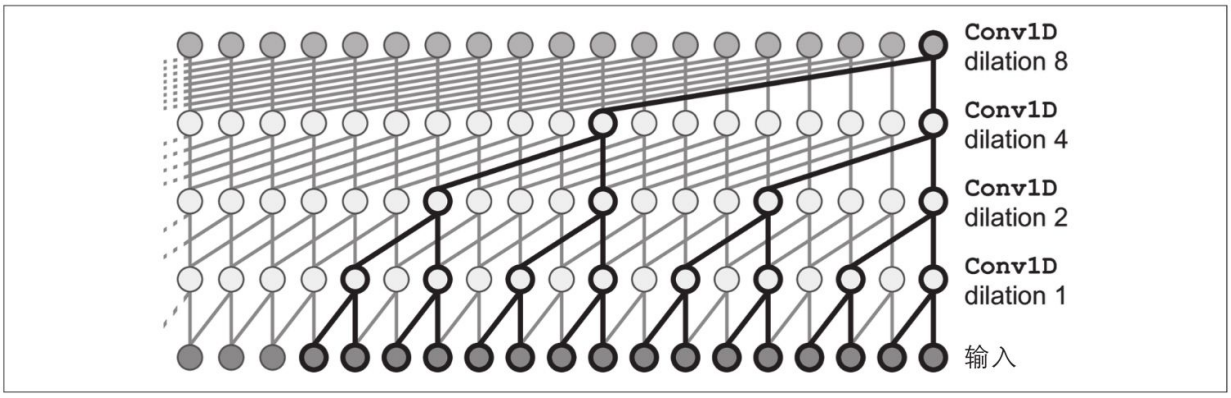


图15-11：WaveNet架构

在WaveNet论文中，作者实际上堆叠了10个卷积层，其扩散率为1、2、4、8，…，256，512，然后又堆叠了另一组10个相同的层（扩散率也分别为1、2、4、8，…，256，512），然后又是另一组相同的10层。它们通过指出具有这些扩散率的10个卷积层的单个堆栈来证明该架构的合理性，就像内核大小为1024的超高效卷积层一样工作（除了更快、更强大且使用更少的参数之外），这就是为什么它们堆叠了3个这样的块。它们还对输入序列进行了左填充，这些零与每个层之前的扩散率相等，以保持整个网络中相同的序列长度。以下实现了简化的WaveNet来处理和之前相同的序列[12]：

```
model = keras.models.Sequential()
model.add(keras.layers.InputLayer(input_shape=[None, 1]))
for rate in (1, 2, 4, 8) * 2:
    model.add(keras.layers.Conv1D(filters=20, kernel_size=2, padding="causal",
                                  activation="relu", dilation_rate=rate))
model.add(keras.layers.Conv1D(filters=10, kernel_size=1))
model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

该Sequential模型从显式输入层开始（这比试图仅在第一层上设置input_shape更为简单），然后继续使用填充为"causal"的一维卷积层：这可确保卷积层在进行预测时不会窥视未来（它等效于在输入的左侧填充正确数量的零且使用"valid"填充）。然后我们添加相似的成对

层，使用不断扩大的扩散率：1、2、4、8，还是1、2、4、8。最后，我们添加输出层：一个具有10个大小为1的滤波器且没有任何激活函数的卷积层。多亏了填充层，每个卷积层都输出与输入序列长度相同的序列，因此我们在训练过程中使用的目标是完整序列，无须裁剪或对它们进行下采样。

到目前为止，最后两个模型在预测我们的时间序列方面提供了最佳性能！在WaveNet论文中，作者在各种音频任务（架构的名字）上获得了最先进的性能，包括从文本到语音的任务，在多种语言中产生了令人难以置信的逼真的声音。他们还使用该模型来生成音乐，一次生成一个音频样本。当你意识到一秒钟的音频可以包含数万个时间步长时，这一成就会更加令人印象深刻，甚至LSTM和GRU都无法处理如此长的序列。

在第16章中，我们将继续探索RNN，我们会看到它们如何解决各种NLP任务。

[1] César Laurent et al.，“Batch Normalized Recurrent Neural Networks”，Proceedings of the IEEE International Conference on Acoustics，Speech，and Signal Processing（2016）：2657–2661.

[2] Jimmy Lei Ba et al.，“Layer Normalization”，arXiv preprint arXiv：1607.06450（2016）.

[3] 从SimpleRNNCell继承会更简单，这样我们就不必创建内部SimpleRNNCell或处理state_size和output_size属性，但是这里的目的是演示如何从头开始创建自定义单元。

[4] 动画电影《海底总动员》和《海底总动员2》中的角色，只有短期记忆。

[5] Sepp Hochreiter and Jürgen Schmidhuber，“Long Short-Term Memory”，Neural Computation 9，no.8（1997）：1735–1780.

[6] Hasçim Sak et al.，“Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition”，arXiv preprint arXiv：1402.1128（2014）.

[7] Wojciech Zaremba et al.，“Recurrent Neural Network Regularization”，arXiv preprint arXiv：1409.2329（2014）.
[8] F.A.Gers and J.Schmidhuber，“Recurrent Nets That Time and Count”，Proceedings of the IEEE-INNSENNS International Joint Conference on Neural Networks（2000）：189－194.
[9] Kyunghyun Cho et al.，“Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation”，Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing（2014）：1724－1734.
[10] Klaus Greff等人在2015年发表的论文“LSTM：A Search Space Odyssey”似乎表明所有LSTM变体的性能大致相同。
[11] Aaron van den Oord et al.，“WaveNet：A Generative Model for Raw Audio”，arXiv preprint arXiv：1609.03499（2016）.
[12] 完整的WaveNet使用了更多技巧，例如像ResNet中的跳过连接，以及类似于GRU单元中的门控激活单元。请参阅notebook以了解更多的详细信息。

15.5 练习题

1.你能想到序列对序列RNN的一些应用吗？序列到向量RNN和向量到序列RNN呢？2.RNN层的输入必须具有多少个维度？每个维度代表什么？它的输出如何？

3.如果要构建深度序列对序列RNN，则哪个RNN层应具有return_sequences=True？序列到向量RNN呢？

4.假设你有一个日常不变的时间序列，并且希望预测接下来的7天。你应该使用哪种RNN架构？

5.训练RNN的主要困难是什么？你如何处理它们？

6.你能勾勒出LSTM单元的架构吗？

7.为什么要在RNN中使用一维卷积层？

8.你可以使用哪种神经网络架构对视频进行分类？

9.为TensorFlow数据集中的SketchRNN数据集训练一个分类模型。

10.下载Bach合唱数据集并解压缩。它由Johann Sebastian Bach作曲的382个合唱组成。每个合唱的长度为100至640个时间步长，每个时间步长包含4个整数，其中每个整数对应于钢琴上一个音符的索引（值0除外，0意味着不演奏任何音符）。训练一个模型（递归、卷积或两者兼而有之），该模型可以预测下一个时间步长（四个音符），给定一个从合唱开始的时间步长序列。然后用这个一次产生一个音符的模型产生类似巴赫的音乐：你可以通过向模型提供合唱的开始并要求其预测下一个时间步长，然后将这些时间步长添加到输入序列中并

询问模型下一个音符，以此类推。另外，请务必检查一下Google的Coconet模型，该模型用来制作有关Bach的漂亮Google涂鸦。

这些练习题的解答在附录A中提供。

第16章　使用RNN和注意力机制进行自然语言处理

当艾伦·图灵（Alan Turing）在1950年想象他著名的图灵测试[1]时，他的目标是评估一台机器的能力来匹配人类的智能。他可以测试很多东西，例如识别图片中的猫、下棋、作曲或逃脱迷宫等的能力，但有趣的是，他选择了一个语言任务。更具体地讲，他设计了一个聊天机器人，能够欺骗其对话者以为它是人类[2]。此测试确实有其弱点：一组硬编码规则可以欺骗毫无戒心或天真的人（例如，机器可以给出模糊的预定义回答来回应某些关键字，也可以假装是在开玩笑或喝醉了来通过这个测试，或者通过应答自己的问题来逃避棘手的问题），人类智力的许多方面都被完全忽略了（例如，解释非语言交流（例如面部表情）的能力，或学习手工任务的能力）。但是测试确实突出了这样一个事实，即掌握语言可以说是智人最大的认知能力。我们可以建造一台可以读写自然语言的机器吗？

自然语言任务的常见方法是使用循环神经网络。我们继续探索RNN（在第15章中介绍），从字符RNN开始，经过训练可以预测句子中的下一个字符。这将使我们能够生成一些原始文本，在此过程中，我们将了解如何在很长的序列上构建TensorFlow数据集。我们首先使用无状态RNN（在每次迭代时，处理文本的随机部分，而在文本的其余部分上没有任何信息），然后我们将创建一个有状态RNN（在RNN训练迭代之间保留隐藏状态，并继续从中断处读取内容，从而允许其学习更长的模式）。接下来，我们将构建一个RNN来执行情感分析（例如，阅读电影评论并提取评分者对电影的感觉），这次将句子视为单词序列，而不是字符。然后，我们将展示如何使用RNN来构建能够执行神经机器翻译（NMT）的编码器-解码器架构。为此，我们使用TensorFlow Addons项目提供的seq2seq API。

在本章的第二部分，我们将研究注意力机制。顾名思义，这些是神经网络组件，可以学习选择输入的一部分，模型的其余部分在每个

时间步长应重点关注输入部分。首先，我们将了解如何通过注意力机制来提高基于RNN的编码器-解码器架构的性能，然后我们将完全放弃RNN，研究一种非常成功的仅有注意力机制的架构，称为Transformer。最后，我们介绍2018年和2019年NLP的一些最重要的进展，包括基于Transformers的功能强大的语言模型，例如GPT-2和BERT。

让我们从一个简单而有趣的模型入手，该模型可以像莎士比亚那样写作。

[1] Alan Turing，“Computing Machinery and Intelligence”，Mind 49（1950）：433－460。
[2] 当然，“聊天机器人”这个词来得很晚。图灵称他的测试为模仿游戏：机器A和人类B通过文字消息与审讯员C聊天。审讯员问一些问题来找出哪个是机器，哪个是人类（A或B）。如果机器可以欺骗审讯员，则机器通过测试，而人类B必须尝试帮助审讯员。

16.1 使用字符RNN生成莎士比亚文本

在2015年著名的博客文章“The Unreasonable Effectiveness of Recurrent Neural Networks”中，Andrej Karpathy展示了如何训练RNN来预测句子中的下一个字符。然后可以将此Char-RNN用于生成新颖的文本，一次生成一个字符。这是Char-RNN模型经过莎士比亚全部作品训练后生成的文本的一小部分样本：

PANDARUS：

Alas, I think he shall be come approached and the day

When little srain would be attain’d into being never fed,

And who is but a chain and subjects of his death,

I should not sleep.

这不是一个好作品，但令人印象深刻的是，该模型仅通过学习预测句子中的下一个字符就能够学习单词、语法、正确的标点符号，等等。让我们从创建数据集开始，逐步研究如何构建Char-RNN。

16.1.1 创建训练数据集

首先，让我们使用Keras方便的get_file（）函数来下载莎士比亚的所有作品，并从Andrej Karpathy的Char-RNN项目中下载数据：

```
shakespeare_url = "https://homl.info/shakespeare" # shortcut URL
lepath = keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

接下来，我们必须将每个字符编码为整数。一种选择是创建自定义的预处理层，就像我们在第13章中所做的那样。但是在这种情况下，使用Keras的Tokenizer类会更简单。首先，我们需要为文本添加一个分词器：它会找到文本中使用的所有字符，并将它们映射到不同的字符ID，从1到不同字符的数量（它不从0开始，所以我们可以使用该值进行屏蔽，如我们在本章稍后看到的那样）：

```
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)
tokenizer.fit_on_texts([shakespeare_text])
```

我们设置char_level=True来得到字符级编码，而不是默认的单词级编码。请注意，默认情况下，该分词器将文本转换为小写（但是，如果你不希望这样做，可以将其设置为lower=False）。现在，分词器可以将一个句子（或句子列表）编码为字符ID列表并返回，并告诉我们有多少个不同的字符以及文本中的字符总数：

```
>>> tokenizer.texts_to_sequences(["First"])
[[20, 6, 9, 8, 3]]
>>> tokenizer.sequences_to_texts([[20, 6, 9, 8, 3]])
['f i r s t']
>>> max_id = len(tokenizer.word_index) # number of distinct characters
>>> dataset_size = tokenizer.document_count # total number of characters
```

让我们对全文进行编码，以便每个字符都由其ID表示（我们减去1即可得到从0到38的ID，而不是从1到39的ID）：

```
[encoded] = np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1
```

在继续之前，我们需要将数据集分为训练集、验证集和测试集。我们不能只是将文本中的所有字符都进行混洗，那么如何拆分顺序数据集呢？

16.1.2　如何拆分顺序数据集

避免训练集、验证集和测试集之间的任何重合都是非常重要的。例如，我们可以将文本的前90%用作训练集，其后的5%作为验证集，最后的5%作为测试集。在两个集合之间留一个空隙也是一个好主意，可以避免出现两个集合中的段落重合。

在处理时间序列时，通常会跨时间进行划分：例如，你可能使用2000年到2012年之间的作为训练集，在2013年到2015年之间的作为验证集，在2016年到2018年之间的作为测试集。但是，在某些情况下，你可以沿其他维度来拆分，这将使你有更长的训练时间。例如，如果你有2000年至2018年间10 000家公司的财务状况数据，你可以按照不同的公司将这些数据拆分。但是，其中许多公司很有可能会高度相关（例如，整个经济部门可能会一起上升或下降），如果你在训练集和测试集中有相关联的公司，那你的测试集不会很有用，因为其对泛化误差的度量偏向于乐观。

因此，跨时间划分通常更安全。但这隐含地假设RNN过去（在训练集中）学习到的模式将来仍会存在。换句话说，我们假设时间序列是稳定的（至少在广义上是这样）[1]。对于许多时间序列，此假设是合理的（例如，化学反应就很好，因为化学定律不会每天改变），但对于其他的并非如此（例如，众所周知，金融市场并非一成不变，因为一旦交易者发现并开始利用它们，模式就会消失）。为了确保时间序列足够稳定，你可以按时间在验证集上绘制模型的误差：如果模型在验证集的第一部分比最后一部分表现更好，则时间序列可能不够稳定，因此最好在较短的时间范围内训练模型。

简而言之，将时间序列分为训练集、验证集和测试集不是一件容易的事，而如何划分将在很大程度上取决于你手头的任务。

现在回到莎士比亚！让我们使用文本的前90%作为训练集（其余部分保留为验证集和测试集），并创建一个tf.data.Dataset，它将从该集合中逐个返回每个字符：

```
train_size = dataset_size * 90 // 100
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
```

16.1.3 将顺序数据集切成多个窗口

训练集现在由超过一百万个字符的单个序列组成，所以我们不能直接在其上训练神经网络：RNN相当于一个超过一百万层的深层网络，我们会有一个（很长的）实例来训练它。取而代之的是，我们会使用数据集的window（）方法将这个长字符序列转换为许多较小的文本窗口。数据集中的每个实例将是整个文本的很短的子字符串，并且RNN仅在这些子字符串的长度上展开。这称为时间截断反向传播。让我们调用window（）方法来创建短文本窗口的数据集：

```
n_steps = 100
window_length = n_steps + 1 # target = input shifted 1 character ahead
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
```

 你可以尝试调整n_steps：在较短的输入序列上训练RNN会更容易，但是当然RNN不能学习比n_steps长的任何模式，因此不要使其太短。

默认情况下，window（）方法会创建不重叠的窗口，但是为了获得最大可能的训练集，我们使用shift=1，以便使第一个窗口包含0到100的字符，第二个窗口包含1到101的字符，以此类推。为确保所有窗口正好是101个字符长度（这使我们不需要进行任何填充就可创建批处理），我们设置drop_remainder=True（否则最后100个窗口将包含100个字符，99个字符，以此类推直至1个字符）。

Window（）方法创建一个包含窗口的数据集，每个窗口也表示为一个数据集。它是一个嵌套的数据集，类似于列表的列表。当你想通过调用窗口的数据集方法来转换每个窗口时（例如，对它们进行混洗或批处理），此功能非常有用。但是，我们不能直接使用嵌套数据集进行训练，因为我们的模型希望输入张量，而不是数据集。因此，我们必须调用flat_map（）方法：它将嵌套的数据集转换为一个展平的数据集（一个不包含数据集的数据集）。例如，假设{1，2，3}表示一个包含张量1、2和3的序列的数据集。如果展平嵌套的数据集{{1，2}，{3，4，5，6}}，你将获得展平的数据集{1，2，3，4，5，6}。此外，flat_map（）方法将一个函数用作参数，它允许你在展平前变换嵌套数据集中的每个数据集。例如，如果你把函数lambda ds：ds.batch（2）传递给flat_map（），则它将嵌套数据集{{1，2}，{3，4，5，6}}转换为平数据集{[1，2]，[3，4]，[5，6]}：它是大小为2的张量的数据集。考虑到这一点，我们准备将数据集展平：

```
dataset = dataset.flat_map(lambda window: window.batch(window_length))
```

注意，我们在每个窗口上调用batch（window_length）：由于所有窗口的长度都恰好相同，因此每个窗口都获得一个张量。现在数据集包含101个字符的连续窗口。由于当训练集中的实例独立且分布相同时，梯度下降效果最好（见第4章），因此我们需要对这些窗口进行混洗。然后，我们可以批处理这些窗口并将输入（前100个字符）与目标（最后一个字符）分开：

```
batch_size = 32
dataset = dataset.shuffle(10000).batch(batch_size)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
```

图16-1总结了到目前为止讨论的数据集准备步骤（显示窗口长度为11而不是101，批量大小为3而不是32）。

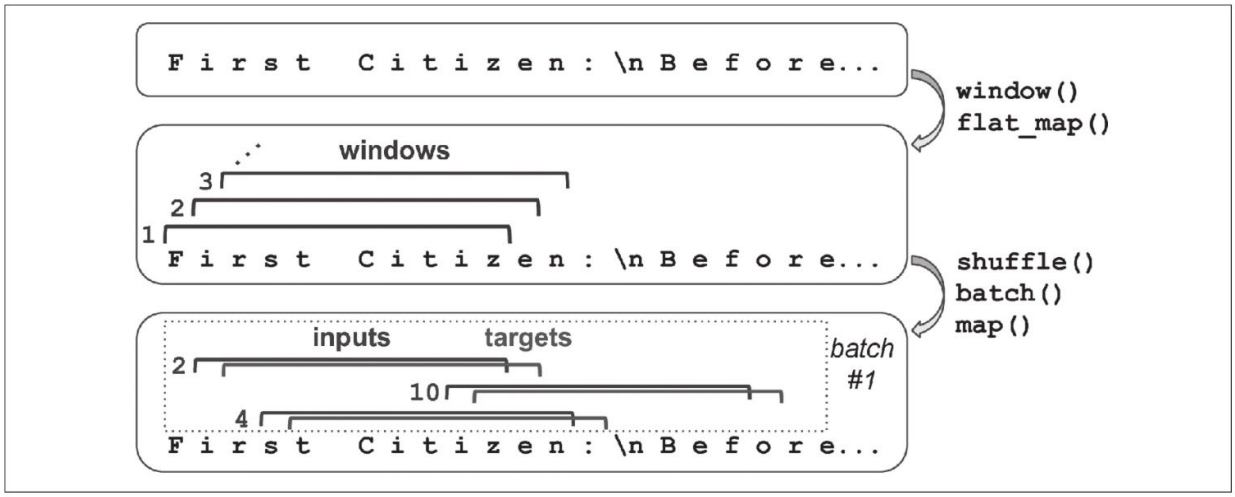


图16-1：准备随机窗口的数据集

如第13章所述，一般应将类别输入特征编码为独热向量或嵌入。在这里，我们将使用一个独热向量对每个字符进行编码，因为只有很少个不同的字符（只有39个）：

```
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
```

最后，我们只需要添加预取：

```
dataset = dataset.prefetch(1)
```

准备数据集是最困难的部分。现在让我们创建模型。

16.1.4 创建和训练Char-RNN模型

要基于前100个字符来预测下一个字符，我们可以使用有2个GRU层的RNN，每个GRU层有128个单元，输入（drop out）和隐藏状态（recurrent_dropout）的dropout率均为20%。如果需要，我们可以稍后调整这些超参数。输出层是一个时间分布的Dense层，就像我们在第15章中看到的那样。这一次该层必须有39个单元（max_id），因为文本中有39个不同的字符，并且我们想为每个可能的字符输出一个概率（在每个时间步长）。每个时间步长的输出概率总和为1，因此我们将softmax激活函数应用于Dense层的输出。然后，我们使用"sparse_categorical_crossen tropy"损失和Adam优化器来编译此模型。最后，我们已经准备好训练几个轮次的模型（这可能需要很多个小时，具体取决于你的硬件）：

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id],
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.GRU(128, return_sequences=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                     activation="softmax"))
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, epochs=20)
```

16.1.5 使用Char-RNN模型

现在我们有了一个可以预测莎士比亚写的文本中下一个字符的模型。为了提供一些文本，我们首先需要像之前一样对它进行预处理，因此我们创建了一个小函数：

```
def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)
```

现在，让我们使用该模型来预测某些文本中的下一个字母：

```
>>> X_new = preprocess(["How are yo"])
>>> Y_pred = model.predict_classes(X_new)
>>> tokenizer.sequences_to_texts(Y_pred + 1)[0][-1] # 1st sentence, last char
'u'
```

成功！模型猜对了。现在让我们使用此模型来生成新文本。

16.1.6 生成假莎士比亚文本

要使用Char-RNN模型来生成新文本，我们可以向其提供一些文本，使模型预测最可能的下一个字母，把它添加在文本末尾，然后将扩展的文本提供给模型来猜测下一个字母，以此类推。但是实际上，这经常导致相同的单词一遍又一遍地重复。相反，我们可以使用TensorFlow的tf.random.categorical（）函数估计出来的概率随机选择下一个字符。这将产生更多不同和有趣的文本。给定类对数概率（logits），categorical（）函数会对随机类索引进行采样。为了更好地控制生成的文本的多样性，我们可以把logits除以一个称为温度的数字，这样可以根据需要进行调整：接近0的温度倾向于高概率的字符，而非常高的温度会给予所有的字符相同的概率。下面的next_char（）函数使用这种方法来选择要添加到输入文本的下一个字符：

```
def next_char(text, temperature=1):
    X_new = preprocess([text])
    y_proba = model.predict(X_new)[0, -1:, :]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1)+ 1
    return tokenizer.sequences_to_texts(char_id.numpy())[0]
```

接下来，我们可以编写一个小函数，该函数会反复调用next_char（）来获得下一个字符并将其添加到给定的文本中：

```
def complete_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text
```

现在我们准备生成一些文本！尝试一下不同的温度：

```
>>> print(complete_text("t", temperature=0.2))
the belly the great and who shall be the belly the
>>> print(complete_text("w", temperature=1))
thing? or why you gremio.
who make which the first
>>> print(complete_text("w", temperature=2))
th no cce:
yeolg-hormer firi. a play asks.
fol rusb
```

显然，我们的莎士比亚模型在接近1的温度下效果最佳。要生成更具说服力的文本，你可以尝试使用更多的GRU层和每层有更多的神经元，训练更长的时间，并添加一些正则化（例如，你可以在GRU层设置recurrent_dropout=0.3）。而且，当前该模型无法学习比n_steps长的模式，仅仅只能学习100个字符。你可以尝试增大此窗口，但这会增加训练难度，甚至LSTM和GRU单元也无法处理很长的序列。或者，你可以使用有状态RNN。

16.1.7 有状态RNN

到目前为止，我们仅仅使用了无状态RNN：在每次训练迭代中，模型都从一个充满零的隐藏状态开始，然后在每个时间步长更新该状态，在最后一个时间步长之后将其丢弃，因为不再需要它了。如果我们告诉RNN在处理一个训练批次后保留此最终状态并将其用作下一个训练批次的初始状态，应该怎么办？这样，尽管反向传播只是通过短序列，模型仍可以学习长期模式。这称为有状态RNN。让我们看看如何创建它。

首先，请注意，只有当批次中的每个输入序列均从上一个批次中对应序列中断的确切位置开始时，有状态RNN才有意义。因此，创建有状态RNN所需要做的第一件事是使用顺序和非重合的输入序列（而不是我们用来训练无状态RNN的混洗和重叠的序列）。因此，在创建Dataset时，我们在调用window（）方法时必须使用shift=n_steps（而不是shift=1）。而且，我们显然不能调用shuffle（）方法。不幸的是，在为有状态RNN准备数据集时，批处理要比无状态RNN困难得多。确实，如果我们要调用batch（32），那32个连续的窗口应该放入同一批处理中，而下一个批处理不会在这些窗口中的每个中断处继续。第一批包含窗口1至32，第二批包含窗口33至64，因此，如果你考虑每个批次的第一个窗口（即窗口1和33），可以看到它们是不连续的。解决此问题的最简单方法是只使用包含单个窗口的“批处理”：

```
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
dataset = dataset.window(window_length, shift=n_steps, drop_remain-der=True)
dataset = dataset.flat_map(lambda window: window.batch(window_length))
dataset = dataset.batch(1)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows [:, 1:]))
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
dataset = dataset.prefetch(1)
```

图16-2总结了第一步。

批处理比较困难，但并非不可能。例如，我们可以将莎士比亚的文本切成等长的32个文本，为每个文本创建一个连续输入序列的数据集，最后使用tf.train.Dataset.zip（datasets）.map（lambda*windows：tf.stack（windows））来创建合适的连续批处理，其中批处理中的第n个输入序列恰好是从上一个批处理中第n个输入序列结束的位置开始（有关完整代码，请参阅notebook）。

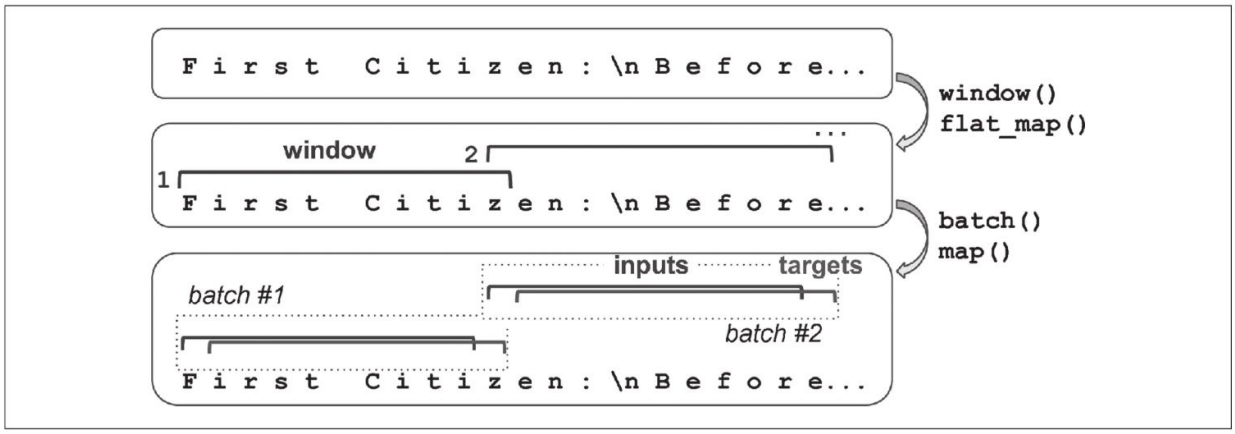


图16-2：为有状态RNN准备的连续序列片段的数据集

现在让我们创建有状态RNN。首先，在创建每个循环层时需要设置stateful=True。其次，有状态RNN需要知道批处理大小（因为它会为批处理中的每个输入序列保留一个状态），因此我们必须在第一层中设置batch_input_shape参数。注意，我们可以不指定第二个维度，因为输入可以有任意长度：

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2,
                     batch_input_shape=[batch_size, None, max_id]),
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                     activation="softmax"))
])
```

在每个轮次结束时，我们需要先重置状态，然后再返回到文本的开头。为此，我们可以使用一个小的回调函数：

```
class ResetStatesCallback(keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

现在，我们可以编译并拟合模型（更多的轮次，因为每个轮次都比先前的短得多，并且每批次只有一个实例）：

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
model.fit(dataset, epochs=50, callbacks=[ResetStatesCallback()])
```

 训练此模型后，将只能使用它来预测与训练期间使用的相同大小的批次。为了避免这种限制，要创建相同的无状态模型，然后将有状态模型的权重复制到该模型中。

现在我们已经创建了字符级模型，是时候来看一下单词级模型并解决一个常见的自然语言处理任务：情感分析。在此过程中，我们将学习如何使用掩码来处理可变长度的序列。

[1] 根据定义，稳定的时间序列的均值、方差和自相关（在给定间隔的时间序列中，值之间的相关性）不会随时间变化。这是非常严格的。例如，它不包括具有趋势或周期性模式的时间序列。RNN可以学习趋势和周期性模式，更能容忍这些模式。

16.2 情感分析

如果MNIST是计算机视觉的“hello world”，那么IMDb评论数据集就是自然语言处理的“hello world”：它提取了来自著名的互联网电影数据库的50 000条英语电影评论（其中25 000条用于训练，25 000条用于测试），每条评论的简单二元目标值表明了该评论是负面（0）还是正面（1）。就像MNIST一样，IMDb评论数据集也很受欢迎，这有充分的理由：它足够简单，可以在合理的时间内用笔记本电脑来处理，但又具有挑战性，可以带来乐趣和收获。Keras提供了一个简单的函数来加载它：

```
>>> (X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data()
>>> X_train[0][:10]
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

电影评论在哪里？好了，正如你所看到的，数据集已经为你进行了预处理：X_train由一个评论列表组成，每条评论都由一个NumPy整数数组表示，其中每个整数代表一个单词。删除了所有标点符号，然后将单词转换为小写字母，用空格分隔，最后按频率索引（因此，小的整数对应于常用单词）。整数0、1和2是特殊的：它们分别表示填充令牌、序列开始（SSS）令牌和未知单词。如果你想可视化评论，可以按以下方式对其进行解码：

```
>>> word_index = keras.datasets.imdb.get_word_index()
>>> id_to_word = {id_ + 3: word for word, id_ in word_index.items()}
>>> for id_, token in enumerate(("<pad>", "<sos>", "<unk>")):
...     id_to_word[id_] = token
...
>>> " ".join([id_to_word[id_] for id_ in X_train[0][:10]])
'<sos> this film was just brilliant casting location scenery story'
```

在实际的项目中，你将不得不自己预处理文本。你可以使用与之前使用的相同的Tokenizer类来执行此操作，但是这次要设置char_level=False（这是默认值）。在对单词进行编码时，它会过滤掉很多字符，包括大多数标点符号、换行符和制表符（但你可以通过设置filter参数来更改）。最重要的是，它使用空格来标识单词边界。对于英语和许多其他在单词之间使用空格的脚本（书面语言）这是可以的，但并非所有脚本都用此方式。中文在单词之间不使用空格，越南语甚至在单词中也使用空格，例如德语之类的语言经常将多个单词附加在一起，而没有空格。即使在英语中，空格也不总是分词的最佳方法：想想“San Francisco”或“#ILoveDeepLearning”。

幸运的是，还有更好的选择！Taku Kudo在2018年发表的论文[1]引入了一种无监督学习技术，用一种独立于语言的方式在子单词级别对文本进行分词和组词，像对待其他字符一样对待空格。使用这种方法，即使你的模型遇到一个从未见过的单词，也仍然可以合理地猜出其含义。例如，它可能在训练期间从未见过“smartest”一词，但可能学会了“smart”一词，并且还学习到后缀“est”的意思是“the most”，因此它可以推断出“smartest”的意思。Google的SentencePiece项目提供了一个开源实现，Taku Kudo和John Richardson在论文中对此进行了描述[2]。

Rico Sennrich等人在较早的论文中提出了另一种选择[3]，探索了创建子单词编码的其他方法（例如使用字节对编码）。TensorFlow团队于2019年6月发布了TF.Text库，该库实现了各种分词策略，包括WordPiece[4]（字节对编码的一种变体）。

如果你想将模型部署到移动设备或Web浏览器上，而又不想每次都编写不同的预处理函数，那么可以使用TensorFlow操作来做预处理，因此其包含在模型本身中。让我们看看怎么做。首先使用TensorFlow数据集（在第13章中介绍）以文本（字节字符串）的形式加载原始的IMDb评论：

```
import tensorflow_datasets as tfds

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
```

接下来写预处理函数：

```
def preprocess(X_batch, y_batch):
    X_batch = tf.strings.substr(X_batch, 0, 300)
    X_batch = tf.strings.regex_replace(X_batch, b"<br\\s*/?>", b" ")
    X_batch = tf.strings.regex_replace(X_batch, b"[^a-zA-Z']", b" ")
    X_batch = tf.strings.split(X_batch)
    return X_batch.to_tensor(default_value=b"<pad>"), y_batch
```

它从截断评论开始，每条评论仅保留前300个字符：这会加快训练速度，并且不会对性能产生太大的影响，因为你通常可以在第一句话或第二句中判断出评论是正面还是负面。它使用正则表达式来用空格替换
标记，还用空格替换字母和引号以外的所有字符。例如，文本"Well，I can't
"将变成"Well I can't"。最后，preprocess（）函数将评论按空格分割，并返回一个不规则的张量，并将该不规则张量转换为密集张量，还使用填充标记"<pad>"来填充所有评论，以使它们都具有相同的长度。

接下来，我们需要构建词汇表。这需要一次遍历整个训练集，应用preprocess（）函数，并使用Counter来对每个单词出现的次数进行计数：

```
from collections import Counter
vocabulary = Counter()
for X_batch, y_batch in datasets["train"].batch(32).map(preprocess):
    for review in X_batch:
        vocabulary.update(list(review.numpy()))
```

让我们看一下三个最常见的词：

```
>>> vocabulary.most_common()[:3]
[(b'<pad>', 215797), (b'the', 61137), (b'a', 38564)]
```

很好！但是，为了获得良好的性能，我们可能不需要模型知道字典中的所有单词，因此让我们截断词汇表，只保留10 000个最常见的单词：

```
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]
```

现在，我们需要添加一个预处理步骤，以把每个单词替换为其ID（即其在词汇表中的索引）。就像在第13章中所做的那样，我们使用1 000 out-of-vocabulary（oov）存储桶来创建一个查找表：

```
words = tf.constant(truncated_vocabulary)
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
num_oov_buckets = 1000
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)
```

然后，我们可以使用此表来查找几个单词的ID：

```
>>> table.lookup(tf.constant([b"This movie was faaaaaantastic".split()]))
<tf.Tensor: [...], dtype=int64, numpy=array([[ 22, 12, 11, 10054]])>
```

请注意，在表中可以找到单词“this”“movie”和“was”，因此它们的ID低于10 000，而单词“faaaaaantastic”没有找到，因此被映射到ID大于或等于10 000的一个oov桶中。

 TF Transform（在第13章中介绍）提供了一些有用的函数来处理此类词汇表。例如，查看tft.compute_and_apply_vocabulary（）函数：它会遍历数据集来查找所有不同的单词并构建词汇表，并将生成对使用此词汇表的单词进行编码所需的TF操作。

现在，我们准备创建最终的训练集。我们对评论进行批处理，然后使用preprocess（）函数将它们转换为单词的短序列，然后使用简单的encode_words（）函数对这些单词进行编码，该函数会使用我们刚才构建的单词表，最后预取下一批次：

```
def encode_words(X_batch, y_batch):
    return table.lookup(X_batch), y_batch

train_set = datasets["train"].batch(32).map(preprocess)
train_set = train_set.map(encode_words).prefetch(1)
```

最后，我们可以创建模型并对其进行训练：

```
embed_size = 128
model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                           input_shape=[None]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
history = model.fit(train_set, epochs=5)
```

第一层是嵌入层，它将单词ID转换为嵌入（在第13章中介绍）。嵌入矩阵需要每个单词ID（vocab_size+num_oov_buckets）一行，每个嵌入维度一列（此示例使用128维，但这是你可以调整的超参数）。模型的输入是形状为[批处理大小，时间步长]的2D张量，而嵌入层的输出是形状为[批处理大小，时间步长，嵌入大小]的3D张量。

该模型的其余部分相当简单直接：它由两个GRU层组成，第二个GRU层仅返回最后一个时间步长的输出。输出层只是使用sigmoid激活函数来输出估计概率的单个神经元，该概率反映了评论表达了与电影有关的正面情绪。然后，我们可以简单地编译模型，并将其拟合到之前准备的数据集中，进行几个轮次的训练。

16.2.1　掩码屏蔽

就目前而言，该模型需要学习应该忽略掉填充令牌。我们已经知道了！那为什么不告诉模型去忽略掉填充令牌呢，以便它可以专注于实际很重要的数据？实际上这很容易：在创建嵌入（Embedding）层时只需添加mask_zero=True。这意味着所有下游层都将忽略填充令牌（其ID为0）[5]。就是这样！

这种工作方式是Embedding层创建一个等于K.not_equal（inputs，0）（其中K=keras.backend）的掩码张量：它是一个布尔张量，其形状与输入相同，它在单词ID为0的任何地方为False，否则为True。只要时间维度被保留，该掩码张量就会由模型自动传播到所有后续层。因此，在此示例中，两个GRU层都会自动接收此掩码，但是由于第二个GRU层不返回序列（它仅返回最后一个时间步长的输出），因此该掩码不会传输到Dense层。每一层可能会以不同的方式来处理掩码，但通常它们只忽略被掩码的时间步长（即掩码为False的时间步长）。例如，当循环层遇到被掩蔽的时间步长时，它仅复制前一时间步长的输出。如果掩码一直以这种方式传播到输出（在输出

序列的模型中，本例中不是这种情况），那么它也会被应用于损失，因此，被掩码的时间步长不会对损失造成影响（它们的损失是0）。

 LSTM和GRU层有一个基于Nvidia的cuDNN库的优化了GPU的实现。但是，此实现不支持掩码。如果你的模型使用了掩码，那么这些层会退回到（慢得多的）默认实现。请注意，优化的实现还需要你使用多个超参数的默认值：activation、recurrent_activation、recurrent_dropout、unroll、use_bias和reset_after。

能接收掩码的所有层都必须支持使用掩码（否则将引发异常）。这包括所有循环层，以及TimeDistributed层和其他一些层。任何支持掩码的层都必须有一个等于True的supports_masking属性。如果你想实现自己的支持掩码的自定义图层，则应在call（）方法中添加mask参数（显然这样会使该方法以某种方式使用mask）。另外，你应该在构造函数中设置self.supports_masking=True。如果你的层不是以Embedding层开始，则可以改用keras.layers.Masking层：它将掩码设置为K.any（K.not_equal（inputs，0），axis=-1），这意味着在最后一个维度都为零的时间步长中，后续层都会被屏蔽（同样，只要存在时间维度）。

使用掩码层和自动掩码传播最适合简单的Sequential模型。它不适用于更复杂的模型，例如当你需要混合Conv1D层与循环层时。在这种情况下，你需要使用函数式API或子类API来显式地计算掩码并将其传递到适当的层。例如，以下模型与先前的模型相同，不同之处在于它是使用函数式API构建并手动处理掩码的：

```
K = keras.backend
inputs = keras.layers.Input(shape=[None])
mask = keras.layers.Lambda(lambda inputs: K.not_equal(inputs, 0))(inputs)
z = keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size)(inputs)
z = keras.layers.GRU(128, return_sequences=True)(z, mask=mask)
z = keras.layers.GRU(128)(z, mask=mask)
outputs = keras.layers.Dense(1, activation="sigmoid")(z)
model = keras.Model(inputs=[inputs], outputs=[outputs])
```

经过几个轮次的训练之后，该模型将非常擅长判断评论是正面还是负面的。如果你使用TensorBoard（）回调函数，你可以在模型学习时用TensorBoard把嵌入可视化：有趣的是，像“真棒”和“令人惊奇”这样的单词会逐渐聚集在嵌入空间的一侧，而“可怕的”和“恐怖的”这样的单词则会在另一侧聚集。有些词并不像你预期的那样是正面的（至少在此模型中如此），例如“好”一词，大概是因为许多负面评论中都包含了“不好”一词。该模型能够仅仅基于25 000条电影评论来学习有用的词嵌入。想象一下，如果我们有数十亿的评论可以用来训练，那么嵌入的效果将是多么出色！不幸的是我们没有，但是也许我们可以重用在其他大型文本语料库（例如Wikipedia文章）上训练的词嵌入，即使它不是由电影评论组成的。毕竟，“惊奇”一词通常具有相同的含义，无论你是用来谈论电影还是其他的。此外，即使嵌入是在其他任务上被训练的，对情感分析还是有用的：因为“真棒”和“令人惊奇”之类的词具有相似的含义，即使对于其他任务，它们也可能会在嵌入空间中聚成一类（例如，预测句子中的下一个单词）。如果所有的正面词和所有的负面词都形成聚类，那么这将有助于情感分析。因此，让我们看看是否可以仅仅重用预训练的嵌入，而不是使用太多的参数来学习词嵌入。

16.2.2　重用预训练的嵌入

TensorFlow Hub项目使你可以轻松地在你自己的模型中重用经过预训练的模型组件。这些模型组件称为模块。只需浏览TF Hub存储库，就能找到你所需要的，然后将代码示例复制到你的项目中，该模块将连同其预先训练的权重一起自动下载并包含在你的模型中。如此简单！

例如，让我们在情感分析模型中使用nnlm-en-dim50句子嵌入模块（版本1）：

```
import tensorflow_hub as hub

model = keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
                   dtype=tf.string, input_shape=[], output_shape=[50]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
```

---

hub.Keras Layer层会从给定的URL下载模块。这个特殊的模块是一个句子编码器：它把字符串作为输入，并将每个字符串编码为单个向量（在这个情况下为50维向量）。在内部，它解析字符串（用空格来分隔单词），并使用在大型语料库（Google News 7B语料库，长70亿个单词！）上预训练的嵌入矩阵来嵌入每个单词。然后，它计算所有词嵌入的均值，其结果就是句子嵌入[6]。然后，我们可以添加两个简单的Dense层来创建一个良好的情感分析模型。默认情况下，hub.Keras Layer是不可训练的，但是你可以在创建它时将它设置为trainable=True来更改它，以便你可以针对你的任务来进行微调。

 并非所有TF Hub模块都支持TensorFlow 2，因此请确保选择一个支持的模块。

接下来，我们只需加载IMDb评论数据集即可——无须对其进行预处理（除了批处理和预取）并直接训练模型：

```
datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
batch_size = 32
train_set = datasets["train"].batch(batch_size).prefetch(1)
history = model.fit(train_set, epochs=5)
```

请注意，TF Hub模块URL的最后一部分指定了我们想要模型的版本1。此版本控制可确保如果发布了新的模块版本，不会破坏我们的模

型。方便的是，如果你在网络浏览器中输入此URL，你会得到此模块的文档。默认情况下，TF Hub会将下载的文件缓存到本地系统的临时目录中。你可能希望把它们下载到一个永久目录中，以避免在每次系统清理后都必须再次下载它们。为此，请将TFHUB_CACHE_DIR环境变量设置为你选择的目录（例如os.environ["TFHUB_CACHE_DIR"]="./my_tfhub_cache"）。

到目前为止，我们已经研究了时间序列，使用Char-RNN来生成文本以及使用单词级RNN模型进行情感分析，训练了我们自己的词嵌入或重用了预训练的嵌入。现在让我们看一下NLP的另一个重要任务：神经机器翻译（NMT），首先使用一个纯编码器-解码器模型，然后使用注意力机制对其进行改进，最后再看一下优秀的Transformer架构。

[1] Taku Kudo，“Subword Regularization：Improving Neural Network Translation Models with Multiple Sub-word Candidates”，arXiv preprint arXiv：1804.10959（2018）.
[2] Taku Kudo and John Richardson，“SentencePiece：A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing”，arXiv preprint arXiv：1808.06226（2018）.
[3] Rico Sennrich et al.，“Neural Machine Translation of Rare Words with Subword Units”，Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics 1（2016）：1715－1725.
[4] Yonghui Wu et al.，“Google’s Neural Machine Translation System：Bridging the Gap Between Human and Machine Translation，”arXiv preprint arXiv：1609.08144（2016）.
[5] 它们的ID为0只是因为它们是数据集中最频繁的“单词”。确保填充令牌始终被编码为0（即使它们不是最频繁的）可能是一个好主意。
[6] 确切地说，句子嵌入等于词嵌入均值乘以句子中单词数的平方根。这弥补了以下事实：随着n的增加，n个向量的平均值变小。

16.3 神经机器翻译的编码器-解码器网络

让我们看一个简单的神经机器翻译模型[1]，它将英语句子翻译成法语（见图16-3）。10简而言之，英语句子被送入到编码器，解码器输出法语翻译。请注意，法语翻译也用作解码器的输入，但向后偏移了一步。换句话说，给解码器的输入是在上一个步长应该输出的单词（无论它实际输出了什么）。对于第一个单词，它被赋予了序列开始（SOS）令牌。解码器希望以序列结尾（EOS）令牌来结束句子。

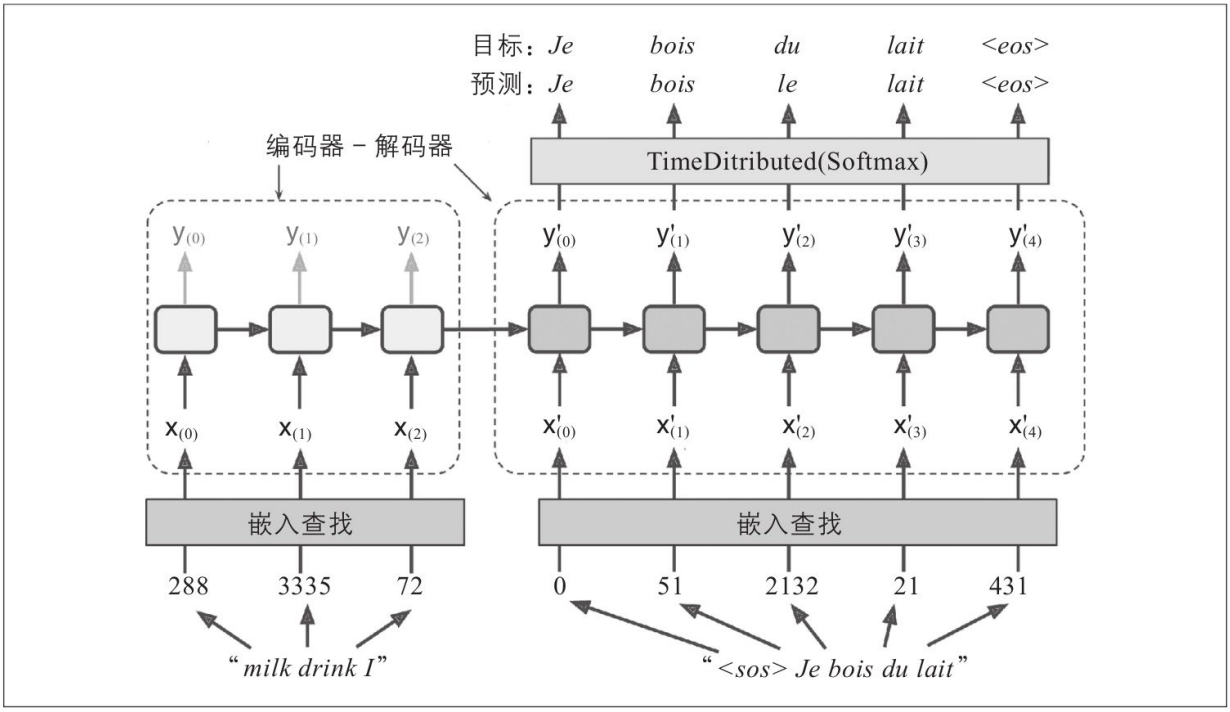


图16-3：一个简单的机器翻译模型

请注意，在将英语句子送入到编码器之前，它们已经被颠倒了。例如，“I drink milk”被反转为“milk drink I”。这确保了英语句子的开头最后被馈送到编码器，这很有用，因为通常这是解码器需要翻译的第一个内容。

每个单词最初都由其ID表示（例如，单词“milk”为288）。接下来，embedding层返回词嵌入。这些词嵌入实际上是馈送到编码器和解码器的。

在每个步长中，解码器为在词汇表（即法语）中的每个单词输出一个分数，然后softmax层将这些分数转换为概率。例如，在第一步中，单词“Je”的概率为20%，单词“Tu”的概率为1%，以此类推。概率最高的单词被输出。这非常类似于常规的分类任务，因此你可以使用“sparse_categorical_crossentropy”损失来训练模型，就像我们在Char-RNN模型中所做的一样。

请注意，在推理期间（训练后），你没有目标句子要馈送到解码器中。取而代之的是只需向解码器送入在上一步中输出的单词，如图16-4所示（这需要在图中未显示的嵌入查找表）。好的，现在你有了全局图。但是，如果要实现此模型，还有更多细节需要处理：

• 到目前为止，我们已经假设所有输入序列（到编码器和到解码器的）的长度都是恒定的。但是显然句子的长度会有所不同。由于规则张量具有固定的形状，因此它们只能容纳相同长度的句子。你可以使用掩码来处理此问题，如前所述。但是，如果句子的长度截然不同，则你不能像进行情感分析那样仅仅只是裁剪它们（因为我们需要完整的翻译，而不是裁剪的翻译）。而是将句子分为相似长度的一组（例如，一个1到6个单词的句子为一组，另一个7到12个单词的句子为一组，以此类推），对较短的序列使用填充来确保一组中的所有句子都具有相同的长度（为此请查看tf.data.experimental.bucket_by_sequence_length（）函数）。例如，“I drink milk”变成“<pad><pad><pad>milk drink I”。

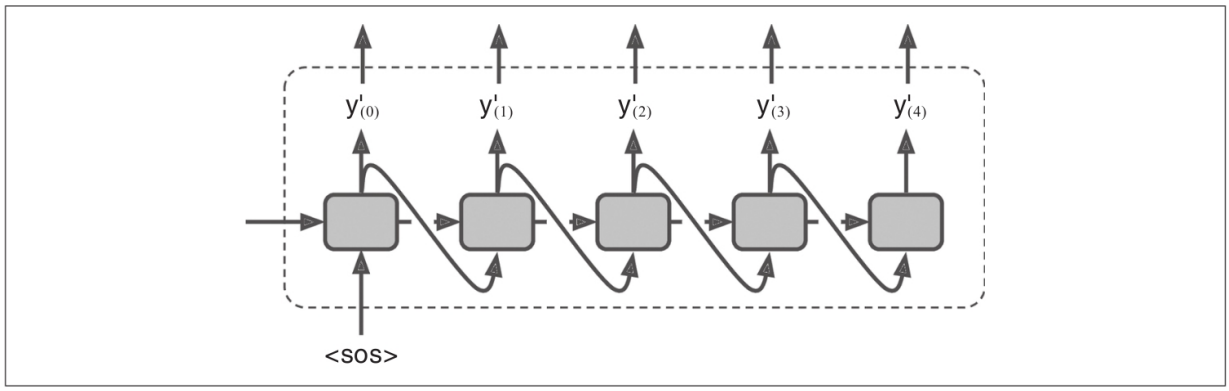


图16-4：在推理期间将以前的输出单词作为输入

• 我们希望忽略EOS令牌之后的任何输出，因此这些令牌不应对损失有贡献（必须将其屏蔽）。例如，如果模型的输出为“Je bois du lait<eos>oui”，那么最后一个单词的损失应忽略不计。

• 当输出词汇量很大时（这里就是这种情况），为每个可能的单词输出概率将非常慢。如果目标词汇表包含例如50 000个法语单词时，则解码器将输出50 000维的向量，在如此大的向量上计算softmax函数需要大量的计算。为了避免这种情况，一种解决方案是仅查看模型为正确单词和不正确单词随机样本的输出的对数，然后根据这些对数来计算损失的一个近似值。这种采样softmax技术是Sébastien Jean等人在2015年发表的[2]。在TensorFlow中，你可以在训练过程中使用tf.nn.sampled_softmax_loss（）函数，在推断时使用通常的softmax函数（采样softmax不能在推理时使用，因为它需要知道目标值）。

TensorFlow Addons项目包含许多序列到序列的工具，它可以使你轻松构建可用于生产环境的编码器-解码器。例如，以下代码创建一个基本的解码器-编码器模型，类似于图16-3中所示的模型：

```
import tensorflow_addons as tfa

encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
sequence_lengths = keras.layers.Input(shape=[], dtype=np.int32)

embeddings = keras.layers.Embedding(vocab_size, embed_size)
```

```
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)

encoder = keras.layers.LSTM(512, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.TrainingSampler()

decoder_cell = keras.layers.LSTMCell(512)
output_layer = keras.layers.Dense(vocab_size)
decoder = tfa.seq2seq.basic_decoder.BasicDecoder(decoder_cell, sampler,
                                                 output_layer=output_layer)
final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings, initial_state=encoder_state,
    sequence_length=sequence_lengths)
Y_proba = tf.nn.softmax(final_outputs.rnn_output)

model = keras.Model(inputs=[encoder_inputs, decoder_inputs, sequence_lengths],
                    outputs=[Y_proba])
```

该代码是不言自明的，但有几点需要注意。首先，我们在创建LSTM层时设置return_state=True，以便获得其最终的隐藏状态并将它传递给解码器。由于我们使用的是LSTM单元，因此实际上返回了两个隐藏状态（短期和长期）。TrainingSampler是TensorFlow Addons中提供的几种采样器之一：它们的作用是在每个步长中告诉解码器它应该假装成先前输出的内容。在推理时，这应该是实际输出令牌的嵌入。在训练期间，它应该是先前目标令牌的嵌入：这就是我们使用TrainingSampler的原因。实际上，从前一个时间步长的目标嵌入开始训练，然后逐渐转换到使用在前一步长输出的实际令牌的嵌入，这通常是一个好办法。Samy Bengio等人在2015年的论文[3]中介绍了这个想法。ScheduledEmbeddingTrainingSampler将在目标值或实际输出之间使用一个概率值做随机选择，你可以在训练过程中逐渐改变这个概率值。

16.3.1 双向RNN

在每个时间步长中，常规循环层在生成其输出之前仅仅查看过去和当前的输入。换句话说，这是“因果关系”，意味着它无法看到未来。预测时间序列时，这种类型的RNN很有意义，但是对于许多NLP任务，例如神经机器翻译，通常最好在给单词编码之前先查看下一个单词。例如，考虑短语“the Queen of the United Kingdom”“the queen of

hearts”和“the queen bee”：为了正确编码单词“queen”，你需要向前看一下。要实现此目的，在同一输入上运行两个循环层，一层从左至右读取单词，另一层从右至左读取单词。然后，只需在每个时间步长上简单地组合它们的输出，通常将它们合并即可。这称为双向循环层（见图16-5）。

为了在Keras中实现双向循环层，要在keras.layers.Bidirectional层中包装一个循环层。例如，以下代码创建一个双向GRU层：

```
keras.layers.Bidirectional(keras.layers.GRU(10, return_sequences=True))
```

 Bidirectional层会创建GRU层的副本（但方向相反），它将运行2个并合并它们的输出。因此，尽管GRU层有10个单元，但是Bidirectional层会在每个时间步长输出20个值。

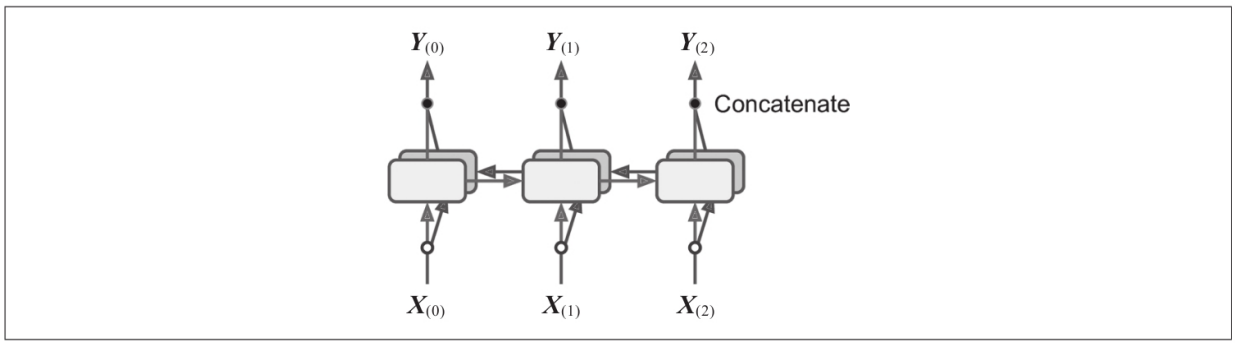


图16-5：一个双向循环层

16.3.2 集束搜索

假设你训练了一个解码器-编码器模型，并使用该模型将法语句子“Comment vas-tu”翻译成英语。你希望它会输出正确的翻译（“How

are you？”），但不幸的是它输出了“How will you？”在训练集上，你注意到许多句子，例如“Comment vas-tu jouer？”的意思是“How will you play？”因此看到“Comment vas”后，模型输出“How will”并不荒谬。不幸的是，在这种情况下是一个错误，该模型无法返回并对其进行修正，因此它尝试尽最大可能完成该句子。通过尽量在每个步长中输出最可能出现的单词，最终得到次优的翻译。我们如何才能使模型有机会返回并更正先前的错误呢？最常见的解决方法之一是集束搜索：它跟踪k个最有希望的句子（例如前三个）的一个短列表，在每个解码器步长中尝试用一个单词它们扩展，仅仅保留k个最有可能的句子。参数k称为集束宽度。

例如，假设你使用该模型通过集束宽度为3的集束搜索来翻译句子“Comment vas-tu”，在第一个解码器步长，模型为每个可能的单词输出一个估计的概率。假设前三名的词分别是“How”（估计概率为75%）、“What”（3%）和“You”（1%）。到目前为止，这是我们的短列表。接下来，我们创建模型的三个副本，并使用它们来查找每个句子的下一个单词。每个模型为词汇表中的每个单词输出一个估计的概率。第一个模型将尝试在句子“How”中找到下一个单词，也许它将以36%的概率输出单词“will”，32%的概率为单词“are”，16%的概率为单词“do”，等等。注意，鉴于句子以“How”开头，这些实际上是条件概率。第二个模型将尝试完成句子“What”。它可能会为单词“are”输出50%的条件概率。假设词汇表有10 000个单词，则每个模型将输出10 000个概率。

接下来，我们计算这些模型会考虑（3×10 000）的30 000个两个单词的句子中每个句子的概率。我们通过将每个单词的估计的条件概率乘以它完成的句子的估计概率来做到这一点。例如，句子“How”的估计概率为75%，而单词“will”的估计条件概率（假设第一个单词为“How”）为36%，因此句子“How will”的估计概率为75%×36%=7%。在计算完所有30 000个两个单词的句子的概率之后，我们仅仅保留前3名。也许它们都以单词“How”开头：“How will”（27%）、“How

are”（24%）和“How do”（12%）。目前，“How will”是赢家，但“How are”也没有被丢掉。

然后，我们重复相同的过程：使用三个模型来预测这三个句子中每个句子的下一个单词，计算所有30 000个三个单词的句子的概率。也许现在排名前三的是“How are you”（10%）、“How do you”（8%）和“How will you”（2%）。在下一步长中，我们可能会得到“How do you do”（7%）、“How are you<eos>”（6%）和“How are you doing”（3%）。注意，“How will”被丢掉了，我们现在有了三个完全合理的翻译。仅仅通过更聪明地使用它，就可以提高编码器-解码器模型的性能，而无须任何额外的训练。

你可以使用TensorFlow Addons轻松地实现集束搜索：

```
beam_width = 10
decoder = tfa.seq2seq.beam_search_decoder.BeamSearchDecoder(
    cell=decoder_cell, beam_width=beam_width, output_layer=output_layer)
decoder_initial_state = tfa.seq2seq.beam_search_decoder.tile_batch(
    encoder_state, multiplier=beam_width)
outputs, _, _ = decoder(
    embedding_decoder, start_tokens=start_tokens, end_token=end_token,
    initial_state=decoder_initial_state)
```

我们首先创建一个BeamSearchDecoder，它将所有解码器的副本（在本例中为10个副本）包装起来。然后，我们为每个解码器副本创建一个编码器的最终状态的副本，并将这些状态以及开始令牌和结束令牌传递给解码器。

有了这些，你就可以获得很短的句子的良好翻译（尤其是如果你使用预训练的词嵌入）。不幸的是，这个模型在翻译长句子时非常不好。问题来自RNN的有限的短期记忆。注意力机制是解决此问题的巨大创新。

[1] Ilya Sutskever et al.，“Sequence to Sequence Learning with Neural Networks”，arXiv preprint arXiv：1409.3215（2014）.
[2] Sébastien Jean et al.，“On Using Very Large Target Vocabulary for Neural Machine Translation，”Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing 1（2015）：1－10.
[3] Samy Bengio et al.，“Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks”，arXiv preprint arXiv：1506.03099（2015）.

16.4 注意力机制

考虑一下图16-3中从单词“milk”到其翻译“lait”的路径：它非常长！这意味着这个单词（以及所有其他单词）的表征在实际使用之前需要进行许多步骤。我们不能缩短这条路吗？这是Dzmitry Bahdanau等人在2014年的突破性论文中的核心思想[1]。他们介绍了一种技术，该技术允许解码器在每个时间步长中专注于适当的单词（由编码器编码）。例如，在解码器需要输出单词“lait”的时间步长上，它会把注意力集中在单词“milk”上。这意味着从输入单词到其翻译的路径变短了，因此RNN的短期记忆限制的影响变小了。注意力机制彻底改变了神经机器翻译（一般来说是NLP），极大地改善了现有技术，特别是对于长句子（超过30个单词）[2]。

图16-6显示了该模型的架构（略有简化）。在左侧有编码器和解码器。现在，不是把编码器的最终隐藏状态发送给解码器（尽管未在图中显示，但仍然做了），而是把其所有输出发送给解码器。在每个时间步长，解码器的记忆单元都会计算所有这些编码器输出的加权总和：这确定了该步长将会把重点关注在哪个单词。权重$\alpha_{(t,\,i)}$是在第t个解码器时间步长处的第i个编码器输出的权重。例如，如果权重$\alpha_{(3,\,2)}$远大于权重$\alpha_{(3,\,0)}$和权重$\alpha_{(3,\,1)}$，则解码器将更加注意第2个单词（“milk”）而不是其他两个单词，至少在这个时间步长。解码器的其余部分的工作方式与之前类似：在每个时间步长，记忆单元都会接收我们刚刚讨论的输入，再加上前一个时间步长的隐藏状态，最后（尽管图中未表示）它接收前一个时间步长中的目标单词（在推断时，为前一个时间步长的输出）。

但是这些$\alpha_{(t,\,i)}$权重从何而来？实际上这很简单：它们是由一种称为对齐模型（或注意力层）的小型神经网络生成的，该网络与其余的编码器-解码器模型一起进行训练。图16-6的右侧表明了此对齐模型。它始于有单个神经元的时间分布Dense层[3]，该层接收所有编码器的输

出作为输入，与解码器先前的隐藏状态（例如$h_{(2)}$）合并。该层为每个编码器的输出（例如$e_{(3,2)}$）而输出一个分数（或熵）：该分数用于衡量每个输出与解码器先前的隐藏状态对齐的程度。最后，所有分数都经过softmax层，以获取每个编码器输出的最终权重（例如$\alpha_{(3,2)}$）。给定解码器时间步长的所有权重加起来为1（因为softmax层未按时间分布）。这种特殊的注意力机制称为Bahdanau注意力（以该论文的第一作者命名）。由于它将编码器输出与解码器的先前隐藏状态合并在一起，因此有时称为合并注意力（或加法注意力）。

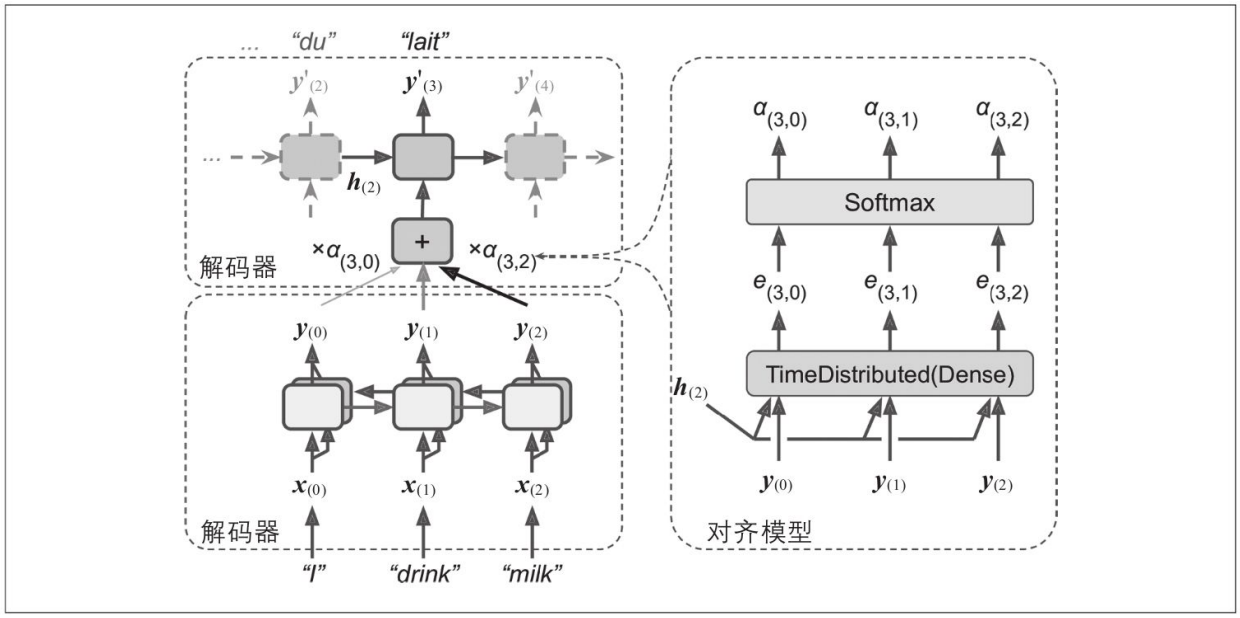


图16-6：使用有注意力模型的编码器-解码器网络的神经机器翻译

 如果输入句子的长度为n个单词，并且假设输出句子的长度大致相同，则此模型需要计算大约n^2个权重。幸运的是，这种二次计算复杂度仍然是很容易处理的，因为即使长句子也没有成千上万个单词。

之后不久，Minh-Thang Luong等人在2015年发表的论文[4]中提出了另一种常见的注意力机制。由于注意力机制的目的是测量编码器的输出之一与解码器的先前隐藏状态之间的相似性，因此作者提出了简单地计算这两个向量的点积（见第4章）。因为这通常是一个相当好的相似

性度量，当前的硬件可以更快地进行计算。为此，两个向量必须具有相同的维度。这被称为Luong注意力（同样是以论文的第一作者命名），有时也称为乘法注意力。点积给出一个分数，所有的分数（在给定的解码器时间步长）都经过softmax层以给出最终的权重，就像在Bahdanau注意力中一样。他们提出的另一种简化方法是在当前时间步长而不是在前一个时间步长中使用解码器的隐藏状态（即$h_{(t)}$）而不是（$h_{(t-1)}$），然后使用注意力机制的输出$\tilde{h}_{(t)}$直接计算解码器的预测（而不是使用它来计算解码器的当前隐藏状态）。他们还提出了一种点积机制的变体，其中编码器的输出在计算点积之前先经过线性变换（即没有偏置项的时间分布Dense层）。这称为“通用”点积方法。他们将两种点积方法与合并注意力机制进行了比较（添加了一个重新缩放参数向量v），他们观察到，点积变体的效果要好于合并注意力。因此，现在很少使用合并注意力。公式16-1总结了这三种注意力机制的公式。

公式16-1：注意力机制

$$\tilde{\boldsymbol{h}}_{(t)} = \sum_i \alpha_{(t,i)} \boldsymbol{y}_{(i)}$$

$$\text{with } \alpha_{(t,t)} = \frac{\exp(e_{(t,i)})}{\sum_{i'} \exp(e_{(t,i')})}$$

$$\text{and } e_{(t,i)} = \begin{cases} \boldsymbol{h}_{(t)}^{\mathrm{T}} \boldsymbol{y}_{(i)} & \text{点} \\ \boldsymbol{h}_{(t)}^{\mathrm{T}} \boldsymbol{W} \boldsymbol{y}_{(i)} & \text{通用} \\ \boldsymbol{v}^{\mathrm{T}} \tanh(\boldsymbol{W}[\boldsymbol{h}_{(t)}; \boldsymbol{y}_{(i)}]) & \text{合并} \end{cases}$$

这是使用TensorFlow Addons将Luong注意力添加到编码器-解码器模型的方法：

```
attention_mechanism = tfa.seq2seq.attention_wrapper.LuongAttention(
    units, encoder_state, memory_sequence_length=encoder_sequence_length)
attention_decoder_cell = tfa.seq2seq.attention_wrapper.AttentionWrapper(
    decoder_cell, attention_mechanism, attention_layer_size=n_units)
```

我们简单地将解码器单元包装在AttentionWrapper中，提供了所需的注意力机制（在此示例中为Luong注意力）。

16.4.1 视觉注意力

注意力机制现在可以用于多种目的。它们在NMT之外的第一个应用之一是使用视觉注意力[5]生成图像标题：卷积神经网络首先处理图像并输出一些特征图，然后具有注意力机制的解码器RNN生成标题，一次生成一个单词。在每个解码器时间步长（每个单词），解码器使用注意力模型将注意力集中在图像的正确部分。例如，在图16-7中，该模型生成了标题“一个女人在公园里扔飞盘”，你可以看到解码器在输出“飞盘”一词时将注意力集中在输入图像的哪一部分上：显然，它的大部分注意力都集中在飞盘上。

图16-7：视觉注意力：输入图像（左）在生成单词“飞盘”之前的模型的关注点（右）[6]

可解释性

注意力机制的另一个好处是，它们使人们更容易理解导致模型产生其输出的原因。这称为可解释性。当模型出错时，它特别有用：例如，如果在雪地里行走的狗的图像被标记为“在雪地里行走的狼”，那么你可以返回并检查模型输出单词“狼”时所关注的是什么。你可能会发现它不仅关注狗，还关注雪，这暗示了可能的解释：也许模型学会区分狼和狗的方式是通过检查周围是否有大雪。然后，你可以通过使用更多没有雪的狼和有雪的狗的图像训练模型来解决此问题。这个示例来自Marco Tulio Ribeiro等人2016年的一篇精彩论文[7]。它对可解释性采用了不同的方法：在分类器的预测局部周围学习一个可解释的模型。

在某些应用程序中，可解释性不仅仅是调试模型的工具，还可能是一项法律要求（可以考虑系统决定是否应授予你贷款）。

注意力机制非常强大以至于你实际上可以仅仅使用它来构建最新模型。

16.4.2 Transformer架构

在2017年的一篇开创性论文中[8]，一个Google研究团队提出了“注意力就是你所需要的一切”。他们设法创建了一个名为Transformer的架构，该架构显著改善了NMT的现有水平，没有使用任何循环层或卷积层[9]，只有注意力机制（加上嵌入层、密集层、归一化层以及其他一些小东西）。额外的好处是该架构的训练速度更快且更易于并行化，因此他们使用了比以前最先进的模型少的时间和成本来进行训练。

图16-8表示了Transformer架构。

让我们看一下该图：

• 左侧是编码器。就像前面一样，它以一批表示为单词ID序列的句子作为输入（输入形状为[批处理大小，最大输入句子长度]），并将每个单词编码为512维的表征（因此编码器的输出形状为[批处理大小，最大输入句子长度，512]。请注意，编码器的顶部堆叠了N次（在本文中，N=6）。

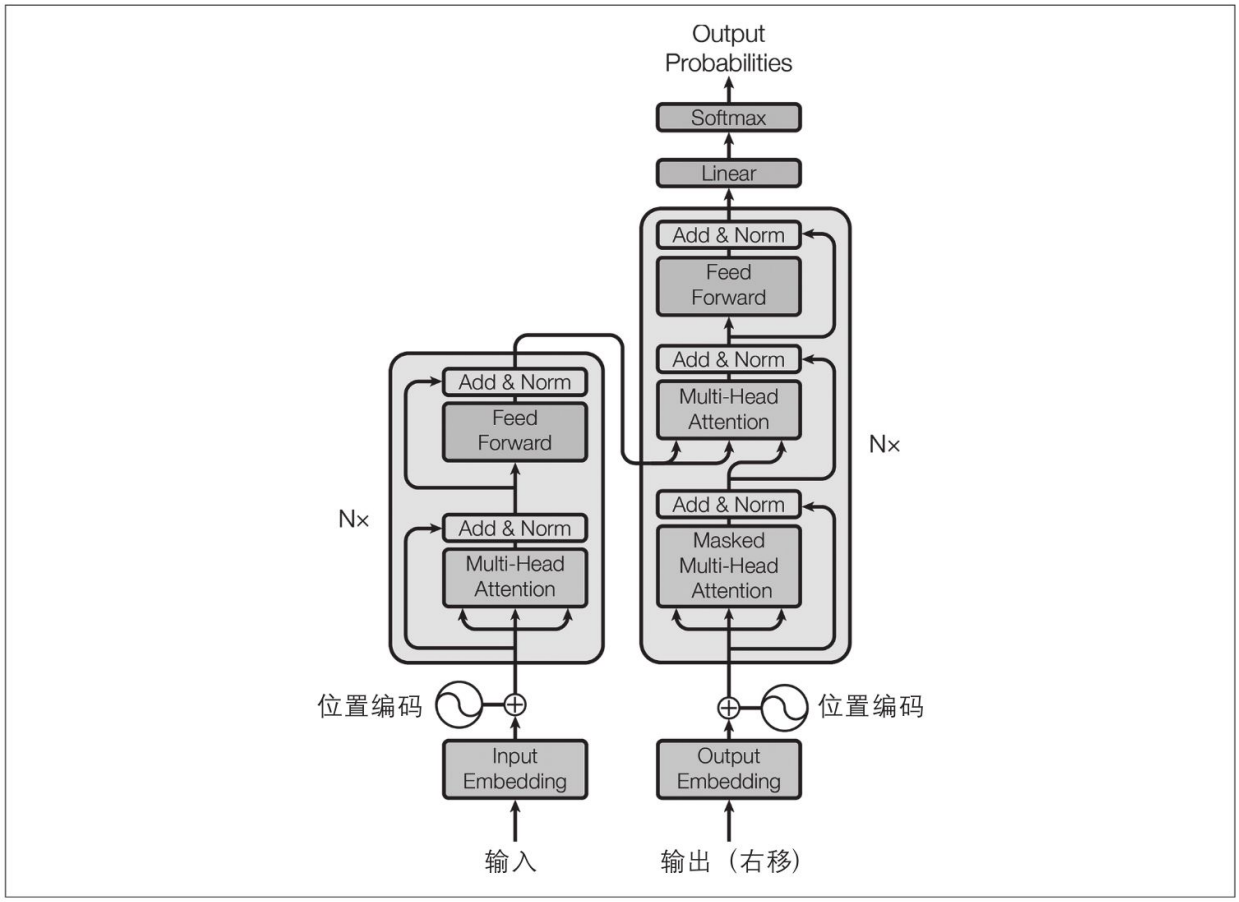


图16-8：Transformer架构[10]

• 右侧是解码器。在训练期间，它以目标句子为输入（也表示为单词ID的序列），向右移动一个时间步长（即在序列的开头插入一个序列开始令牌）。它也接收编码器的输出（即来自左侧的箭头）。请注意，解码器的顶部也堆叠了N次，编码器堆的最终输出在N层的每层中馈送到解码器。与前面一样，解码器在每个时间步长输出每个可能的下一个单词的概率（其输出形状为[批处理大小，最大输出句子长度，词汇表长度]）。

• 在推理期间，无法向解码器提供目标值，因此我们向其提供先前输出的单词（从序列开始令牌开始）。因此，需要反复调用该模型，并在每个回合中预测一个字（在下一个回合中将其馈送到解码器，直到输出序列结束令牌）。

• 仔细观察，你会发现你已经熟悉大多数组件：有两个嵌入层；5×N个跳过连接，每个后面都有一个归一化层；2×N个前馈模块，每个模块由两个密集层组成（第一个使用ReLU激活函数，第二个没有激活函数），最后输出层是使用softmax激活函数的密集层。所有这些层都是时间分布的，因此每个单词都独立于所有其他单词进行处理。但是，如何一次只看一个单词就能翻译一个句子呢？好吧，这就需要新组件出现：

• 编码器的多头注意力（Multi-Head Attention）层对同一句子中每个单词与其他单词之间的关系进行编码，更加关注最相关的单词。例如，句子“They welcomed the Queen of the United Kingdom”中的“Queen”单词的这一层的输出取决于句子中的所有词，它可能会更关注“United”和“Kingdom”，而不是“They”或“welcomed”。这个注意力机制被称为“自我注意力”（句子关注自身）。我们稍后将详细讨论它是如何工作的。解码器的掩码多头注意力（Masked MultiHead Attention）层执行相同的操作，但是每个单词只能被允许关注位于其前面的单词。最后，解码器的多头注意力层上部是解码器关注输入句子中单词的地方。例如，当解码器要输出这个单词的翻译时，解码器可能会密切注意输入句子中的单词“Queen”。

• 位置嵌入只是表示单词在句子中的位置的密集向量（很像词嵌入一样）。第n个位置嵌入被添加到每个句子中的第n个单词的词嵌入中。这使模型可以访问每个单词的位置，这是必需的，因为“多头注意力”层不考虑单词的顺序或位置，只看它们的关系。由于所有其他层都是由于时间分布的，它们无法知道每个单词的位置（无论是相对还是绝对的）。显然，相对和绝对的词位置很重要，我们需要以某种方式将此信

息提供给Transformer，而位置嵌入是实现此目的的好方法。让我们再看一下Transformer架构中的这两个新颖的组件，从位置嵌入开始。

位置嵌入

位置嵌入是对单词在句子中的位置进行编码的密集向量：第i个位置嵌入被添加到句子中第i个单词的词嵌入中。这些位置嵌入可以通过模型学习，但是在本论文中，作者更喜欢使用固定的位置嵌入，该固定位置嵌入是使用不同频率的正弦和余弦函数定义的。位置嵌入矩阵P在公式16-2中定义，并在图16-9的底部表示（已转置），其中$P_{p,\ i}$是位于句子中第p个位置的单词的嵌入的第i个分量。

公式16-2：正弦/余弦位置嵌入

$$P_{p,\ 2i} = \sin(p / 10\,000^{\ 2i/d})$$

$$P_{p,\ 2i+1} = \cos(p / 10\,000^{\ 2i/d})$$

该解决方法具有与学习的位置嵌入相同的性能，但可以扩展到任意长的句子，因此受到青睐。在将位置嵌入添加到单词嵌入之后，模型的其余部分可以访问句子中每个单词的绝对位置，因为每个位置都有唯一的位置嵌入（例如，位于句子中第22个位置的单词的位置嵌入由图16-9左下角的垂直虚线表示，你可以看到它对于该位置是唯一的）。此外，振荡函数（正弦和余弦）的选择使模型也可以学习相对位置。例如，相隔38个单词的单词（例如，在位置p=22和p=60处）在嵌入维度i=100和i=101中始终具有相同的位置嵌入值，如图16-9。这就解释了为什么每个频率都需要正弦和余弦：如果我们仅使用正弦（i=100处的波形），则该模型将无法区分位置p=22和p=35（标记为十字）。

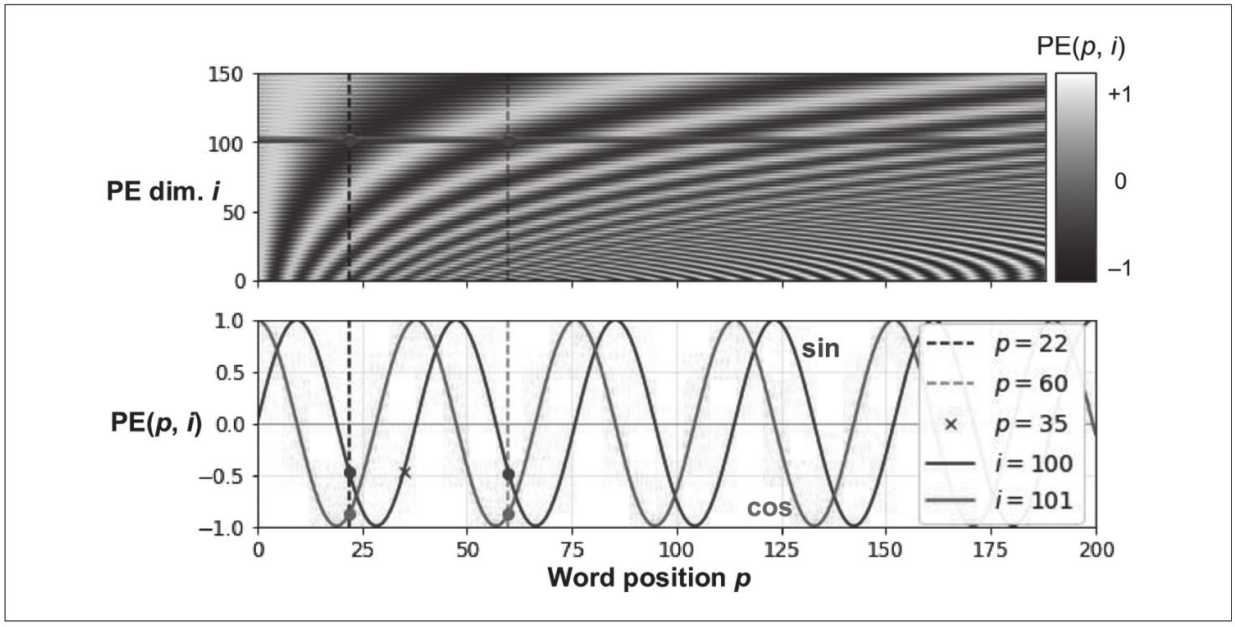


图16-9：正弦/余弦位置嵌入矩阵（转置，上），关注两个i值（下）

TensorFlow中没有PositionalEmbedding层，但是创建起来很容易。出于效率原因，我们预先计算了构造函数中的位置嵌入矩阵（因此我们需要知道最大句子长度max_steps和每个单词表示的维度max_dims）。然后call（）方法将该嵌入矩阵裁剪为输入的大小，并将其加到输入中。由于我们在创建位置嵌入矩阵时增加了一个额外的大小为1的第一维度，因此广播法则将确保把矩阵添加到输入中的每个句子中：

```
class PositionalEncoding(keras.layers.Layer):
    def __init__(self, max_steps, max_dims, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
        if max_dims % 2 == 1: max_dims += 1 # max_dims must be even
        p, i = np.meshgrid(np.arange(max_steps), np.arange(max_dims // 2))
        pos_emb = np.empty((1, max_steps, max_dims))
        pos_emb[0, :, ::2] = np.sin(p / 10000**(2 * i / max_dims)).T
        pos_emb[0, :, 1::2] = np.cos(p / 10000**(2 * i / max_dims)).T
        self.positional_embedding = tf.constant(pos_emb.astype(self.dtype))
    def call(self, inputs):
        shape = tf.shape(inputs)
        return inputs + self.positional_embedding[:, :shape[-2], :shape[-1]]
```

然后，我们可以创建Transformer的第一层：

```
embed_size = 512; max_steps = 500; vocab_size = 10000
encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)
positional_encoding = PositionalEncoding(max_steps, max_dims=embed_size)
encoder_in = positional_encoding(encoder_embeddings)
decoder_in = positional_encoding(decoder_embeddings)
```

现在让我们更深入地了解Transformer模型的核心：多头注意力层。

多头注意力

要理解“多头注意力”层如何工作，我们必须首先理解“缩放点积注意力”层。让我们假设编码器分析了输入句子“They played chess”，并设法理解单词“They”是主语，单词“played”是动词，因此用这些词的表征来编码这些信息。现在假设解码器已经翻译了主语，认为接下来应该翻译动词。为此，它需要从输入句子中获取动词。这类似于字典查找：好像编码器创建了一个字典{“subject”：“They”，“verb”：“played”，…}，解码器想要查找与键“verb”相对应的值。但是，该模型没有具体的令牌来表示键（例如“subject”或“verb”）；它具有这些概念的向量化表示（它是在训练期间学到的），因此用于查找的键（称为查询）不完全匹配字典中的任何键。解决方法是计算查询和字典中每个键之间的相似度，然后使用softmax函数将这些相似度分数转换为加起来为1的权重。如果表示动词的键到目前为止是最相似的查询，那么该键的权重将接近1。然后，该模型可以计算相应值的加权和，因此，如果“verb”键的权重接近1，则该加权和将非常接近单词“played”的表征。简而言之，你可以将整个过程视为可区分的字典查找。像Luong注意力一样，Transformer使用的相似性度量只是点积。实际上，除了比例因子外，该公式与Luong注意力的相同。公式16-3中显示了以向量化形式存在的等式。

公式16-3：缩放点积注意力

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_{keys}}}\right)\boldsymbol{V}$$

在此等式中：

- Q是一个矩阵，每个查询包含一行。其形状为$[n_{queries}, d_{keys}]$，其中$n_{queries}$是查询数，而d_{keys}是每个查询和每个键的维度。

- K是一个矩阵，每个键包含一行。其形状为$[n_{keys}, d_{keys}]$，其中n_{keys}是键和值的数量。

- V是一个矩阵，每个值包含一行。其形状为$[n_{keys}, d_{values}]$，其中d_{values}是每个值的数量。

- $Q\ K^{T}$的形状是$[n_{queries}, n_{keys}]$：每一对查询/键有一个相似性分数。softmax函数的输出具有相同的形状，但所有行的总和为1。最终输出的形状为$[n_{queries}, d_{values}]$：每个查询有一行，其中每一行代表了查询结果（值的加权和）。

- 比例因子会缩小相似度分数，以避免softmax函数饱和，这会导致很小的梯度。

- 可以在计算softmax之前，通过将一个非常大的负值加到相应的相似性分数中来屏蔽一些键/值对。这在“掩码多头注意力”层中很有用。

在编码器中，这个等式应用于批处理中的每个输入句子，其中Q、K和V均等于输入句子中的单词列表（因此，这个句子里的每个单词都会和同一个句子里的每个单词进行比较，包括它自己）。类似地，在解码器的屏蔽注意力层中，该等式应用于批处理中的每个目标句子，其中Q、K和V都等于目标句子中的单词列表，但这次是使用掩码来防止任何单词将自己与其后面的单词进行比较（在推理时，解码器只能访问它已经输出的单词，而不能访问将来的单词，因此在训练期间，我们必须屏蔽掉以后输出的令牌）。在解码器的注意力层上部，键K和值V仅是编码器生成的单词编码的列表，而查询Q是解码器生成的单词编码的列表。

Keras.layers.Attention层实现了缩放点积注意力，将等式16-3有效地应用于一个批量中的多个句子。它的输入就像Q、K和V一样，除了有额外的批处理维度（第一个维度）。

 在TensorFlow中，如果A和B是具有两个以上维度的张量，例如形状分别为[2，3，4，5]和[2，3，5，6]，则tf.matmul（A，B）将这些张量视为2×3的数组，其中每个单元都包含一个矩阵，并且它将乘以对应的矩阵：A中的第i行和第j列的矩阵将与B中的第i行和第j列相乘。由于4×5的矩阵与5×6的矩阵的乘积是4×6的矩阵，因此tf.matmul（A，B）将返回形状为[2，3，4，6]的数组。

如果我们忽略跳过连接、归一化层、前馈块，那事实是这是缩放点积注意力而不是多头注意力，可以像下面这样实现Transformer模型的其余部分：

```
Z = encoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True)([Z, Z])

encoder_outputs = Z
Z = decoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True, causal=True)([Z, Z])
    Z = keras.layers.Attention(use_scale=True)([Z, encoder_outputs])
```

```
outputs = keras.layers.TimeDistributed(
    keras.layers.Dense(vocab_size, activation="softmax"))(Z)
```

use_scale=True参数创建了一个附加参数，该参数可以让层来学习如何适当降低相似性分数。这与Transformer模型有所不同，后者始终用相同的因子$\left(\sqrt{d_{keys}}\right)$来降低相似性分数。创建第二个注意力层时，causal=True参数可确保每个输出令牌仅注意先前的输出令牌，而不是将来的。

现在是时候看最后一个难题了：什么是多头注意力层？其架构如图16-10所示。

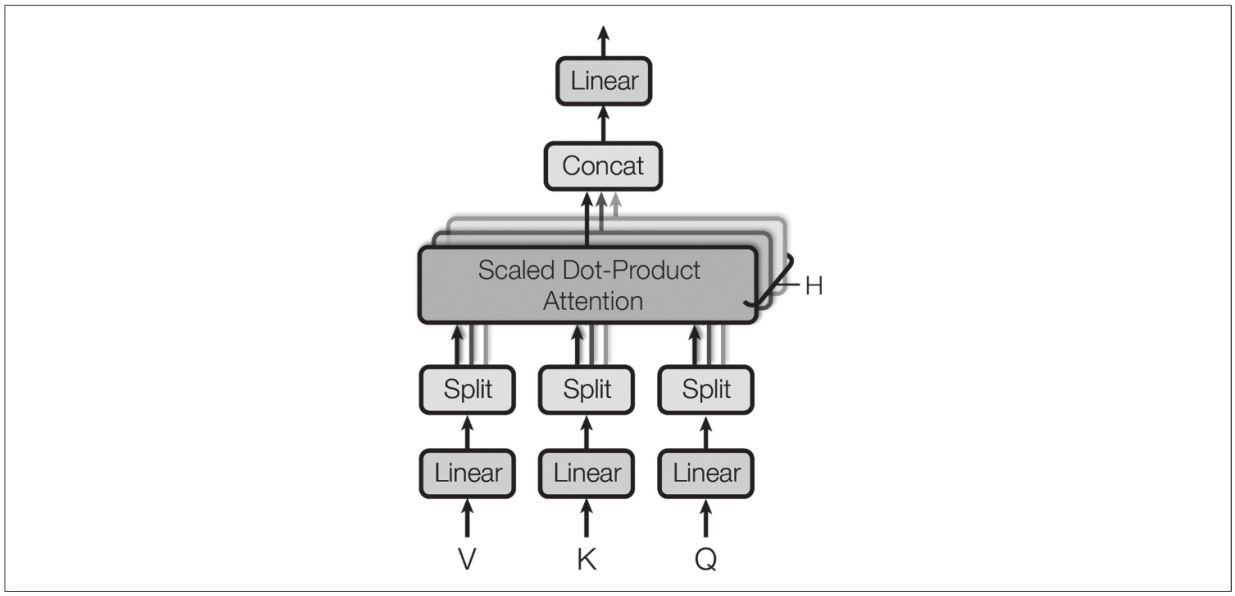


图16-10：多头注意力层架构[11]

如你所见，它只是一堆可缩放的点积注意力层，每层之前都有一个值、键和查询的线性变换（即没有激活函数的时间分布的Dense层）。所有输出都被简单地合并起来，并经过最终的线性变换（又是时间分布的）。为什么？这种架构背后的直觉是什么？好吧，考虑一下我们前面讨论的“played”一词（在句子“They played chess”中）。编码器足够聪明，可以编码其为动词。但是，由于位置编码，单词表征也包括

其在文本中的位置，它可能包含了许多其他对其翻译有用的特征，例如过去时态。简而言之，单词表征编码了单词的许多不同特征。如果我们仅使用单个缩放点积注意力层，则只能一次查询所有这些特征。这就是多头注意力层应用于多个不同的值、键和查询线性变换的原因：这允许模型将许多不同的单词表征投影于不同的子空间，每个子空间都专注于一个单词特征的子集。也许某个线性层会将单词表征投射到一个子空间中，在该子空间中只剩下单词是动词的信息，另一个线性层仅仅提取其过去时态，以此类推。然后，缩放点积注意力层实现了查找阶段，最后我们将所有结果合并起来，并将其投影回原始空间。

在撰写本书时，TensorFlow 2没有可用的Transformer类或MultiHeadAttention类。但是，你可以查看TensorFlow出色的tutorial for building a Transformer model for language understanding。此外，TF Hub团队目前正在将多个基于Transformer的模块移植到TensorFlow 2中，并且应该很快就可以使用。同时，我希望我已经证明自己实现一个Transformer并不难，当然这是一个很棒的练习！

[1] Dzmitry Bahdanau et al.，“Neural Machine Translation by Jointly Learning to Align and Translate”，arXiv preprint arXiv：1409.0473（2014）.

[2] NMT中最常用的指标是双语评估学习（BLEU）评分，该评分将模型产生的每个翻译与几个良好的人工翻译进行比较：它计算出现在任何目标译文中的n-gram（n个单词的序列）的数量，并调整分数以考虑目标译文中生成的n-gram的频率。

[3] 回想一下，时间分布的Dense层相当于你在每个时间步长上独立应用的常规Dense层（只是速度更快）。

[4] Minh-Thang Luong et al.，“Effective Approaches to Attention-Based Neural Machine Translation”，Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing（2015）：1412－1421.

[5] Kelvin Xu et al.，“Show，Attend and Tell：Neural Image Caption Generation with Visual Attention”，Proceedings of the

32nd International Conference on Machine Learning（2015）：2048－2057.
[6] 这是论文图3的一部分。经作者授权转载。
[7] Marco Tulio Ribeiro et al.，“‘Why Should I Trust You?’：Explaining the Predictions of Any Classifier”，Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining（2016）：1135－1144.
[8] Ashish Vaswani et al.，“Attention Is All You Need”，Proceedings of the 31st International Conference on Neural Information Processing Systems（2017）：6000－6010.
[9] 由于Transformer使用时间分布的Dense层，因此可以说它使用了内核大小为1的一维卷积层。
[10] 这是论文的图1，经作者授权转载。
[11] 这是论文图2的右侧，经作者授权转载。

16.5 最近语言模型的创新

2018年被称为“NLP的ImageNet时刻”：进展惊人，越来越大的LSTM和基于Transformer的架构在庞大的数据集上得到了训练。我强烈建议你查看以下2018年发表的论文：

• Matthew Peters撰写的ELMo论文[1]介绍了从语言模型来的嵌入（Embeddings from Language Model，ELMo）：这些是从深度双向语言模型的内部状态中学到的上下文词嵌入。例如，“queen”一词在“Queen of the United Kingdom”和“queen bee”中的嵌入将不同。

• Jeremy Howard和Sebastian Ruder撰写的ULMFiT论文[2]证明了无监督预训练对NLP任务的有效性：作者使用自监督学习的方法（即根据数据自动生成标签）在庞大的文本语料库上训练了LSTM语言模型，然后他们在各种任务上进行微调。他们的模型在6个文本分类任务上的表现远远超过了现有技术（在大多数情况下，错误率降低了18－24%）。而且，他们表明，通过仅仅在100个带标签的示例上微调预训练的模型，它们可以实现与从10 000个示例中从头开始训练的模型有相同的性能。

• Alec Radford和其他OpenAI研究人员撰写的GPT论文[3]也展示了无监督预训练的有效性，但是这次使用了类似于Transformer的架构。作者在大型数据集上对一个很大但相当简单的架构进行了预训练，该架构由12个Transformer模块（仅使用掩码多头注意力层）的堆叠组成，并再次使用自监督学习进行训练。然后，他们在各种语言任务上对其进行了微调，对每个任务仅进行了少量修改。任务非常多样：它们包括文本分类、蕴含（句子A是否蕴含句子B）[4]、相似度（例如，“今天天气不错”与“晴天”非常相似），以及回答问题（鉴于文本的几段给出了某些上下文，该模型必须回答一些多项选择题）。仅仅

几个月后，即2019年2月，Alec Radford、Jeffrey Wu和其他OpenAI研究人员发表了GPT-2论文[5]，提出了非常相似的架构，但仍然很大（参数超过15亿！），他们表明它可以在许多任务上获得良好的性能，而无须进行任何微调。这称为零次学习（Zero-Shot Learning，ZSL）。可以在https://github.com/openai/gpt-2上获得GPT-2模型的较小版本（参数只有1.17亿），以及其预训练的权重。

• Jacob Devlin和其他Google研究人员的BERT论文[6]还展示了在大型语料库上进行自监督预训练的有效性，它使用了与GPT类似的架构，但没有非掩码多头注意力层（就像Transformer的编码器）。这意味着模型本质是双向的。这是BERT（Bidirectional Encoder Representations from Transformers）中的B的来源。最重要的是，作者提出了两个预训练任务，这些任务可以解释该模型的大部分优势：

掩码语言模型（Masked Language Model，MLM）

句子中的每个单词被屏蔽的可能性为15%，模型经过训练来预测被屏蔽的单词。例如，如果原始句子是“She had fun at the birthday party”，则可以把句子“She<mask>fun at the<mask>party”赋予模型，模型必须预测单词“had”和“birthday”（其他输出将被忽略）。更准确地说，每个选定的词都有80%的机会被屏蔽，有10%的机会被随机词替换（以减少预训练和微调之间的差异，因为模型在微调时不会看到<mask>令牌），并且有10%的机会被保留（使模型偏向正确的答案）。

下一句预测（Next Sentence Prediction，NSP）

训练模型来预测两个句子是否连续。例如，应该预测“The dog sleeps”和“It snores loudly”是连续的句子，而“The dog sleeps”和“The Earth orbits the Sun”不是连续的句子。这是一

项具有挑战性的任务，当对诸如回答问题或蕴含问题等任务进行微调时，它可以显著地改进模型的性能。

如你所见，2018年和2019年的主要创新是更好的子单词分词化、从LSTM转换为Transformers，并使用自监督学习来预训练通用语言模型，然后通过很少的架构更改来进行微调（或完全没有更改）。事情发展很快。没有人可以说明年将流行什么架构。今天，它显然是Transformers，但明天可能是CNN（例如，请参阅Maha Elbayad等人2018年的论文[7]，研究人员使用掩码2D卷积层来做序列到序列的任务）。或者甚至可能是RNN，如果它们卷土重来的话（例如，请参阅Shuai Li等人2018年的论文[8]，通过在给定的RNN层中使神经元彼此独立，可以训练更深层的RNN，使该RNN能够学习更长的序列）。

在下一章中，我们将讨论如何使用自动编码器以无监督方式学习深度表征，并且将使用生成式对抗网络（GAN）来生成图像等！

[1] Matthew Peters et al.，“Deep Contextualized Word Representations”，Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics：Human Language Technologies 1（2018）：2227-2237.

[2] Jeremy Howard and Sebastian Ruder，“Universal Language Model Fine-Tuning for Text Classification”，Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics 1（2018）：328-339.

[3] Alec Radford et al.，“Improving Language Understanding by Generative Pre-Training”（2018）.

[4] 例如，“简在她的朋友的生日聚会上玩得很开心”这样的句子蕴含了“简很享受聚会”，但是却与“每个人都讨厌聚会”相矛盾，与“地球是平坦的”无关。

[5] Alec Radford et al.，“Language Models Are Unsupervised Multitask Learners”（2019）.
[6] Jacob Devlin et al.，“BERT：Pre-training of Deep Bidirectional Transformers for Language Understanding”，Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics：Human Language Technologies 1（2019）.
[7] Maha Elbayad et al.，“Pervasive Attention：2D Convolutional Neural Networks for Sequence-to-Sequence Prediction”，arXiv preprint arXiv：1808.03867（2018）.
[8] Shuai Li et al.，“Independently Recurrent Neural Network（IndRNN）：Building a Longer and Deeper RNN”，Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition（2018）：5457－5466.

16.6 练习题

1. 使用有状态RNN与无状态RNN有什么优缺点？

2. 人们为什么使用编码解码RNN而不是简单序列对序列RNN进行自动翻译？

3. 如何处理可变长度的输入序列？可变长度的输出序列呢？

4. 什么是集束搜索，为什么要使用它？你可以使用哪种工具来实现它？

5. 什么是注意力机制？它有什么帮助？

6. Transformer架构中最重要的层是什么？目的是什么？

7. 你何时需要使用采样softmax？

8. Hochreiter和Schmidhuber在有关LSTM的论文中使用了嵌入式Reber语法。它们是人工语法，可产生诸如“BPBTSXXVPSEPEPE”之类的字符串。请查阅Jenny Orr对这个主题的精彩介绍。选择一种特定的嵌入式Reber语法（例如，Jenny Orr主页上表示的语法），然后训练RNN以识别字符串是否符合该语法。首先，你需要编写一个能够生成训练批量处理的函数，其中包含大约50%符合语法的字符串和50%不符合语法的字符串。

9. 训练一个可以将日期字符串从一种格式转换为另一种格式的编码解码模型（例如，从“2019年4月22日”到“2019-04-22”）。

10. 阅读TensorFlow的带注意力机制的神经机器翻译教程。

11.使用最新的语言模型之一（例如BERT）来生成更具说服力的莎士比亚文本。

这些练习题的解答在附录A中提供。

第17章　使用自动编码器和GAN的表征学习和生成学习

自动编码器是一种人工神经网络，不需要任何监督（即无标记训练集）即可学习输入数据的密集表征，称为潜在表征或编码。这些编码的维度通常比输入数据低得多，这使得自动编码器可用于降低维度（见第8章），尤其是可用于可视化。自动编码器还充当特征检测器，还可用于深度神经网络的无监督预训练（如我们在第11章中讨论的）。最后，一些自动编码器是生成模型：它们能够随机生成看起来与训练数据非常相似的新数据。例如，你可以在人脸图片上训练自动编码器，然后就可以生成新的人脸。但是生成的图像通常是模糊的且并不完全真实。

相比之下，由生成式对抗网络（GAN）生成的人脸现在令人信服，很难相信他们所代表的人不存在。你可以访问https://thispersondoesnotexist.com/来自己做出判断，该网站显示了由最近的GAN架构StyleGAN生成的面孔（你也可以通过https://thisrentaldoesnotexist.com/来查看一些Airbnb的卧室）。GAN现在广泛用于超分辨率（提高图像的分辨率）、着色、强大的图像编辑（例如，用逼真的背景替换照片）、将简单的草图变成逼真的图像、预测视频中的下一帧、扩充数据集（以训练其他模型）、生成其他类型的数据（例如文本、音频和时间序列）、识别其他模型中的弱点并加以增强，等等。

自动编码器和GAN都是非监督的，都学习密集表征，都可以用作生成模型，并且具有许多相似的应用。但是它们的工作方式截然不同：

• 自动编码器只需学习将其输入复制到其输出即可。这听起来像是一项琐碎的任务，但是我们会看到以各种方式约束网络会使其变得相当困难。例如，你可以限制潜在表征的大小，或者向输入添加噪声

并训练网络来恢复原始输入。这些限制使自动编码器无法将输入简单地直接复制到输出，这会迫使它学习有效的数据表示方法。简而言之，编码是自动编码器在某些约束下学习恒等函数的副产品。

• GAN由两个神经网络组成：一个试图生成看起来与训练数据相似数据的生成器，以及一个试图从虚假数据中分辨出真实数据的判别器。这种架构在深度学习中非常创新的，因为生成器和判别器在训练期间相互竞争：生成器就像是一个试图做假币的犯罪分子，而判别器就像是一个警察，试图从真币中分辨出假币。对抗训练（训练竞争性神经网络）被广泛认为是近年来最重要的想法之一。在2016年，Yann LeCun甚至说这是“近十年来机器学习中最有趣的想法。”

在本章中，我们开始更深入地探讨自动编码器的工作原理，以及如何把它们用于降维、特征提取、无监督的预训练或作为生成模型。这会自然地把我们引向GAN。我们首先构建一个简单的GAN来生成伪图像，但是我们会看到训练通常非常困难。我们会讨论你在对抗训练中会遇到的主要困难，以及解决这些困难的一些主要技巧。让我们从自动编码器开始吧！

17.1 有效的数据表征

你发现以下哪个数字顺序最容易记忆？

- 40，27，25，36，81，57，10，73，19，68

- 50，48，46，44，42，40，38，36，34，32，30，28，26，24，22，20，18，16，14

乍一看，第一个序列似乎应该更容易些，因为它要短得多。但是，如果仔细看第二个序列，你会发现它只是从50到14的偶数列表。一旦你注意到这个模式，第二个序列就比第一个序列更容易记忆，因为你只需要记住模式（即偶数递减）以及开始和结束的数字（即50和14）。请注意，如果你可以快速轻松地记住很长的序列，就不会关心第二个序列中是否存在模式。你会认真地记忆每个数字，就是这样。很难记住长序列这一事实使模式识别变得很有用，希望这能澄清为什么在训练过程中约束自动编码器会促使其发现和利用数据中的模式。

记忆、感知和模式匹配之间的关系在1970年代初期由William Chase和Herbert Simon进行了著名的研究[1]。他们观察到，国际象棋专家可以通过观察棋盘上的位置只需5秒钟就能记住所有棋子的位置，这是大多数人无法实现的任务。但是，只有把棋子放在真实位置（根据实际棋局）时才是这种情况，而不是将棋子随机放置。国际象棋专家的记忆力不比你我的更好。多亏了他们在象棋中的经验，他们才更容易看到国际象棋的模式。注意到模式可以帮助他们有效地存储信息。

就像这个记忆实验中的国际象棋棋手一样，自动编码器会查看输入，将其转换为有效的潜在表征，然后输出一些看起来非常接近输入的东西。自动编码器通常由两部分组成：将输入转换为潜在表征的编码器（或识别网络），然后是将内部表征转换为输出的解码器（或生成网络）（见图17-1）。

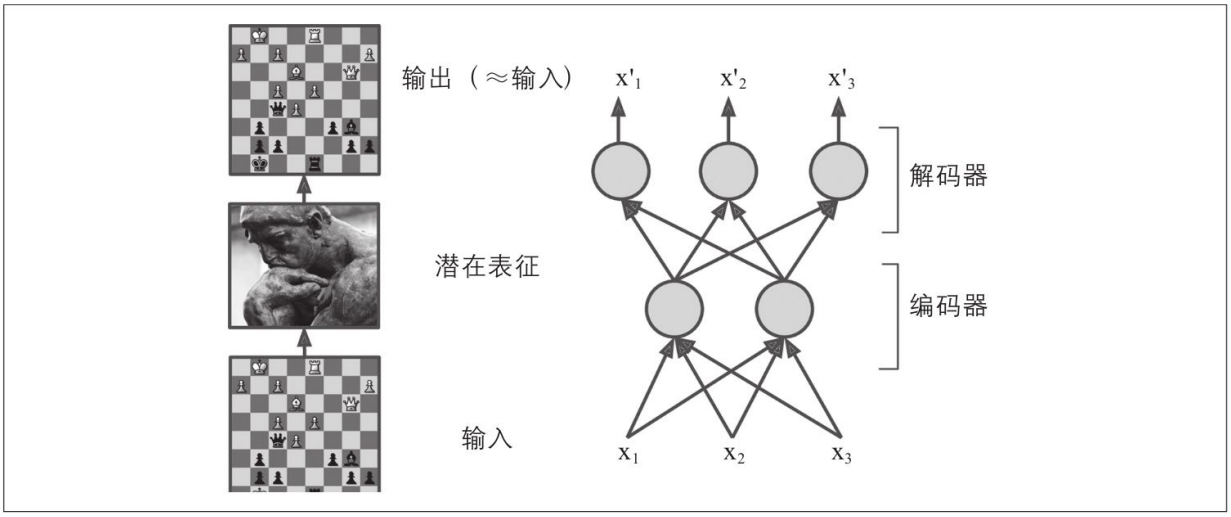


图17-1：国际象棋记忆实验（左）和一个简单的自动编码器（右）

如你所见，自动编码器通常具有与多层感知器（MLP，见第10章）相同的架构，除了输出层中的神经元数量必须等于输入的数量。在此示例中，只有一个由两个神经元组成的隐藏层（编码器），以及一个由三个神经元组成的输出层（解码器）。输出通常称为重构，因为自动编码器会试图重构输入，并且成本函数包含一个重构损失，当重构与输入不同时会惩罚这个模型。

因为内部表征的维度比输入数据的维度低（它是2D而不是3D），所以自动编码器被认为是不完整的。不完整的自动编码器无法将其输入简单地复制到编码中，必须找到一种输出其输入副本的方法。它被迫学习输入数据中最重要的特征（并删除不重要的特征）。让我们看看如何实现一个非常简单的不完整的自动编码器来降低维度。

[1] William G.Chase and Herbert A.Simon，“Perception in Chess”，Cognitive Psychology 4，no.1（1973）：55－81.

17.2 使用不完整的线性自动编码器执行PCA

如果自动编码器仅使用线性激活，并且成本函数是均方误差（MSE），则最后执行的是主成分分析（PCA，见第8章）。

以下代码构建了一个简单的线性自动编码器，来对3D数据集执行PCA，将其投影到2D：

```
from tensorflow import keras

encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])
autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=0.1))
```

这段代码实际上与我们在第16章中构建的所有MLP没有太大的区别，但是需要注意以下几点：

- 我们将自动编码器分为两个子组件：编码器和解码器。两者都是具有单一Den se层的常规Seq uential模型，而自动编码器是包含编码器和解码器的Seq uential模型（请记住，一个模型可以用作另一个模型中的一个层）。

- 自动编码器的输出数量等于输入的数量（即3）。

- 为了执行简单的PCA，我们不使用任何激活函数（即所有神经元都是线性的），成本函数为MSE。很快我们会看到更复杂的自动编码器。

现在让我们在一个简单生成的3D数据集上训练模型，并使用它对同一数据集进行编码（即将其投影到2D）：

```
history = autoencoder.fit(X_train, X_train, epochs=20)
codings = encoder.predict(X_train)
```

注意，相同的数据集X_train用作输入和目标。图17-2显示了原始3D数据集（在左侧）和自动编码器的隐藏层（即在右侧的编码层）的输出。如你所见，自动编码器找到了将数据投影到的最佳2D平面，并尽可能地保留了数据中的方差（就像PCA一样）。

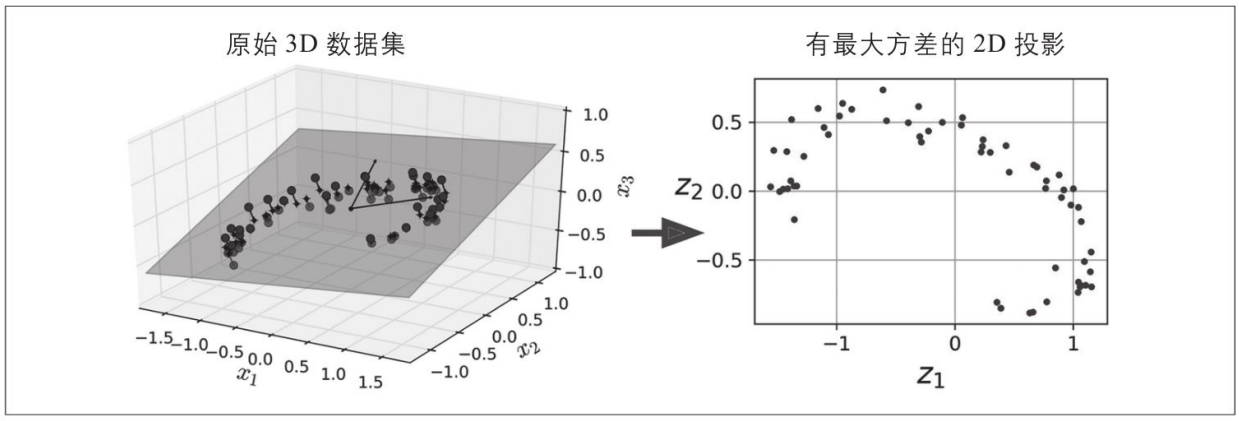


图17-2：由不完整线性自动编码器执行的PCA

 你可以将自动编码器视为一种自监督式的学习形式（即使用具有自动生成的标签的有监督学习技术，在这种情况下，该标签就等于输入）。

17.3 堆叠式自动编码器

与我们讨论过的其他神经网络一样，自动编码器可以具有多个隐藏层。在这种情况下，它们被称为堆叠式自动编码器（或深度自动编码器）。添加更多的层有助于自动编码器学习更多复杂的编码。就是说，要注意不要使自动编码器过于强大。想象一个强大的编码器，它只是学会了把每个输入映射到单个任意数字（而解码器则学习反向映射）。显然，这样的自动编码器可以完美地重建训练数据，但是它不会学到任何有用的数据表征（并且不太可能将其很好地泛化到新实例中）。

堆叠式自动编码器的架构典型地相对于中间隐藏层（编码层）对称。简单来说，它看起来像三明治。例如，MNIST的自动编码器（在第3章中介绍）可能具有784个输入，其后是具有100个神经元的隐藏层，然后是具有30个神经元的中间隐藏层，然后是具有100个神经元的另一个隐藏层，以及有784个神经元的一个输出层。这种堆叠式自动编码器如图17-3所示。

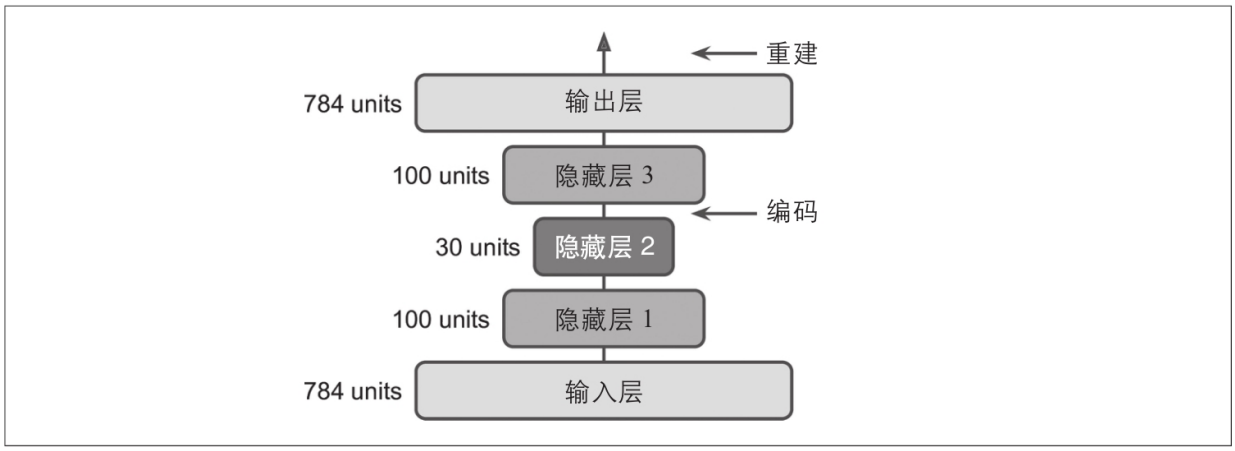


图17-3：堆叠式自动编码器

17.3.1 使用Keras实现堆叠式自动编码器

你可以像实现常规的深层MLP那样来实现堆叠式自动编码器。特别地，我们可以使用在第11章中用于训练深度网络的相同技术。例如，以下代码使用SELU激活函数为Fashion MNIST构建了一个堆叠式自动编码器（如第10章所述进行加载和归一化）：

```
stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu"),
])
stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
stacked_ae = keras.models.Sequential([stacked_encoder, stacked_decoder])
stacked_ae.compile(loss="binary_crossentropy",
                   optimizer=keras.optimizers.SGD(lr=1.5))
history = stacked_ae.fit(X_train, X_train, epochs=10,
                         validation_data=[X_valid, X_valid])
```

让我们看一下这段代码：

• 就像之前一样，我们将自动编码器模型分为两个子模型：编码器和解码器。

• 编码器使用28×28像素的灰度图像，先将它们展平，以便把每个图像表示为大小为784的向量，然后通过两个尺寸递减的Dense层（100个神经元然后是30个）来处理这些向量，两个都使用SELU激活函数（你可能还希望添加LeCun常规初始化，但是网络不是很深，因此不会有太大的不同）。对于每个输入图像，编码器输出大小为30的向量。

• 解码器使用大小为30的编码（由编码器输出），并通过两个大小递增的D e n s e层（100个神经元然后为784个）来处理它们，并将最终向量重构为28×28的数组，因此解码器的输出具有与编码器输入相同的形状。

• 在编译堆叠式自动编码器时，我们使用二元交叉熵损失代替均方误差。我们将重建任务视为多标签二元分类问题：每个像素强度代表像素应为黑色的概率。以这种方式（而不是回归问题）往往会使模型收敛得更快[1]。

• 最后，我们使用X_train作为输入和目_标来训练模型（类似地，我们使用X_valid作为验证输入和目标）。

17.3.2 可视化重构

确保对自动编码器进行了恰当训练的一种方法是比较输入和输出：差异不应该很明显。让我们从验证集中绘制一些图像，以及它们的重构：

```
def plot_image(image):
    plt.imshow(image, cmap="binary")
    plt.axis("off")

def show_reconstructions(model, n_images=5):
    reconstructions = model.predict(X_valid[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plot_image(X_valid[image_index])
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plot_image(reconstructions[image_index])

show_reconstructions(stacked_ae)
```

图17-4显示了生成的图像。

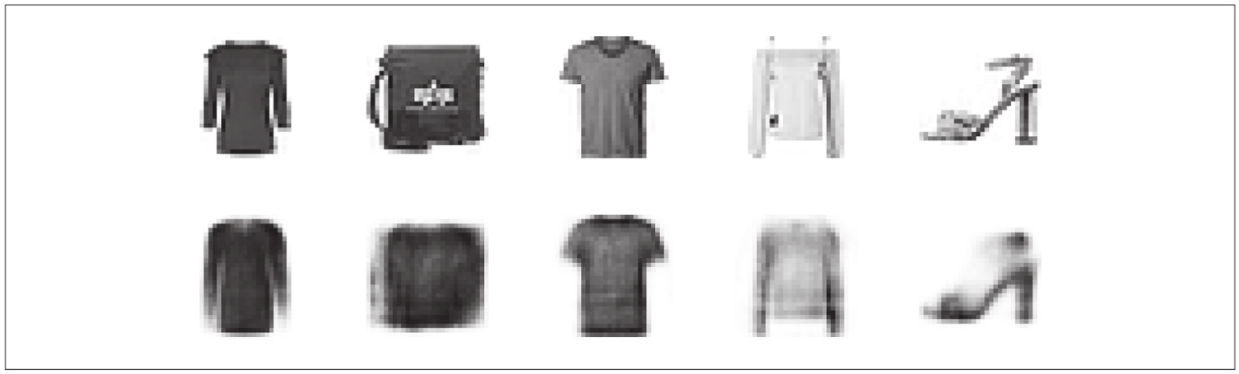


图17-4：原始图像（上）及其重构（下）

重构是可识别的，但损失有点太大。我们可能需要训练模型更久一点，或者使编码器和解码器更深，或者使编码更大。但是如果我们使网络过于强大，它就能够在没有学习到数据中任何有用模式的情况下，进行完美的重构。现在，让我们看看该模型。

17.3.3 可视化Fashion MNIST数据集

现在我们已经训练了一个堆叠式自动编码器，可以使用它来减少数据集的维度。对于可视化而言，与其他降维算法（例如第8章中讨论的算法）相比，它并没有给出很好的结果，但是自动编码器的一大优势是它们可以处理具有许多实例和许多特征的大型数据集。因此，一种策略是使用自动编码器将维度降低到合理水平，然后使用另一维降维算法进行可视化。让我们使用这种策略来可视化Fashion MNIST。首先，我们使用堆叠式自动编码器中的编码器将维度减小到30，然后使用Scikit-Learn的t-SNE算法的实现将维度减小到2来进行可视化：

```
from sklearn.manifold import TSNE

X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE()
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

绘制数据集：

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
```

图17-5显示了生成的散点图（通过显示一些图像进行了美化）。t-SNE算法识别出几个与类合理匹配的集群（每个类用不同颜色表示）。

图17-5：使用了自动编码器和t-SNE的Fashion MNIST可视化

因此，自动编码器可用于降维。另一个应用是无监督的预训练。

17.3.4 使用堆叠式自动编码器的无监督预训练

正如我们在第11章中讨论的那样，如果你要处理一个复杂的有监督任务，但是没有很多标记好的训练数据，一种解决方案是找到执行类似任务的神经网络并重用其较低层网络。这样就可以使用很少的训练数据来训练一个高性能模型，因为你的神经网络不必学习所有的底层特征，它只会重用现有网络学到的特征检测器。

同样，如果你有一个大型数据集，但大多数未被标记，你可以先使用所有的数据训练一个堆叠的自动编码器，然后重用较低层网络为你的实际任务创建神经网络，并使用标记的数据对其进行训练。例如，图17-6显示了如何使用堆叠式自动编码器对分类神经网络执行无监督预训练。在训练分类器时，如果你确实没有太多标记的训练数据，你可能想要冻结预训练的层（至少是较低的层）。

 拥有大量未标记数据和少量标记数据是很常见的。创建大型的未标记数据集通常很便宜（例如，一个简单的脚本即可从互联网上下载数百万个图像），但是标记这些图像（例如将它们分类为可爱或不可爱）通常只能由人工来可靠地完成。标记实例既耗时又昂贵，因此通常只有数千个人工标记的实例是正常的。

该实现没有什么特别的：只需要使用所有训练数据（标记的和未标记的）来训练自动编码器，然后重用其编码器层来创建一个新的神经网络（有关示例请参阅本章末的练习）。

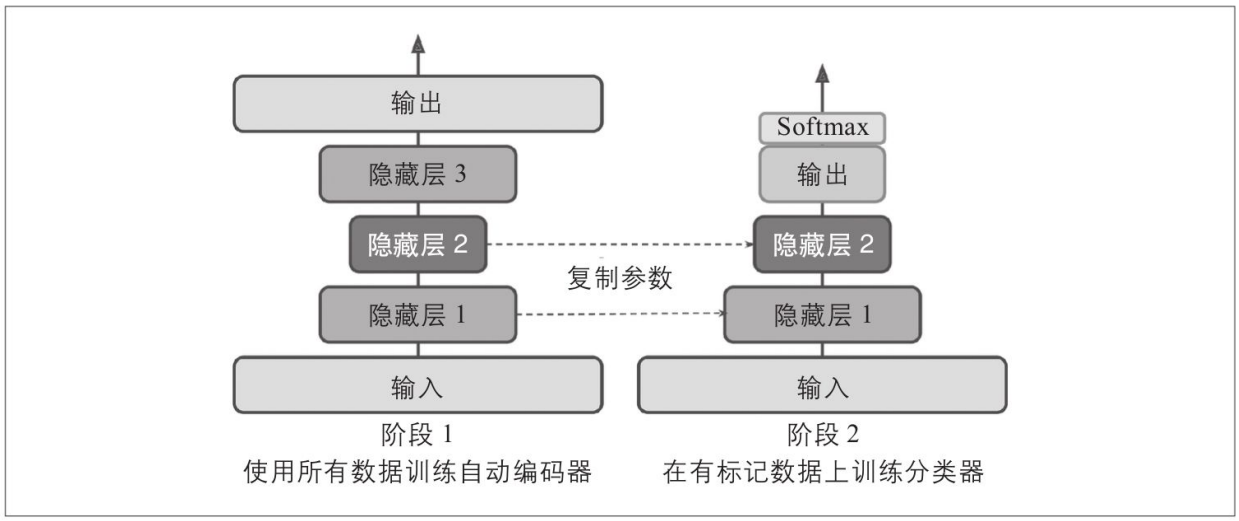


图17-6：使用自动编码器的无监督预训练

接下来，让我们看一些训练堆叠式自动编码器的技术。

17.3.5 绑定权重

当一个自动编码器整齐对称时，就像我们刚刚构建的那样，一种常见的技术是将解码器层的权重与编码器层的权重绑定起来。这样可以将模型中权重的数量减少一半，从而加快训练速度并降低过拟合的风险。具体来说，如果自动编码器总共有N层（不计算输入层），并且$\mathbf{W}_L$表示第L层的连接权重（例如，层1是第一个隐藏层，层N/2是编码层，层N是

输出层），则解码器层权重可以简单定义为：$W_{N-L+1}=W^T_L$（其中L=1，2，…，N/2）。

为了使用Keras绑定各层之间的权重，让我们定义一个自定义层：

```
class DenseTranspose(keras.layers.Layer):
    def __init__(self, dense, activation=None, **kwargs):
        self.dense = dense
        self.activation = keras.activations.get(activation)
        super().__init__(**kwargs)
    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="bias", initializer="zeros",
                                      shape=[self.dense.input_shape[-1]])
        super().build(batch_input_shape)
    def call(self, inputs):
        z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(z + self.biases)
```

这个自定义层的作用类似于常规的Dense层，但是它使用了另一个Dense层的权重，并进行了转置（设置transpose_b=True等效于转置第二个参数，但是它的效率更高，因为它可以在matmul（）操作中即时执行转置）。但是，它使用自己的偏置向量。接下来，我们可以构建一个新的堆叠式自动编码器，与前一个非常相似，但是将解码器的Dense层绑定到了编码器的密集层：

```
dense_1 = keras.layers.Dense(100, activation="selu")
dense_2 = keras.layers.Dense(30, activation="selu")

tied_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    dense_1,
    dense_2
])

tied_decoder = keras.models.Sequential([
    DenseTranspose(dense_2, activation="selu"),
    DenseTranspose(dense_1, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

tied_ae = keras.models.Sequential([tied_encoder, tied_decoder])
```

与前一个模型相比，该模型的重构误差略低，参数数量几乎减少了一半。

17.3.6 一次训练一个自动编码器

与其像一次训练整个堆叠式自动编码器那样，不如一次训练一个浅层自动编码器，然后将它们全部堆叠成一个堆叠式自动编码器（因此得名），如图17-7所示。如今，这种技术已很少使用，但是你仍然可能会遇到有关“贪婪的分层训练”的论文，因此最好知道它的含义。

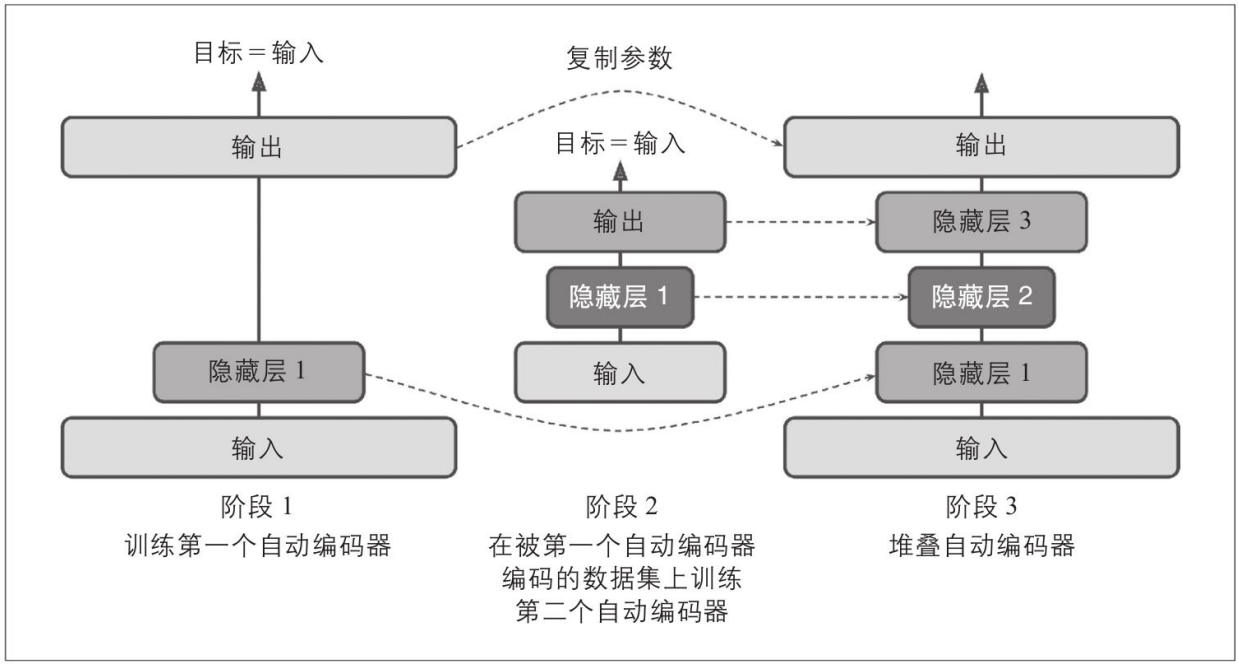


图17-7：一次训练一个自动编码器

在训练的第一阶段，第一个自动编码器学习重建输入。然后，我们使用第一个自动编码器对整个训练集进行编码，这给我们提供了一个新的（压缩）训练集。然后，我们在此新数据集上训练第二个自动编码器，这是训练的第二阶段。最后，我们使用所有这些自动编码器来构建一个“大三明治”，如图17-7所示（我们首先堆叠每个自动编码器的隐藏层，然后以相反的顺序堆叠输出层）。这给了我们最终的堆叠式自动编码器（有关实现，请参见notebook中的“Training One Autoencoder

at a Time”部分）。我们可以通过这种方式轻松地训练更多的自动编码器，从而构建一个非常深的堆叠式自动编码器。

正如我们之前讨论的那样，当前对深度学习感兴趣的触发因素之一是Geoffrey Hinton等人在2006年的发现。深层神经网络可以使用这种贪婪的分层方法以无监督的方式进行预训练。他们为此使用了受限的Boltzmann机器（RBM，见附录E），但在2007年，Yoshua Bengio等人[2]证明了自动编码器也能很好地工作。多年来，这是训练深度网络的唯一有效方法，直到第11章介绍的许多技术使一次训练深度网络成为可能。

自动编码器不仅限于密集网络，你还可以构建卷积自动编码器，甚至是循环自动编码器。让我们现在来看看这些。

[1] 你可能会想使用精度指标，但是它不能正常工作，因为该指标期望每个像素的标签为0或1。你可以通过创建自定义指标轻松解决此问题，该指标将目标和预测取整为0或1后计算精度。

[2] Yoshua Bengio et al.，“Greedy Layer-Wise Training of Deep Networks”，Proceedings of the 19th International Conference on Neural Information Processing Systems（2006）：153–160.

17.4 卷积自动编码器

如果你要处理图像，那么到目前为止我们所看到的自动编码器将无法很好地工作（除非图像非常小）：正如我们在第14章中看到的那样，卷积神经网络比密集网络更适合于处理图像。因此，如果你要为图像构建自动编码器（例如用于无监督预训练或降维），则需要构建卷积自动编码器[1]。编码器是由卷积层和池化层组成的常规CNN。它通常会减小输入的空间尺寸（即高度和宽度），同时会增加深度（即特征图的数量）。解码器必须进行相反的操作（放大图像并减少其深度到原始尺寸），为此你可以使用转置卷积层（或者可以将上采样层与卷积层组合在一起）。以下是用于Fashion MNIST的简单卷积自动编码器：

```
conv_encoder = keras.models.Sequential([
    keras.layers.Reshape([28, 28, 1], input_shape=[28, 28]),
    keras.layers.Conv2D(16, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(64, kernel_size=3, padding="same",activation="selu"),
    keras.layers.MaxPool2D(pool_size=2)
])
conv_decoder = keras.models.Sequential([
    keras.layers.Conv2DTranspose(32, kernel_size=3, strides=2, padding="valid",
                                 activation="selu",
                                 input_shape=[3, 3, 64]),
    keras.layers.Conv2DTranspose(16, kernel_size=3, strides=2, padding="same",
                                 activation="selu"),
    keras.layers.Conv2DTranspose(1, kernel_size=3, strides=2, padding ="same",
                                 activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
conv_ae = keras.models.Sequential([conv_encoder, conv_decoder])
```

---

[1] Jonathan Masci et al.，“Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction”，Proceedings of the 21st International Conference on Artificial Neural Networks 1（2011）：52－59.

17.5 循环自动编码器

如果要为序列构建自动编码器，例如时间序列或文本（例如，用于无监督学习或降维），那么递归神经网络（见第15章）可能比密集网络更适合。构建循环自动编码器非常简单直接：编码器通常是序列到向量的RNN，它将输入序列压缩为单个向量。解码器是向量到序列RNN，做相反的处理：

```
recurrent_encoder = keras.models.Sequential([
    keras.layers.LSTM(100, return_sequences=True, input_shape=[None, 28]),
    keras.layers.LSTM(30)
])
recurrent_decoder = keras.models.Sequential([
    keras.layers.RepeatVector(28, input_shape=[30]),
    keras.layers.LSTM(100, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(28, activation="sigmoid"))
])
recurrent_ae = keras.models.Sequential([recurrent_encoder, recurrent_ decoder])
```

这种循环自动编码器可以处理任何长度的序列，每个时间步长具有28个维度。这意味着它可以通过把每个图像视为一系列行来处理Fashion MNIST图像：在每个时间步，RNN将处理一个28像素的行。显然你可以对任何类型的序列使用循环自动编码器。请注意，我们把RepeatVector层用作解码器的第一层，以确保其输入向量在每个时间步长都馈送到解码器。

好，让我们退后一步。到目前为止，我们已经看过了各种各样的自动编码器（基本的、堆叠式的、卷积和循环编码器），并且已经研究了如何训练它们（一次性或逐层）。我们还研究了一些应用：数据可视化和无监督预训练。

到目前为止，为了强制自动编码器学习有趣的特征，我们限制了编码层的大小，使其成为不完整自动编码器。实际上，还有许多其他

类型的约束可以使用，包括使编码层与输入一样大甚至更大，从而成为一个完整自动编码器。现在让我们看看其中一些方法。

17.6　去噪自动编码器

强制自动编码器学习有用特征的另一种方法是向其输入中添加噪声，训练它来恢复原始的无噪声输入。这个想法自1980年代开始就存在（在Yann LeCun 1987年的硕士论文中提到过）。在2008年的论文中[1]，Pascal Vincent等人表明自动编码器也可以用于特征提取。在2010年的论文中[2]，Vincent等人提出了堆叠式去噪自动编码器。

噪声可以是添加到输入的纯高斯噪声，也可以是随机关闭的输入，就像dropout（在第11章中介绍）一样。图17-8显示了这两种方法。

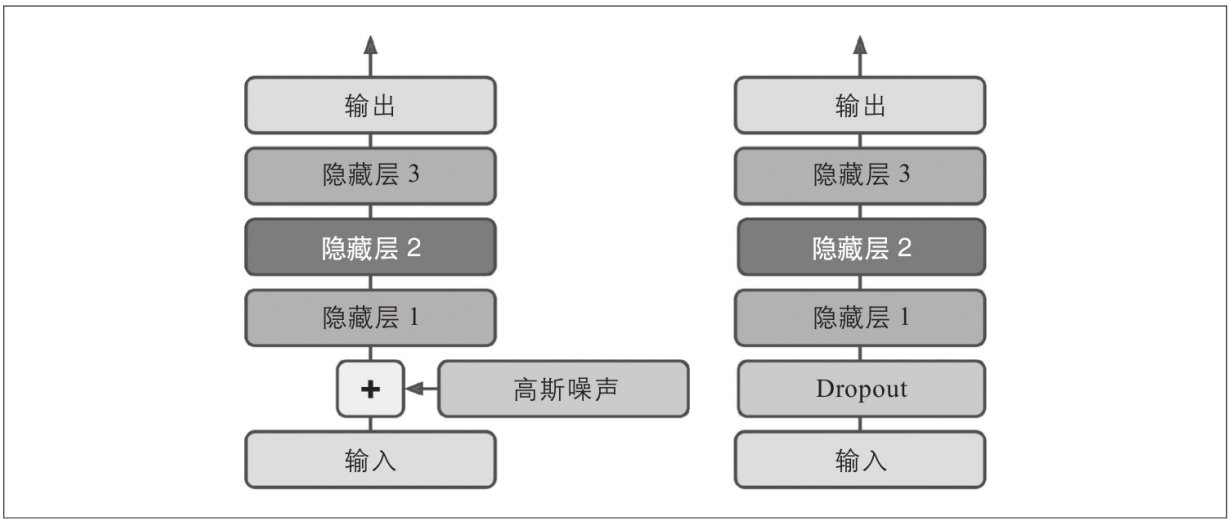


图17-8：具有高斯噪声（左）或dropout（右）的去噪自动编码器

实现很简单直接：这是一个常规的堆叠式自动编码器，在编码器的输入中附加了一个Dropout层（或者你可以改用GaussianNoise层）。回想一下，Dropout层仅在训练期间处于激活状态（GaussianNoise层也是如此）：

```
dropout_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(100, activation="selu"),
```

```
    keras.layers.Dense(30, activation="selu")
])
dropout_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
dropout_ae = keras.models.Sequential([dropout_encoder, dropout_decoder])
```

图17-9显示了一些有噪声的图像（其中一半像素关闭了），以及基于dropout的去噪自动编码器重建的图像。请注意自动编码器是如何猜测实际上不在输入中的详细信息，例如白衬衫的上部（下面一行，第四幅图像）。如你所见，与到目前为止我们讨论过的其他自动编码器一样，去噪自动编码器不仅可以用于数据可视化或无监督预训练，而且还可以非常简单有效地用于图像中的噪声去除。

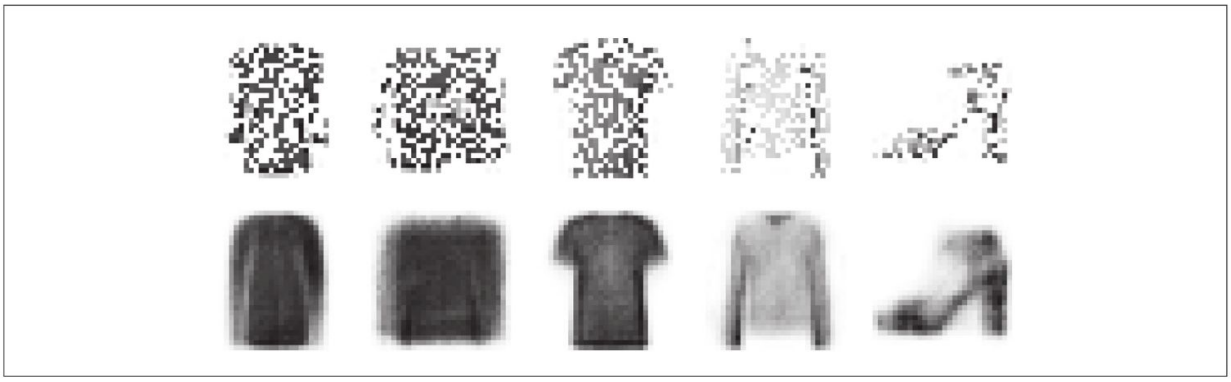


图17-9：有噪声的图像（上部）及其重构（下部）

[1] Pascal Vincent et al.，“Extracting and Composing Robust Features with Denoising Autoencoders”，Proceedings of the 25th International Conference on Machine Learning（2008）：1096–1103.

[2] Pascal Vincent et al.，“Stacked Denoising Autoencoders：Learning Useful Representations in a Deep Network with a Local Denoising Criterion”，Journal of Machine Learning Research 11（2010）：3371–3408.

17.7 稀疏自动编码器

另外一种会导致良好特征提取的约束是稀疏性：通过在成本函数中添加适当的函数项，强迫自动编码器减少编码层中活动神经元的数量。例如，可以强迫其在编码层中平均仅有5%的显著活动神经元。这迫使自动编码器将每个输入表示为少量活动神经元的组合。结果，编码层中的每个神经元最终会代表一个有用的特征（如果你每个月只能说几个单词，你可能会尝试使它们值得聆听）。

一种简单的方法是在编码层中使用sigmoid激活函数（将编码限制为0到1之间的值），使用较大的编码层（例如有300个神经元），并向编码层的激活添加一些ℓ_1正则化（解码器只是常规解码器）：

```
sparse_l1_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid"),
    keras.layers.ActivityRegularization(l1=1e-3)
])
sparse_l1_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_l1_ae = keras.models.Sequential([sparse_l1_encoder, sparse_l1_decoder])
```

此ActivityRegularization层只返回其输入，但作为副作用，它添加了等于其输入的绝对值之和的训练损失（该层仅在训练期间起作用）。同样，你可以删除ActivityRegularization层，并在上一层中设置activity_regularizer=keras.regularizers.l1（1e-3）。这个惩罚会鼓励神经网络产生接近于0的编码，但是如果它不能正确地重构输入，由于也会受到惩罚，因此它不得不输出至少一些非零值。使用ℓ_1规范而不是ℓ_2规范会迫使神经网络保留最重要的编码，同时消除输入图像不需要的编码（而不仅仅是减少所有编码）。

经常会产生更好结果的另一种方法是在每次训练迭代时测量编码层的实际稀疏度，并在测量的稀疏度与目标稀疏度不同时对模型进行惩罚。我们通过在整个训练批次中计算编码层中每个神经元的平均激活来实现。批次大小不能太小，否则平均值会不准确。

一旦我们获得了每个神经元的平均激活，便希望通过向成本函数添加稀疏损失来惩罚过于活跃或不够活跃的神经元。例如，如果我们测量一个神经元的平均激活为0.3，但目标稀疏度为0.1，则必须对其进行惩罚来降低激活。一种方法是简单地将平方误差（0.3-0.1）^{2}加到成本函数中，但是在实践中，更好的方法是使用Kullback-Leibler（KL）散度（在第4章中进行了简要讨论），它的梯度要比均方误差大得多，如图17-10所示。

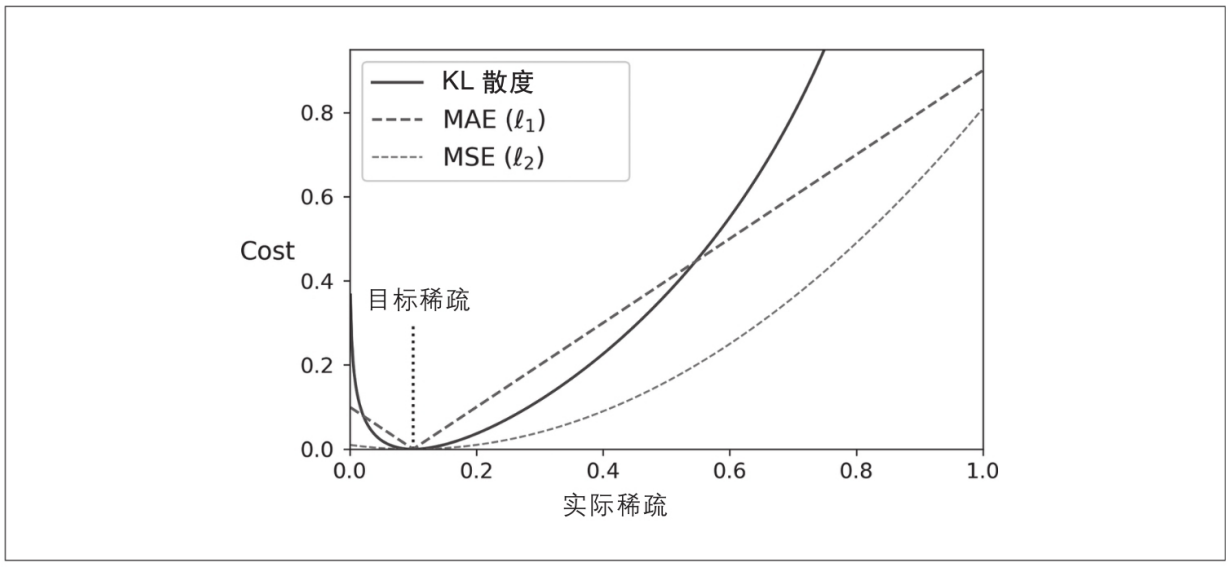


图17-10：稀疏损失

给定两个离散的概率分布P和Q，可以使用公式17-1计算这些分布之间的KL散度，记为D_{KL}（P||Q）。

公式17-1：Kullback－Leibler散度

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

在我们的示例中，我们要测量编码层中一个神经元要激活的目标概率p与实际概率q（即训练批次的平均激活）之间的散度。因此，KL散度简化为公式17-2。

公式17-2：目标稀疏度p与实际稀疏度q之间的KL散度

$$D_{\mathrm{KL}}(p \parallel q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}$$

一旦我们计算了编码层中每个神经元的稀疏损失，就把这些损失相加并将结果加到成本函数中。为了控制稀疏损失和重建损失的相对重要性，我们可以将稀疏损失乘以稀疏权重超参数。如果此权重过高，则模型会接近目标稀疏度，但可能无法正确重构输入，从而使模型没什么用处。相反，如果它太低，则该模型会忽略稀疏性目标，不会学习任何有趣的特征。

现在我们拥有了实现基于KL散度的稀疏自动编码器所需的一切。首先，让我们创建一个自定义正则化来应用KL散度正则化：

```
K = keras.backend
kl_divergence = keras.losses.kullback_leibler_divergence

class KLDivergenceRegularizer(keras.regularizers.Regularizer):
    def __init__(self, weight, target=0.1):
        self.weight = weight
        self.target = target
    def __call__(self, inputs):
        mean_activities = K.mean(inputs, axis=0)
        return self.weight * (
            kl_divergence(self.target, mean_activities) +
            kl_divergence(1. - self.target, 1. - mean_activities))
```

现在，我们可以构建这个稀疏自动编码器，使用KLDivergenceRegularizer来进行编码层的激活：

```
kld_reg = KLDivergenceRegularizer(weight=0.05, target=0.1)
sparse_kl_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid", activity_regular-izer=kld_reg)
])
sparse_kl_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_kl_ae = keras.models.Sequential([sparse_kl_encoder, sparse_kl_ decoder])
```

在Fashion MNIST上训练了这个稀疏自动编码器后，编码层中神经元的激活大部分接近0（所有激活中的大约70%低于0.1），如图17-11所示，所有神经元的平均激活约为0.1（所有神经元中约90%的平均激活在0.1到0.2之间）。

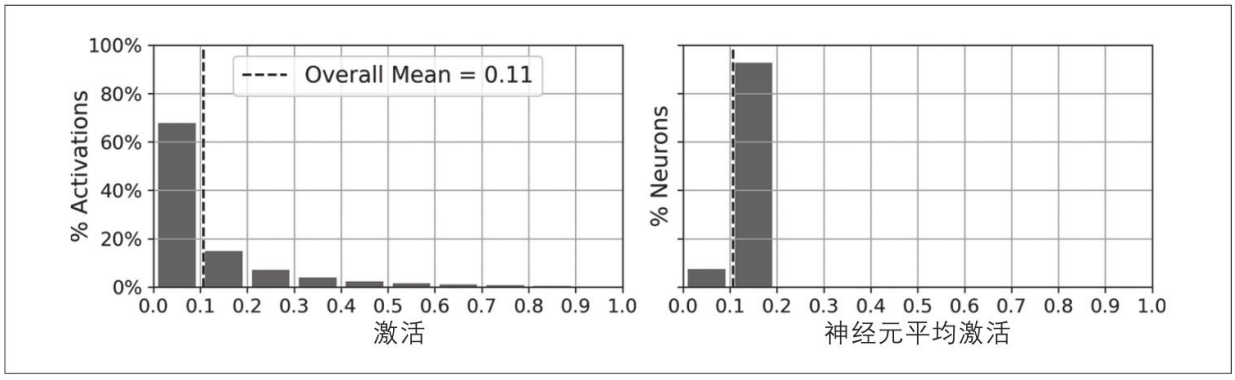


图17-11：编码层中所有激活的分布（左）和每个神经元的平均激活分布（右）

17.8 变分自动编码器

Diederik Kingma和Max Welling于2013年推出了自动编码器的另一个重要类别，并迅速成为最受欢迎的自动编码器类型之一：变分自动编码器[1]。

它们与到目前为止我们讨论过的所有自动编码器有很大的不同，它们具有以下特殊的地方：

- 它们是概率自动编码器，这意味着即使在训练后，它们的输出会部分由概率决定（与仅在训练期间使用随机性的去噪自动编码器相反）。

- 最重要的是，它们是生成式自动编码器，这意味着它们可以生成看起来像是从训练集中采样的新实例。

这两个属性使它们与RBM相当类似，但它们更易于训练，并且采样过程要快得多（使用RBM，你需要等待网络稳定到“热平衡”，然后才能采样新实例）。确实，顾名思义，变分自动编码器执行变分贝叶斯推理（在第9章中介绍），这是执行近似贝叶斯推理的有效方法。

让我们看一下它们是如何工作的。图17-12（左）显示了一种变分自动编码器。你可以看到所有自动编码器的基本结构，其中编码器后面是解码器（在此示例中，它们都具有两个隐藏层），但是有一个不同之处：不是直接为给定输入生成编码，而是编码器产生平均编码 μ 和标准差 σ 。然后实际编码是从均值 μ 和标准差 σ 的高斯分布中随机采样的。之后，解码器正常解码采样到的编码。图17-12的右侧部分显示了通过此自动编码器的一个训练实例。首先，编码器产生 μ 和 σ ，然后编码被随机采样（注意它实际上不位于 μ 处），最后对该编码进行解码。最终的输出和训练实例类似。

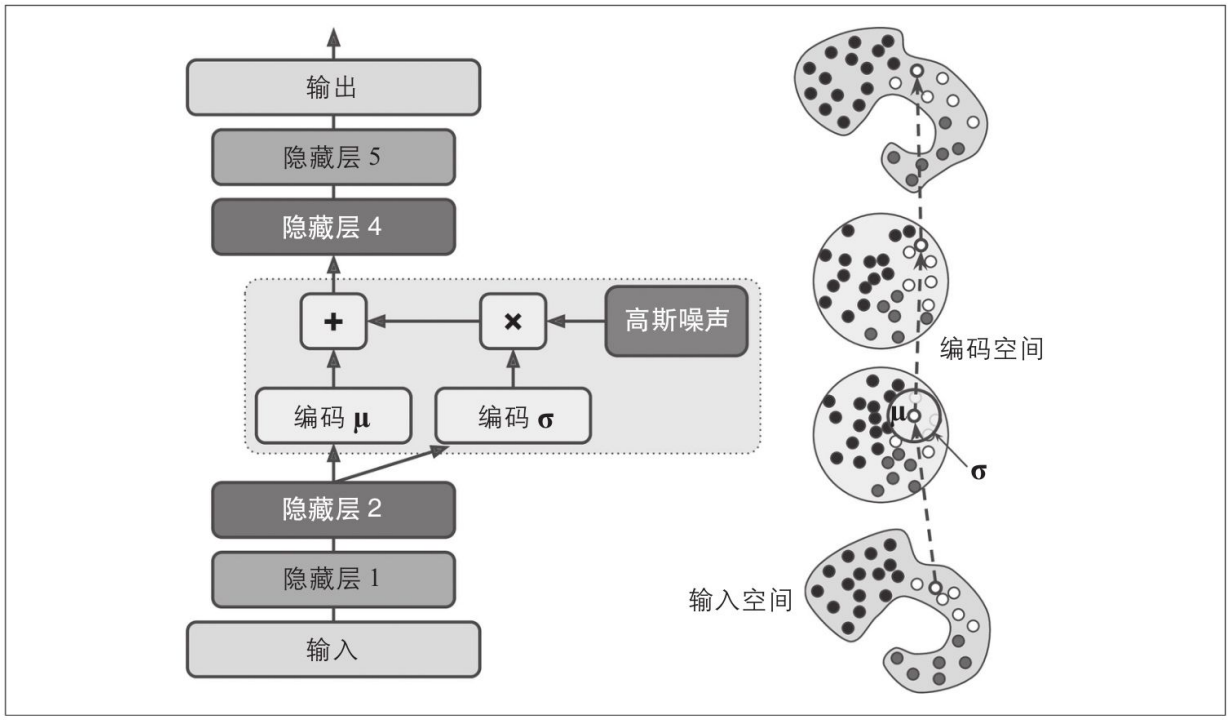


图17-12：变分自动编码器（左）和通过它的一个实例（右）

从图中可以看出，尽管输入的分布可能极端复杂，但变分自动编码器会产生看起来像是从简单的高斯分布[2]中采样的编码：在训练过程中，成本函数（下面讨论）迫使编码逐渐地在编码空间（也称为潜在空间）内移动，最终看起来像高斯点云。一个很好的结果是，在训练了变分自动编码器之后，你可以非常轻松地生成一个新实例：只需从高斯分布中采样一个随机编码，对其进行解码，然后就伪造出来了！

现在，让我们看一下成本函数。它由两部分组成。第一个是通常的重构损失，它会迫使自动编码器重现其输入（如前所述，我们可以使用交叉熵）。第二个是潜在损失，它使自动编码器的编码看起来像是从简单的高斯分布中采样得到的：它是目标分布（高斯分布）和编码的实际分布之间的KL散度。在数学上比稀疏自动编码器要复杂一些，特别是由于高斯噪声，它限制了可以传输到编码层的信息量（因此迫使自动编码器学习有用的特征）。幸运的是方程简化了，因此使用方程17-3[3]，可以容易地计算出潜在损失。

公式17-3：变分自编码器的潜在损失

$$\mathscr{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + \log(\sigma_i^2) - \sigma_i^2 - \mu_i^2$$

在这个等式中，L是潜在损失，n是编码的维度，μ_i和σ_i是编码中第i个分量的均值和标准差。向量μ和σ（包含所有μ_i和σ_i）由编码器输出，如图17-12（左）所示。

变分自动编码器架构的常见调整是使编码器输出γ=log（σ^2）而不是σ。然后可以如公式17-4所示计算潜在损失。这种方法在数值上更稳定，而且可以加快训练速度。

公式17-4：变分自编码器的潜在损失，用γ=log（σ^2）重写

$$\mathscr{L} = -\frac{1}{2}\sum_{i-1}^{K} 1 + \gamma_i - \exp(\gamma_i) - \mu_i^2$$

让我们开始为Fashion MNIST构建一个变分自动编码器（如图17-12所示，但使用γ调整）。首先，给定μ和γ，我们需要一个自定义层来采样编码：

```
class Sampling(keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return K.random_normal(tf.shape(log_var)) * K.exp(log_var / 2) + mean
```

该Sampling层接受两个输入：mean（μ）和log_var（γ），它使用函数K.random_normal（）从正态分布中采样一个均值0和标准差为1的随机向量（与γ形状相同），然后将其乘以exp（γ/2）（等于σ，你可以验证），最后将μ加起来并返回结果。该方法从均值μ和标准差σ的正态分布中采样一个编码向量。

接下来，我们可以使用函数式API来创建编码器，因为模型不是完全顺序的：

```
codings_size = 10

inputs = keras.layers.Input(shape=[28, 28])
z = keras.layers.Flatten()(inputs)
z = keras.layers.Dense(150, activation="selu")(z)
z = keras.layers.Dense(100, activation="selu")(z)
codings_mean = keras.layers.Dense(codings_size)(z) # μ
codings_log_var = keras.layers.Dense(codings_size)(z) # γ
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = keras.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])
```

注意，输出codings_mean（μ）和codings_log_var（γ）的Dense层具有相同的输入（第二个Dense层的输出）。然后我们将codings_mean和codings_log_var都传递给Sampling层。最后，如果你想检查codings_mean和codings_log_var的值，variational_encoder模型具有三个输出。我们要使用的唯一输出是最后一个（codings）。现在让我们构建解码器：

```
decoder_inputs = keras.layers.Input(shape=[codings_size])
x = keras.layers.Dense(100, activation="selu")(decoder_inputs)
x = keras.layers.Dense(150, activation="selu")(x)
x = keras.layers.Dense(28 * 28, activation="sigmoid")(x)
outputs = keras.layers.Reshape([28, 28])(x)
variational_decoder = keras.Model(inputs=[decoder_inputs], outputs=  [outputs])
```

对于此解码器，我们可以使用顺序API而不是函数式API，因为它实际上只是一个简单的层堆栈，实际上与我们到目前为止构建的许多解码器相同。最后我们建立变分自动编码器模型：

```
_, _, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = keras.Model(inputs=[inputs], outputs=[recon-structions])
```

请注意，我们忽略了编码器的前两个输出（我们只想将编码馈送到解码器）。最后，我们必须加上潜在损失和重构损失：

```
latent_loss = -0.5 * K.sum(
    1 + codings_log_var - K.exp(codings_log_var) - K.square(codings_mean),
    axis=-1)
variational_ae.add_loss(K.mean(latent_loss) / 784.)
variational_ae.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

我们首先应用公式17-4来计算该批次中每个实例的潜在损失（在最后一个轴上求和）。然后，我们计算该批次中所有实例的平均损失，将结果除以784，以确保它与重构损失相比有合适的比例标度。确实，变分自编码器的重建损失应该是像素重建误差的总和，但是当Keras计算"binary_crossentropy"损失时，它计算所有784个像素的均值，而不是总和。因此重构损失比我们需要的少784倍。我们可以定义一个自定义损失来计算总和而不是平均值，但是把潜在损失除以784更为简单（最终损失要比其应该的小784倍，但这只是意味着我们要使用更大一点的学习率）。

请注意，我们使用RMSprop优化器，该优化器在这个示例下效果很好。最后我们可以训练自动编码器！

```
history = variational_ae.fit(X_train, X_train, epochs=50, batch_size=128,
                             validation_data=[X_valid, X_valid])
```

生成Fashion MNIST图像

现在，让我们使用这种变分自动编码器来生成看起来像时尚物品的图像。我们需要做的就是从高斯分布中采样随机编码并对它们进行解码：

```
codings = tf.random.normal(shape=[12, codings_size])
images = variational_decoder(codings).numpy()
```

图17-13显示了12个生成的图像。

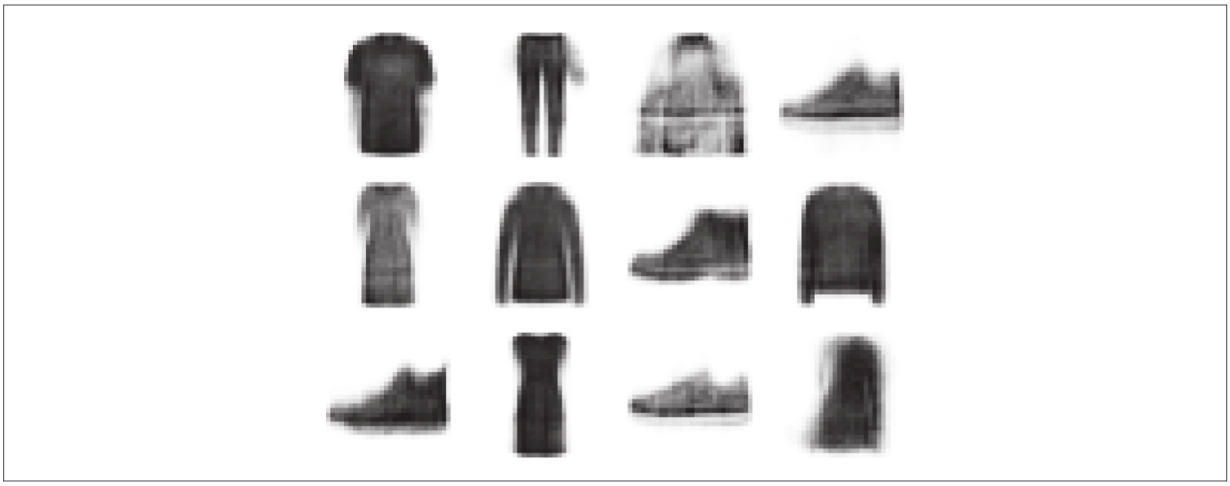


图17-13：变分自动编码器生成的Fashion MNIST图像

这些图像中的大多数看上去令人信服，即使有些过于模糊。其余的效果都不是很好，但是不要对自动编码器太苛刻，它只学习了几分钟！多给它一些微调和训练时间，这些图像应该看起来会更好。

可变自动编码器使语义插值成为可能：我们可以在编码级别进行插值，而不是在像素级别插值两个图像（看起来好像两个图像被叠加了一样）。我们首先让两个图像通过编码器，然后对获得的两个编码进行插值，最后对插值的编码进行解码来获得最终图像。它看起来像是常规的Fashion MNIST图像，但它是原始图像之间的中间图像。在下面的代码示例中，我们采用刚刚生成的12个编码，将它们组织在3×4网格中，然后使用TensorFlow的tf.image.resize（）函数将该网格的大小调整为5×7。默认情况下，resize（）函数会执行双线性插值，因此每隔一行和一列会包含插值编码。然后，我们使用解码器生成所有图像：

```
codings_grid = tf.reshape(codings, [1, 3, 4, codings_size])
larger_grid = tf.image.resize(codings_grid, size=[5, 7])
interpolated_codings = tf.reshape(larger_grid, [-1, codings_size])
images = variational_decoder(interpolated_codings).numpy()
```

图17-14显示了生成的图像。原始图像被加了一个框，其余图像是附近图像之间的语义插值的结果。请注意，例如第4行和第5列中的鞋子是位于其上方和下方的两双鞋子之间的很好插值。

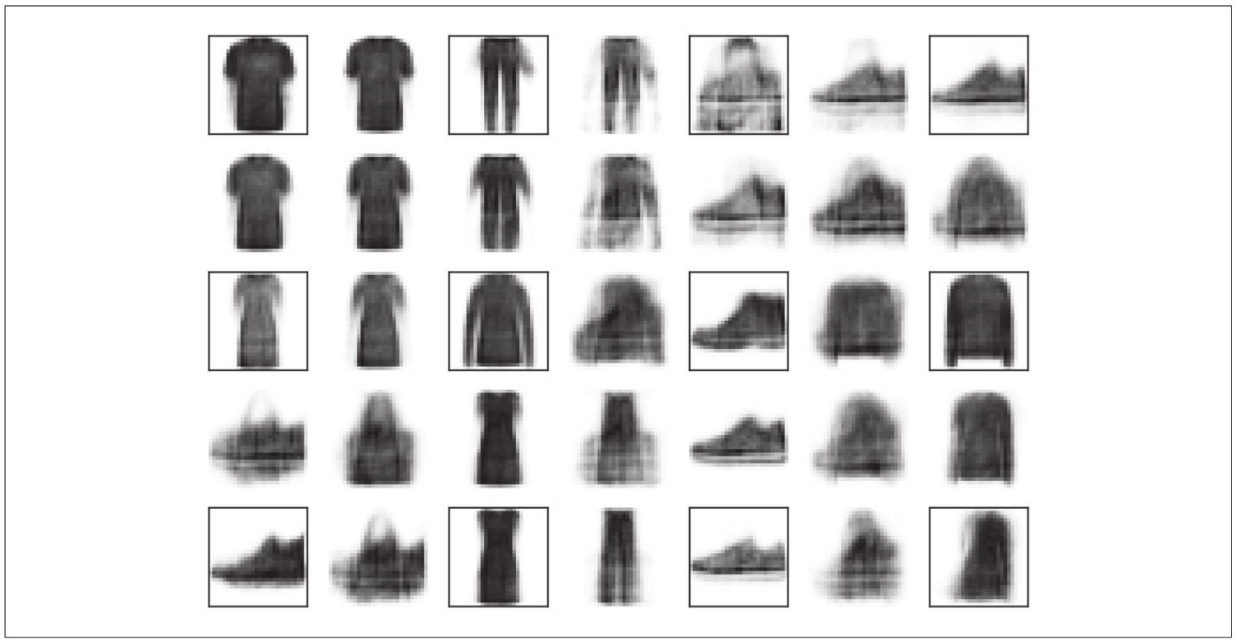


图17-14：语义插值

几年来，变分自动编码器非常流行，但是GAN最终占了领先位置，是因为它们能够生成更加逼真的图像。因此，让我们将注意力转向GAN。

[1] Diederik Kingma and Max Welling，“Auto-Encoding Variational Bayes”，arXiv preprint arXiv：1312.6114（2013）.
[2] 变分自动编码器实际上更通用。编码不限于高斯分布。
[3] 有关更多数学细节，请查看有关变分自动编码器的原始论文，或Carl Doersch的精彩教程（2016）。

17.9 生成式对抗网络

Ian Goodfellow等人在2014年的论文[1]中提出了生成式对抗网络，尽管这个想法几乎立刻使研究人员们兴奋不已，但还是花了几年时间才克服了训练GAN的一些困难。就像许多伟大的想法一样，事后看起来似乎很简单：让神经网络相互竞争，希望这种竞争能够促使它们变得更好。如图17-15所示，GAN由两个神经网络组成：

生成器

以随机分布作为输入（通常是高斯分布），并输出一些数据（通常是图像）。你可以将随机输入视为要生成的图像的潜在表征（即编码）。因此你可以看到，生成器提供的功能与变分自动编码器中的解码器相同，并且可以使用相同的方式来生成新图像（只需馈入一些高斯噪声，就会输出一个新图片）。但是我们很快就会看到，它的训练方式大不相同。

判别器

输入从生成器得到的伪图像或从训练集中得到的真实图像，并且必须猜测输入图像是伪图像还是真实图像。

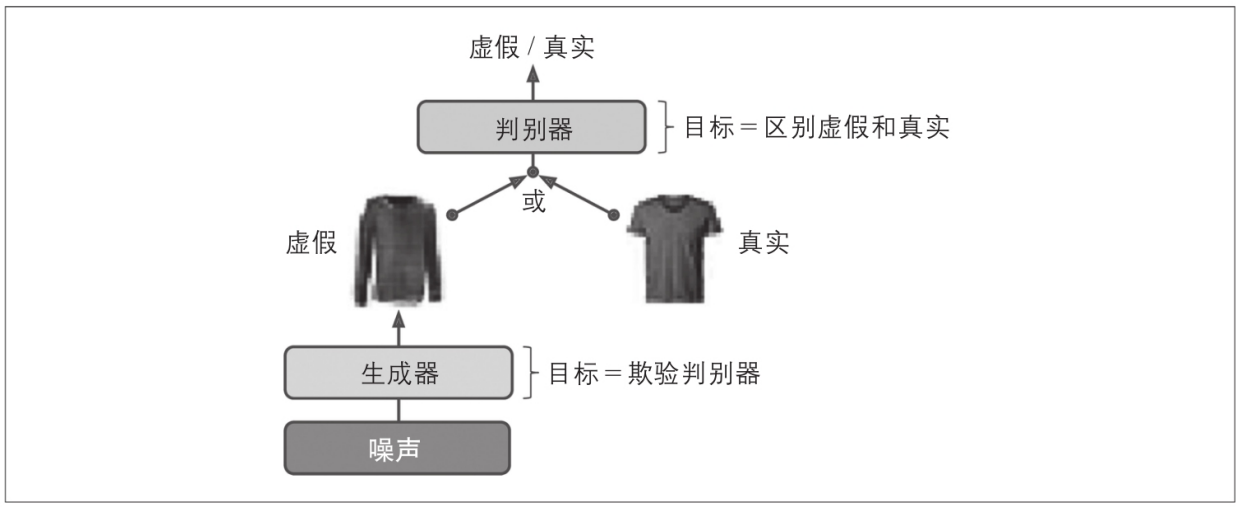


图17-15：一个生成式对抗网络

在训练过程中，生成器和判别器有相反的目标：判别器试图从真实图像中分辨出虚假图像，而生成器则试图产生看起来足够真实的图像来欺骗判别器。由于GAN由不同目标的两个网络组成，因此无法像常规神经网络一样对其进行训练。每个训练迭代都分为两个阶段：

• 在第一阶段，我们训练判别器。从训练集中采样一批真实图像，再加上用生成器生成的相等数量的伪图像组成训练批次。对于伪图像，将标签设置为0；对于真实图像，将标签设置为1，并使用二元交叉熵损失在该被标签的批次上对判别器进行训练。重要的是，在这个阶段反向传播只能优化判别器的权重。

• 在第二阶段，我们训练生成器。首先使用它来生成另一批伪图像，然后再次使用判别器来判断图像是伪图像还是真实图像。在这个批次中我们不添加真实图像，并且所有标签都设置为1（真实）：换句话说，我们希望生成器能产生判别器会（错误地）认为是真实的图像！至关重要的是，在此步骤中，判别器的权重会被固定，因此反向传播只会影响生成器的权重。

 生成器实际上从未看到过任何真实的图像，但是它逐渐学会产生令人信服的伪图像！它所得到的是流经判别器的回流梯度。幸运的是，判别器越好，这些二手梯度中包含的真实图像信息就越多，因此生成器可以取得很大进步。让我们继续前进，给Fashion MNIST构建一个简单的GAN。

首先，我们需要构建生成器和判别器。生成器类似于自动编码器的解码器，判别器是常规的二元分类器（它将图像作为输入，包含单个神经元和使用sigmoid激活函数的Dense层）。对于每个训练迭代的第二阶段，我们还需要一个完整的GAN模型，其中包含生成器，后面跟随一个判别器：

```
codings_size = 30

generator = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[codings_size]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
discriminator = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])
```

接下来，我们需要编译这些模型。由于判别器是二元分类器，我们自然可以使用二元交叉熵损失。生成器仅通过gan模型进行训练，因此我们根本不需要对其进行编译。gan模型也是二元分类器，因此它可以使用二元交叉熵损失。重要的是，判别器不应该在第二阶段进行训练，因此在编译gan模型之前，我们将其设为不可训练：

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

 Keras仅在编译模型时才考虑可训练属性，因此在运行此代码后，如果我们调用其fit（）方法或其train_on_batch（）方法（我们将使用），则判别器是可训练的；当我们在gan模型上调用这些方法时，则判别器是不可训练的。

由于训练循环不寻常，因此我们不能使用常规的fit（）方法。相反，我们要编写一个自定义训练循环。为此，我们首先需要创建一个数据集来遍历图像：

```
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

现在我们准备编写训练循环。让我们将其包装在train_gan（）函数中：

```
def train_gan(gan, dataset, batch_size, codings_size, n_epochs=50):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        for X_batch in dataset:
            # phase 1 - training the discriminator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            generated_images = generator(noise)
            X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
            y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
            discriminator.trainable = True
            discriminator.train_on_batch(X_fake_and_real, y1)
            # phase 2 - training the generator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            y2 = tf.constant([[1.]] * batch_size)
            discriminator.trainable = False
            gan.train_on_batch(noise, y2)

train_gan(gan, dataset, batch_size, codings_size)
```

如前所述，你可以在每次迭代中看到两个阶段：

• 在第一阶段，我们将高斯噪声馈送到生成器以生成伪图像，然后通过合并相等数量的真实图像来组成这一批次。对于伪图像，目标y1被设置为0；对于真实图像，目标y1被设置为1。然后我们在这个批次上进行判别器训练。注意我们将判别器的可训练属性设置为T r u e：这只是为了消除Keras注意到在编译模型时可训练属性为True，但是现在为False时显示的警告（反之亦然）。

• 在第二阶段，我们向GAN馈入一些高斯噪声。它的生成器首先会生成伪图像，然后判别器会尝试猜测这些图像是伪图像还是真实图像。我们希望判别器相信伪图像是真实的，因此将目标y2设置为1。请注意，我们再次将可训练属性设置为F a ls e，以避免再次发出警告。

这就是所有的内容！如果显示生成的图像（见图17-16），则在第一个轮次结束时可以看到它们已经开始看起来像（有很多噪声）Fashion MNIST图像。

图17-16：经过一个轮次训练后GAN生成的图像

不幸的是，图像不会比这更好了，而且你甚至可能会发现在一些轮次中，GAN似乎忘记了所学到的内容。这是为什么？事实证明训练GAN可能很有挑战性。让我们看看为什么。

17.9.1 GAN的训练难点

在训练过程中，生成器和判别器在零和游戏中不断地试图超越彼此。随着训练的进行，游戏可能会在一种博弈论者称为纳什均衡的状态中结束，这种均衡以数学家约翰·纳什（John Nash）的名字命名：在这种情况下，假设其他玩家都没有改变他们的策略，那么没有人会改变自己的策略使自己更好。例如，当每个人都在道路左侧行驶时，达到了纳什均衡：没有任何一个驾驶员会成为唯一切换道路的人而变得更好。当然，还有第二种可能的纳什均衡：当每个人都在道路的右侧行驶时。不同的初始状态和动态发展可能会导致一个平衡或另一个平衡。在此示例中，一旦达到平衡（即与其他所有人在同一侧行驶），就存在一个最

佳策略，但是一个纳什均衡可能涉及多种竞争策略（例如，捕食者追捕猎物，猎物试图逃避，改变它们的策略也不会变得更好）。

那么这如何适用于GAN呢？论文的作者证明了GAN只能达到单个纳什均衡：当生成器产生完美逼真的图像时，判别器只能被迫猜测（50%真实，50%伪造）。这个事实非常令人鼓舞：你似乎只需要训练GAN足够长的时间，它就会最终达到均衡，从而给你一个完美的生成器。不幸的是，并不是那么简单：没有任何东西可以保证达到均衡。

最大的困难被称为模式崩溃：就是当生成器的输出逐渐变得不太多样化时。这是怎么发生的？假设生成器在产生逼真的鞋子方面比其他任何类都更好。它就会用鞋子来更多地欺骗判别器，这会鼓励它生成更多的鞋子图像。逐渐地它会忘记如何产生其他任何东西。同时判别器看到的唯一伪造图像将是鞋子，因此它也会忘记如何辨别其他类别的伪造图像。最终当判别器在进行假鞋与真鞋区分时，生成器将被迫转移到另一类。这样一来，它可能会变得擅长于生成衬衫，而忘记了鞋子，然后判别器也会跟着生成器。GAN可能会逐渐在几个类别中循环，而不会擅长于生成任何一个类别。

此外由于发生器和判别器不断地相互竞争，因此它们的参数可能最终会振荡并变得不稳定。训练开始时可能会很好，然后由于这些不稳定而突然发散，没有明显的原因。而且有许多因素会影响这些复杂的动态过程，因此GAN对超参数非常敏感：你可能不得不花费大量的精力来微调它们。

自2014年以来，研究人员忙于解决这些问题：针对该问题发表了许多论文，其中一些提出了新的成本函数[2]（尽管Google研究人员在2018年发表了一篇论文[3]质疑其效率）或技术来解决训练稳定性或避免模式崩溃的问题。例如，一种称为重播体验的流行技术包括将生成器在每次迭代中生成的图像存储在重播缓冲区中（逐渐删除较早生成的图像），使用真实图像以及从该缓冲区取出的伪图像来训练判别器（而不

是由当前生成器生成的伪图像）。这减少了判别器过拟合最新生成器输出的图像的机会。另一种常见的技术称为小批量判别：它可测量跨批次中相似图像的程度，并将此统计信息提供给判别器，因此判别器可以轻松拒绝缺乏多样性的一整个批次的伪图像。这会鼓励生成器生成更多样性的图像，从而减少模式崩溃。其他论文只是提出了一些表现良好的特定网络架构。

简而言之，这仍然是一个非常活跃的研究领域，并且对GAN的动态过程还没有完全了解。但是，好消息是已经取得了巨大的进步，其中一些结果确实令人震惊！让我们看一些最成功的架构，首先是深度卷积GAN，这是几年前的最新技术。然后，我们将研究两种最新（更复杂）的架构。

17.9.2 深度卷积GAN

2014年的GAN原始论文试验了卷积层，但是只是生成了小图像。不久之后，许多研究人员试图基于更深的卷积网络为更大的图像构建GAN。由于训练非常不稳定，所以被证明是棘手的，但是Alec Radford等人在实验了许多不同的架构和超参数之后，终于在2015年末取得了成功。他们称其架构为深度卷积GAN（DCGAN）[4]。以下是他们为构建稳定的卷积GAN提出的主要指导：

- 用跨步卷积（在判别器中）和转置卷积（在生成器中）替换所有池化层。

- 除生成器的输出层和判别器的输入层外，在生成器和判别器中都使用批量归一化。

- 删除全连接的隐藏层以获得更深的架构。

- 对生成器中的所有层使用ReLU激活函数，除了输出层应该使用tanh。

• 对判别器中的所有层使用leaky ReLU激活函数。

这些准则在许多情况下都会起作用，但并非总是如此，因此你可能需要试验不同的超参数（实际上，仅仅更改随机种子并再次训练相同的模型有时会起作用）。例如，这是一个小型DCGAN，在Fashion MNIST数据集上可以很好地工作：

```
codings_size = 100

generator = keras.models.Sequential([
    keras.layers.Dense(7 * 7 * 128, input_shape=[codings_size]),
    keras.layers.Reshape([7, 7, 128]),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2, padding= "same",
                                 activation="selu"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2, padding= "same",
                                  activation="tanh")
])
discriminator = keras.models.Sequential([
    keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same",
                        activation=keras.layers.LeakyReLU(0.2),
                        input_shape=[28, 28, 1]),
    keras.layers.Dropout(0.4),
    keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same",
                        activation=keras.layers.LeakyReLU(0.2)),
    keras.layers.Dropout(0.4),
    keras.layers.Flatten(),
    keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])
```

生成器使用大小为100的编码，将其投影到6272维度（7×7×128），对结果进行重构以获得7×7×128张量。该张量被批量归一化后，馈入步幅为2的转置卷积层，将其从7×7上采样至14×14，将深度从128减小至64。其结果再次被批量归一化，并馈入另一个步幅为2的转置卷积层，将其从14×14上采样到28×28，将深度从64减小到1。该层使用tanh激活函数，因此输出范围为-1到1。因此在训练GAN之前，我们需要将训练集重新按照比例调整为相同的范围。还需要重构形状来添加通道维度：

```
X_train = X_train.reshape(-1, 28, 28, 1) * 2. - 1. # reshape and rescale
```

判别器看起来很像是用于二元分类的常规CNN，除了不是使用最大池化层对图像进行下采样，而是使用跨步卷积（strides=2）。请注意，我们使用了leaky ReLU激活函数。

总的来说，我们尊重DCGAN准则，除了我们用Dropout层替换了判别器中的BatchNormalization层（否则这种情况下训练不稳定），并且在生成器中将ReLU替换为SELU。你可以自由调整此架构：你会看到它对超参数（特别是两个网络的相对学习率）有多敏感。

最后，构建数据集，然后编译和训练该模型，我们使用与之前完全相同的代码。经过50轮训练后，生成器将生成如图17-17所示的图像。它仍然不完美，但是其中许多图像都令人信服。

图17-17：经过50轮训练后，DCGAN生成的图像

你如果扩展此架构并在大型的人脸数据集上训练，则可以获得相当逼真的图像。实际上，DCGAN可以学习非常有意义的潜在表征，如图17-18所示：生成了许多图像，手动选择其中的9个图像（左上），包括三个戴着眼镜的男人、三个不戴眼镜的男人、三个不戴眼镜的女人。对于这些类别中的每一类，对用于生成图像的编码进行平均，然后根据所得的平均编码生成了图像（左下）。简而言之，三个左下图像均代表位于

其上方的三个图像的均值。但这不是在像素级别计算的简单均值（这会导致三个重叠的人脸），而是在潜在空间中计算的均值，因此图像看起来仍然像正常人脸。令人惊讶的是，如果你计算戴眼镜的男人，减去不戴眼镜的男人，再加上不戴眼镜的女人，其中每个项对应于其中一个均值编码，然后生成与该编码相对应的图像，则该图像位于右边的3×3网格的中心：一个戴着眼镜的女人！它周围的其他8幅图像是根据相同的向量加上一点噪声生成的，这说明了DCGAN的语义插值能力。能够在脸上进行算术运算感觉就像是科幻小说！

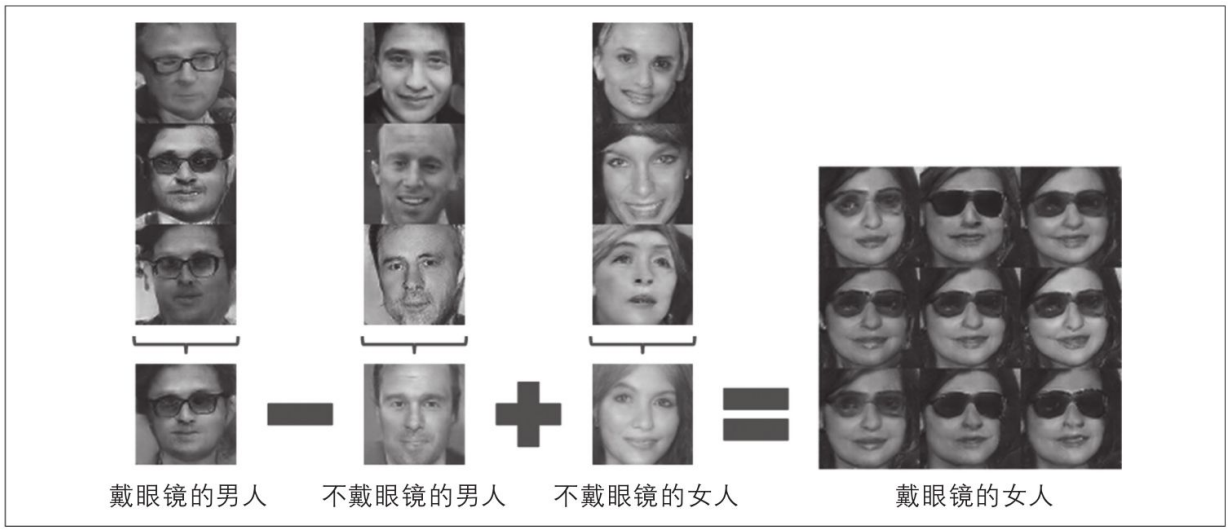


图17-18：视觉概念的向量算法（DCGAN论文图7的一部分）[5]

 如果将每个图像的类别作为生成器和判别器的额外输入，则它们都会学习每个类别看起来像什么，从而可以控制生成器生成的每个图像类别。这称为条件GAN（CGAN）[6]。

不过DCGAN并不完美。例如，当你尝试使用DCGAN来生成非常大的图像时，通常会遇到局部令人信服的特征，但总体上会出现不一致（例如，一个袖子的衬衫长于另一个袖子的衬衫）。该如何解决呢？

17.9.3　GAN的逐步增长

Nvidia的研究人员Tero Karras等人在2018年的一篇论文[7]中提出了一项重要技术：他们建议在训练开始时生成小图像，然后向生成器和判别器中逐渐添加卷积层以生成越来越大的图像（4×4、8×8、16×16、512×512、1024×1024）。这种方法类似于堆叠式自动编码器的贪婪分层训练。额外的层添加在生成器的末尾和判别器的开始处，并且先前训练过的层仍然是可训练的。

例如，当生成器的输出从4×4增长到8×8（见图17-19）时，一个上采样层（使用最近邻滤波）被添加到现有的卷积层中，因此它输出8×8特征图，然后将其馈送到新的卷积层（使用"same"填充和1的步幅，因此其输出也是8×8）。这个新层之后是新的输出卷积层：这是内核大小为1的常规卷积层，它将输出向下投影到所需数量的颜色通道（例如3）。为了避免在添加新卷积层时破坏第一个卷积层的训练权重，最终输出是原始输出层（现在输出8×8的特征图）和新输出层的加权和。新输出的权重是α，而原始输出的权重是1-α，并且α从0缓慢增加到1。换句话说，新的卷积层（在图17-19中用虚线表示）逐渐增强，而原始输出层逐渐减弱。在新的卷积层添加到判别器时，使用类似的增强/减弱技术（随后是用于下采样的平均池化层）。

这篇论文还介绍了其他几种旨在增加输出多样性（避免模式崩溃）和使训练更稳定的技术：

小批次标准差层

在判别器末端附近添加。对于输入中的每个位置，它计算批次中所有通道和所有实例的标准差（S=tf.math.reduce_std（inputs，axis=[0，-1]））。然后，将这些标准差在所有点上取平均值，得到一个单一值（v=tf.reduce_mean（S））。最后，向批处理中的每个实例中添加一个额外的特征图，并用计算值填充（tf.concat（[inputs，tf.fill（[batch_size，height，width，1]，v）]，axis=-1））。这有什么帮助？好吧，如果生成器生成的图像变化不大，那么判别器中特征图上的标准差就会很小。多亏了这一层，判别器就可以轻松访问此统

计信息，从而减少了被生成器（生成很少的多样性）欺骗的可能性。这会鼓励生成器产生更多不同的输出，从而降低模式崩溃的风险。

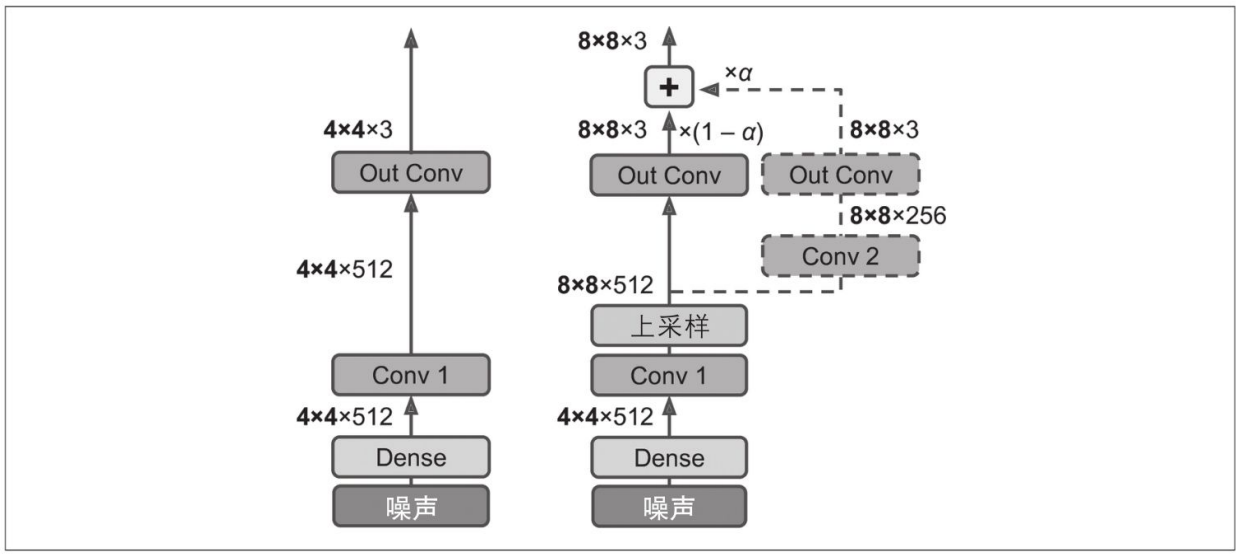


图17-19：渐进式GAN：GAN生成器输出4×4彩色图像（左）。我们扩展它以输出8×8图像（右）

均衡的学习率

使用均值为0和标准差为1的简单高斯分布而不是使用He初始化来初始化所有权重。但是，权重会在运行时（即每次执行层时）按与He初始化相同的系数进行缩小：权重除以$\sqrt{2/n_{\text{inputs}}}$，其中ninputs是层输入的数量。论文表明，当使用RMSProp、Adam或其他自适应梯度优化器时，该技术显著提高了GAN的性能。实际上，这些优化器通过估计的标准差对梯度更新进行归一化（见第11章），因此动态范围[8]较大的参数要花费较长的训练时间，而动态范围较小的参数可能更新得太快，从而导致不稳定。通过将权重调整作为模型本身的一部分，而不仅仅是在初始化时进行权重调整，这种方法可确保在整个训练过程中，所有参数的动态范围都相同，因此它们都以相同的速度学习。这加快和稳定了训练过程。

像素归一化层

在生成器中的每个卷积层之后添加。它基于同一图像中和同一位置但所有通道之间的所有激活对每个激活进行归一化（除以均方激活的平方根）。在TensorFlow代码中，这是inputs/tf.sqrt（tf.reduce_mean（tf.square（X），axis=-1，keepdims=True）+1e-8）（需要平滑项1e-8来避免被零除）。该技术避免了由于生成器和判别器之间过度竞争而导致的激活爆炸。

所有这些技术的组合使作者能够生成令人信服的高清人脸图像。但是到底什么叫“令人信服”？评估是使用GAN的一大挑战：尽管可以自动评估所生成图像的多样性，但是判断图像的质量是一项更加棘手和主观的任务。一种技术是使用人工评估，但这昂贵且费时。因此作者建议考虑在每个比例尺度来测量生成图像的局部图像结构与训练图像之间的相似性。这个想法把他们引向了另一个突破性的创新：StyleGAN。

17.9.4 StyleGAN

相同的Nvidia团队在2018年发表的一篇论文[9]中提出了高分辨率图像生成的最新技术，该论文介绍了流行的StyleGAN架构。作者在生成器中使用了风格转换技术，以确保生成的图像在各个尺度上都具有与训练图像相同的局部结构，从而极大地提高了所生成的图像质量。判别器和损失函数没有被修改，仅仅修改了生成器。让我们看一下StyleGAN，它由两个网络组成（请参见图17-20）。

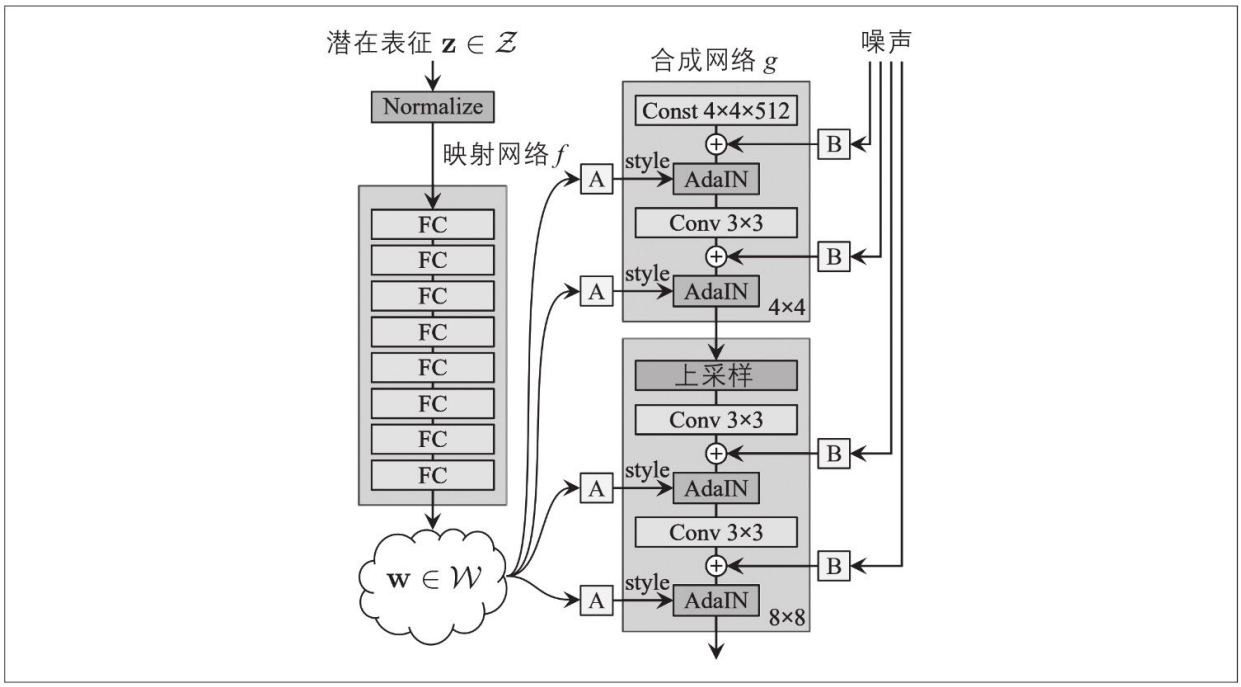


图17-20：StyleGAN的生成器架构（来自StyleGAN论文的图1的一部分）[10]

映射网络

一个8层的MLP把潜在表示z（即编码）映射到向量w。然后，通过多个仿射变换（没有激活函数的Dense层，在图17-20中用“A”框表示）发送此向量，从而生成多个向量。这些向量从细粒度的纹理（例如头发的颜色）到高级特征（例如成人或儿童）在不同级别上控制生成图像的风格。简而言之，映射网络将编码映射到多个风格向量。

合成网络

负责生成图像。它具有恒定的学习输入（需要明确的是，此输入在训练后将保持不变，但是在训练过程中，它会经过反向传播而不断调整）。如前所述，它通过多个卷积和上采样层处理此输入，但是有两处调整：首先，在卷积层的输入和所有输出中添加了一些噪声（在激活函数之前）。其次，每个噪声层后面是一个自适应实例归一化（AdaIN）层：它独立地归一化每个特征图（通过减去特征图的均值并除以其标准

差），然后使用风格向量确定每个特征图的比例和偏移量（风格向量为每个特征图包含一个比例和一个偏置项）。

独立于编码来增加噪声的想法非常重要。图像的某些部分非常随机，例如每个雀斑或头发的确切位置。在较早的GAN中，这种随机性要么来自编码，要么是生成器自身产生的一些伪随机噪声。如果它来自编码，则意味着生成器使用了编码的表征力的很大一部分来存储噪声：这非常浪费。而且，噪声必须能够流经网络并到达生成器的最后一层：这似乎是不必要的约束，可能会减慢训练速度。最后，可能会出现一些人工视觉，因为在不同层次使用了相同的噪声。相反，如果生成器试图产生自己的伪随机噪声，该噪声可能看起来不那么令人信服，从而导致出现更多人工视觉。另外，生成器的权重的一部分用于产生伪随机噪声，这似乎又是浪费的。通过增加额外的噪声输入，可以避免所有这些问题。GAN能够使用所提供的噪声为图像的每个部分添加适当数量的随机性。

每个级别增加的噪声都不相同。每个噪声输入由一个充满了高斯噪声的单个特征图组成，该噪声会广播到所有（给定级别的）的特征图，并在添加之前使用学习到的每个特征图的比例因子进行缩放（在图17-20中由“B”框表示）。

最后，StyleGAN使用一种称为混合正则化（或风格混合）的技术，使用两种不同的编码生成一定百分比的生成图像。具体来说，编码c_1和c_2通过映射网络发送，给出两个风格向量w_1和w_2。然后，合成网络基于第一个级别的风格w_1和其余级别的样式w_2生成图像。截断级别是随机选择的。这可以防止网络假设相邻级别的风格是相关联的，这反过来又鼓励了GAN中的局部性，意味着每个风格向量仅影响所生成图像中有限数量的特征。

GAN种类繁多，因此需要一本书才能涵盖它们。希望这个介绍给了你主要的思想，最重要的是能使你渴望了解更多。如果你在数学概念上有困难，那么有博客文章可以帮助你更好地理解它。然后继续实现你自

己的GAN，如果一开始学习困难，不要灰心。不幸的是，这是正常现象，需要很多耐心才能使它工作，但结果很值得。如果你在实现细节上苦苦挣扎，则可以查看许多Keras或TensorFlow的实现。实际上，如果你想要的只是快速获得惊人的结果，那么你可以使用预先训练的模型（例如Keras提供了预先训练的StyleGAN模型）。

在下一章中，我们将转到深度学习的一个完全不同的分支：深度强化学习。

[1] Ian Goodfellow et al.，“Generative Adversarial Nets”，Proceedings of the 27th International Conference on Neural Information Processing Systems 2（2014）：2672－2680.
[2] 为了更好地比较主要的GAN损失部分，请检查Hwalsuk Lee这个伟大的GitHub项目。
[3] Mario Lucic et al.，“Are GANs Created Equal?A Large-Scale Study”，Proceedings of the 32nd International Conference on Neural Information Processing Systems（2018）：698－707.
[4] Alec Radford et al.，“Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”，arXiv preprint arXiv：1511.06434（2015）.
[5] 经作者授权转载。
[6] Mehdi Mirza and Simon Osindero，“Conditional Generative Adversarial Nets”，arXiv preprint arXiv：1411.1784（2014）.
[7] Tero Karras et al.，“Progressive Growing of GANs for Improved Quality，Stability，and Variation”，Proceedings of the International Conference on Learning Representations（2018）.
[8] 变量的动态范围是它可能取的最大值和最小值之间的比率。
[9] Tero Karras et al.，“A Style-Based Generator Architecture for Generative Adversarial Networks”，arXiv preprint arXiv：1812.04948（2018）.
[10] 经作者授权转载。

17.10 练习题

1. 自动编码器的主要任务是什么？

2. 假设你要训练一个分类器，并且你有很多未标记的训练数据，但是只有几千个标记的实例。自动编码器怎么能够帮助你？你会如何进行？

3. 如果自动编码器能完美地重构输入，是否一定是一个好的自动编码器？如何评估自动编码器的性能？

4. 什么是不完整的自动编码器和完整的自动编码器？自动编码器过于不完整的主要风险是什么？自动编码器过于完整的主要风险是什么？

5. 如何将权重绑定在堆叠式自动编码器中？这样做有什么意义？

6. 什么是生成模型？你能说出一种生成型自动编码器吗？

7. 什么是GAN？你能说出一些使用GAN的任务吗？

8. 训练GAN的主要困难是什么？

9. 尝试使用去噪自动编码器来预训练一个图像分类器。你可以使用MNIST（最简单的选项）。如果你想要更大的挑战，也可以使用更复杂的图像数据集，例如CIFAR10。不考虑你使用什么数据集，请按照下列步骤操作：

- 将数据集分为训练集和测试集。在完整的训练集上训练深度去噪自动编码器。

• 检查图像是否重构得很好。可视化编码层中的最激活每个神经元的图像。

• 重新使用自动编码器的较低层构建分类DNN。仅使用训练集中的500张图像进行训练。在进行或不进行预训练的情况下，效果是否更好？

10. 在你选择的图像数据集上训练变分自动编码器，并使用它来生成图像。或者，你可以尝试查找你感兴趣的未标记数据集，来看看是否可以生成新样本。

11. 训练一个DCGAN来处理你选择的图像数据集，并使用它来生成图像。添加重播体验，看看是否有帮助。将其转换为条件GAN，你可以控制生成的类别。

这些练习题的解答在附录A中提供。

第18章　强化学习

强化学习（RL）是当今机器学习最令人兴奋的领域之一，也是最古老的领域之一。自20世纪50年代以来一直存在，多年来产生了许多有趣的应用[1]，特别是在游戏（例如TD-Gammon，西洋双陆棋游戏程序）和机器控制中，但很少成为头条新闻。但是在2013年发生了一场革命，当时英国一家名为DeepMind的初创公司的研究人员展示了一种可以从头开始学习几乎所有Atari游戏的系统[2]，最终该系统的大多数性能均优于人类[3]，仅仅使用了原始像素作为输入，而没有游戏规则的先验知识[4]。这是一系列令人惊叹的壮举中的第一个，直到2016年3月，他们的系统AlphaGo击败了传奇的职业围棋选手李世石（Lee Sedol），并于2017年5月战胜了世界冠军柯洁。没有哪个程序可以击败这个游戏的大师，更不用说世界冠军了。如今，RL的整个领域都在不断涌现新想法，并具有广泛的应用范围。DeepMind在2014年被Google以超过5亿美元收购。

那么DeepMind是如何实现这一切的呢？事后看来，这似乎很简单：他们把深度学习的能力应用于强化学习领域，并且运行效果超出了他们最疯狂的梦想。在本章中，我们将首先解释强化学习是什么以及它的优点，然后介绍深度强化学习中最重要的两种技术：策略梯度和深度Q网络（DQN），包括对马尔可夫决策过程（MDP）的讨论。我们将使用这些技术来训练模型以平衡移动小车上的杆；然后将介绍TFAgents库，它使用先进的算法极大地简化了构建强大RL系统的过程，并且我们使用该库训练一个智能体来玩著名的Atari游戏Breakout。本章最后将讨论该领域的最新进展。

[1] 更多详细信息，请查看Richard Sutton和Andrew Barto关于RL的书Reinforcement Learning：An Introduction（MIT出版社）。

[2] Volodymyr Mnih et al.“，Playing Atari with Deep Reinforcement Learning”，arXiv preprint arXiv：

1312.5602（2013）.
[3] Volodymyr Mnih et al.，“Human-Level Control Through Deep Reinforcement Learning”，Nature 518（2015）：529－533.
[4] 在https://homl.info/dqn3上查看DeepMind系统学习来玩Space Invaders、Breakout和其他视频游戏。

18.1 学习优化奖励

在强化学习中，软件智能体在环境中进行观察并采取行动，作为回报，它会获得奖励。它的目标是学会以一种可以随时间推移最大化其预期回报的方式来采取行动。如果你不介意拟人化，则可以把正面奖励视为愉悦，把负面奖励视为痛苦（在这种情况下，“奖励”一词有点误导）。简而言之，该智能体在环境中行动，并通过反复试错来学习，以最大限度地提高其愉悦并最大限度地减少其痛苦。

这是一个相当广泛的设定，可以应用于各种各样的任务。以下是一些示例（见图18-1）：a.该智能体可以是控制机器人的程序。在这种情况下，环境就是现实世界，智能体通过一组传感器（例如摄像头和触摸传感器）来观察环境，其动作包括发送信号以激活电动马达。它可能被编程为在到达目的地时获得正奖励，而在浪费时间或走错方向时获得负奖励。

b.该智能体可以是控制Ms.Pac-Man的程序。在这种情况下，环境是Atari游戏的模拟，动作是9个可能的操纵杆位置（左上、下、中心等），观察结果是屏幕截图，而奖励是游戏点数。

c.类似地，智能体可以是玩棋盘游戏（例如围棋）的程序。

d.智能体不必控制物理（或虚拟）移动的事物。例如，它可以是一个智能恒温器，只要温度接近目标温度并节省能源，它就会获得正回报；而当人们需要调节温度时，它就会获得负回报，因此智能体商必须学会预测人类的需求。

e.智能体可以观察股市价格并决定每秒要买卖多少。奖励显然是金钱的盈利或者损失。

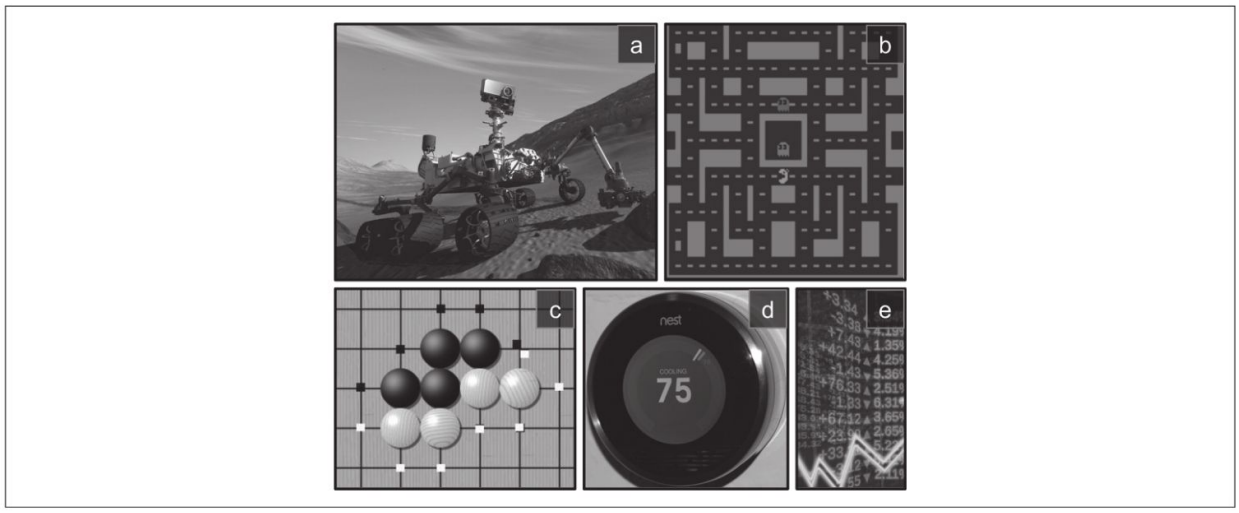


图18-1：强化学习示例：（a）机器人技术；（b）Ms.Pac-Man；（c）围棋选手；（d）恒温器；（e）自动交易员[1]

注意，可能根本没有任何正奖励。例如，智能体可能在迷宫中四处走动，在每个步骤中都获得负奖励，因此最好尽快找到出口！强化学习还有许多其他适合的任务，例如自动驾驶汽车、推荐系统、在网页上放置广告或控制图像分类系统应集中注意力的地方。

[1] 图片（a）来自NASA（公共领域）。（b）是Ms.Pac-Man游戏的截图，该游戏的版权为Atari（在本章中合法使用）。图片（c）和（d）摘自Wikipedia。（c）由用户Stevertigo创建，并根据Creative Commons BY-SA 2.0发布。（d）公共领域。（e）是根据Creative Commons CC0发布的Pixabay所复制的。

18.2 策略搜索

软件智能体用来确定其动作的算法称为其策略。该策略可以是一个神经网络，将观察作为输入并输出要采取的行动（见图18-2）。

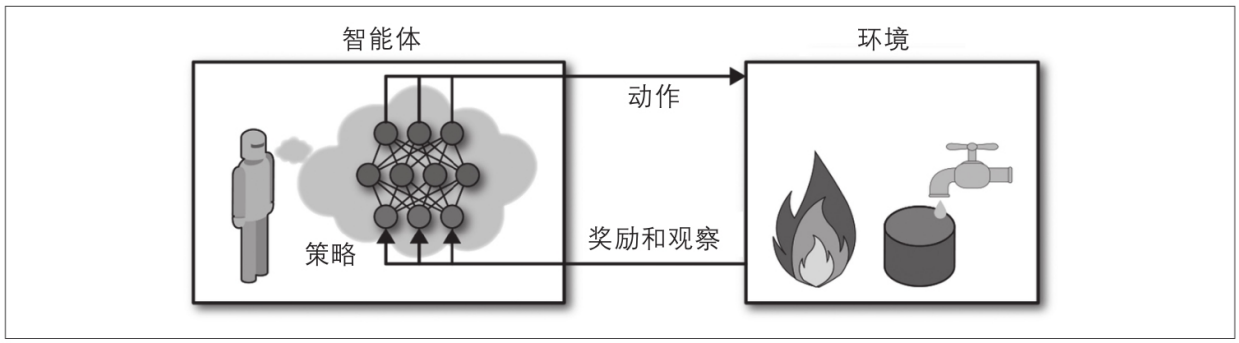


图18-2：使用神经网络策略进行强化学习

该策略可能是你能想到的任何算法，并且不必具有确定性。实际上，在某些情况下，甚至不需要观察环境！例如，考虑一个机器人吸尘器，其奖励是在30分钟内吸走的灰尘量。它的策略可能是以每秒概率p向前移动，或以概率1-p随机向左或向右旋转。旋转角将是-r和+r之间的随机角度。由于此策略涉及某种随机性，因此称为随机策略。机器人具有不确定的轨迹，这保证了它最终将到达任何位置并拾起所有灰尘。问题是，在30分钟内会吸走多少灰尘？

你将如何训练这样的机器人？你可以调整两个策略参数：概率p和角度范围r。一种可能的学习算法可能是为这些参数尝试许多不同的值，然后选择效果最佳的组合（见图18-3）。这是策略搜索的示例，在这种情况下使用暴力解决方法。当策略空间太大时（通常是这种情况），以这种方式查找一组好的参数就像大海捞针一样。

探索策略空间的另一种方法是使用遗传算法。例如，你可以随机创建第一代100条策略并进行尝试，然后“杀死”80条最差的策略[1]，并让20个幸存者各自产生4个后代。一个后代是其父代[2]的副本，外加一

些随机变异。幸存的策略及其后代共同构成了第二代。你可以继续以这种方式迭代很多代，直到找到一个好的策略为止[3]。

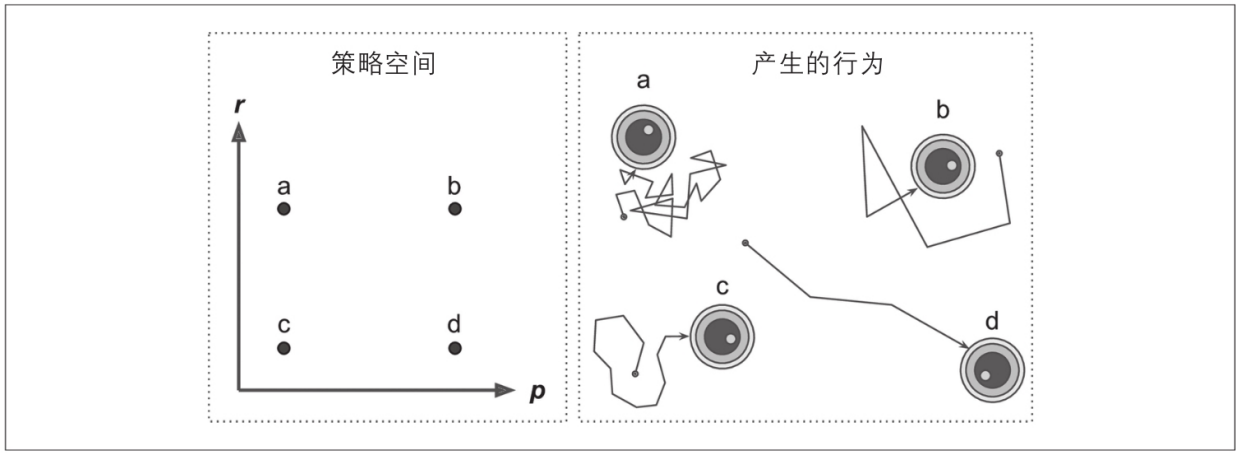


图18-3：策略空间中的4个点（左）和智能体的相应行为（右）

另一种方法是使用优化技术，通过评估与策略参数有关的奖励梯度，然后按照梯度向更高奖励的方向调整这些参数[4]。我们讨论的这种方法称为政策梯度（Policy Gradient，PG），本章稍后将更详细地介绍。回到吸尘器机器人，你可以稍微增加p并评估这样做是否会增加机器人在30分钟内吸收的灰尘量。如果会，则将p增加一些，否则减小p。我们会使用TensorFlow来实现一种流行的PG算法，但在此之前，我们需要创建一个智能体可以生活的环境——现在该介绍OpenAI Gym了。

[1] 通常最好让表现欠佳的人有生存的机会，以便在“基因库”中保留一些多样性。

[2] 如果有单亲，这称为无性繁殖。有两个（或更多）父母，这被称为有性生殖。后代的基因组（在这种情况下是一组策略参数）由其父本基因组的一部分随机组成。

[3] 用于增强学习的遗传算法的一个有趣示例是增强拓扑神经进化（NEAT）算法。

[4] 这称为“梯度上升”。它就像梯度下降，但方向相反：最大化而不是最小化。

18.3 OpenAI Gym介绍

强化学习的挑战之一是为了训练智能体，你首先需要拥有一个工作环境。如果你想为一个智能体编程来学习玩Atari游戏，则需要Atari游戏模拟器。如果你要为步行机器人编程，那么环境就是真实世界，你可以在这个环境中直接训练机器人，但有其局限性：如果机器人从悬崖上掉下来，你不能只单击“撤消”。你也无法加快时间，增加更多的计算能力也不会使机器人移动得更快。而且并行训练1000个机器人通常太昂贵了。简而言之，在现实世界中，训练既困难又缓慢，因此通常至少需要一个模拟环境才能进行引导训练。例如，你可以使用PyBullet或MuJoCo之类的库进行3D物理模拟。

OpenAI Gym[1]是一个工具包，提供了广泛的模拟环境（Atari游戏、棋盘游戏、2D和3D物理模拟等），因此你可以训练智能体，进行比较或开发新的RL算法。

在安装工具包之前，如果你使用virtualenv创建了隔离环境，则首先需要激活它：

```
$ cd $ML_PATH                # Your ML working directory (e.g., $HOME/ml)
$ source my_env/bin/activate # on Linux or MacOS
$ .\my_env\Scripts\activate  # on Windows
```

接下来，安装OpenAI Gym（如果不使用虚拟环境，则需要添加--user选项，或具有管理员权限）：

```
$ python3 -m pip install -U gym
```

根据你的系统，你可能还需要安装Mesa OpenGL实用程序（GLU）库（例如在Ubuntu 18.04上，你需要运行apt install libglu1-mesa）。渲染第一个环境需要这个库。接下来，打开一个Python shell或Jupyter notebook，并使用make（）创建一个环境：

```
>>> import gym
>>> env = gym.make("CartPole-v1")
>>> obs = env.reset()
>>> obs
array([-0.01258566, -0.00156614, 0.04207708, -0.00180545])
```

在这里我们创建了一个CartPole环境。这是一个2D模拟，其中小推车可以向左或向右加速，以平衡放置在其顶部的杆子（见图18-4）。你可以通过运行Gym.envs.registry.all（）来获取所有可用环境的列表。创建环境后，必须使用reset（）方法对其进行初始化。这会返回第一个观察值。观察值取决于环境的类型。对于CartPole环境，每个观测值都是一个包含4个浮点数的一维NumPy数组：这些浮点数代表小推车的水平位置（0.0=中心）、速度（右方向为正）、极角（0.0=垂直）和角速度（顺时针为正）。

现在，通过调用其render（）方法来显示此环境（见图18-4）。在Windows上，这需要首先安装X服务器，例如VcXsrv或Xming：

```
>>> env.render()
True
```

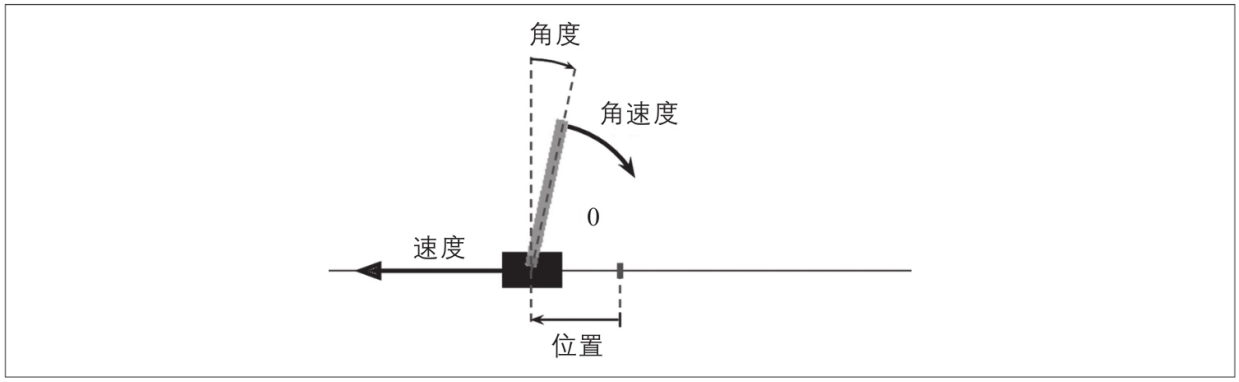


图18-4：CartPole环境

 如果你使用的是无屏幕服务器，例如云上的虚拟机，渲染会失败。避免这种情况的唯一方法是使用伪X服务器，例如Xvfb或Xdummy。你可以安装Xvfb（在Ubuntu或Debian上运行apt install xvfb）并使用以下命令启动Python：xvfb-run-s"-screen 0 1400x900x24"python3。或者安装Xvfb和pyvirtualdisplay库并在程序的开始处运行pyvirtualdisplay.Display（visible=0，size=（1400，900））.start（）。

如果想要render（）把渲染的图像作为NumPy数组返回，则可以设置mode="rgb_array"（奇怪的是，这个环境也将环境渲染到屏幕上）：

```
>>> img = env.render(mode="rgb_array")
>>> img.shape # height, width, channels (3 = Red, Green, Blue)
(800, 1200, 3)
```

让我们问一下环境可能采取的行动：

```
>>> env.action_space
Discrete(2)
```

Discrete（2）表示可能的动作是整数0和1，它们表示向左加速（0）或向右加速（1）。其他环境可能会有其他的离散动作或其他种类的动作（例如连续动作）。由于杆子向右倾斜（obs[2]>0），让我们把推车向右加速：

```
>>> action = 1 # accelerate right
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.01261699, 0.19292789, 0.04204097, -0.28092127])
>>> reward
1.0
>>> done
False
>>> info
{}
```

Step（）方法执行给定的动作并返回4个值：

Obs

这是新观察。现在，小推车向右移动（obs[1]>0）。杆子仍向右倾斜（obs[2]>0），但其角速度现在为负（obs[3]<0），因此在下一步之后极有可能向左倾斜。

reward

在这个环境下，无论你做什么，每一步都会获得1.0的奖励，因此目标是使小推车尽可能长时间地运行。

done

当整个回合结束时，此值为True。当杆子倾斜太大，离开了屏幕或经过200步后（在最后一种情况下，你赢了），就会发生这种情况。之后必须重设环境，然后才能再次使用。

info

该环境特定的字典可以提供一些额外的信息，你可能会发现这些信息对于调试或训练很有用。例如在某些游戏中，它可能指示智能体有多少生命。

 使用完环境后，应调用其close（）方法来释放资源。

让我们硬编码一个简单的策略，当杆子向左倾斜时向左加速，而当杆子向右倾斜时向右加速。我们执行这个政策来看看它获得500个回合以上的平均奖励：

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

希望该代码是不言自明的。让我们看一下结果：

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(41.718, 8.858356280936096, 24.0, 68.0)
```

即使进行了500次尝试，该策略也无法将杆保持直立超过68个连续的步骤。效果不是很好。如果你在Jupyter notebook中查看模拟环境，

会看到小推车左右摆动越来越强烈，直到杆子倾倒。让我们看看神经网络是否可以提出更好的策略。

[1] OpenAI是一家人工智能研究公司，部分由Elon Musk资助。该公司的既定目标是促进和发展有利于人类（而不是消灭人类）的友好型AI。

18.4 神经网络策略

让我们创建一个神经网络策略。就像我们之前进行硬编码的策略一样，该神经网络把观察值作为输入，输出要执行的动作。更准确地说，它将估计每个动作的概率，然后根据估计的概率随机选择一个动作（见图18-5）。在CartPole环境中，只有两个可能的动作（左或右），因此我们只需要一个输出神经元。它将输出动作0（左）的概率为p，当然动作1（右）的概率为1-p。例如，如果输出为0.7，那么我们将以70%的概率选择动作0，或以30%的概率选择动作1。

你可能想知道为什么我们要根据神经网络给出的概率来选择随机动作，而不是仅仅选择得分最高的动作。这种方法使智能体可以在探索新动作和利用已知运行良好的动作之间找到适当的平衡。这里有一个比喻：假设你是第一次去餐厅，所有的菜肴看起来都一样吸引人，因此你随机选择一个。如果事实证明很好吃，你可以增加下次购买的可能性，但是不应将其可能性提高到100%，否则你将永远不会尝试其他菜肴，其中一些有可能比你尝试过的要更好吃。

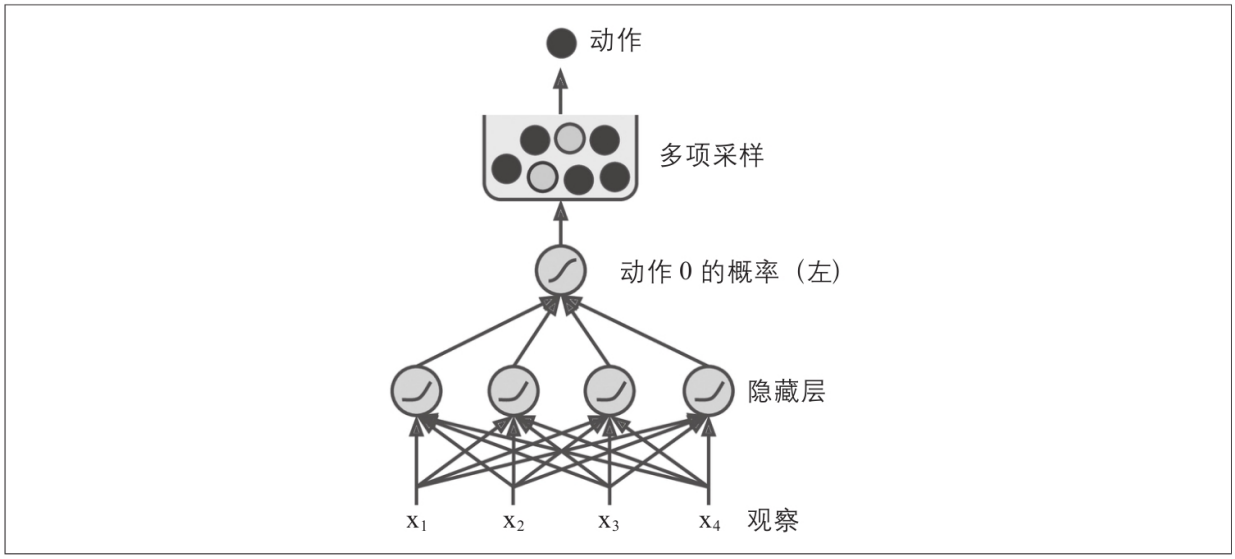


图18-5：神经网络策略

另请注意，在这个特定环境中，过去的动作和观察可以被安全地忽略，因为每个观察都包含了环境的完整状态。如果存在某种隐藏状态，那么你可能还需要考虑过去的动作和观察。例如，如果环境仅仅显示小推车的位置而不显示其速度，那你不仅必须考虑当前观测值，还必须考虑先前的观测值，以便估算当前速度。另一个示例是观察结果是否有噪声。在那种情况下，你通常希望使用过去的一些观察来估计最可能的当前状态。因此CartPole问题非常简单。观测结果无噪声，并且包含环境的完整状态。

以下是使用tf.keras构建此神经网络策略的代码：

```
import tensorflow as tf
from tensorflow import keras

n_inputs = 4 # == env.observation_space.shape[0]

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[n_inputs]),
    keras.layers.Dense(1, activation="sigmoid"),
])
```

导入之后，我们使用简单的顺序模型来定义策略网络。输入的数量是观察空间的大小（在CartPole示例的情况下为4），我们只有5个隐藏单元，因为这是一个简单的问题。最后，我们想输出一个单一的概率（向左移动的概率），因此我们使用sigmoid激活函数来输出一个单一的神经元。如果存在多于两个的可能的动作，则每个动作有一个输出神经元，那我们会使用softmax激活函数。

我们现在有了一个神经网络策略，该策略将进行观察并输出动作概率。但是我们如何训练呢？

18.5 评估动作：信用分配问题

如果我们在每一个步骤都知道最佳动作是什么，那我们可以按照通常的方法来训练神经网络，方法是使估计的概率分布与目标概率分布之间的交叉熵最小。这是常规的有监督学习。但是在“强化学习”中，智能体得到的唯一指导是通过奖励，而奖励通常是稀疏的和延迟的。例如，如果智能体使杆子平衡了100个步骤，那么如何知道执行的100个动作中哪个是好的，哪些是不好的呢？它所知道的是，杆子在最后一个动作之后掉下了，但可以肯定的是，这个最后的动作并不完全负责。这称为信用分配问题：当智能体得到报酬时，很难知道应该归功于哪些动作（或归咎于哪些动作）。想想看，一只狗在表现良好之后几小时得到了奖励，它会理解因为什么而获得回报吗？

为了解决这个问题，一种常见的策略是基于动作后获得的所有奖励的总和来评估一个动作，通常在每个步骤中应用一个折扣因子 γ（gamma）。折扣后回报的总和称为动作回报。考虑图18-6中的示例。如果一个智能体决定连续三次向右走，并且在第一步之后获得+10奖励，在第二步之后获得0，最后在第三步之后获得－50，假设我们使用折扣因子 $\gamma=0.8$，则第一个动作将得到 $10+\gamma\times 0+\gamma^2\times(-50)=-22$ 的回报。如果折扣因子接近于0，则与立即回报相比，未来回报将不起作用。相反，如果折扣因子接近1，则远期回报将几乎等于立即回报。典型的折扣因子从0.9到0.99不等。折扣因子为0.95，未来13个步骤的回报大约是即时回报的一半（因为 $0.95^{13}\approx 0.5$），而折扣因子为0.99，未来69个步骤的回报是即时奖励的一半。在CartPole环境中，动作有短期影响，因此选择0.95的折扣因子似乎是合理的。

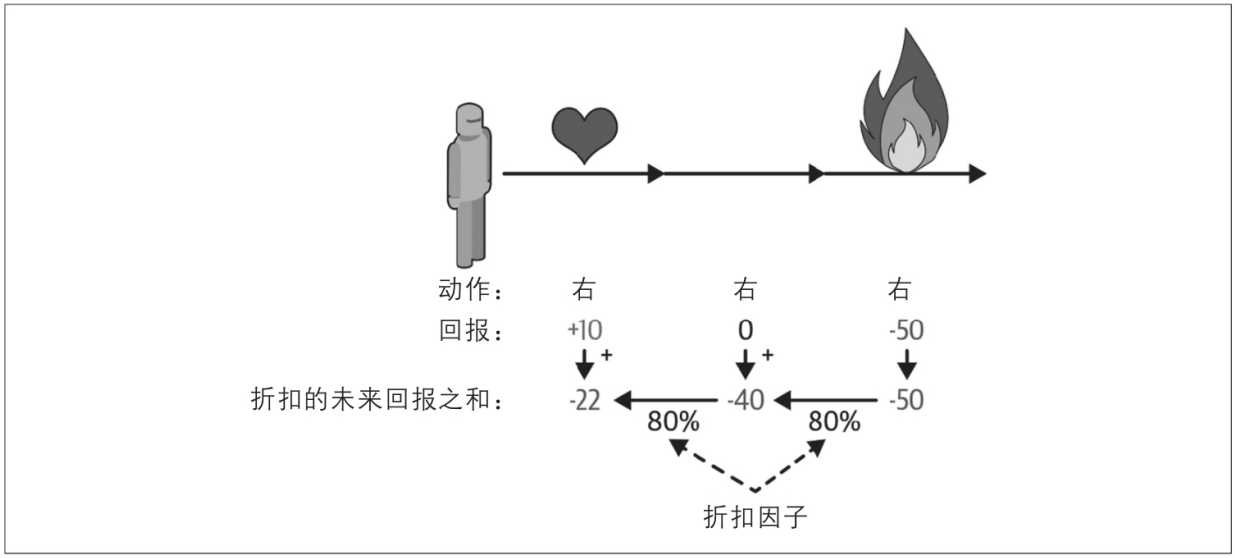


图18-6：计算一个动作的回报：折扣的未来回报的总和

当然，在一个好的动作之后，可能会跟随发生一些不好的动作，这些动作会导致杆子快速倾斜，导致这个好动作获得了较低的回报（类似地，一个好的演员有时可能出演了烂电影）。但是，如果我们有足够多的时间玩游戏，则平均而言，好的动作会比坏的动作获得更高的回报。我们想要估计一个动作与其他可能动作相比平均好或者坏多少。这就是所谓的动作优势。为此，我们必须运行许多回合并归一化所有动作的回报（通过减去均值并除以标准差）。我们可以合理地假设，具有负优势的动作是不好的，而具有正优势的动作是好的。太好了！现在，我们已经有了评估每个动作的方法，我们已经准备好使用策略梯度来训练第一个智能体。让我们看看如何来做。

18.6 策略梯度

如前所述，PG算法通过跟随朝着更高回报的梯度来优化策略的参数。Ronald Williams于1992年提出了一种流行的PG算法类别[1]，称为REINFORCE算法。以下是一个常见的变体：

1. 首先，让神经网络策略多次参与游戏，然后在每个步骤中计算梯度，使所选择的动作更有可能发生——但不要使用这些梯度。

2. 一旦运行了几个回合，就可以计算每个动作的优势（使用18.5节中介绍的方法）。

3. 如果某个动作的优势为正，则表示该动作可能很好，并且你希望应用较早计算出的梯度来使该动作将来更有可能被选择。但是，如果该动作的优点是负面的，则表示该动作可能是不好的，你希望应用相反的梯度以使该动作在将来被选择的可能性较小。解决方法是简单地把每个梯度向量乘以相应动作的优势。

4. 最后，计算所有得到的梯度向量的均值，并使用其执行“梯度下降”步骤。

让我们使用tf.keras来实现该算法。我们训练我们先前建立的神经网络策略，以便它学会平衡购物车上的杆子。首先，我们需要一个运行一个步骤的函数。现在，我们假装认为采取的任何动作都是正确的操作，以便我们可以计算损失及其梯度（这些梯度将被保存一会儿，稍后我们将根据操作的好坏来对其进行修改）：

```
def play_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba)
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
```

```
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))
    grads = tape.gradient(loss, model.trainable_variables)
    obs, reward, done, info = env.step(int(action[0, 0].numpy()))
    return obs, reward, done, grads
```

让我们来看一下这个函数：

• 在GradientTape块（见第12章）中，我们首先调用模型，并为其提供单个观察值（我们对观察值进行重构，以便使其成为包含单个实例的批处理，就像模型期望的那样）。这个输出向左移动的概率。

• 接下来，我们采样一个介于0和1之间的随机浮点数，并检查它是否大于left_proba。如果概率是left_proba，该动作为False，如果概率为1-left_proba，则该动作为True。一旦把此布尔值强制转换为数字，这个动作是以适当的概率为0（左）或1（右）。

接下来，我们定义向左移动的目标概率：它是1减去动作（转换到浮点数）。如果动作为0（向左），则向左走的目标概率为1。如果动作为1（右），则目标概率为0。

然后我们使用给定的损失函数来计算损失，并使用tape来计算关于模型的可训练变量的损失梯度。同样，这些梯度将在以后的应用之前进行调整，具体取决于动作的好坏。

最后，我们执行选定的动作，返回新的观察结果、奖励、是否结束这个回合，当然还返回我们刚刚计算出的梯度。

现在让我们创建另一个函数，该函数依赖play_one_step（）函数来执行多个回合，并返回每个回合和每个步骤的所有回报和梯度：

```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):
    all_rewards = []
    all_grads = []
```

```
    for episode in range(n_episodes):
        current_rewards = []
        current_grads = []
        obs = env.reset()
        for step in range(n_max_steps):
            obs, reward, done, grads = play_one_step(env, obs, model, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads)
            if done:
                break
        all_rewards.append(current_rewards)
        all_grads.append(current_grads)
    return all_rewards, all_grads
```

此代码返回一个奖励列表的列表（每个回合一个奖励列表，包含每个步骤一个奖励），以及梯度列表的列表（每个回合一个梯度列表，每个梯度列表包含每个步骤一个梯度元组，每个元组又包含每个可训练变量一个梯度张量）。

该算法执行play_multiple_episodes（）函数多次（例如10次），然后返回查看所有奖励，对其进行折价并将其标准化。为此，我们需要更多的函数：第一个函数将计算每个步骤中未来折扣奖励的总和，第二个函数将减去平均数并除以标准差，从而对许多回合中的所有这些折扣奖励（回报）进行归一化：

```
def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted
def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = [discount_rewards(rewards, discount_factor)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
            for discounted_rewards in all_discounted_rewards]
```

检查一下是否可行：

```
>>> discount_rewards([10, 0, -50], discount_factor=0.8)
array([-22, -40, -50])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]],
...                                discount_factor=0.8)
...
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([1.26665318, 1.0727777 ])]
```

调用Discount_rewards（）会返回我们期望的结果（见图18-6）。你可以验证函数Discount_and_normalize_rewards（）确实返回了两个回合中每个动作的归一化动作优势。请注意，第一个回合要比第二个回合差很多，因此其归一化的优势都是负的。第一个回合的所有动作都将被视为坏的，相反第二个回合的所有动作都将被视为好的。

我们几乎可以运行该算法了！现在让我们定义超参数。要运行150个训练迭代，每个迭代执行10个回合，每一个回合最多持续200个步骤。我们使用0.95的折扣因子：

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

我们还需要一个优化器和损失函数。常规的学习率为0.01的Adam优化器可以很好地完成工作，我们使用二元交叉熵损失函数，因为正在训练的是二元分类器（有两种可能的动作：向左或向右）：

```
optimizer = keras.optimizers.Adam(lr=0.01)
loss_fn = keras.losses.binary_crossentropy
```

准备创建并运行训练循环：

```
for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)
    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)
    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean(
            [final_reward * all_grads[episode_index][step][var_index]
            for episode_index, final_rewards in enumerate(all_final_rewards)
                for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)
    optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))
```

让我们看一下这段代码：

• 在每次训练迭代中，此循环都会调用play_multiple_episodes（）函数，该函数会玩10次游戏，并为每一个回合和每一个步骤返回所有奖励和梯度。

• 然后我们调用Discount and normalize rewards___（）来计算每个动作的归一化优势（在代码中我们称为final_reward）。这可以衡量事后的每个动作的实际好坏。

• 接下来，我们遍历每个可训练变量，并针对每个变量计算所有回合和所有步骤中该变量的梯度的加权平均值，以final_reward加权。

• 最后，我们使用优化器来应用这些均值梯度：调整模型的可训练变量，并希望该策略会更好。

我们完成了！该代码会训练神经网络策略，会成功的学习平衡小推车上的杆子（你可以在Jupyter notebook的“Policy Gradients”部分中尝试一下）。每个回合的平均奖励非常接近200（这是此环境默认情况下的最高奖励）。成功！

 研究人员试图找到一些算法，即使智能体最初对环境一无所知，也能很好工作。但是，除非你正在撰写论文，否则你应该毫不犹豫地给智能体输入先验知识，因为它会大大提高训练速度。例如由于你知道杆子应尽可能垂直，那么你可以添加与杆子角度成正比的负回报。这会降低回报的稀疏度，并加快训练速度。同样，如果你已经有了一个相当不错的策略（例如硬编码），则可能在使用策略梯度进行改进之前，你可以训练神经网络来模仿它。

我们刚刚训练的简单策略梯度算法解决了CartPole任务，但无法很好地扩展到更大和更复杂的任务。实际上，它的样本效率极低，这意味着它需要探索很长时间才能取得重大进展。正如我们所看到的，这是因为它必须运行多个回合才能估算每个动作的优势。但是，它是功能更强大的算法的基础，例如Actor-Critic算法。（我们将在本章末尾简要讨论它）。

现在，我们看看另一个流行的算法系列。PG算法直接尝试优化策略来增加奖励，而我们现在要看的算法则不太直接：智能体学习估计每个状态或每个状态中每个动作的预期回报，然后把这些知识用于决定如何行动。为了理解这些算法，我们必须首先介绍马尔可夫决策过程。

[1] Ronald J.Williams，“Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Leaning”，Machine Learning 8（1992）：229－256.

18.7 马尔可夫决策过程

在20世纪初期，数学家安德烈·马尔可夫（Andrey Markov）研究了无记忆的随机过程，称为马尔可夫链。这个过程具有固定数量的状态，并且在每个步骤中它都从一种状态随机演变到另一种状态。它从状态s演变为状态s'的概率是固定的，并且仅取决于（s，s'），而不取决于过去的状态（这就是为什么我们说系统没有记忆）。

图18-7显示了具有四个状态的马尔可夫链的示例。

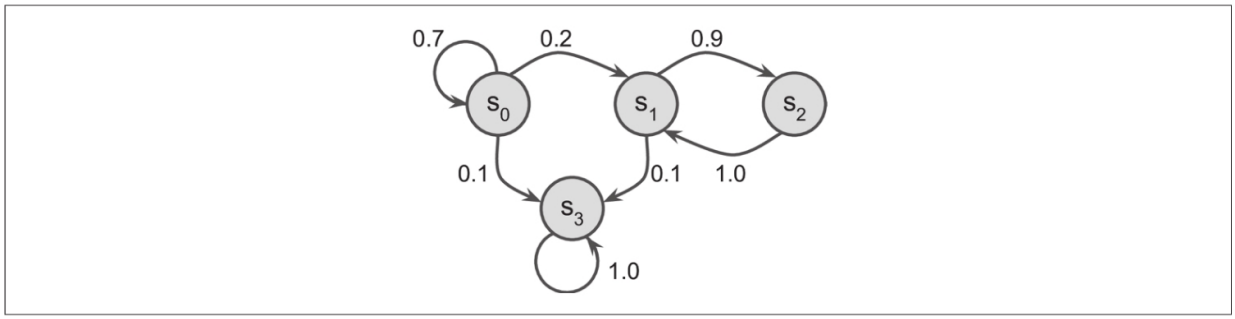


图18-7：马尔可夫链的示例

假设该过程在状态s_0开始，并且有70%的机会在下一步继续保持该状态。最终它必然会离开该状态，而且再也不会回来，因为没有其他状态指向s_0。如果它进入状态s_1，则很可能会进入状态s_2（概率为90%），然后立即返回状态s_1（概率为100%）。它可以在这两个状态之间交替多次，但是最终它会落入状态s3并永久停留在那里（这是终端状态）。马尔可夫链可以具有截然不同的动态过程，并且在热力学、化学、统计等领域大量使用。马尔可夫决策过程最早是在1950年代由理查德·贝尔曼（Richard Bellman）描述的[1]。它们类似于马尔可夫链，但有一个变化：在每个步骤中，智能体可以选择几种可能的动作之一，而转移概率取决于所选的动作。此外，一些状态转换会返回一定的奖励（正的或负的），而智能体的目标是找到一种能够随着时间的推移最大化奖励的策略。

例如，图18-8中表示的MDP具有三个状态（用圆环表示）和每个步骤最多三个可能的离散动作（用菱形表示）。

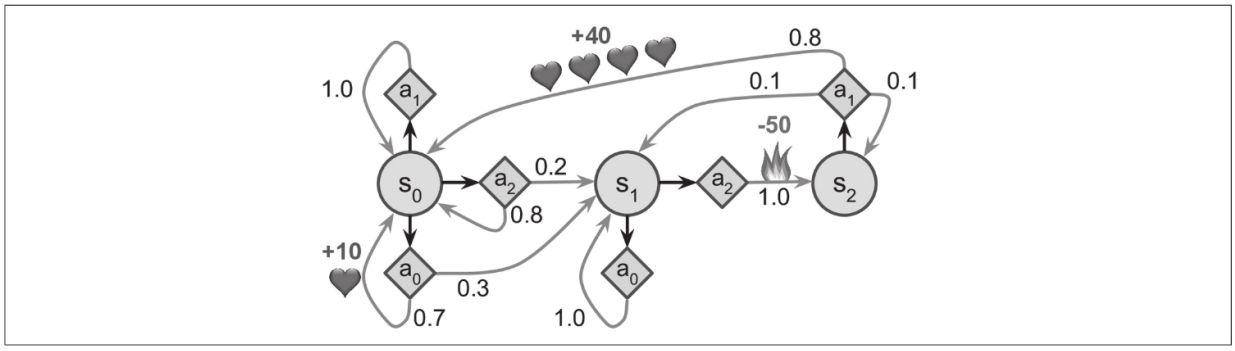


图18-8：马尔可夫决策过程的示例

如果它以状态s_0开始，则智能体可以在动作a_0、a_1或a_2之间进行选择。如果选择动作a_1，则它肯定会保持在状态s_0中，而没有任何回报。因此，如果愿意，它可以永远待在那里。但是，如果选择动作a_0，则它有70%的概率获得+10的奖励并保持在状态s_0中。然后，它可以一次又一次地尝试以得到尽可能多的奖励，但是在某一时刻，它将以状态s_1结束。在状态s_1中，它只有两个可能的动作：a_0或a_2。它可以通过重复选择动作a_0来选择保持不变，或者可以选择进入状态s_2并获得-50的负奖励。在状态s_2中，除了采取行动a_1之外别无选择，这很可能将其带回到状态s_0，并获得+40的奖励。通过查看这个MDP，你能猜出哪种策略随着时间的推移会获得最大的回报吗？很明显，在状态s_0中，动作a_0是最佳选择，在状态s_2中，智能体别无选择，只能采取动作a_1，但是在状态s_1中，智能体是应继续执行（a_0）还是（a_2）。

贝尔曼找到了一种方法来估计任何状态s的最佳状态值，记为V*（s），这是智能体在达到状态s时平均预期的所有折扣未来奖励的总和，假设在状态s时动作最佳。他表明，如果智能体有最佳动作，则适用Bellman最优方程式（见方程式18-1）。这个递归方程式表示，如果智能体采取最佳行动，则当前状态的最佳值等于采取一个最佳行动后平均获得的回报，加上该行动可能导致的所有可能的下一状态的预期最佳值。

公式18-1：贝尔曼最优方程

$$V^*(s) = \max_a \sum_s T(s,a,s')[R(s,a,s') + \gamma \cdot V^*(s')]，对于所有 s$$

在此等式中：

• T（s，a，s’）是从状态s到状态s’的转移概率，给定智能体选择了行动a。例如，在图18-8中，T（s_2，a_1，s_0）=0.8。

• R（s，a，s’）是智能体从状态s进入状态s所获得的奖励，给定智能体选择了动作a。例如，在图18-8中，R（s_2，a_1，s_0）=+40。

• γ是折扣因子。

该方程式直接导致可以精确估算每个可能状态的最佳状态值的算法：首先将所有状态估算值初始化为零，然后使用值迭代算法迭代更新它们（见公式18-2）。一个了不起的结果是，给定足够的时间，可以保证这些估计会收敛到与最佳策略相对应的最佳状态值。

公式18-2：值迭代算法

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \cdot V_k(s')]，对于所有 s$$

在此等式中，V_k（s）是算法第k次迭代时状态s的估计值。

 此算法是动态规划编程的一个示例，动态规划将一个复杂的问题分解为可迭代解决的易处理的子问题。

知道最佳状态值可能很有用，特别是对于评估策略很有用，但是并不能为我们提供智能体的最佳策略。幸运的是，贝尔曼发现了一种非常

相似的算法来估算最佳状态动作值，通常称为Q值（质量值）。状态动作对（s，a）的最佳Q值，记为Q*（s，a），是智能体在到达状态s选择行动a后平均期望获得的折扣未来回报之和，但在看到该动作的结果之前，假设它在该动作之后表现最佳。

它是这样工作的：再一次，你把所有Q值估计初始化为零，然后使用Q值迭代算法更新它们（见公式18-3）。

公式18-3：Q值迭代算法

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s.a.s') + \gamma \cdot \max_{a'} Q_k(s',a')], \text{对所有}(s',a')$$

一旦你有了最佳Q值，就可以轻松地定义最佳策略，记为π*（s）：当智能体处于状态s时，它应该为该状态选择具有最高Q值的动作：$\pi^*(s) = \max_a Q^*(s,a)$。

让我们将此算法应用于图18-8所示的MDP。首先，我们需要定义MDP：

```
transition_probabilities = [ # shape=[s, a, s']
        [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
        [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
        [None, [0.8, 0.1, 0.1], None]]
rewards = [ # shape=[s, a, s']
        [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
        [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
        [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

例如，要知道在执行动作a_1之后从s_2到s_0的转移概率，我们要查找transition_probabilities[2][1][0]（即0.8）。同样，要获得相应的奖励，我们要查找rewards[2][1][0]（即+40）。为了获得s_2中可能采取的动作列表，我们要查找possible_actions[2]（在这个情况下，只

能采取行动a_1）。接下来，我们必须将所有Q值初始化为0（不可能的动作除外，为此我们要把Q值设置为 - ∞）：

```
Q_values = np.full((3, 3), -np.inf) # -np.inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0  # for all possible actions
```

现在让我们运行Q值迭代算法。对于每个状态和每个可能的动作，它将公式18-3重复应用于所有Q值：

```
gamma = 0.90 # the discount factor

for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                    transition_probabilities[s][a][sp]
                    * (rewards[s][a][sp] + gamma * np.max(Q_prev[sp]))
                for sp in range(3)])
```

产生的Q值如下所示：

```
>>> Q_values
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.        ,         -inf, -4.87971488],
       [       -inf, 50.13365013,        -inf]])
```

例如，当智能体处于状态s_0并选择动作a_1时，折扣的未来奖励的预期总和约为17.0。

对于每个状态，让我们看一下具有最高Q值的动作：

```
>>> np.argmax(Q_values, axis=1) # optimal action for each state
array([0, 0, 1])
```

当使用0.90的折扣因子时，这为我们提供了这个MDP的最佳策略：在状态s_0中，选择动作a_0；在状态s_1中，选择动作a_0（即保持原状）；在状态s_2中，选择动作a_1（唯一可能的操作）。有趣的是，如果我们将折扣因子提高到0.95，则最佳策略将发生变化：在状态s_1中，最佳动作变为a_2。这是有道理的，因为你对未来的回报越看重，就越愿意为未来的幸福而付出一些痛苦。

[1] Richard Bellman，“A Markovian Decision Process”，Journal of Mathematics and Mechanics 6，no.5（1957）：679–684.

18.8 时序差分学习

离散动作的强化学习问题通常可以用马尔可夫决策过程建模，但智能体最初不知道转移概率（它不知道T（s，a，s’）），也不知道奖励（它不知道R（s，a，s’））。它必须至少经历一次每个状态和每个转移一次才能知道奖励，并且如果要对转移概率进行合理的估计，则必须多次经历。时序差分学习（TD学习）算法与值迭代算法非常相似，但进行了调整，以考虑到智能体仅仅对MDP有部分了解的事实。通常，我们假设智能体最初仅知道可能的状态和动作，仅此而已。智能体使用探索策略（例如，纯随机策略）来探索MDP，并且随着它的发展，TD学习算法会根据实际观察到的转变和奖励来更新状态值的估计值（见公式18-4）。

公式18-4：TD学习算法

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

或等效地：

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$$

有 $\delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$

在此等式中：

- α 是学习率（例如0.01）。

• $r+\gamma \cdot V_k(s')$被称为TD目标。

• $\delta_k(s, r, s')$被称为TD误差。

此等式第一种形式的一种更简洁的方法是使用符号$a \underset{\alpha}{\leftarrow} b$，表示$a_{k+1} \leftarrow (1-\alpha) \cdot a_k + \alpha \cdot b_k$。因此，公式18-4的第一行可以被改写成$V(s) \underset{\alpha}{\leftarrow} r + \gamma \cdot V(s')$。

 TD学习与随机梯度下降有很多相似之处，特别是它一次处理一个样本。而且，就像随机梯度下降一样，它只有在逐渐降低学习率的情况下才能真正收敛（否则它将不断在最佳Q值附近震荡）。

对于每个状态s，此算法仅跟踪智能体离开该状态时获得的即时奖励的运行平均值，加上其预期在以后获得的奖励（假设其动作最佳）。

18.9 Q学习

同样，Q学习算法是Q值迭代算法对最初未知转移概率和奖励的情况的一种改进（见公式18-5）。Q学习的工作方式是观看智能体的活动（随机进行），逐步改善其对Q值的估算。一旦得到准确的Q估算值（或足够接近），则最佳策略就是选择具有最高Q值的动作（即贪婪策略）。

公式18-5：Q学习算法

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} Q(s',a')$$

对于每个状态动作对（s，a），此算法都会跟踪智能体在离开状态s时使用动作a所获得的报酬r的运行平均值，加上预期获得的折扣未来报酬之和。为了估算此总和，我们假设下一个状态s′的Q估算值为最大值，因为我们假设目标策略从那时起将以最佳动作运行。实现Q学习算法，首先需要让智能体探索环境。为此，我们需要一个步骤函数，以便智能体可以执行一个动作并得到结果状态和奖励：

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

现在，让我们实现智能体的探索策略。由于状态空间很小，因此简单的随机策略就足够了。如果我们运行算法足够长的时间，则智能体会多次访问每个状态，并且还会尝试多次可能的动作：

```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

接下来，像之前一样初始化Q值之后，就可以运行具有学习率衰减的Q学习算法（使用功率调度，在第11章中介绍）：

```
alpha0 = 0.05 # initial learning rate
decay = 0.005 # learning rate decay
gamma = 0.90 # discount factor
state = 0 # initial state

for iteration in range(10000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = np.max(Q_values[next_state])
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state
```

该算法将收敛到最佳Q值，但是需要很多次迭代，并且可能需要进行大量的超参数微调。如你在图18-9中所见，Q值迭代算法（左）收敛非常快，迭代少于20次，而Q学习算法（右）则需要约8000次迭代收敛。显然，不知道转移概率或回报会使找到最佳策略变得非常困难！

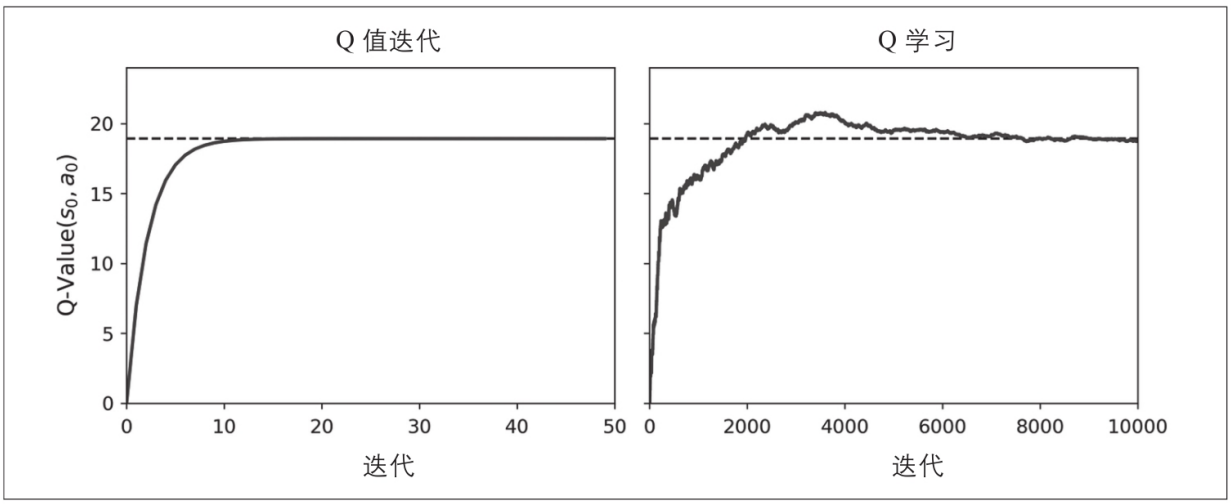


图18-9：Q值迭代算法（左）与Q学习算法（右）

Q学习算法称为异策略（off-policy）算法，因为训练的策略不一定是要被执行的策略：在前面的代码示例中，所执行的策略（探索策略）是完全随机的，而所训练的策略则始终选择Q值最高的动作。相反，策略梯度算法是一种同策略（on-policy）的算法：它使用训练的策略来探索世界。出乎意料的是，Q学习能够通过仅仅观察智能体的随机行为来学习最佳策略（想象一下，当你的老师是一只喝醉的猴子时学习打高尔夫球）。我们能做得更好吗？

18.9.1 探索策略

当然只有在探索策略充分探索MDP的情况下，Q学习才能起作用。尽管可以保证纯随机策略最终可以多次访问每个状态和每个转换，但是这样做可能要花很长时间。因此，一个更好的选择是使用“ε贪婪策略”（ε-greedy policy）：在每个步骤中，它以概率ε随机行动，或以概率1-ε贪婪地行动（即选择带有最高Q值的动作）。ε贪婪策略（与完全随机的策略相比）的优势在于，随着Q值的估算越来越好，它会花费越来越多的时间探索环境中有趣的部分，同时仍然花一些时间来访问MDP的未知区域。从较高的ε值开始（例如1.0）然后逐渐减小（例如，降低仅仅0.05）是很常见的。

另外，不仅仅只是依赖探索的机会，另一种方法是鼓励探索策略尝试以前未曾尝试过的动作。如公式18-6所示，这可以作为加到Q值估算值的奖金来实现。

公式18-6：使用探索函数进行Q学习

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} f(Q(s',a'), N(s',a'))$$

在此等式中：

- N（s’，a’）是在状态s’中选择a’动作的计数次数。

• f（Q，N）是一个探索函数，例如f（Q，N）=Q+κ/（1+N），其中κ是一个好奇心超参数，用于衡量智能体被吸引到未知对象的程度。

18.9.2 近似Q学习和深度Q学习

Q学习的主要问题在于它无法很好地扩展到具有许多状态和动作的大型（甚至中型）MDP。例如，假设你想使用Q学习来训练智能体玩Ms.Pac-Man（见图18-1）。Ms.Pac-Man可以吃大约150个小丸，每个可以存在或不存在（即已经食用）。因此，可能状态的数量大于$2^{150} \approx 10^{45}$。如果你为所有小丸和Ms.Pac-Man添加所有可能的位置组合，则可能状态的数量大于我们星球中原子的数量。因此，绝对不可能跟踪每个Q值的估算值。解决方案是使用一些可管理数量的参数（由参数向量θ给出），找到一个近似任何状态对（s，a）的Q值的函数Q_θ（s，a）。这称为近似Q学习。多年来，它推荐使用从状态中提取的手工特征的线性组合（例如最近的小丸的距离，它们的方向等）来估计Q值，但在2013年，DeepMind显示使用深度神经网络可以更好地工作，尤其是对于复杂的问题，不需要任何特征工程。用于估计Q值的DNN被称为深度Q网络（DQN），而将DQN用于近似Q学习则称为深度Q学习。

现在，我们如何训练DQN呢？好吧，考虑由DQN计算的给定状态-动作对（s，a）的近似Q值。多亏了Bellman，我们知道我们希望这个近似Q值尽可能接近我们在状态s下进行动作a后实际观察到的奖励r，加上从那时起最佳动作的折扣值。为了估算未来折价奖励的总和，我们可以简单地在下一个状态s'上，对所有可能的动作a'执行DQN。对于每个可能的动作，我们都会得到一个近似的未来Q值。然后，我们选择最高的（因为我们假设我们将以最佳动作执行）并对其进行打折扣，这给我们一个未来折价奖励的总和的估计。通过把奖励r和将来的折扣值的估算值相加，可以得到状态-动作对（s，a）的目标Q值y（s，a），如公式18-7所示。

公式18-7：目标Q值

$$Q_{\text{target}}(s,a) = r + \gamma \cdot \max_{a'} Q_{\theta}(s',a')$$

有了这个目标Q值，我们可以使用任何梯度下降算法来进行训练。具体来说，我们通常试图最小化估计的Q值Q（s，a）与目标Q值之间的平方误差（或Huber损失来降低算法对大误差的敏感性）。这就是基本的深度Q学习算法的全部！让我们看看如何实现它来解决CartPole环境。

18.10 实现深度Q学习

我们需要的第一件事是深度Q网络。从理论上讲，你需要一个输入一个状态-动作对并且输出近似Q值的神经网络，但实际上，使用一个输入状态并且为每个可能的动作输出一个近似Q值的神经网络效率更高。为了解决CartPole环境，我们不需要非常复杂的神经网络。几个隐藏层会执行以下操作：

```
env = gym.make("CartPole-v0")
input_shape = [4] # == env.observation_space.shape
n_outputs = 2 # == env.action_space.n

model = keras.models.Sequential([
    keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    keras.layers.Dense(32, activation="elu"),
    keras.layers.Dense(n_outputs)
])
```

要使用此DQN选择动作，我们选择具有最大预测Q值的动作。为了确保智能体能探索环境，我们使用“ε贪婪”策略（即我们将选择概率为ε的随机动作）：

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(2)
    else:
        Q_values = model.predict(state[np.newaxis])
        return np.argmax(Q_values[0])
```

我们不是仅仅根据最新的经验来训练DQN，而是把所有的经验存储在重播缓冲区（或重播内存）中，并在每次训练迭代中从中随机抽样一个训练批次。这有助于减少训练批次中的经验之间的相关性，从而极大地帮助训练。为此，我们仅使用双端口队列：

```
from collections import deque
replay_buffer = deque(maxlen=2000)
```

 双端口队列是一个链表，其中每个元素都指向下一个和上一个元素。它使插入和删除元素的速度非常快，但是双端口队列越长，随机访问的速度就越慢。如果你需要很大的重放缓冲区，请使用循环缓冲区；请参阅notebook的“Deque vs Rotating List”部分来了解其实现。

每个经验由五个元素组成：状态，智能体采取的行动，所获得的报酬，达到的下一个状态，最后是一个布尔值，表明这个回合是否在该点结束（完成）。我们需要一个小函数来从重播缓冲区中随机抽取一批经验。它将返回与5个经验元素相对应的5个NumPy数组：

```
def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]
    states, actions, rewards, next_states, dones = [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(5)]
    return states, actions, rewards, next_states, dones
```

让我们还创建一个函数，该函数使用 ε 贪婪策略来执行单个步骤，然后将所获得的经验存储在重播缓冲区中：

```
def play_one_step(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, done, info = env.step(action)
    replay_buffer.append((state, action, reward, next_state, done))
    return next_state, reward, done, info
```

最后，让我们创建最后一个函数，该函数将从重播缓冲区中采样一批经验，并通过对该批量执行单个梯度下降步骤来训练DQN：

```
batch_size = 32
discount_factor = 0.95
optimizer = keras.optimizers.Adam(lr=1e-3)
loss_fn = keras.losses.mean_squared_error
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    max_next_Q_values = np.max(next_Q_values, axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_factor * max_next_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

让我们看一下这段代码：

• 首先，我们定义一些超参数，然后创建优化器和损失函数。

• 然后，我们创建training_step（）函数。首先从一批经验中取样，然后使用DQN来预测每个经验的下一个状态中每个可能动作的Q值。由于我们假设智能体会最佳执行，因此我们仅为每个下一个状态保留最大Q值。接下来，我们使用公式18-7计算每个经验的状态-动作对的目标Q值。

• 接下来，我们使用DQN为每个有经验的状态动作对计算Q值。但是，DQN还将输出其他可能动作的Q值，而不仅仅是智能体实际选择的动作。因此我们需要屏蔽所有不需要的Q值输出。使用tf.one_hot（）函数可以很容易地把动作索引数组转换成这样的掩码。例如，如果前三个经验分别包含动作1、1、0，则掩码将以[[0，1]，[0、1]，[1，0]，...]开头。然后，我们可以把DQN的输出与此掩码相乘，这把我们不需要的所有Q值设置为零。然后，我们在轴1上求和来消除所有零，仅仅保留有经验的状态动作对的Q值。这为我们提供了张量Q_Values，包含了批次中每个经验的一个预测Q值。

• 然后我们计算损失：这是经验的状态操作对的目标Q值与预测Q值之间的均方误差。

• 最后，我们执行梯度下降步骤，来最小化模型的可训练变量的损失。

这是最难的部分。现在训练模型很直接：

```
for episode in range(600):
    obs = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info = play_one_step(env, obs, epsilon)
        if done:
            break
    if episode > 50:
        training_step(batch_size)
```

我们运行600个回合，每回合最多200个步骤。在每一步骤中，我们首先计算 ε 贪婪策略的epsilon值：它会在不到500个回合内从1线性降低到0.01。然后我们调用play_one_step（）函数，该函数会使用 ε 贪婪策略来选择一个动作，然后执行该动作并将经验记录在重播缓冲区中。如果这个回合完成，我们将退出循环。最后，如果超过了第50个回合，则调用training_step（）函数对从重播缓冲区采样的一个批次进行模型训练。我们玩50个回合而不训练的原因是给重播缓冲区一些时间来填充（如果我们没有等待足够的时间，则重播缓冲区中就没有足够的多样性）。就这样我们实现了深度Q学习算法！

图18-10显示了智能体在每个回合中获得的总奖励。

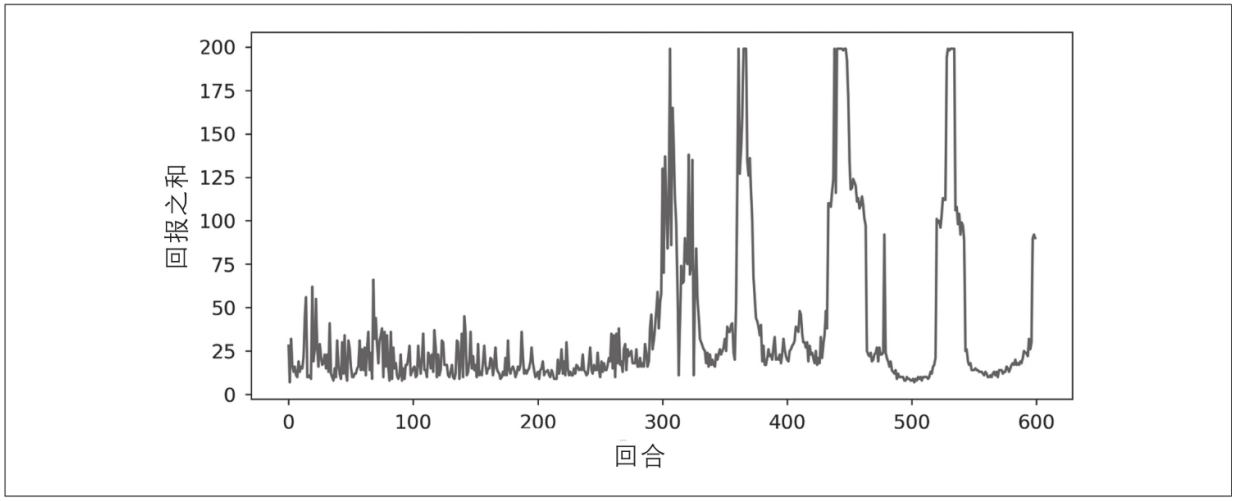


图18-10：深度Q学习算法的学习曲线

如你所见，该算法在将近300个回合中根本没有任何明显的进展（部分是因为ε在开始时很高），然后其性能突然暴涨至200（这是环境中最大可能的性能）在这种环境下）。这真是个好消息：该算法运行良好，并且实际上比策略梯度算法运行得要快！但是等一下……只是几个回合，它忘记了所有的一切，它的性能下降到了25以下！这称为灾难性遗忘，这几乎是所有RL算法所面临的主要问题之一：随着智能体探索环境，它会更新其策略，但是在环境的一部分中学到的东西可能会破坏先前在环境的其他部分中学到的东西。经验是非常相关的，学习环境也在不断变化——这对于梯度下降而言并不理想！如果增加重播缓冲区的大小，这个算法会较少受到这个问题的影响。降低学习率也可能有所帮助。但是事实是，强化学习很困难：训练通常是不稳定的，在找到有效的参数组合之前，你可能需要尝试许多超参数值和随机种子。例如，如果你尝试将前面每层的神经元数量从32更改为30或34，则性能将永远不会超过100（DQN有一个而不是两个隐藏层可能会更稳定）。

 众所周知，强化学习非常困难，这主要是由于训练的不稳定性以及对超参数值和随机种子选择的巨大敏感性[1]。正如研究人员Andrej Karpathy所说：“[监督学习]想要工作……RL必须一定要工作。”你需要时间、耐心、恒心和一点运气。这是RL没有像常规深度学

习（例如卷积网络）那样被广泛采用的主要原因。但是除了AlphaGo和Atari游戏外，还有一些实际的应用：例如，Google使用RL来优化其数据中心的成本，并且将其用于某些机器人应用程序，超参数调整和推荐系统中。

你也许想知道为什么我们没有画出损失。事实证明，损失不是模型性能的好指标。损失可能会减少，但是智能体的性能可能会变差（例如，当智能体卡在环境的一个小区域中并且DQN开始过度拟合该区域时，可能会发生这种情况）。相反，损失可能会增加，但智能体商可能会表现更好（例如，如果DQN低估了Q值，并且它开始正确地增加其预测，智能体商可能会表现得更好，获得更多的回报，但是损失可能会更大，因为DQN也会设置更大的目标）。

到目前为止，我们一直在使用的基本的深度Q学习算法过于不稳定而无法学习Atari游戏。那么DeepMind是如何做到的？好吧，他们调整了算法！

[1] Alex Irpan在2018年发表的一篇精彩文章很好地阐述了RL的最大困难和局限性。

18.11 深度Q学习的变体

让我们看一下可以稳定和加速训练的深度Q学习算法的一些变体。

18.11.1 固定的Q值目标

在基本的深度Q学习算法中，该模型既用来进行预测，也可以用来设置自己的目标。

这可能导致类似于狗追尾的情况。这种反馈回路可能使网络不稳定：它可能发散、振荡、冻结等。为了解决这个问题，DeepMind的研究人员在2013年的论文中使用了两个DQN（而不是一个）：第一个是在线模型，该模型在每个步骤中学习并用于移动智能体，另一个是仅仅用来定义目标的目标模型。目标模型只是在线模型的克隆：

```
target = keras.models.clone_model(model)
target.set_weights(model.get_weights())
```

然后，在training_step（）函数中，在计算下一个状态的Q值时，我们只需要更改一行即可使用目标模型而不是在线模型：

```
next_Q_values = target.predict(next_states)
```

最后，在训练循环中，我们必须定期（例如每50个回合）把在线模型的权重复制到目标模型中：

```
    if episode % 50 == 0:
        target.set_weights(model.get_weights())
```

由于与在线模型相比，目标模型的更新频率要低得多，因此Q值目标更加稳定，我们前面讨论的反馈回路受到抑制，其影响也较小。这种方法是DeepMind的研究人员在其2013年论文中的主要贡献之一，它允许智能体可以从原始像素学习玩Atari游戏。为了稳定训练，他们使用了0.00025的微小学习率，每10 000步更新目标模型（而不是以前的代码示例中的50步），并且使用了非常大的重播缓冲区，包含了100万个经验。它们非常缓慢的降低epsilon，在一百万步中从1降到0.1，并且让算法运行了5千万步。在本章的后面，我们将使用TF-Agents库来训练DQN智能体及使用这些超参数来玩Breakout，但是在这之前，让我们看一下另一个DQN变体，该变体再次成功击败了现有的最新技术。

18.11.2 双DQN

在2015年的一篇论文中[1]，DeepMind的研究人员调整了他们的DQN算法，提高了性能并在一定程度上稳定了训练。他们称此变体为Double DQN。这个更新基于以下观察结果：目标网络易于高估Q值。确实，假设所有动作都一样好：目标模型估计的Q值应该相同，但是由于它们是近似值，因此偶然的原因，某些Q值可能比其他值大一些。目标模型始终选择最大的Q值，该值略大于平均Q值，很有可能高估了真实的Q值（有点像在测量泳池深度时计算最高随机波浪的高度）。为了解决这个问题，他们建议在选择下一个状态的最佳动作时使用在线模型而不是目标模型，并且仅仅使用目标模型来估计这些最佳动作的Q值。这是更新后的training_step（）函数：

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    best_next_actions = np.argmax(next_Q_values, axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    next_best_Q_values = (target.predict(next_states) * next_mask).sum(axis=1)
```

```
target_Q_values = (rewards +
                   (1 - dones) * discount_factor * next_best_Q_values)
mask = tf.one_hot(actions, n_outputs)
[...] # the rest is the same as earlier
```

仅仅几个月后，DQN算法的另一种改进被提出来了。

18.11.3 优先经验重播

不从重播缓冲区中均匀采样一些经验，为什么不更频繁地采样重要的经验呢？这种想法称为重要性采样（Importance Sampling，IS）或优先经验重放（Prioritized Experience Replay，PER），DeepMind的研究人员在2015年的论文中提出了这一想法[2]（再次！）。65更具体地说，如果经验有可能导致快速的学习进度，则被认为是“重要的”。但是我们如何估计呢？一种合理的方法是测量TD误差的大小 $\delta = r + \gamma \cdot V(s') - V(s)$。较大的TD错误表明转移（s，r，s’）非常令人惊奇，因此可能值得学习[3]。当经验被记录在重播缓冲区中时，其优先级被设置为非常大的值，以确保至少采样一次。但是，一旦被采样（每次都被采样时），就会计算出TD误差 δ，并且将这个经验的优先级设置为 $p = |\delta|$（加上一个小常数，以确保每个经验都具有非零的采样概率）。对优先级为p的体验进行采样的概率P与 p^{ζ} 成正比，其中 ζ 是一个超参数，它控制我们希望重要性采样的贪婪程度：当 $\zeta = 0$ 时，我们得到均匀采样；当 $\zeta = 1$ 时，我们得到了完全重要性抽样。在本论文中，作者使用 $\zeta = 0.6$，但最佳值将取决于任务。

但是有一个陷阱：由于样本会偏向重要经验，因此我们必须在训练期间根据它们的重要性降低权重来弥补这种偏差，否则该模型将过度拟合重要的经验。需要明确的是，我们希望对重要的经验进行更多的采样，但这也意味着在训练过程中必须给予它们较低的权重。为此，我们将每个经验的训练权重定义为 $w = (np)^{-\beta}$，其中n是重放缓冲区中的经验数量，β 是一个超参数，它控制我们要补偿重要采样偏差的程度（0表示完全不，而1表示完全）。在本论文中，作者在训练开

始时使用β=0.4，并在训练结束时将其线性增加到β=1。同样，最佳值取决于任务，但是如果你增加一个，通常也希望增加另一个。

现在让我们看一下DQN算法的最后一个重要变体。

18.11.4 竞争DQN

DeepMind研究人员在2015年的另一篇论文[4]中引入了Dueling DQN算法（DDQN，不要与Double DQN混淆，尽管两种技术可以轻松的组合在一起）。要理解它是如何工作的，我们必须首先注意到一个状态动作对（s，a）的Q值可以表示为Q（s，a）=V（s）+A（s，a），与处于该状态的所有其他可能的动作相比，V（s）是状态s的值，而A（s，a）是在状态s中采取动作a的优势。此外，状态的值等于该状态的最佳动作a*的Q值（因为我们假设最佳策略会选择最佳动作），因此V（s）=Q（s，a*），这意味着A（s，a*）=0。在竞争DQN中，模型同时估算状态值和每个可能动作的优势。由于最佳动作的优势应为0，因此模型从所有预测的优势中减去最大的预测优势。这是一个使用函数API实现的简单的竞争DQN模型：

```
K = keras.backend
input_states = keras.layers.Input(shape=[4])
hidden1 = keras.layers.Dense(32, activation="elu")(input_states)
hidden2 = keras.layers.Dense(32, activation="elu")(hidden1)
state_values = keras.layers.Dense(1)(hidden2)
raw_advantages = keras.layers.Dense(n_outputs)(hidden2)
advantages = raw_advantages - K.max(raw_advantages, axis=1, keepdims=True)
Q_values = state_values + advantages
model = keras.Model(inputs=[input_states], outputs=[Q_values])
```

该算法的其余部分与之前的相同。实际上，你可以创建双竞争DQN，并将其与优先级经验回放结合起来！更广泛地说，如DeepMind在2017年的一篇论文中证明的那样，可以将许多RL技术进行组合[5]。该论文的作者将6种不同的技术组合到一个名为Rainbow的智能体中，该智能体的性能大大超过了现有的最新技术。

不幸的是，实现所有这些技术，对其进行调试，微调以及对模型进行训练需要大量的工作。因此，与其重新发明轮子，最好还是重用可扩展且经过良好测试的库，例如TFAgent。

[1] Hado van Hasselt et al.，“Deep Reinforcement Learning with Double Q-Learning”，Proceedings of the 30th AAAI Conference on Artificial Intelligence（2015）：2094－2100.
[2] Tom Schaul et al.，“Prioritized Experience Replay”，arXiv preprint arXiv：1511.05952（2015）.
[3] 回报也可能是有噪声的，在这种情况下，有更好的方法来评估经验的重要性（有关示例，请参见本论文）。
[4] Ziyu Wang et al.，“Dueling Network Architectures for Deep Reinforcement Learning”，arXiv preprint arXiv：1511.06581（2015）.
[5] Matteo Hessel et al.，“Rainbow：Combining Improvements in Deep Reinforcement Learning”，arXiv preprint arXiv：1710.02298（2017）：3215－3222.

18.12 TF-Agents库

TF-Agents库是一个基于TensorFlow的强化学习库，由Google开发并于2018年开源。与OpenAI Gym一样，它提供了许多现成的环境（包括所有OpenAI Gym环境的包装器），以及它支持的PyBullet库（用于3D物理模拟）、DeepMind的DM Control库（基于MuJoCo的物理引擎）和Unity的ML-Agents库（用于模拟许多3D环境）。它还实现了许多RL算法，包括REINFORCE、DQN和DDQN，以及各种RL组件，例如有效的重放缓冲区和度量。它快速，可扩展，易于使用且可自定义：你可以创建自己的环境和神经网络，可以自定义几乎任何组件。在本节中，我们使用TF-Agents和DQN算法（如果愿意，你可以轻松地切换到另一种算法）来训练智能体玩著名的Atari游戏Breakout（见图18-11[1]）。

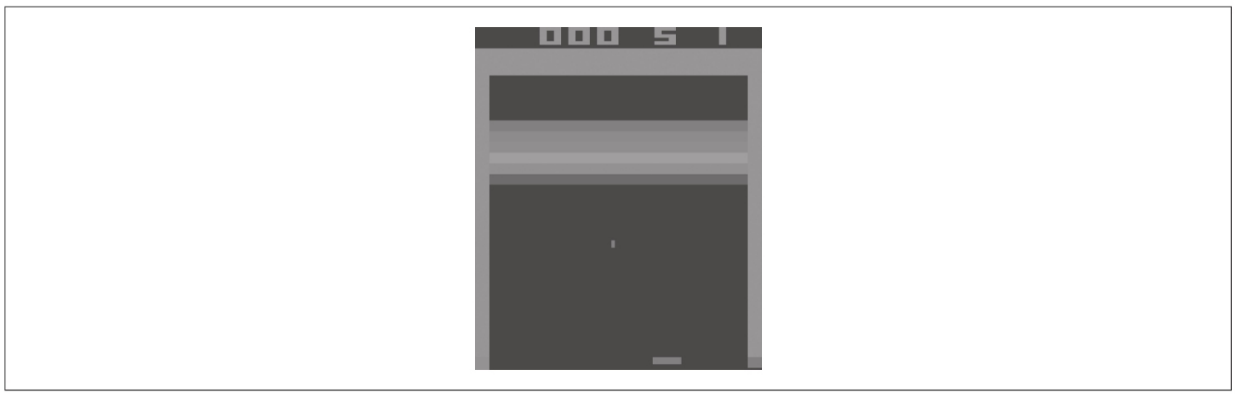


图18-11：著名的Breakout游戏

18.12.1 安装TF-Agents

让我们从安装TF-Agents开始。这个可以使用pip完成（与往常一样，如果你使用的是虚拟环境，请确保先激活它；否则，需要使用--user选项或具有管理员权限）：

```
$ python3 -m pip install -U tf-agents
```

 在撰写本书时，TF-Agents仍然是相当新的并且每天都在改进，因此在你阅读本书时，API可能会有所变化。但是总体情况及大多数代码都应该保持不变。如果有任何问题，我会相应地更新Jupyter notebook，因此请务必查看更新。

接下来，让我们创建一个包装了OpenAI GGym的Breakout环境的TF-Agents环境。为此，你必须首先安装OpenAI Gym的Atari依赖库：

```
$ python3 -m pip install -U 'gym[atari]'
```

在其他库中，此命令将安装atari-py，这是Arcade Learning Environment（ALE）（在Atari 2600模拟器Stella之上构建的框架）的Python接口。

18.12.2 TF-Agents环境

如果一切顺利，你应该可以导入TF-Agent并创建一个Breakout环境：

```
>>> from tf_agents.environments import suite_gym
>>> env = suite_gym.load("Breakout-v4")
>>> env
<tf_agents.environments.wrappers.TimeLimit at 0x10c523c18>
```

这只是OpenAI Gym环境的包装，你可以通过gym属性访问它：

```
>>> env.gym
<gym.envs.atari.atari_env.AtariEnv at 0x24dcab940>
```

TF-Agents环境与OpenAI Gym环境非常相似，但有一些区别。首先，reset（）方法不返回观察值；相反它返回一个包装了观察结果的TimeStep对象以及一些其他信息：

```
>>> env.reset()
TimeStep(step_type=array(0, dtype=int32),
         reward=array(0., dtype=float32),
         discount=array(1., dtype=float32),
         observation=array([[[0., 0., 0.], [0., 0., 0.],...]]], dtype=float32))
```

step（）方法也返回一个TimeStep对象：

```
>>> env.step(1) # Fire
TimeStep(step_type=array(1, dtype=int32),
         reward=array(0., dtype=float32),
         discount=array(1., dtype=float32),
         observation=array([[[0., 0., 0.], [0., 0., 0.],...]]], dtype=float32))
```

reward和observation属性是不言自明的，与OpenAI Gym相同（除了奖励以NumPy数组表示）。step_type属性对于回合中的第一个时间步骤等于0，中间的时间步骤等于1，最后的时间步骤等于2。你可以调用时间步骤的is_last（）方法来检查它是否为最后一个。最后，discount属性指示这个时间步骤要使用的折扣因子。在此示例中，它等于1，因此根本没有折扣。你可以通过在加载环境时设置discount参数来定义折扣因子。

 你可以随时通过调用环境的current_time_step（）方法来访问环境的当前时间步骤。

18.12.3　环境规范

TF-Agents环境方便地提供了观察、动作和时间步骤的规范，包括其形状、数据类型和名称，以及其最小值和最大值：

```
>>> env.observation_spec()
BoundedArraySpec(shape=(210, 160, 3), dtype=dtype('float32'), name=None,
                 minimum=[[[0. 0. 0.], [0. 0. 0.],...]],
                 maximum=[[[255., 255., 255.], [255., 255., 255.], ...]])
>>> env.action_spec()
BoundedArraySpec(shape=(), dtype=dtype('int64'), name=None,
                 minimum=0, maximum=3)
>>> env.time_step_spec()
TimeStep(step_type=ArraySpec(shape=(), dtype=dtype('int32'), name='step_type'),
         reward=ArraySpec(shape=(), dtype=dtype('float32'), name='reward'),
         discount=BoundedArraySpec(shape=(), ..., minimum=0.0, maximum=1.0),
         observation=BoundedArraySpec(shape=(210, 160, 3), ...))
```

如你所见，这些观察结果只是Atari的屏幕截图，表现出NumPy数组[210、160、3]的形式。如果要渲染环境，你可以调用env.render（mode="human"），如果要以NumPy数组的形式获取图像，只需调用env.render（mode="rgb_array"）（在OpenAI Gym中，这是默认模式）。

这里有4个可用的动作。Gym的Atari环境有一个额外的方法，你可以调用该方法来了解每个动作对应的内容：

```
>>> env.gym.get_action_meanings()
['NOOP', 'FIRE', 'RIGHT', 'LEFT']
```

 规范可以是规范类的实例，规范的嵌套列表或字典。如果规范是嵌套的，则指定的对象必须与规范的嵌套结构相匹配。例如，如果观察规范为{"sensors"：ArraySpec（shape=[2]），"camera"：ArraySpec（shape=[100，100]）}，则有效观察值为{"sensors"：np.array（[1.5，3.5]），"camera"：np.array（...）}。tf.nest包提供了处理此类嵌套结构的工具。

观测值相当大，因此我们要对其进行下采样并将其转换为灰度。这会加快训练速度和使用更少的RAM。为此，我们可以使用环境包装器。

18.12.4 环境包装器和Atari预处理

TF-Agents在tf_agents.environments.wrappers包中提供了几个环境包装器。顾名思义，它们包装了一个环境，将每个调用转发给该环境，还添加了一些额外的功能。以下是一些可用的包装器：

ActionClipWrapper

将动作修改为动作规范。

ActionDiscretizeWrapper

将连续的动作空间量化为离散的动作空间。例如，如果原始环境的动作空间是从－1.0到+1.0的连续范围，但是你想要使用仅支持离散动作空间的算法（例如DQN），则可以使用discrete_env=ActionDiscretizeWrapper（env，num_actions=5）来包装环境，新的discrete_env具有一个离散的动作空间，其中包含5个可能的动作：0、1、2、3、4。这些动作对应于原始环境中的动作－1.0、－0.5、0.0、0.5和1.0。

ActionRepeat

在累积奖励的同时，以n步重复每个动作。在许多环境中，这可以显著的加快训练速度。

RunStats

记录环境的统计信息，例如步骤数和回合数。

TimeLimit

如果运行时间超过了最大步骤数，则会中断环境。

VideoWrapper

给环境录制一个视频。

要创建一个包装的环境，你必须创建一个包装器，然后将包装的环境传递给构造函数。就是这样！例如，以下代码将我们的环境包装在ActionRepeat包装器中，每个动作重复四次：

```
from tf_agents.environments.wrappers import ActionRepeat
repeating_env = ActionRepeat(env, times=4)
```

OpenAI Gym在gym.wrappers包中有一些自己的环境包装器。但是，它们旨在包装Gym环境，而不是TF-Agents环境，因此要使用它们，你必须先使用Gym包装器包装Gym环境，然后使用TF-Agents包装器包装所得到的环境。suite_gym.wrap_env（）函数会为你执行这些操作，只要你为其提供了Gym环境、Gym包装器列表和/或TF-Agents包装器列表。或者，s uite_gym.load（）函数会创建Gym环境并为你包装它，如果你提供了一些包装器。每个包装器可以不需要任何参数而被创建，因此如果要设置一些参数，则必须传递一个lambda。例如，以下代码创建一个Breakout环境，该环境将在每个回合中最多运行10 000个步骤，每个动作重复4次：

```
from gym.wrappers import TimeLimit

limited_repeating_env = suite_gym.load(
    "Breakout-v4",
    gym_env_wrappers=[lambda env: TimeLimit(env, max_episode_steps=10000)],
    env_wrappers=[lambda env: ActionRepeat(env, times=4)])
```

对于Atari环境，大多数使用它们的论文都采用了一些标准的预处理步骤，因此TFAgents提供了一个方便的AtariPreprocessing包装器来实现它们。这是它支持的预处理步骤的列表：

灰度和下采样

观测值将被转换为灰度并进行下采样（默认为84×84像素）。

最大池化

游戏的最后两帧使用1×1滤波器来进行最大池化。这是为了消除由于Atari 2600可以在每帧中显示的数量有限的精灵而在某些Atari游戏中发生的闪烁。

跳帧

智能体只能看到游戏的每n个帧（默认情况下n=4），动作每帧重复，收集所有的奖励。从智能体的角度来看，这有效地加快了游戏的速度，并且由于奖励的延迟减少，因此也加快了训练速度。

失去生命的结束

在某些游戏中，奖励只是基于分数，因此智能体不会因为失去生命而立即受到惩罚。一种解决方案是每当失去生命时立即结束游戏。关于此策略的实际好处还有争议，因此默认情况下处于关闭状态。

由于默认的Atari环境已经应用了随机跳帧和最大池化，我们需要加载称为"Breakout-NoFrameskip-v4"的原始，不跳帧的变体。此外，Breakout游戏中的单个帧不足以知道小球的方向和速度，这就使智能体很难正确地玩游戏（除非它是RNN智能体，它保留了步骤之间的一些内部状态）。解决此问题的一种方法是使用环境包装器，该包装器会输出由沿通道维度彼此堆叠的多个帧组成的观察结果。该策略由

FrameStack4包装器实现，该包装器返回四帧的堆叠。让我们创建包装好的Atari环境！

```
from tf_agents.environments import suite_atari
from tf_agents.environments.atari_preprocessing import AtariPreprocessing
from tf_agents.environments.atari_wrappers import FrameStack4

max_episode_steps = 27000 # <=> 108k ALE frames since 1 step = 4 frames
environment_name = "BreakoutNoFrameskip-v4"

env = suite_atari.load(
    environment_name,
    max_episode_steps=max_episode_steps,
    gym_env_wrappers=[AtariPreprocessing, FrameStack4])
```

所有这些预处理的结果如图18-12所示。你会看到分辨率降低了很多，但是足以玩游戏了。此外，帧沿通道维度堆叠，因此红色代表三步前的帧，绿色代表两步前的帧，蓝色代表前一帧，粉红色代表当前帧[2]。从这个单一观察中，智能体可以看到球正朝着左下角移动，并且应该继续将拍子向左移动（与之前的步骤一样）。

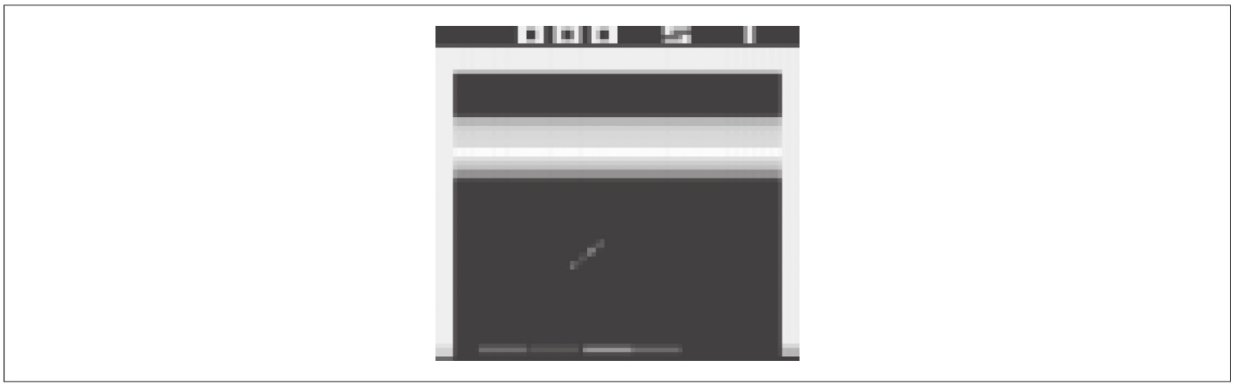


图18-12：预处理的Breakout观察值

最后，我们可以将环境包装在TFPyEnvironment中：

```
from tf_agents.environments.tf_py_environment import TFPyEnvironment

tf_env = TFPyEnvironment(env)
```

在TensorFlow图中可以使用这个环境（在后台，这个类依赖于tf.py_function（），该函数允许图调用任意的Python代码）。多亏了TFPyEnvironment类，TF-Agents支持纯Python环境和基于TensorFlow的环境。更一般而言，TF-Agent支持并提供纯Python和基于TensorFlow的组件（智能体、重播缓冲区、指标等）。

现在我们已经有了一个不错的Breakout环境，具有所有合适的预处理和TensorFlow支持，我们必须创建DQN智能体，和我们需要进行对其训练的其他组件。让我们看一下我们要构建的系统架构。

18.12.5　训练架构

TF-Agents训练程序通常分为两个并行的部分，如图18-13所示：在左侧，驱动者使用收集策略选择操作来探索环境，然后收集轨迹（即经验），然后将其发送给观察者，从而将其保存在重播缓冲区；在右侧，智能体从重播缓冲区中提取了一批次的轨迹，并训练了一些网络，这些网络是收集策略所使用的。简而言之，左侧部分探索环境并收集轨迹，而右侧部分学习并更新收集策略。

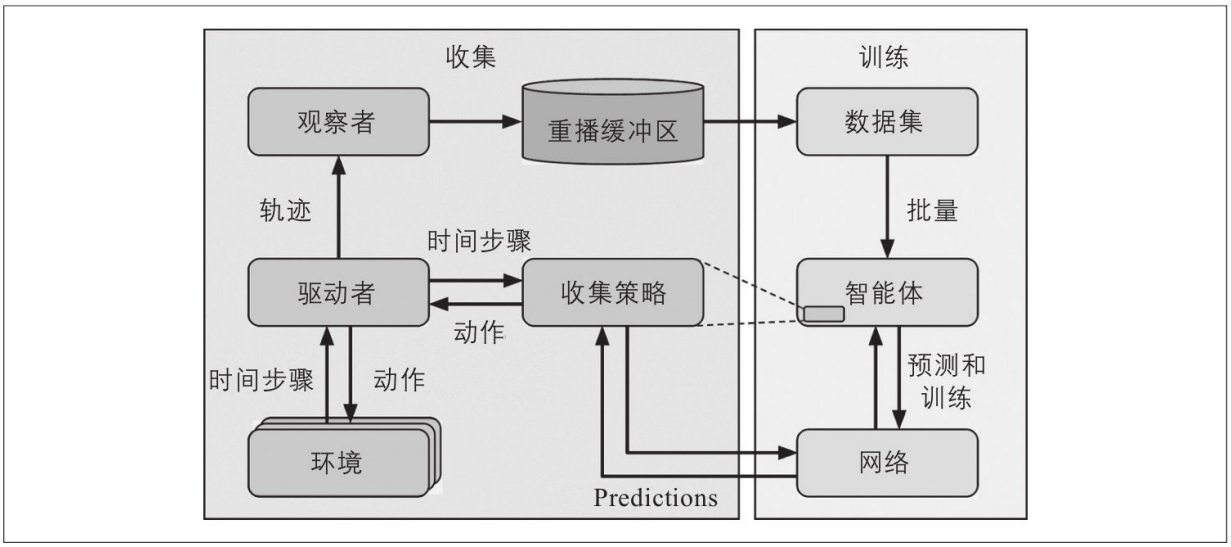


图18-13：典型的TF-Agents训练架构

这个图提出了几个问题，我在这里尝试回答一下：

- 为什么会有多个环境？通常，你不希望探索单个环境，而是希望驱动者能并行探索环境的多个副本，利用所有CPU内核的功能，充分利用GPU训练，并为训练算法提供较少的相关轨迹。

- 什么是轨迹？它是从一个时间步长到下一个时间步长转换的简明表示，或者是从时间步长n到时间步长n+t的一系列连续转换。驱动者收集的轨迹被传递给观察者，被保存在重播缓冲区中，随后由智能体对其进行采样并用于训练。

- 我们为什么需要观察者？驱动者不能直接保存轨迹吗？确实可以，但这会使架构不太灵活。例如，如果你不想使用重播缓冲区怎么办？如果你想将轨迹用于其他方面，例如计算指标，该怎么办？实际上，观察者就是将轨迹作为参数的任何函数。你可以使用观察者将轨迹保存到重播缓冲区中，或将它们保存到TFRecord文件中（见第13章），或用于计算指标或其他任何内容。此外，你可以把多个观察者传递给驱动者，然后驱动者将轨迹广播给所有观察者。

 尽管这种架构是最常见的，但是你可以随意自定义它，甚至可以用自己的组件替换某些组件。实际上，除非你正在研究新的RL算法，否则你很想为你的任务使用自定义环境。为此，你只需要在tf_agents.environments.py_environment包中创建一个继承自PyEnvironment类的自定义类，并重写适当的方法，例如action_spec（）、observation_spec（）、_reset（）和_step（）（有关示例，请参见notebook的“Creating a Custom TF_Agents Environment”部分）。

现在，我们创建了所有这些组件：首先是深度Q网络，然后是DQN智能体（它将负责创建收集策略），然后是重播缓冲区和观察者以对其进行写入，然后是一些训练指标，然后是驱动者，最后是数据集。一旦所

有组件都准备就绪，我们就使用一些初始轨迹填充重放缓冲区，然后运行主训练循环。因此让我们从创建深度Q网络开始。

18.12.6 创建深度Q网络

TF-Agents库在tf_agents.networks包及其子包中提供了许多网络。我们使用了tf_agents.networks.q_network.QNetwork类：

```
from tf_agents.networks.q_network import QNetwork

preprocessing_layer = keras.layers.Lambda(
                          lambda obs: tf.cast(obs, np.float32) / 255.)
conv_layer_params=[(32, (8, 8), 4), (64, (4, 4), 2), (64, (3, 3), 1)]
fc_layer_params=[512]

q_net = QNetwork(
    tf_env.observation_spec(),
    tf_env.action_spec(),
    preprocessing_layers=preprocessing_layer,
    conv_layer_params=conv_layer_params,
    fc_layer_params=fc_layer_params)
```

这个QNetwork将观察值作为输入，每个动作输出一个Q值，因此我们必须给它观察值和动作的规范。它从预处理层开始：一个简单的Lambda层，将观察值转换为32位浮点并将其归一化（值范围为0.0到1.0）。观察值包含无符号字节，其使用的内存空间比32位浮点数少4倍，这就是我们先前没有将观察值转换为32位浮点的原因。我们想在重播缓冲区中节省RAM。接下来，这个网络应用三个卷积层：第一层有32个8×8滤波器，使用步幅为4；第二层有64个4×4的滤波器和步幅为2；第三层有64个3×3的滤波器，步幅为1。最后，它应用了一个具有512个单元的密集层，紧接一个具有4个单元的密集输出层，每个Q值一个输出（每个动作一个值）。默认情况下，除输出层外，所有卷积层和密集层都使用ReLU激活函数（你可以通过设置activation_fn参数来更改此函数）。输出层不使用任何激活函数。

在网络内部，一个QNetwork由两部分组成：一个处理观察结果的编码网络，紧跟一个密集输出层，每个动作输出一个Q值。TF-Agent的

EncodingNetwork类实现了各种智能体中的神经网络架构（见图18-14）。

它可能有一个或多个输入。例如，如果每个观测值由一些传感器数据和来自摄像机的图像组成，则你有两个输入。每个输入可能需要一些预处理步骤，在这种情况下，你可以通过preprocessing_layers参数来指定Keras层的列表，每个输入有一个预处理层，网络会把每个层应用于相应的输入（如果一个输入需要多个预处理，你可以传递整个模型，因为一个Keras模型始终可以看作一个层）。如果有两个或两个以上输入，你还必须通过preprocessing_combiner参数传递一个额外的层，来将预处理层的输出合并为一个输出。

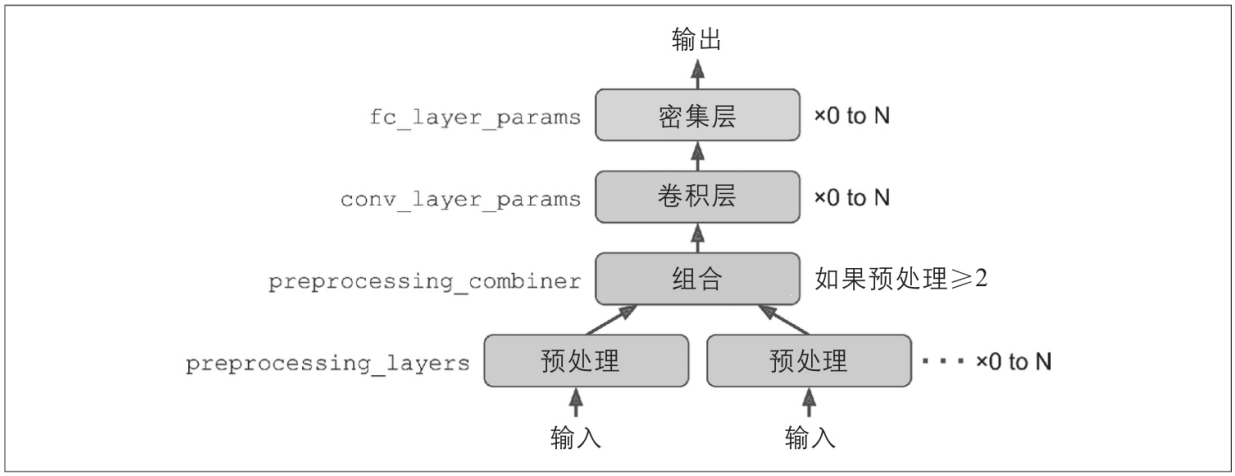


图18-14：编码网络的架构

接下来，编码网络可以选择按照顺序应用卷积列表，如果你通过conv_layer_params参数指定它们的参数。这必须是一个由三元组（每个卷积层一个）组成的列表，表明滤波器的数量、内核大小和步幅。在这些卷积层之后，如果设置了fc_layer_params参数，则编码网络可以选择应用一系列密集层：它必须是一个包含每个密集层的神经元数量的列表。可选择地，如果你想要在每个密集层之后应用dropout，则可以通过dropout_layer_params参数传递dropout率的列表（每个密集层一个）。QNetwork得到编码网络的输出，并将其传递到密集输出层（每个动作一个单元）。

 QNetwork类足够灵活可以构建许多不同的架构，但是如果你需要额外的灵活性，则总是可以构建自己的网络：扩展tf_agents.networks.Network类并将其像常规的自定义Keras层一样实现。tf_agents.networks.Network类是keras.layers.Layer类的子类，它添加了某些智能体所需要的一些功能，例如可以轻松创建网络的浅拷贝（例如，复制网络的架构而不是其权重）。例如，DQNAgent使用它来创建在线模型的副本。

有了DQN之后，我们就可以构建DQN智能体了。

18.12.7 创建DQN智能体

TF-Agents在tf_agents.agents包及其子包中实现了许多类型的智能体，我们将使用tf_agents.agents.dqn.dqn_agent.DqnAgent类：

```
from tf_agents.agents.dqn.dqn_agent import DqnAgent

train_step = tf.Variable(0)
update_period = 4 # train the model every 4 steps
optimizer = keras.optimizers.RMSprop(lr=2.5e-4, rho=0.95, momentum=0.0,
                                     epsilon=0.00001, centered=True)
epsilon_fn = keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=1.0, # initial ε
    decay_steps=250000 // update_period, # <=> 1,000,000 ALE frames
    end_learning_rate=0.01) # final ε
agent = DqnAgent(tf_env.time_step_spec(),
                 tf_env.action_spec(),
                 q_network=q_net,
                 optimizer=optimizer,
                 target_update_period=2000, # <=> 32,000 ALE frames
                 td_errors_loss_fn=keras.losses.Huber(reduction="none"),
                 gamma=0.99, # discount factor
                 train_step_counter=train_step,
                 epsilon_greedy=lambda: epsilon_fn(train_step))
agent.initialize()
```

让我们看一下这段代码：

- 我们首先创建一个变量，该变量将计算训练步骤的数量。

• 然后，我们使用与2015年DQN论文相同的超参数构建优化器。

• 接下来，我们创建一个PolynomialDecay对象，该对象将为 ε 贪婪收集策略计算 ε 值，在给定当前训练步骤的情况下（通常用于降低学习率，因此使用这个参数的名称，但可以降低其他任何值）。在100万个ALE帧中，它将从1.0下降到0.01（2015 DQN论文中使用的值），这相当于250 000步，因为我们使用周期为4的跳帧。此外，每4个步骤（即16个ALE帧），我们会训练智能体，因此 ε 实际上会衰减超过62 500个训练步骤。

• 然后，我们构建DQNAge nt，并传递时间步长和动作规范、要训练的QNetwork、优化器、目标模型更新之间的训练步数、要使用的损失函数、折扣因子、t r a i n_s t e p变量以及一个返回 ε 值的函数（必须不带参数，这就是我们需要一个lambda来传递tr a in_ste p的原因）。请注意，损失函数必须为每个实例返回一个误差，而不是平均误差，这就是我们设置reduction="none"的原因。

• 最后，我们初始化智能体。

接下来，让我们构建重播缓冲区和将写入该缓冲区的观察者。

18.12.8 创建重播缓冲区和相应的观察者

TF-Agents库在tf_agents.replay_buffers包中提供了各种重播缓冲区的实现。有些是用纯Python编写的（它们的模块名称以py_开头），而另一些则基于TensorFlow编写（它们的模块名称以tf_开头）。我们将在tf_agents.replay_buffers.tf_uniform_replay_buffer包中使用TFUniformReplayBuffer类。它提供了具有均匀采样的重放缓冲区的高性能实现[3]：

```
from tf_agents.replay_buffers import tf_uniform_replay_buffer

replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
    batch_size=tf_env.batch_size,
    max_length=1000000)
```

让我们看一下这些参数：

data_spec

数据规范将被保存在重播缓冲区中。DQN智能体知道收集到的数据是什么样子，并通过其collect_data_spec属性使数据规范可以使用，这就是我们提供给重放缓冲区的内容。

batch_size

在每个步骤中将要被添加的轨迹数目。在我们的示例中，这是一个，因为驱动者一个步骤执行一个动作并收集一个轨迹。如果环境是一个批量处理环境，这意味着该环境在每个步骤中都要执行一批动作并返回一批观测值，那么驱动者将不得不在每个步骤中保存一批轨迹。由于我们使用的是TensorFlow重播缓冲区，因此它需要知道将要处理的批处理的大小（以构建计算图）。批处理环境的一个示例是ParallelPyEnvironment（来自tf_agents.environments.parallel_py_environment程序包）：它在单独的进程中并行运行多个环境（只要它们具有相同的动作和观察规范，它们就可以不同），并在每个步骤中使用一批动作并在环境中执行它们（每个环境一个动作），然后返回所有的结果观察值。

max_length

重放缓冲区的最大大小。我们创建了一个很大的重放缓冲区，可以存储一百万个轨迹（就像2015年DQN论文所做的那样）。这需要大量的RAM。

 当我们存储两个连续的轨迹时，它们包含了两个连续的观测值，每个观测值有四个帧（因为我们使用了FrameStack4包装器），不幸的是，第二个观测值中的四个帧中的三个是多余的（它们已经存在于第一个观测值中）。换句话说，我们使用了必需的四倍RAM。为避免这种情况，你可以改用tf_agents.replay_buffers.py_hashed_replay_buffer包中的PyHashedReplay Buffer：它会沿着观测值的最后一个轴对存储轨迹中的数据进行重复数据删除。

现在，我们可以创建观察者，它把轨迹写入重播缓冲区。观察者只是一个带有轨迹参数的函数（或可调用对象），因此我们可以直接把add_method（）方法（绑定到replay_buffer对象）用作观察者：

```
replay_buffer_observer = replay_buffer.add_batch
```

如果你要创建自己的观察者，你可以编写带有轨迹参数的任何函数。如果必须具有状态，则可以写一个带有__call__（self，trajectory）的类。例如，有一个简单的观察者，每次调用计数器都会增加一（轨迹表示两个回合之间的边界时，不算作一步），并且计数器每增加100，它就会显示一个总数（回车符\r和end=""确保显示的计数保持在同一行上）：

```
class ShowProgress:
    def __init__(self, total):
        self.counter = 0
        self.total = total
    def __call__(self, trajectory):
        if not trajectory.is_boundary():
            self.counter += 1
        if self.counter % 100 == 0:
            print("\r{}/{}".format(self.counter, self.total), end="")
```

现在让我们创建一些训练指标。

18.12.9 创建训练指标

TF-Agents在tf_agents.metrics包中实现了几个RL度量指标，其中一些是在纯Python中实现，有些是基于TensorFlow的。让我们创建一些来计算回合数目、采取的步骤数，最重要的是每个回合的平均回报和平均回合的长度：

```
from tf_agents.metrics import tf_metrics

train_metrics = [
    tf_metrics.NumberOfEpisodes(),
    tf_metrics.EnvironmentSteps(),
    tf_metrics.AverageReturnMetric(),
    tf_metrics.AverageEpisodeLengthMetric(),
]
```

 对奖励打折扣对于训练或实现策略是有意义的，因为它可以使即刻奖励与将来的奖励重要性之间取得平衡。但是，当回合结束后，我们可以通过汇总未折扣的奖励来评估整体效果。因此，AverageReturnMetric会为每个回合计算未折扣奖励的总和，并且会跟踪其遇到的所有回合中这些总和的平均值。

你随时可以通过调用它们的result（）方法（例如，train_metrics[0].result（））来得到每个指标的值。或者，你可以通过调用log_metrics（train_metrics）来记录所有的指标（此函数位于tf_agents.eval.metric_utils包中）：

```
>>> from tf_agents.eval.metric_utils import log_metrics
>>> import logging
>>> logging.get_logger().set_level(logging.INFO)
>>> log_metrics(train_metrics)
[...]
NumberOfEpisodes = 0
EnvironmentSteps = 0
```

```
AverageReturn = 0.0
AverageEpisodeLength = 0.0
```

接下来，让我们创建收集驱动者。

18.12.10 创建收集驱动者

如图18-13所示，驱动者是使用给定策略来探索环境，收集经验并将其广播给一些观察者的对象。在每个步骤中，都会发生以下情况：

• 驱动者将当前时间步骤传递到收集策略，该策略使用该时间步骤选择一个动作，并返回包含该动作的动作步骤对象。

• 然后，驱动者将动作传递到环境，该环境返回下一个时间步骤。

• 最后，驱动者创建一个轨迹对象来表示此转换并将其广播给所有观察者。

某些策略，例如RNN策略，是有状态的：它们基于给定的时间步骤和它们自己的内部状态来选择动作。有状态策略会在操作步骤中返回其自己的状态以及所选的操作。然后，驱动程序将在下一时间步将此状态传递回策略。此外，驱动程序将策略状态保存到轨迹（在policy_info字段中），因此它最终位于重播缓冲区中。这对于训练有状态策略非常重要：当智能体对轨迹进行采样时，必须将策略的状态设置为采样时间步时的状态。

同样，如前所述，环境可以是批处理环境，在这种情况下，驱动程序将批处理时间步长传递给策略（即一个时间步长对象，其中包含一批观察值、一批步骤类型、一批奖励和一批折扣，所有四批大小相同）。驱动程序还传递了一批先前的策略状态。然后，该策略将返回一个包含一批操作和一批策略状态的批处理操作步骤。最后，驱动程序会创建一个批处理的轨迹（即包含一批步骤类型、一批观察值、一批动作、一批

奖励，更一般地说是针对每个轨迹属性的一批轨迹，其中所有批次相同的尺寸）。

有两个主要的驱动者：DynamicStepDriver和DynamicEpisodeDriver。第一个收集给定步骤数的经验，第二个收集给定回合数的经验。我们想要为每次训练迭代收集四个步骤的经验（就像在2015年DQN论文中所做的那样），因此让我们创建一个DynamicStepDriver：

```
from tf_agents.drivers.dynamic_step_driver import DynamicStepDriver

collect_driver = DynamicStepDriver(
    tf_env,
    agent.collect_policy,
    observers=[replay_buffer_observer] + training_metrics,
    num_steps=update_period) # collect 4 steps for each training iteration
```

我们给它提供一个可以使用的环境、智能体的收集策略、一个观察者列表（包括重播缓冲区观察者和训练指标），最后是要运行的步骤数（在这个情况下为4）。现在我们可以通过调用run（）方法来运行它，但是最好使用纯随机策略收集的经验来预热重播缓冲区。为此，我们可以使用RandomTFPolicy类并创建第二个驱动者，该驱动者会运行这个策略20 000个步骤（相当于2015年DQN论文中的80 000个模拟帧）。我们可以使用ShowProgress观察者来显示进度：

```
from tf_agents.policies.random_tf_policy import RandomTFPolicy

initial_collect_policy = RandomTFPolicy(tf_env.time_step_spec(),
                                        tf_env.action_spec())
init_driver = DynamicStepDriver(
    tf_env,
    initial_collect_policy,
    observers=[replay_buffer.add_batch, ShowProgress(20000)],
    num_steps=20000) # <=> 80,000 ALE frames
final_time_step, final_policy_state = init_driver.run()
```

我们几乎准备好开始训练循环了！我们只需要最后一个组件：数据集。

18.12.11 创建数据集

要从重播缓冲区中采样一批轨迹，需要调用其get_next（）方法。这将返回这一批轨迹以及一个包含样本标识符及其采样概率的BufferInfo对象（这对于某些算法可能有用，例如PER）。例如，以下代码是采样两个轨迹（子回合）的一小批量，每个轨迹包含三个连续的步骤。这些子回合如图18-15所示（每行包含一个回合的三个连续步骤）：

```
>>> trajectories, buffer_info = replay_buffer.get_next(
...     sample_batch_size=2, num_steps=3)
...
>>> trajectories._fields
('step_type', 'observation', 'action', 'policy_info',
 'next_step_type', 'reward', 'discount')
>>> trajectories.observation.shape
TensorShape([2, 3, 84, 84, 4])
>>> trajectories.step_type.numpy()
array([[1, 1, 1],
       [1, 1, 1]], dtype=int32)
```

trajectories对象是具有7个字段的命名元组。每个字段都包含一个张量，其前两个维度为2和3（因为有两个轨迹，每个轨迹具有三个步长）。这就解释了为什么观察值字段的形状为[2，3，84，84，4]：这是两个轨迹，每个轨迹有三个步骤，而每一步的观察值都是84×84×4。类似地，step_type张量的形状为[2，3]：在此示例中，两个轨迹在一个回合的中间都包含三个连续步骤（类型1、1、1）。在第二个轨迹中，你几乎看不到第一个观测值的左下角的球，并且在接下来的两个观测值中球消失了，因此智能体即将要失去生命，但是该回合不会立即结束，因为还有几条生命。

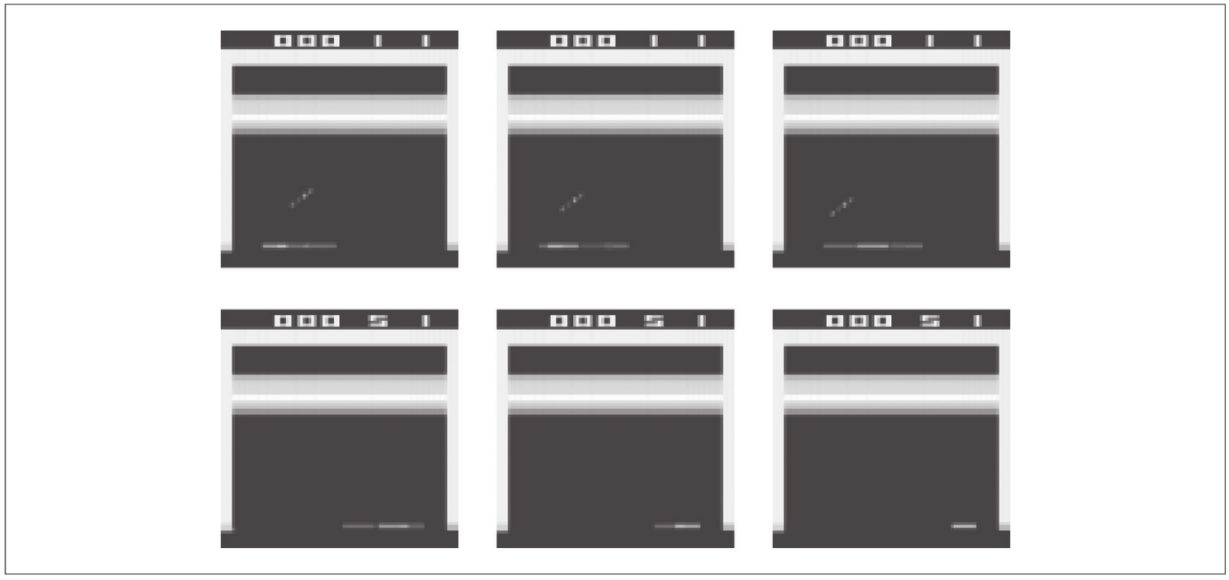


图18-15：两个包含三个连续步骤的轨迹

每个轨迹都是连续时间步骤和动作步骤序列的简明表示，旨在避免冗余。为何如此？如图18-16所示，转换n由时间步骤n、动作步骤n和时间步骤n+1组成，而转换n+1由时间步骤n+1、动作步骤n+1和时间步长n+2组成。如果我们将这两个转换直接存储在重播缓冲区中，则时间步骤n+1会有重复。为了避免这种重复，第n个轨迹步骤仅包含时间步骤n的类型和观察结果（不包括其奖励和折扣），不包含时间步骤n+1的观察结果（但是，它确实包含了下一步骤的类型的副本，这是唯一的重复）。

图18-16：轨迹、转换、时间步骤和动作步骤

因此，如果你有一批轨迹，其中每条轨迹有t+1步（从时间步骤n到时间步骤n+t），那么它包含从时间步骤n到时间步骤n+t的所有数据，除了从时间步骤n的奖励和折扣（但包含时间步长n+t+1的奖励和折扣）。这表示了t个转换（n到n+1，n+1到n+2，…，n+t－1到n+t）。

tf_agents.trajectories.trajectory模块中的to_transition（）函数将批处理的轨迹转换为包含了批处理的time_step，批处理的action_step和批处理的next_time_step的列表。请注意，第二个维度是2而不是3，因为在t+1个时间步骤之间没有t转换（如果你有点困惑，请不要担心；你会掌握的）：

```
>>> from tf_agents.trajectories.trajectory import to_transition
>>> time_steps, action_steps, next_time_steps = to_transition(trajectories)
>>> time_steps.observation.shape
TensorShape([2, 2, 84, 84, 4]) # 3 time steps = 2 transitions
```

 采样轨迹实际上可能会有两个（或更多个）回合重叠的情况！在这种情况下，它将包含边界转换，即step_type等于2（结束）和next_step_type等于0（开始）的转换。当然，TF-Agent可以正确处理这样的轨迹（例如，在遇到边界时重置策略状态）。轨迹的is_boundary（）方法返回一个张量，表明每个步骤是否为边界。

对于我们的主训练循环，我们使用tf.data.Dataset而不是调用get_next（）方法。这样我们就可以受益于Data API的强大功能（例如，并行和预取）。为此，我们调用重播缓冲区的as_dataset（）方法：

```
dataset = replay_buffer.as_dataset(
    sample_batch_size=64,
```

```
    num_steps=2,
    num_parallel_calls=3).prefetch(3)
```

我们在每个训练步骤中抽样64个轨迹的批次（如2015年DQN论文中所述），每个包含2个步骤（即2个步骤=1个完全变换，包括下一步的观察）。该数据集将并行处理三个单元，并预取三个批次。

 对于同策略（on-policy）的算法，例如策略梯度，应将每种经验采样一次，在训练中使用，然后丢弃。在这种情况下，你仍然可以使用重播缓冲区，但是可以不使用数据集，你可以在每次训练迭代时调用重播缓冲区的collect_all（）方法，来得到包含迄今为止记录的所有轨迹的张量，然后使用它们来执行训练步骤，最后通过调用其clear（）方法清除重播缓冲区。

现在我们已经准备好所有组件，可以训练模型了！

18.12.12　创建训练循环

为了加快训练速度，我们把主要函数转换为TensorFlow函数。为此，我们使用tf_agents.utils.common.function（）函数，它包装了tf.function（），并提供一些额外的实验性选项：

```
from tf_agents.utils.common import function

collect_driver.run = function(collect_driver.run)
agent.train = function(agent.train)
```

让我们创建一个小函数，它将运行主训练循环：

```
def train_agent(n_iterations):
    time_step = None
```

```
    policy_state = agent.collect_policy.get_initial_state(tf_env.batch_size)
    iterator = iter(dataset)
    for iteration in range(n_iterations):
        time_step, policy_state = collect_driver.run(time_step, policy_state)
        trajectories, buffer_info = next(iterator)
        train_loss = agent.train(trajectories)
        print("\r{} loss:{:.5f}".format(
            iteration, train_loss.loss.numpy()), end="")
        if iteration % 1000 == 0:
            log_metrics(train_metrics)
```

该函数首先向收集策略询问其初始状态（给定环境批处理大小，在这个情况下为1）。由于该策略是无状态的，因此将返回一个空的元组（因此我们可以写policy_state=（））。接下来，我们在数据集上创建一个迭代器，然后运行训练循环。在每次迭代中，我们都调用驱动者的run（）方法，并向其传递当前时间步长（最初为None）和当前策略状态。它会运行收集策略并收集四个步骤的经验（就像我们之前配置的那样），将收集到的轨迹广播到重播缓冲区和度量指标。接下来，我们从数据集中采样一批轨迹，并将其传递给智能体的train（）方法。它返回一个train_loss对象，该对象可能会根据智能体的类型而有所不同。接下来，我们显示迭代次数和训练损失，并且每进行1000次迭代，我们就会记录所有指标。现在，你只需调用train_agent（）进行一定数量的迭代，就可以看到智能体逐渐学会玩Breakout！

```
train_agent(10000000)
```

这将需要大量计算能力和大量耐心（可能需要数小时甚至数天，具体取决于你的硬件），另外，你可能需要使用不同的随机种子多次运行该算法才能获得良好的结果，但是一旦完成后，智能体是超过人类的（至少在Breakout上）。你还可以尝试在其他Atari游戏上训练DQN智能体：它可以在大多数动作游戏中获得超过人类的技能，但在一些长故事情节的游戏中却不那么出色[4]。

[1] 如果你不了解这款游戏，它很简单：一个球反弹并在碰到砖块时将其打碎。你可以控制屏幕底部附近的板子。板子可以向左或向右移动，你必须让球打破每块砖，同时防止其触碰屏幕底部。

[2] 因为只有三种原色，所以不能仅显示具有4个颜色通道的图像。因此，我将最后一个通道与前三个通道结合在一起来得到此处表示的RGB图像。粉色实际上是蓝色和红色的混合，但是该智能体看到4个独立的通道。

[3] 在撰写本书时，还没有优先经验重播缓冲区，但它很快就会开源。

[4] 有关该算法在各种Atari游戏上的性能的比较，请参见DeepMind 2015年论文中的图3。

18.13 一些流行的RL算法概述

在结束本章之前，让我们快速浏览一些流行的RL算法：

Actor-Critic算法

一系列RL算法，结合了策略梯度和深度Q网络。一个Actor-Critic智能体包含两个神经网络：策略网络和DQN。通过从智能体的经验中学习，可以对DQN进行常规的训练。与常规的策略梯度相比，策略网络的学习方式有所不同（学习速度更快）：不是通过经历多个回合来估算每个动作的值，然后对每个动作的未来折扣奖励进行相加，最后对它们进行归一化，智能体（actor）依赖于根据DQN（critic）估算得到的动作值。有点像运动员（智能体）在教练（DQN）的帮助下学习。

Asynchronous Advantage Actor-Critic[1]（A3C）

DeepMind的研究人员于2016年推出的一种重要的Actor-Critic变体，多个智能体并行学习，探索环境的不同副本。每个智能体以固定的时间间隔但不同步（因此得名）将一些权重更新推送到主网络，然后从该主网络获取最新的权重。因此，每个智能体都有助于改善主网络并受益于其他智能体学到的知识。此外，DQN不会估算Q值，而是估算每个动作的优势（名称中的第二个A），这样可以稳定训练。

Advantage Actor-Critic（A2C）

A3C算法的一种变体，它去除了异步性。所有模型更新都是同步的，因此梯度更新是在较大的批次上执行的，这使模型可以更好地利用GPU的功能。

Soft Actor-Critic[2]（SAC）

Tuomas Haarnoja和其他加州大学伯克利分校的研究人员于2018年提出了一种Actor-Critic变体。它不仅学习奖励，而且还最大化其动作的熵。换句话说，它试图尽可能地不可预测，同时仍然获得尽可能多的奖励。这会鼓励智能体探索环境，从而加快训练速度，并使得当DQN产生不完美的估计值时，重复执行同一动作的可能性降低。该算法已证明了有惊人的效率（与之前学习非常慢的所有算法相反）。

SAC在TF-Agents中可用。

Proximal Policy Optimization（PPO）[3]

一种基于A2C的算法，修改了损失函数以避免过多的权重更新（这通常会导致训练不稳定）。PPO是对先前的Trust Region Policy Optimization[4]（TRPO）算法的简化，该算法由John Schulman和其他OpenAI研究人员提出。OpenAI于2019年4月发布了他们的AI，该AI基于PPO算法，名为OpenAI Five，在多人游戏Dota 2中击败了世界冠军。PPO在TF-Agents中也可用。

Curiosity-based exploration[5]

RL中反复出现的问题是奖励的稀疏性，这使得学习非常缓慢且效率低下。DeepakPathak和加州大学伯克利分校（UC Berkeley）的其他研究人员提出了一种令人振奋的方法来解决该问题：为什么不忽略这些奖励，而只是让智能体极端好奇地探索环境？因此，奖励成为智能体所内在固有的，而不是来自环境。同样，激发孩子的好奇心比单纯奖励孩子取得了好成绩更可能产生良好的效果。这是如何运作的呢？智能体不断尝试预测其动作的结果，并寻找结果与预测不符的情况。换句话说，它需要感到惊奇。如果结果是可预见的（无聊的），那么它将转移到其他地方。但是，如果结果是不可预测的，但是智能体发现自己无法控制它，则一段时间后也会感到无聊。只是出于好奇，作者成功地在许多视频游戏中训练了一个智能体：即使智能体没有因失

败而受到惩罚，游戏还是重新开始，这很无聊，因此它学习避免这种情况。

本章涵盖了许多主题：策略梯度、马尔可夫链、马尔可夫决策过程、Q学习、近似Q学习和深度Q学习及其主要变体（固定Q值目标、Double DQN、Duling DQN和优先经验重播）。我们讨论了如何使用TF-Agents来大规模训练智能体，最后，我们快速浏览了其他一些流行的算法。强化学习是一个巨大而激动人心的领域，每天都会涌现出新的想法和算法，因此我希望本章能激发你的好奇心：有一个广阔的世界值得探索！

[1] Volodymyr Mnih et al.，“Asynchonous Methods for Deep Reinforcement Learning”，Proceedings of the 33rd International Conference on Machine Learning（2016）：1928－1937.
[2] Tuomas Haarnoja et al.，“Soft Actor-Critic：Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”，Proceedings of the 35th International Conference on Machine Learning（2018）：1856－1865.
[3] John Schulman et al.，“Proximal Policy Optimization Algorithms”，arXiv preprint arXiv：1707.06347（2017）.
[4] John Schulman et al.，“Trust Region Policy Optimization”，Proceedings of the 32nd International Conference on Machine Learning（2015）：1889－1897.
[5] Deepak Pathak et al.，“Curiosity-Driven Exploration by Self-Supervised Prediction”，Proceedings of the 34th International Conference on Machine Learning（2017）：2778－2787.

18.14 练习题

1.你如何定义强化学习？它与常规的有监督学习或无监督学习有何不同？

2.你能想到本章中未提到的RL的三种可能的应用吗？对于每一个来说，环境是什么？什么是智能体？有哪些可能的动作？有什么奖励？

3.折扣因子是多少？如果你修改折扣因子，最优策略会改变吗？

4.你如何衡量强化学习智能体的性能？

5.什么是贡献分配问题？什么时候发生？如何缓解呢？

6.使用重播缓冲区有什么意义？

7.什么是异策略（off-policy）RL算法？

8.使用策略梯度来解决OpenAI Gym的LunarLander-v2环境。你需要安装Box2D依赖包（python3-m pip install-u gym[box2d]）。

9.使用TF-Agents和任何可用的算法来训练一个智能体，在SpaceInvaders-v4达到超过人类的水平。

10.如果你有大约100美元的余钱，你可以购买Raspberry Pi 3以及一些廉价的机器人组件，在Pi上安装TensorFlow，然后做你如何想做的事情！例如，查看Lukas Biewald的有趣文章，或者看一下GoPiGo或BrickPi。从简单的目标开始，例如使机器人转身来找到最亮的方向（如果有光传感器）或最近的物体（如果有声呐传感器），然后朝那

个方向移动。然后你可以使用深度学习：例如，如果机器人具有摄像头，则可以尝试实现物体检测算法，来检测人员并向其移动。你还可以尝试使用RL来使智能体自己学习如何使用电马达来实现这个目标。祝你玩得开心！

这些练习题的解答在附录A中提供。

第19章　大规模训练和部署TensorFlow模型

一旦有了一个可以做出惊人预测的漂亮模型，你将如何处理？好吧，你需要将其投入生产环境！这可能很简单，例如对一批数据运行模型，然后编写每晚运行该模型的脚本。但是它通常涉及很多事情。基础架构的各个部分可能需要在实时数据上使用此模型，在这种情况下，你可能希望将模型包装在Web服务中：这样基础架构的任何部分都可以使用简单的REST API（或某些其他协议）随时查询模型，如我们在第2章中讨论的。但是随着时间的流逝，你需要定期对模型进行新数据的重新训练并将更新后的版本推向生产环境。你必须处理模型的版本控制，从一个模型平稳过渡到下一个模型，以防万一出现问题时可能回滚到前一个模型，并可能并行运行多个不同的模型以进行A/B实验[1]。如果产品成功了，你的服务可能开始每秒获得大量查询（QPS），必须扩展它以支持负载。正如我们将在本章中看到的那样，扩展服务的一个很好的解决方案是在你自己的硬件基础结构上使用TF Serving或通过云服务（例如Google Cloud AI Platform）。它会负责有效地为你的模型提供服务，处理平滑的模型转换等。如果使用云平台，你还将获得许多额外的功能，例如强大的监视工具。

此外，如果有大量的训练数据和计算密集型模型，那么训练时间可能会很长。如果产品需要快速适应变化，那么很长的训练时间可能会成为一个障碍（想想新闻推荐系统宣传上周的新闻）。可能更重要的是，很长的训练时间将妨碍你尝试新的想法。在机器学习中（与许多其他领域一样），很难事先知道哪些想法会起作用，因此你应该尽可能快地尝试尽可能多的想法。加快训练速度的一种方法是使用硬件加速器，例如GPU或TPU。为了更快地运行，你可以在多台机器上训练模型，每台机器都配备多个硬件加速器。我们会看到，TensorFlow简单而强大的分布式策略API使这一过程变得容易。

在本章中，我们将研究如何将模型部署到TF Serving，然后再部署到Google Cloud AI平台。我们还将快速浏览将模型部署到移动应用程序、嵌入式设备和Web应用程序的过程。最后，我们将讨论如何使用GPU加快计算速度，以及如何使用分布式策略API在多个设备和服务器之间训练模型。本章有很多话题需要讨论，让我们开始吧！

[1] A/B实验包括在不同的用户子集上测试产品的两个不同版本，以便检查哪个版本最有效并获得其他洞见。

19.1 为TensorFlow模型提供服务

一旦训练完一个TensorFlow模型，就可以轻松地在任何Python代码中使用它：如果它是tf.keras模型，只需调用predict（）方法即可！但是随着基础架构的增长，有时最好将模型包装在一个小型服务中，该服务的唯一作用是进行预测并让其余基础架构查询（例如通过REST或gRPC API）[1]。这样可以使模型与基础架构的其余部分隔离，从而可以轻松切换模型版本或根据需要扩展服务（独立于基础架构的其余部分），执行A/B测试并确保所有软件组件依赖相同的模型版本。它还简化了测试和开发等。你可以使用任何所需的技术（例如使用Flask库）创建自己的微服务，但是当你可以使用TF SERVING时，为什么要重新发明轮子呢？

19.1.1 使用TensorFlow Serving

TF Serving是使用C++编写的非常有效的、经过测试的模型服务器。它可以承受很高的负载，可以为多个模型版本提供服务，还可以查看模型库并自动部署最新版本（见图19-1）。

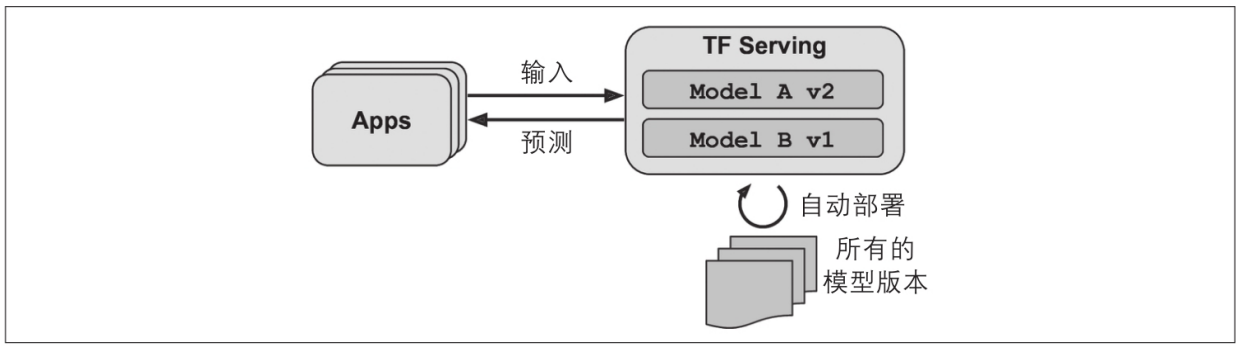


图19-1：TF Serving可以为多个模型提供服务，并自动部署每个模型的最新版本

假设你已经使用tf.keras训练了MNIST模型，并且希望将其部署到TF Serving。你要做的第一件事是将该模型导出为TensorFlow的

SavedModel格式。

导出保存的模型

TensorFlow提供了一个简单的tf.saved_model.save（）函数，可将模型导出为SavedModel格式。你所要做的就是为它提供模型，并指定其名称和版本号，然后该函数将保存该模型的计算图及其权重：

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit([...])

model_version = "0001"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

另外，你也可以只使用模型的save（）方法（model.save（model_path））：只要文件的扩展名不是.h5，就将使用SaveModel格式而不是HDF5格式保存模型。

将所有预处理层包括在导出的最终模型中通常是一个好主意，这样一旦将模型部署到生产环境中就可以以自然的形式读取数据。这避免了在使用该模型的应用程序内单独进行预处理。将预处理步骤捆绑在模型中还可以使其在以后进行更新，从而简化了模型，并降低了模型与其所需的预处理步骤之间不匹配的风险。

 由于SavedModel保存了计算图，因此只能与基于TensorFlow操作的模型一起使用，不包括tf.py_function（）操作（包装任意Python代码）的模型。它也排除了动态tf.keras模型（见附录G），因为这些模型无法转换为计算图。需要使用其他工具（例如Flask）为动态模型提供服务。

SavedModel代表模型的一个版本。它存储为包含一个save_model.pb文件的目录，该文件定义了计算图（表示为序列化的协议缓冲区），以及一个包含变量值的variables子目录。对于包含大量权重的模型，可以将这些变量值拆分为多个文件。SavedModel还包括一个assets子目录，该子目录可能包含其他数据，例如词汇表文件、类名或该模型的某些示例实例。目录结构如下（在此示例中我们没有使用assets）：

```
my_mnist_model
└─ 0001
    ├─ assets
    ├─ saved_model.pb
    └─ variables
        ├─ variables.data-00000-of-00001
        └─ variables.index
```

如你所料，可以使用tf.saved_model.load（）函数加载SavedModel。但是返回的对象不是Keras模型：它表示SavedModel，包括其计算图和变量值。你可以像使用函数一样使用它，进行预测（确保将输入作为正确类型的张量传递）：

```
saved_model = tf.saved_model.load(model_path)
y_pred = saved_model(tf.constant(X_new, dtype=tf.float32)))
```

或者，可以使用keras.model.load_model（）函数将此SavedModel直接加载到keras模型中：

```
model = keras.models.load_model(model_path)
y_pred = model.predict(tf.constant(X_new, dtype=tf.float32))
```

TensorFlow还带有一个小的saved_model_cli命令行工具，用于检查SavedModel：

```
$ export ML_PATH="$HOME/ml" # point to this project, wherever it is
$ cd $ML_PATH
$ saved_model_cli show --dir my_mnist_model/0001 --all
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:
signature_def['__saved_model_init_op']:
  [...]
signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['flatten_input'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 28, 28)
        name: serving_default_flatten_input:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['dense_1'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 10)
        name: StatefulPartitionedCall:0
  Method name is: tensorflow/serving/predict
```

一个SavedModel包含一个或多个元图。元图是一个计算图，外加一些函数签名定义（包括它们的输入和输出名称、类型和形状）。每个元图由一组标签标识。例如，你可能想要一个包含完整计算图的元图，包括训练操作（这可能被标记为"train"），而另一个元图包含仅具有预测操作的修剪后的计算图，包括一些特定GPU的操作（这可以标记为"serve""gpu"）。但是，当将tf.keras模型传递给tf.saved_model.save（）函数时，默认情况下，该函数会保存更简单的SavedModel：它保存一个标记为"serve"的元图，其中包含两个函数签名定义：初始化函数（__saved_model_init_op，你无须担心）和默认服务函数（serving_default）。保存tf.keras模型时，默认服务函数对应于模型的call（）函数，这当然可以做预测。

saved_model_cli工具还可以被用来做预测（用于测试，而不是用于生产）。假设你有一个NumPy数组（X_new），其中包含你要对其进行预测的三个手写数字图像。你首先需要将它们导出为NumPy的npy格式：

```
np.save("my_mnist_tests.npy", X_new)
```

接下来，使用如下的saved_model_cli命令：

```
$ saved_model_cli run --dir my_mnist_model/0001 --tag_set serve \
                      --signature_def serving_default \
                      --inputs flatten_input=my_mnist_tests.npy
[...] Result for output key dense_1:
[[1.1739199e-04 1.1239604e-07 6.0210604e-04 [...] 3.9471846e-04]
 [1.2294615e-03 2.9207937e-05 9.8599273e-01 [...] 1.1113169e-07]
 [6.4066830e-05 9.6359509e-01 9.0598064e-03 [...] 4.2495009e-04]]
```

该工具的输出包含3个实例中每个实例的10个类概率。现在你已经有一个可以使用的SavedModel了，下一步就是安装TF Serving。

安装TensorFlow Serving

安装TF Serving的方法有很多：使用Docker映像[2]、使用系统的程序包管理器、从源代码安装等。让我们使用Docker安装，它是TensorFlow团队强烈推荐的，因为它易于安装，不会与你的系统混淆，并且提供了高性能。你首先需要安装Docker。然后下载官方的TF Serving Docker映像：

```
$ docker pull tensorflow/serving
```

现在，你可以创建一个Docker容器来运行该映像：

```
$ docker run -it --rm -p 8500:8500 -p 8501:8501 \
             -v "$ML_PATH/my_mnist_model:/models/my_mnist_model" \
             -e MODEL_NAME=my_mnist_model \
             tensorflow/serving
[...]
2019-06-01 [...] loaded servable version {name: my_mnist_model version: 1}
2019-06-01 [...] Running gRPC ModelServer at 0.0.0.0:8500 ...
2019-06-01 [...] Exporting HTTP/REST API at:localhost:8501 ...
[evhttp_server.cc : 237] RAW: Entering the event loop ...
```

TF Serving正在运行。它加载了我们的MNIST模型（版本1），并通过gRPC（端口8500）和REST（端口8501）提供服务。下面是所有命令行选项的含义：

-it

使容器具有交互性（因此你可以按Ctrl-C停止它）并显示服务器的输出。

--rm

停止容器时将其删除（不会使中断的容器搞乱机器）。但是它不会删除映像。

-p 8500：8500

使Docker引擎将主机的TCP端口8500转发到容器的TCP端口8500。默认情况下，TF Serving使用此端口为gRPC API服务。

-p 8501：8501

将主机的TCP端口8501转发到容器的TCP端口8501。默认情况下，TF Serving使用此端口为REST API服务。

-v"$ML_PATH/my_mnist_model：/models/my_mnist_model"

使主机的$ML_PATH/my_mnist_model目录可用于容器的路径/models/mnist_model。

在Windows上，你可能需要在主机路径（而不是容器路径）中用\替换/。

-e MODEL_NAME=my_mnist_model

设置容器的MODEL_NAME环境变量，因此TF Serving知道要使用哪个模型。默认情况下，它将在/models目录中查找模型，并且它将自动服务找到的最新版本。

tensorflow/serving

这是要运行的映像的名称。

现在，让我们回到Python并查询该服务器，首先使用REST API，然后使用gRPC API。

通过REST API查询TF Serving

让我们从创建查询开始。它必须包含要调用的函数签名的名称，当然还要包含输入数据：

```
import json

input_data_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

请注意，JSON格式是100%基于文本的，因此X_new的NumPy数组格式必须转换为Python列表，然后格式化为JSON：

```
>>> input_data_json
'{"signature_name": "serving_default", "instances": [[[0.0, 0.0, 0.0, [...]
0.3294117647058824, 0.725490196078431, [...very long], 0.0, 0.0, 0.0, 0.0]]]}'
```

现在，我们通过发送HTTP POST请求将输入数据发送到TF Serving。这可以使用requests库轻松完成（它不是Python标准库的一部分，因此你需要首先安装它，例如使用pip）：

```
import requests

SERVER_URL = 'http://localhost:8501/v1/models/my_mnist_model:predict'
response = requests.post(SERVER_URL, data=input_data_json)
response.raise_for_status() # raise an exception in case of error
response = response.json()
```

响应是一个包含"predictions"键的字典。对应的值是预测列表。该列表是Python列表，因此我们将其转换为NumPy数组，并将其包含的浮点数舍入到第二个小数：

```
>>> y_proba = np.array(response["predictions"])
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 0.  , 0.  ],
       [0.  , 0.  , 0.99, 0.01, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.96, 0.01, 0.  , 0.  , 0.  , 0.  , 0.01, 0.01, 0.  ]])
```

我们得到预测了！该模型100%确信第一幅图像是7，99%确信第二幅图像是2，以及96%确信第三幅图像是1。

REST API非常好用和简单，当输入和输出数据不太大时，它可以很好地工作。而且几乎任何客户端应用程序都可以进行REST查询而没有其他依赖关系，而其他协议并非总是那么容易获得的。但是它基于JSON，是文本格式且相当冗长。我们必须将NumPy数组转换为Python列表，并且每个浮点数最终都表示为字符串。无论是在序列化/反序列化耗时上（将所有浮点数转换为字符串然后返回）还是在有效负载大小方面，这都是非常低效的：许多浮点数使用超过15个字符表示，这会把32位浮点数转化为超过120位！传输大NumPy数组时，这会导致高延迟和带宽占用[3]。因此让我们改用gRPC。

 传输大量数据时，最好使用gRPC API（如果客户端支持），因为它基于紧凑的二进制格式和有效的通信协议（基于HTTP/2框架）。

通过gRPC API查询TF Serving

gRPC API期望将序列化的PredictRequest协议缓冲区作为输入，并输出序列化的PredictResponse协议缓冲区。这些protobuf是tensorflow-serving-api库的一部分，你必须先安装该库（例如使用pip）。首先，让我们创建请求：

```
from tensorflow_serving.apis.predict_pb2 import PredictRequest

request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0]
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

此代码创建一个PredictRequest协议缓冲区，并填写要求的字段，包括模型名称（先前定义）、要调用的函数签名名称，最后以Tensor协议缓冲区的形式输入数据。tf.make_tensor_proto（）函数基于给定的张量或NumPy数组（在本例中为X_new）创建Tensor协议缓冲区。

接下来，我们将请求发送到服务器并获得响应（为此你需要grpcio库，可以使用pip安装该库）：

```
import grpc
from tensorflow_serving.apis import prediction_service_pb2_grpc

channel = grpc.insecure_channel('localhost:8500')
predict_service = prediction_service_pb2_grpc.PredictionServiceStub (channel)
response = predict_service.Predict(request, timeout=10.0)
```

代码非常简单直观：import后，我们在TCP端口8500上为本地主机创建了一个gRPC通信通道，然后在该通道上创建了一个gRPC服务，并使用它发送请求，超时时间为10秒（它将一直阻塞直到收到响应或超时时间到期为止）。在此示例中，通道是不安全的（不加密，不进行身份验证），但是gRPC和TensorFlow Serving也支持基于SSL/TLS的安全通道。

接下来，我们将PredictResponse协议缓冲区转换为张量：

```
output_name = model.output_names[0]
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto
```

如果你运行此代码并打印y_proba.numpy（）.round（2），你将获得与之前完全相同的估计类概率。这就是全部：只需几行代码，你现在就可以使用REST或gRPC远程访问TensorFlow模型。

部署新的模型版本

现在，让我们创建一个新的模型版本，并将SavedModel导出到my_mnist_model/0002目录，就像之前一样：

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit([...])

model_version = "0002"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

TensorFlow Serving以固定间隔（延迟是可配置的）来检查新模型版本。如果找到一个，它将自动优雅地处理过渡：默认情况下，它将使

用以前的模型版本来响应待处理的请求（如果有），同时使用新的版本来处理新的请求[4]。一旦每个待处理的请求都得到了答复，将卸载先前的模型版本。你可以在TensorFlow Serving日志中看到这一点：

```
[...]
reserved resources to load servable {name: my_mnist_model version: 2}
[...]
Reading SavedModel from: /models/my_mnist_model/0002
Reading meta graph with tags { serve }
Successfully loaded servable version {name: my_mnist_model version: 2}
Quiescing servable version {name: my_mnist_model version: 1}
Done quiescing servable version {name: my_mnist_model version: 1}
Unloading servable version {name: my_mnist_model version: 1}
```

这种方法提供了平稳的过渡，但是可能会使用过多的RAM（尤其是GPU RAM，通常是有限的）。在这种情况下，你可以配置TF Serving，以使它用先前模型版本处理完所有待处理请求，并在加载和使用新模型版本之前将其卸载。此配置将避免同时加载两个模型版本，但是该服务在短期内不可用。

如你所见，TF Serving使部署新模型变得非常简单。此外，如果发现版本2的运行效果不理想，则可以回滚到版本1，就像删除my_mnist_model/0002目录一样简单。

 TF Serving的另一个重要功能是其自动批处理功能，你可以在启动时使用--enable_batching选项来激活它。当TF Serving在短时间内（延迟是可以配置的）接收到多个请求时，它将在使用模型之前自动将它们一起进行批处理。通过利用GPU的功能，可以显著提高性能。模型返回预测后，TF Serving会将每个预测发送到正确的客户端。你可以通过增加批处理延迟以一点延迟来换取更大的吞吐量（请参阅--batching_parameters_file选项）。

如果希望每秒获得许多查询，则需要在多个服务器上部署TF Serving并对查询进行负载均衡（见图19-2）。这需要在这些服务器之间部署和管理许多TF Serving容器。解决该问题的一种方法是使用诸如Kubernetes之类的工具，该工具是一个开放源代码系统，用于简化许多服务器之间的容器平衡。如果你不想购买、维护和升级所有硬件基础架构时，可以在云平台上使用虚拟机，例如Amazon AWS、Microsoft Azure、Google Cloud Platform、IBM Cloud、Alibaba Cloud、Oracle Cloud或其他一些Platform-as-a-Service（PaaS）。管理所有这些虚拟机，处理容器平衡（甚至在Kubernetes的帮助下），查看TF Serving配置，调整和监视——所有这些都是全职工作。幸运的是，某些服务提供商可以为你解决所有这些问题。在本章中，我们将使用Google Cloud AI平台，因为它是当今唯一具有TPU的平台，它支持TensorFlow 2，提供了一套不错的AI服务套件（例如AutoML、视觉API、自然语言API），而且我对其最有经验。但该领域还有其他提供商，例如Amazon AWS SageMaker和Microsoft AI Platform，它们也能够为TensorFlow模型提供服务。

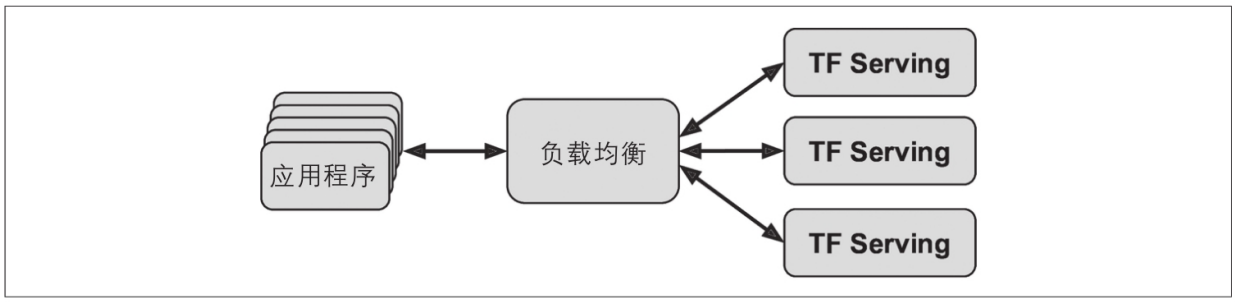


图19-2：通过负载平衡扩大TF Serving的规模

现在，让我们看看如何在云上提供我们出色的MNIST模型！

19.1.2 在GCP AI平台上创建预测服务

在部署模型之前，需要进行一些设置：

1.登录到你的Google账户，然后转到Google Cloud Platform（GCP）控制台（见图19-3）。

如果你没有Google账户，则必须创建一个。

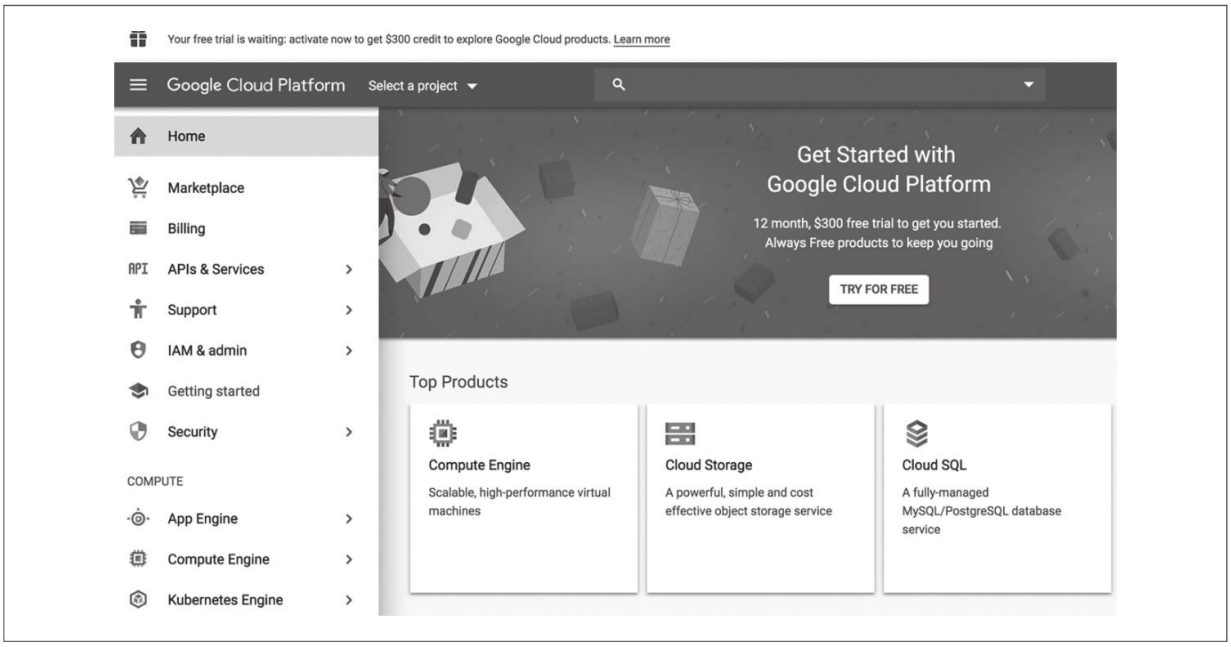


图19-3：Google Cloud Platform控制台

2.如果你是第一次使用GCP，则必须阅读并接受条款和条件。如果需要，请单击“Tour Console”。在撰写本书时，为新用户提供了免费试用版，包括可在12个月内使用的价值300美元的GCP信用额。你只需支付其中的一小部分即可完成本章要使用的服务。注册免费试用版后，你仍然需要填写一个付款信息并输入你的信用卡号：该卡用于验证（可能是为了避免人们多次使用免费试用版），但是不会向你收费。根据要求激活并升级你的账户。

3.如果你以前使用过GCP并且免费试用期已过，那么本章要使用的服务会使你花费一些钱。不会太多，尤其是当你记得不再需要服务时将其关闭。在运行任何服务之前，请确保你了解并同意定价条件。如果服务的最终费用超出你的预期，我在此不承担任何责任！另外请确保你的

付款账户有效。要进行检查，请打开左侧的导航菜单，然后点击“Billing”，确保你已设置付款方式并且该付款账户处于激活状态。

4.GCP中的每个资源都属于一个项目。这包括你可能使用的所有虚拟机、你存储的文件以及你运行的训练任务。创建账户时，GCP会自动为你创建一个名为“My First Project”的项目。如果需要，可以通过转到项目设置来更改其显示名称：在导航菜单中（位于左侧屏幕），选择“IAM&admin→Settings”，更改项目的显示名称，然后单击“Save”。请注意，该项目还具有唯一的ID和编号。你可以在创建项目时选择项目ID，但以后不能更改。项目编号是自动生成的，不能更改。如果要创建新项目，请单击页面顶部的项目名称，然后单击“New Project”并输入项目ID。确保此新项目的账单处于激活状态。

 当你知道只需要几个小时的服务时，请始终设置一个警报以提醒自己关闭服务，否则可能会使它们运行数天或数月，从而招致高昂的潜在费用。

5.现在你已激活了GCP账户并已激活了账单，就可以开始使用这些服务了。你需要的第一个是Google Cloud Storage（GCS）：你将在其中放置SavedModel、训练数据等。在导航菜单中，向下滚动到“Storage”部分，然后单击“Storage→Browser”。你的所有文件都将放入一个或多个存储桶中。点击“Create Bucket”，然后选择存储桶名称（你可能需要先激活Storage API）。GCS为存储桶使用一个全球范围的命名空间，因此像“machine-learning”这样的简单名称很可能不可用。确保存储区名称符合DNS命名约定，因为它可能在DNS记录中使用。此外存储桶名称是公共名称，因此不要在其中放置任何私有内容。通常将域名或公司名称用作前缀以确保唯一性，或者仅使用随机数用作名称的一部分。选择你要存储桶的保存位置，默认情况下其余选项应该都可以。然后单击“Create”。

6.将你之前创建的my_mnist_model文件夹（包括一个或多个版本）上传到存储桶。为此只需转到GCS浏览器，单击存储桶，然后将my_mnist_model文件夹从系统拖放到存储桶中（见图19-4）。或者你可以单击“Upload folder”，然后选择要上传的my_mnist_model文件夹。默认情况下，SavedModel的最大大小为250MB，但是可以请求更大的容量。

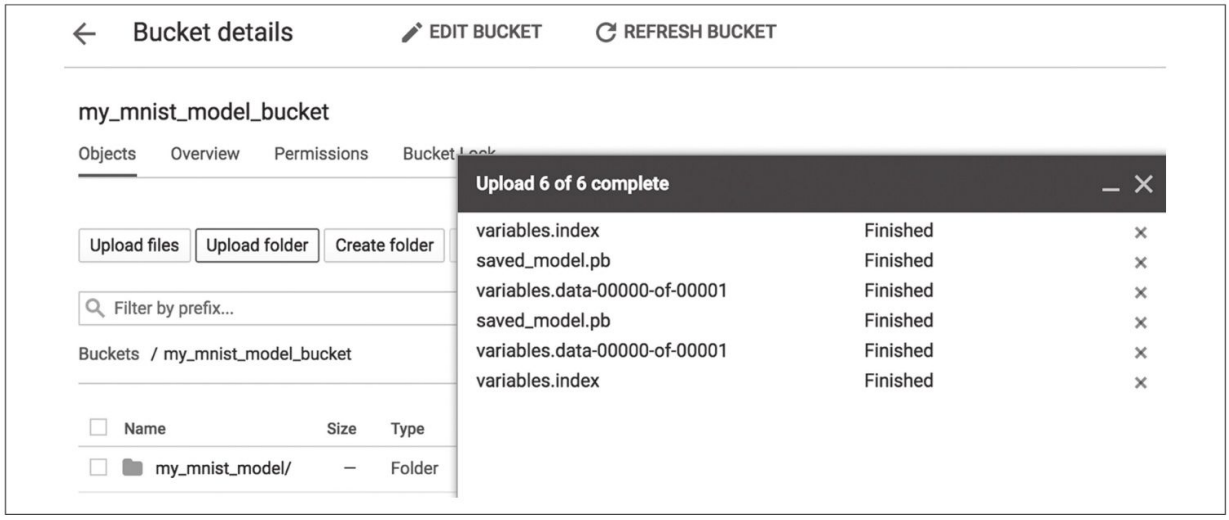


图19-4：将SavedModel上传到Google Cloud Storage

7.现在你需要配置AI平台（以前称为ML Engine），以便它知道要使用的模型和版本。在导航菜单中，向下滚动到“Artificial Intelligence”部分，然后单击“AI Platform→Models”。再单击“Activate API”（需要几分钟），然后单击“Create model”。填写模型详细信息（见图19-5），然后单击“Create”。

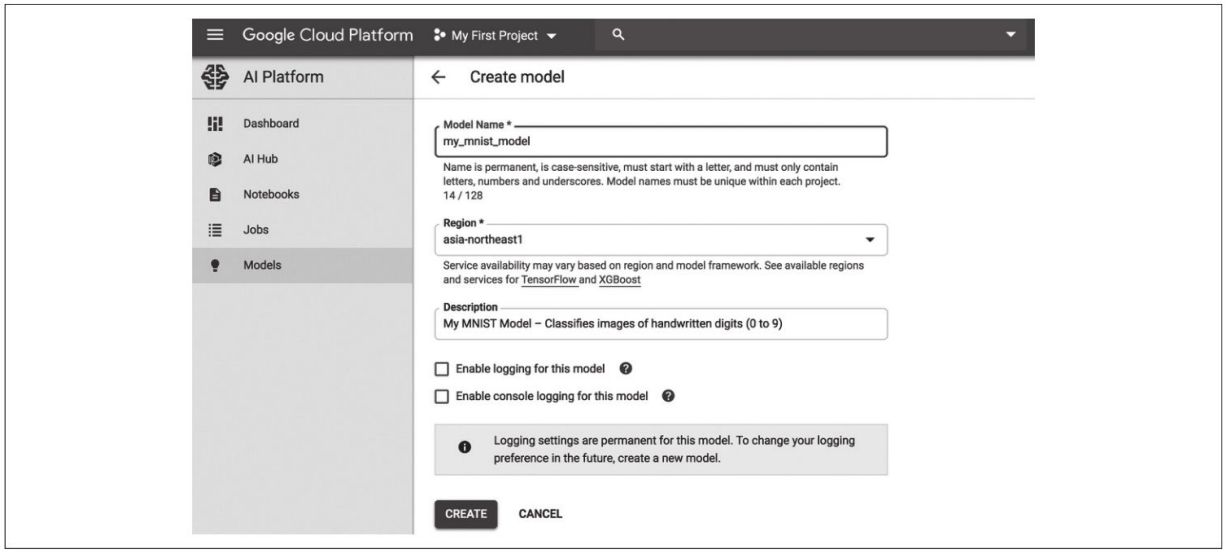


图19-5：在Google Cloud AI平台上创建新模型

8.既然你已经在AI平台上建立了模型，还需要创建一个模型版本。在模型列表中，单击刚刚创建的模型，然后单击“Create version”并填写版本详细信息（见图19-6）：设置名称、描述、Python版本（3.5或更高版本）、框架（TensorFlow）、框架版本（2.0（如果可用），或1.13）[5]、ML运行时版本（2.0（如果可用）或1.13）、机器类型（现在选择“Single core CPU”）、GCS上的模型路径（这是实际版本文件夹的完整路径，例如，gs：//my-mnist-model-bucket/my_mnist_model/0002/）、缩放（选择自动）和在所有时间运行的TF Serving容器的最小数量。然后点击Save。

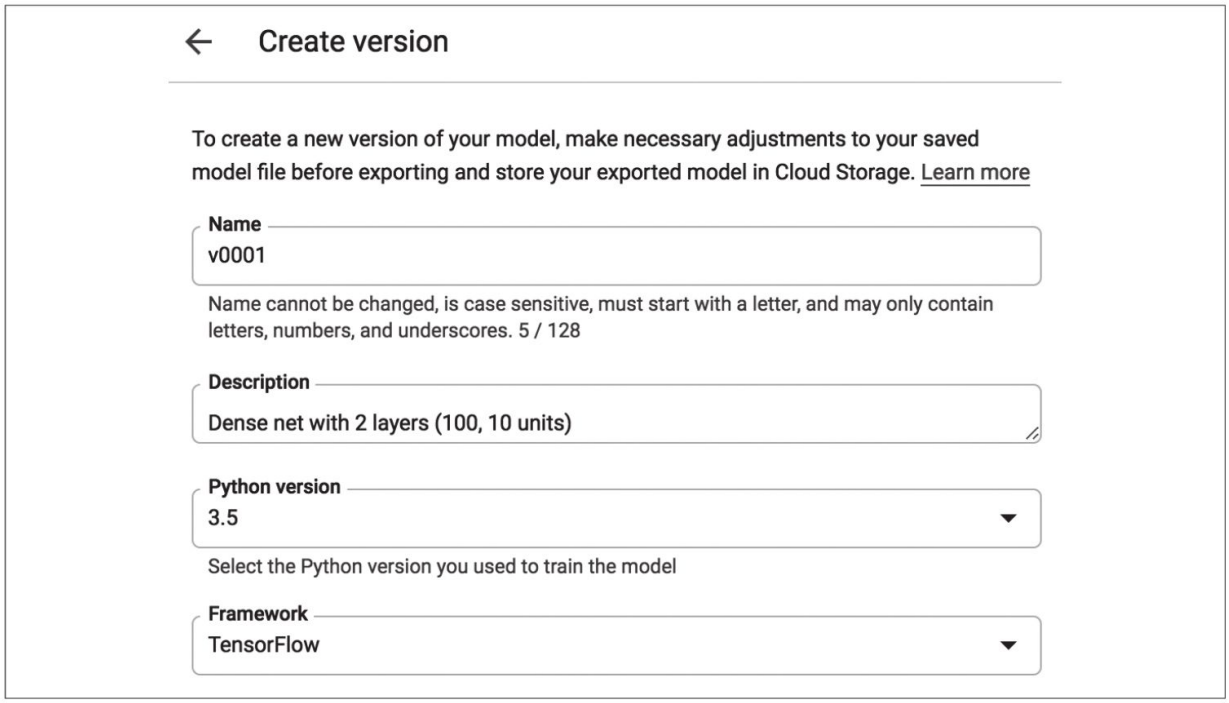


图19-6：在Google Cloud AI平台上创建新的模型版本

恭喜，你已经在云上部署了第一个模型！因为你选择了自动缩放，所以当每秒查询数量增加时，AI平台将启动更多的TF Serving容器，并且在它们之间平衡查询。如果QPS下降，它将自动停止容器。因此成本直接与QPS（以及你选择的机器类型以及在GCS上存储的数据量）相关联。这种定价模型对于偶尔使用的用户以及具有使用高峰的服务以及初创企业特别有用：价格保持低廉，直到初创企业真正启动。

 如果你不使用预测服务，则AI平台将停止所有容器。这意味着你只需支付你使用的存储量（每月每GB几美分）。请注意当你查询服务时，AI平台将启动TF Serving容器，这需要几秒钟。如果无法接受此延迟，则在创建模型版本时，必须将TF Serving容器的最小数量设置为1。当然这意味着至少一台机器将持续运行，因此月费会更高。

现在让我们查询该预测服务！

19.1.3 使用预测服务

在后台，AI平台仅运行TF Serving，因此原则上，如果你知道要查询的URL，就可以使用与之前相同的代码。还有一个问题：GCP还负责加密和身份验证。加密基于SSL/TLS，身份验证基于令牌：必须在每个请求中将秘密的身份验证令牌发送到服务器。因此在你的代码可以使用预测服务（或任何其他GCP服务）之前，它必须获得令牌。我们很快就会看到如何执行此操作，但是首先你需要配置身份验证，并为你的应用程序提供对GCP的适当访问权限。你有两种验证方式：

• 你的应用程序（即查询预测服务的客户端代码）可以使用带有你自己的谷歌登录名和密码的用户凭证进行身份验证。使用用户凭证将为你的应用程序赋予与GCP完全相同的权限，这肯定超出了所需的权限范围。此外你将必须在应用程序中部署凭证，任何具有访问权限的人都可以窃取你的凭证并完全访问你的GCP账户。简而言之，不要选择此选项。仅在极少数情况下（例如当你的应用程序需要访问用户的GCP账户时）才需要使用此功能。

• 客户端代码可以使用服务账户进行身份验证。这是代表应用程序而不是用户的账户。通常，它具有非常有限的访问权限：严格限制所需的权限，仅此而已。这是推荐的选项。

让我们为你的应用程序创建一个服务账户：在导航菜单中，转到IAM&admin→Service accounts，然后单击“Create Service Account”，填写表格（服务账户名称、ID、描述），然后单击“Create”（见图19-7）。接下来，你必须授予该账户一些访问权限。选择ML Engine Developer角色：这将使服务账户可以进行预测，而仅能进行预测。或者你可以授予某些用户访问服务账户的权限（当你的GCP用户账户是组织的一部分，并且你希望授权组织中的其他用户部署，基于该服务账户或管理服务账户本身）。接下来，单击“Create Key”以导出服务账户的私钥，选择JSON，然后单击“Create”。这将以JSON文件的形式下载私钥。确保将私钥保密！

现在让我们写一个小脚本来查询预测服务。Google提供了几个库来简化对其服务的访问：

Google API Client库

这是OAuth 2.0（用于身份验证）和REST之上的一层。你可以将其与所有GCP服务一起使用，包括AI平台。你可以使用pip进行安装：该库名为google-api-python-client。

Google Cloud Client库

这些是更高级别的库：每个都专用于一种特定的服务，例如GCS、Google BigQuery、Google Cloud Natural Language和Google Cloud Vision。可以使用pip安装所有这些库（例如，GCS客户端库称为google-cloud-storage）。当客户端库可用于给定服务时，建议使用它而不是Google API客户端库，因为它实现了所有的最佳实践，而且通常会使用gRPC而不是REST以获得更好的性能。

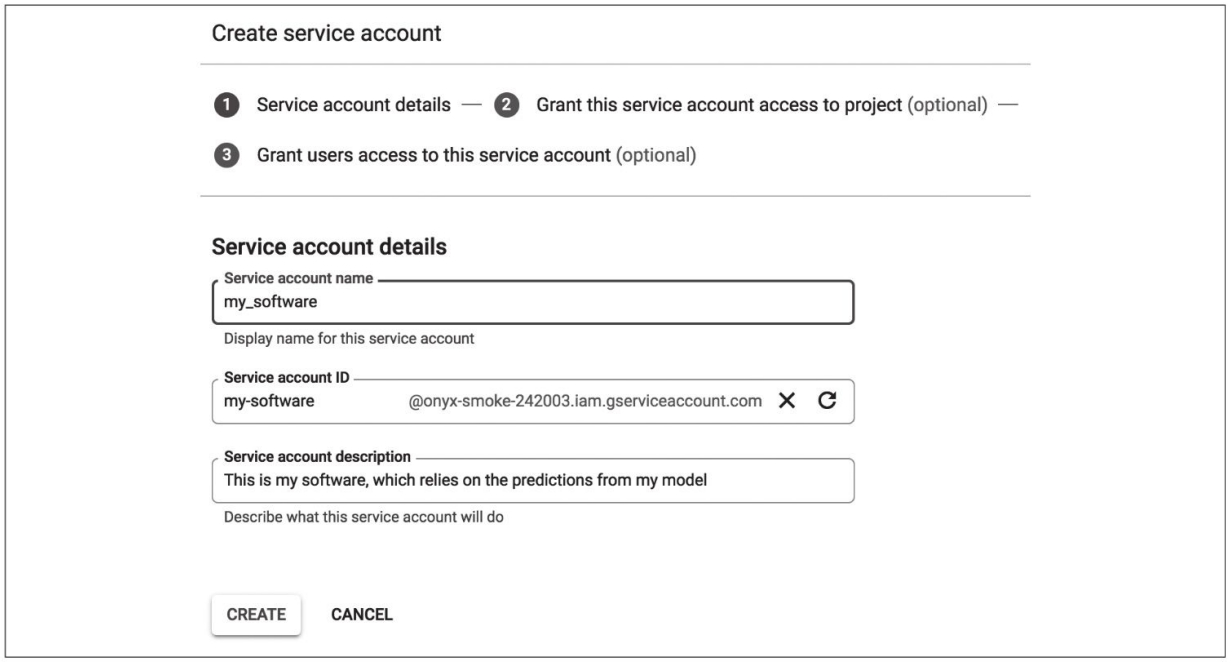


图19-7：在Google IAM中创建一个新的服务账户

在撰写本书时，还没有AI平台的客户端库，因此我们将使用Google API客户端库。它需要使用服务账户的私钥；你可以在启动脚本之前或在如下脚本内通过设置GOOGLE_APPLICATION_CREDENTIALS环境变量来告知其位置：

```
import os

os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "my_service_account_key.json"
```

 如果你将应用程序部署到Google Cloud Engine（GCE）上的虚拟机，或者使用Google Cloud Kubernetes Engine部署在容器中，或者作为Web应用程序部署在Google Cloud App Engine上，或者作为微服务部署在Google Cloud Functions上，并且GOOGLE_APPLICATION_CREDENTIALS环境变量没有设置，则库将使用默认的服务账户为主机服务（如果你的应用程序在GCE上运行，则默认GCE服务账户）。

接下来，你必须创建一个资源对象，该资源对象包装了对预测服务的访问权[6]：

```
import googleapiclient.discovery

project_id = "onyx-smoke-242003" # change this to your project ID
model_id = "my_mnist_model"
model_path = "projects/{}/models/{}".format(project_id, model_id)
ml_resource = googleapiclient.discovery.build("ml", "v1").projects()
```

请注意，你可以将/versions/0001（或任何其他版本号）附加到model_path上以指定要查询的版本：这对于A/B测试或在广泛发布新版本之前在一小群用户中测试新版本很有用（这叫作金丝雀

（canary））。接下来，让我们编写一个小函数，该函数将使用资源对象调用预测服务并返回预测：

```
def predict(X):
    input_data_json = {"signature_name": "serving_default",
                       "instances": X.tolist()}
    request = ml_resource.predict(name=model_path, body=input_data_json)
    response = request.execute()
    if "error" in response:
        raise RuntimeError(response["error"])
    return np.array([pred[output_name] for pred in response["predictions"]])
```

该函数使用一个包含输入图像的NumPy数组，并准备一个字典，客户端库将其转换为JSON格式（就像我们之前所做的那样）。然后它准备一个预测请求并执行它。如果响应中包含错误，它将引发异常，否则它将提取每个实例的预测并将其合并在NumPy数组中。让我们看看它是否有效：

```
>>> Y_probas = predict(X_new)
>>> np.round(Y_probas, 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 0.  , 0.  ],
       [0.  , 0.  , 0.99, 0.01, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.96, 0.01, 0.  , 0.  , 0.  , 0.  , 0.01, 0.01, 0.  ]])
```

是的，现在你已经在云上运行了一个不错的预测服务，该服务可以自动扩展到任意数量的QPS，而且你可以从任何地方安全地使用它。而且不使用它时，你几乎不需付出任何费用：你每月只需为GCS上使用的每千兆字节支付几美分。你还可以使用Google Stackdriver获取详细的日志和性能指标。

但是如果你要将模型部署到移动应用程序该怎么办？嵌入式设备呢？

[1] REST（或RESTful）API是使用标准HTTP动词（例如GET、POST、PUT和DELETE）并使用JSON输入和输出的API。gRPC协议更复杂但效率更高。使用协议缓冲区来交换数据（见第13章）。

[2] 如果你不熟悉Docker，则可以下载Docker映像，里面有打包好的一组应用程序（包括所有依赖项，通常还包括一些好的默认配置），然后使用Docker引擎在系统上运行它们。当运行映像时，引擎会创建一个Docker容器，以使应用程序与你自己的系统保持良好的隔离（但如果需要，可以给它一些有限的访问权限）。它类似于虚拟机，但是速度更快，更轻量级，因为容器直接依赖于主机的内核。这意味着映像不需要包含或运行其自己的内核。

[3] 公平地说，可以通过在创建REST请求之前先对数据进行序列化并将其编码为Base64来缓解这种情况。此外还可以使用gzip压缩REST请求，从而大大减少有效负载大小。

[4] 如果SavedModel在asset/extra目录中包含一些示例实例，则可以配置TF Serving在这些实例上执行模型，然后再开始为新请求提供服务。这称为模型预热：它确保正确加载所有内容，从而避免了第一个请求的较长响应时间。

[5] 在撰写本书时，TensorFlow 2在AI平台上尚不可用，但是没关系：你可以使用1.13，它将很好地运行TF 2 SavedModel。

[6] 如果收到错误消息，提示未找到模块google.appengine，请在对build（）方法的调用中设置cache_discovery=False。参见https://stackoverflow.com/q/55561354。

19.2 将模型部署到移动端或嵌入式设备

如果你需要将模型部署到移动端或嵌入式设备，则大型模型可能要花很长时间下载并使用过多的RAM和CPU，所有这些都会使你的应用程序无响应，使设备发热并消耗电池电量。为避免这种情况，你需要在不牺牲精度的前提下，创建一个适合移动设备使用，轻巧且高效的模型。TFLite库提供了几种工具[1]，帮助你将模型部署到移动端和嵌入式设备，其主要目标是三个：

- 减小模型尺寸，缩短下载时间并减少RAM使用量。
- 减少每个预测所需的计算量以减少延时、电池使用量和发热量。
- 使模型适应特定设备的约束。

为了减小模型的大小，TFLite的模型转换器采用了SavedModel并将其压缩为基于FlatBuffers的更轻量的格式。这是Google最初为游戏而创建的高效跨平台的序列化库（有点类似于协议缓冲区）。它的设计使你无须任何预处理就可以直接将FlatBuffer加载到RAM：这减少了加载时间和内存占用。一旦将模型加载到移动或嵌入式设备中，TFLite解释器将执行该模型以进行预测。这是把SavedModel转换为FlatBuffer并将其保存到一个.tflite文件的方法：

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_path)
tflite_model = converter.convert()
with open("converted_model.tflite", "wb") as f:
    f.write(tflite_model)
```

 你也可以使用from_keras_model（）将tf.keras模型直接保存到FlatBuffer中。

转换器还优化了模型，缩小模型并减少其延迟。它会修剪所有进行预测不需要的操作（例如训练操作），并在可能的情况下优化计算。例如，3×a+4×a+5×a将转换为（3+4+5）×a。它还会尽可能地融合操作。例如批量归一化层最终会尽可能地折叠到前一层的加法和乘法运算中。要了解TFLite可以多大程度上优化模型，请下载一个预训练的TFLite模型，解压缩文件，然后打开优秀的Netron图形可视化工具并上传.pb文件以查看原始模型。这是一个大而复杂的图，接下来打开优化的.tflite模型并惊叹于它的精简吧！

减少模型大小的另一种方法（除了简单地使用较小的神经网络架构之外）是使用较小的位宽：例如，如果你使用半浮点数（16位）而不是常规浮点数（32位），则模型尺寸会缩小2倍，但会降低精度（通常较小）。而且训练会更快，你将使用大约一半的GPU RAM。

通过将模型权重量化为定点8位整数，TFLite的转换器可以做到的更好！与使用32位浮点数相比，这使大小减小了四倍。最简单的方法称为训练后量化：它只是使用相当基本但有效的对称量化技术对训练后的权重进行量化。它找到最大绝对权重值m，然后将浮点范围-m至+m映射到定点（整数）范围-127至+127。例如（见图19-8），如果权重范围为-1.5至+0.8，则字节-127、0和+127将分别对应于浮点数-1.5、0.0和+1.5。注意使用对称量化时，0.0始终映射为0（还请注意，不使用字节值+68至+127，因为它们映射为大于+0.8的浮点数）。

要执行此训练后量化，只需在调用convert（）方法之前将OPTIMIZE_FOR_SIZE添加到转换器优化列表中即可：

```
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
```

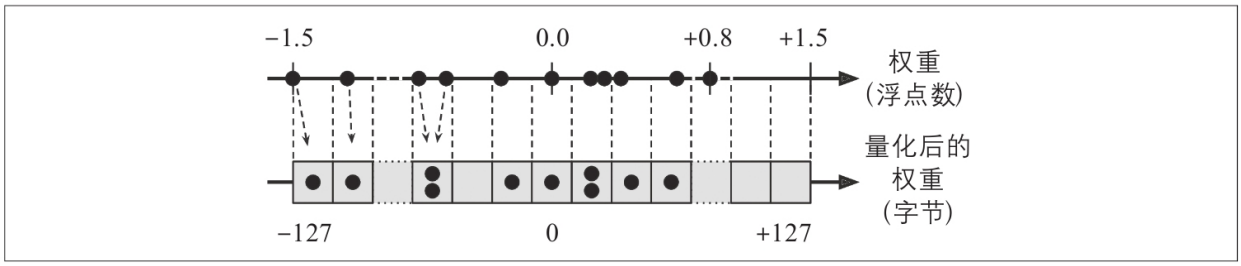


图19-8：使用对称量化，从32位浮点数到8位整数

该技术极大地减小了模型的大小，因此下载和存储速度更快。但是，在运行时，量化权重在使用之前会转换回浮点数（这些恢复的浮点数与原始浮点数并不完全相同，但相差不大，因此精度损失通常是可以接受的）。为避免始终对它们进行重新计算，将对被恢复的浮点进行缓存，因此不会减少RAM使用量。而且计算速度也不会降低。

减少延时和功耗的最有效方法是量化激活，以便可以完全使用整数完成计算，而无须任何浮点运算。即使使用相同的位宽（例如32位整数而不是32位浮点数），整数计算也会占用较少的CPU周期、消耗较少的能量并产生较少的热量。而且如果你还减少了位宽（例如减少到8位整数），则可以大大提高速度。此外，某些神经网络加速器设备（例如Edge TPU）只能处理整数，因此必须对权重和激活进行完全量化。这可以在训练后完成。它需要一个校准步骤来找到激活的最大绝对值，因此你需要向TFLite提供一个有代表性的训练数据样本（它不需要很多），它将通过模型处理数据并测量量化所需的激活统计信息（这个步骤通常很快）。

量化的主要问题是它失去了一点精度：这等同于在权重和激活中添加噪声。如果准确性下降太严重，则可能需要使用量化意识训练（quantization-aware training）。这意味着要在模型中添加伪量化操作，以便在训练过程中可以学会忽略量化噪声。最终权重将对量化更有鲁棒性。此外在训练过程中可以自动执行校准步骤，从而简化了整个过程。

我已经解释了TFLite的核心概念，但是要对移动应用程序或嵌入式程序进行编码需要另外一本书。幸运的是，存在这样一本书：如果你想了解有关为移动和嵌入式设备构建TensorFlow应用程序的更多信息，请阅读Pete Warden（他领导TFLite团队）和DanielSitunayake撰写的O’Reilly书TinyML：Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers。

浏览器中的TensorFlow

如果你想在网站中使用模型，直接在用户的浏览器中运行，该怎么办？这在许多情况下很有用，例如：

- 当你的Web应用程序经常在用户的连接是间歇性或缓慢的情况下使用时（例如远足者的网站），因此直接在客户端运行模型是使网站可靠的唯一方法。

- 当你需要模型的响应速度尽可能快时（例如对于在线游戏）。消除查询服务器进行预测的需求肯定会减少延迟，并使网站的响应速度更快。

- 当你的Web服务基于某些私有用户数据进行预测时，你希望通过在客户端上进行预测来保护用户的隐私，使私有数据不必离开用户的计算机[2]。

对于所有这些情况，你都可以将模型导出为可以由TensorFlow.js JavaScript库加载的特殊格式。该库可以使你的模型直接在用户的浏览器中进行预测。TensorFlow.js项目包含一个tensorflowjs_converter工具，该工具可以将Tensor FlowSavedModel或Keras模型文件转换为TensorFlow.js Layers格式：这是一个目录，包含一组二进制格式的共享权重文件和一个model.json文件，该文件描述模型的结构并链接到权重文件。该格式经过优化使其在网络上有效下载。用户可以使用

TensorFlow.js库下载模型并在浏览器中运行预测。这是一个代码段，可让你大致了解JavaScript API的作用：

```
import * as tf from '@tensorflow/tfjs';
const model = await tf.loadLayersModel('https://example.com/tfjs/model. json');
const image = tf.fromPixels(webcamElement);
const prediction = model.predict(image);
```

再说一次，要对这个话题做个公正的判断就需要整本书。如果你想了解更多有关TensorFlow.js的信息，请阅读Anirudh Koul、Siddha Ganju和Meher Kasam撰写的O’Reilly书Practical Deep Learning for Cloud，Mobile，and Edge。

接下来，我们将看到如何使用GPU来加快计算速度！

[1] 还要检查TensorFlow的图变换工具，以修改和优化计算图。
[2] 如果你对此主题感兴趣，请查看联邦学习。

19.3 使用GPU加速计算86

在第11章中，我们讨论了可大大加快训练速度的几种技术：更好的权重初始化、批量归一化、复杂的优化器等。但是即使采用了所有这些技术，使用单个CPU在单台机器上训练大型神经网络也可能需要数天甚至数周的时间。

在本节中，我们将研究如何使用GPU加速模型。我们还将看到如何将计算拆分到多个设备上，包括CPU和多个GPU设备（见图19-9）。现在我们将在单个机器上运行所有内容，但是在本章稍后，我们将讨论如何在多个服务器之间分配计算。

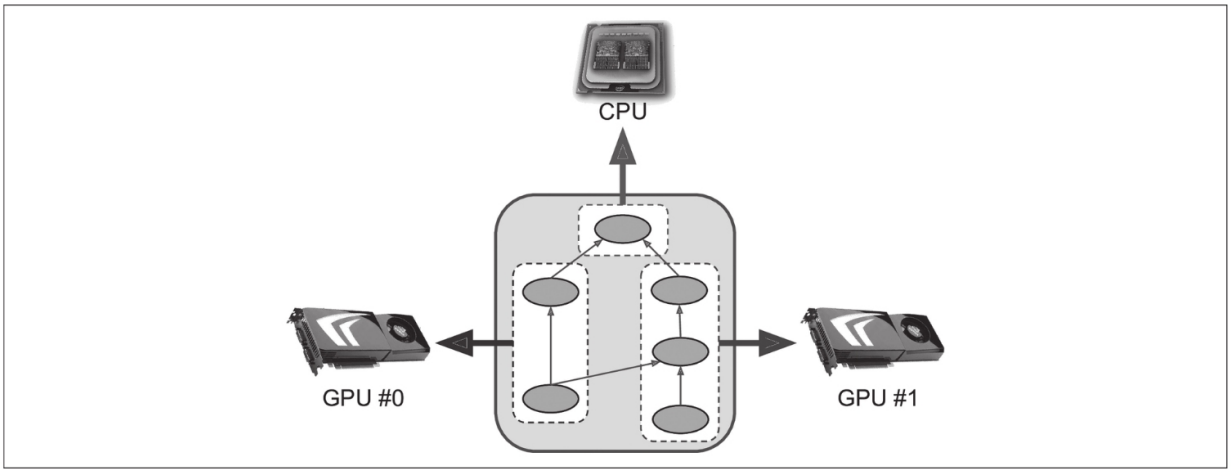


图19-9：跨多个设备并行执行TensorFlow图

多亏了GPU，你不必等待数天或数周才能完成训练算法，而只需等待几分钟或几小时。这不仅节省了大量时间，而且还意味着你可以更轻松地尝试各种模型，并经常在新数据上对模型进行重新训练。

 通常只需将GPU卡添加到一台计算机上就可以显著提高性能。实际上在许多情况下这已经足够了。你根本不需要使用多台计算机。例如由于分布式设置中网络通信带来的额外延迟，通常可以在一台机器上使用四个GPU而不是在多台机器上使用八个GPU来训练神经网络。同样使用单个功能强大的GPU通常比使用多个速度较慢的GPU更可取。

第一步是让你手头里有GPU。有两种选择：你可以购买自己的GPU，也可以在云上使用配备有GPU的虚拟机。让我们从第一个选项开始。

19.3.1 拥有你自己的GPU

如果你选择购买GPU卡，则需要一些时间做出正确的选择。Tim Dettmers写了一篇出色的博客文章来帮助你选择，他会定期进行更新：我鼓励你仔细阅读它。在撰写本书时，TensorFlow仅支持具有CUDA Compute Capability 3.5+的Nvidia卡（当然还有Google的TPU），但它可能会将其支持扩展到其他制造商。尽管目前仅在GCP上提供TPU，但很有可能在不久的将来出售类似TPU的卡，而且TensorFlow可能会支持它们。简而言之，请务必查看TensorFlow的文档以了解其支持哪些设备。

你如果要购买Nvidia GPU卡，则需要安装适当的Nvidia驱动程序和几个Nvidia库[1]。

其中包括Compute Unified Device Architecture（CUDA）库，该库使开发人员可以利用支持CUDA的GPU来进行各种计算（不仅是图形加速），还有CUDA深度神经网络库（cuDNN），这是GPU加速的DNN算子库。cuDNN提供了常见DNN计算的优化实现，例如激活层、归一化、前向和后向卷积以及池化（见第14章）。它是Nvidia深度学习SDK的一部分（注意你需要创建Nvidia开发人员账户才能下载该软件库）。TensorFlow使用CUDA和cuDNN来控制GPU卡并加速计算（见图19-10）。

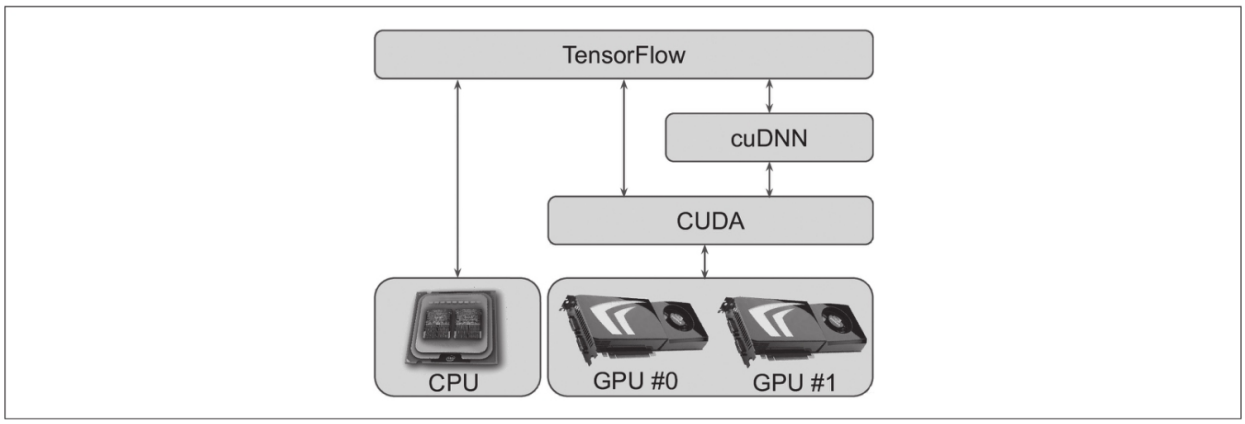


图19-10：TensorFlow使用CUDA和cuDNN来控制GPU和加速DNN

一旦安装了GPU卡以及所有必需的驱动程序和库，就可以使用nvidia-smi命令检查CUDA是否已正确安装。它列出了可用的GPU卡以及每个卡上运行的进程：

```
$ nvidia-smi
Sun Jun  2 10:05:22 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 410.79       CUDA Version: 10.0     |
|-------------------------------+----------------------+----------------------+
| GPU Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   61C    P8    17W /  70W |      0MiB / 15079MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

在撰写本书时，你还需要安装TensorFlow的GPU版本（即tensorflow-gpu库）。但是针对CPU和GPU机器制定统一的安装过程仍是一项正在进行的工作，因此请查看安装文档以了解应安装哪个库。在任何情况下，正确安装每个必需的库都耗时长且棘手（如果你未安装正确的库版本，那么所有事情都会变得一团糟），因此TensorFlow提供了一个Docker映像，其中包含你需要的所有内容。为了使Docker容器能够访问GPU，你仍然需要在主机上安装Nvidia驱动程序。

要检查TensorFlow实际上是否可以看到GPU，请运行以下测试：

```
>>> import tensorflow as tf
>>> tf.test.is_gpu_available()
True
>>> tf.test.gpu_device_name()
'/device:GPU:0'
>>> tf.config.experimental.list_physical_devices(device_type='GPU')
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

is_gpu_available（）函数检查是否至少有一个GPU可用。gpu_device_name（）函数给出第一个GPU的名称：默认情况下操作将在此GPU上运行。list_physical_devices（）函数返回所有可用GPU设备的列表（在此示例中为一个）[2]。

现在，如果你不想花费时间和金钱来购买自己的GPU卡，该怎么办？只需在云上使用GPU VM！

19.3.2 使用配备GPU的虚拟机

现在所有主要的云平台都提供GPU VM，其中一些已预先配置了所需的所有驱动程序和库（包括TensorFlow）。Google Cloud Platform会在全球和每个区域实施各种GPU配额：未经Google事先授权，你不能创建数千个GPU VM[3]。默认情况下，全球GPU配额为零，因此你无法使用任何GPU VM。你要做的第一件事就是要求更高的全球配额。在GCP控制台中，打开导航菜单，然后转到IAM&admin→Quotas。单击“Metric”，单击“None”以取消选中所有地点，然后搜索“GPU”，选择“GPUs（all regions）”以查看相应的配额。如果此配额的值为零（或不足以满足你的需要），则选中它旁边的框（应该是唯一选中的那个），然后单击“Edit quotas”。填写所需的信息，然后单击“Submit request”。配额请求可能需要几个小时（或最多几天）才能处理并被接受。默认情况下，每个区域和每种GPU类型的配额也是一个。你也可以请求增加这些配额：单击“Metric”，选择“None”以取

消选中所有指标，搜索“GPU”，然后选择所需的GPU类型（例如NVIDIA P4 GPU）。然后单击“Location”下拉菜单，单击“None”以取消选中所有指标，然后单击所需的地点。选中要更改的配额旁边的框，然后单击“Edit quotas”以提交请求。

一旦你的GPU配额请求获得批准，就可以立即使用Google Cloud AI Platform的深度学习VM映像创建配备有一个或多个GPU的VM：请访问https://homl.info/dlvm，单击“View Console”，然后单击“Launch on Compute Engine”，填写VM配置表。请注意有些地点没有所有类型的GPU，而有些地点根本没有GPU（更改地点以查看可用的GPU类型）。确保选择Tensor Flow 2.0作为框架，并选中“Install NVIDIA GPU driver automatically on first startup”。最好选中“Enable access to JupyterLab via URL instead of SSH”：这会使启动在此GPU VM上运行的Jupyter notebook非常容易，JupyterLab是运行Jupyter notebook的替代Web界面。创建虚拟机后，向下导航菜单至“Artificial Intelligence”部分，然后单击“AI Platform→Notebooks”。一旦Notebook实例出现在列表中（这可能需要几分钟，因此请不时单击“Refresh”直到出现），单击其“Open JupyterLab”链接。这将在VM上运行JupyterLab并将你的浏览器连接到它。你可以创建notebook并在此VM上运行所需的任何代码，这些代码会充分利用GPU。

但是，如果你想和同事进行一些快速测试或共享notebook，那应该尝试使用Colaboratory。

19.3.3 Colaboratory

访问GPU VM的最简单、最便宜的方法是使用Colaboratory（简称为Colab）。它完全免费！只需访问https://colab.research.google.com/并创建一个新的Python3 notebook：这将创建一个Jupyter notebook，保存在你的Google Drive上（或者你可以在GitHub或Google Drive上打开任何notebook，甚至可

以上传自己的notebook）。Colab的用户界面与Jupyter相似，不同之处在于你可以像普通的Google Docs一样共享和使用notebook，并且还有一些其他细微的差异（你可以使用代码中的特殊注释来创建方便的小部件）。

当你打开Colab notebook时，它会在你的免费Google VM（称为Colab运行时）（见图19-11）上运行notebook。默认情况下，运行时仅用于CPU，但是你可以通过以下方法更改：Runtime→“Change runtime type”，在“Hardware accelerator”下拉菜单中选择GPU，然后单击“Save”。实际上，你甚至可以选择TPU！（是的，你实际上可以免费使用TPU。我们将在本章后面讨论TPU，现在仅选择GPU）。

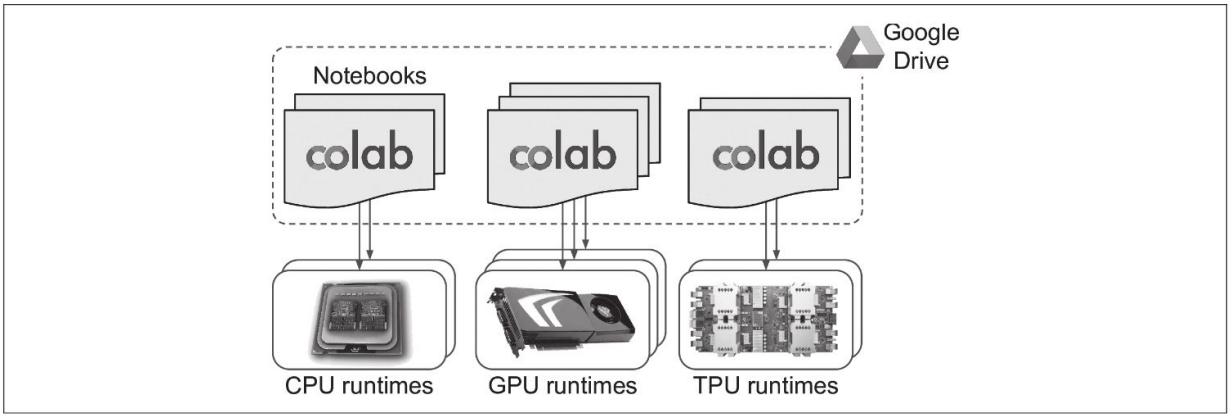


图19-11：Colab运行时和notebook

Colab确实有一些限制：首先，可以同时运行的colab notebook数量有限制（目前每个运行时类型的数量限制为5）。而且，如FAQ所述，“Colaboratory旨在用于交互式使用。长时间运行的后台计算（尤其是在GPU上）可能会停止。请不要使用Colaboratory进行加密货币的挖矿”。如果你暂时不使用Web界面（约30分钟），Web界面将自动从Colab运行时断开连接。当你重新连接到Colab运行时，它可能已被重置，因此请确保导出你关心的所有数据（下载或者保存到Google Driver）。即使你从未断开连接，Colab运行时也会在12小时后自动关闭，因为它不适合长时间计算。尽管有这些限制，它还是一个出色的工具，可以轻松地运行测试，快速获得结果，与你的同事进行协作。

19.3.4 管理GPU内存

默认情况下，TensorFlow会在你第一次运行计算时自动获取所有可用GPU中的所有RAM。这么做是为了防止GPU内存碎片。这意味着如果你启动第二个TensorFlow程序（或任何需要GPU的程序），内存将很快用完。这不会像你想的那样频繁发生，因为你通常会在机器上运行单个TensorFlow程序：通常是训练脚本、TF Serving节点或Jupyter notebook。如果出于某种原因需要运行多个程序（例如在同一台机器上并行训练两个不同的模型），则需要在这些进程之间更均匀地分配GPU内存。

如果你的计算机上有多个GPU卡，一个简单的解决方法是将每个GPU卡分配给一个进程。为此你可以设置CUDA_VISIBLE_DEVICES环境变量，以便每个进程只能看到相应的GPU卡。还要将CUDA_DEVICE_ORDER环境变量设置为PCI_BUS_ID，以确保每个ID始终引用相同的GPU卡。例如你有四个GPU卡，则可以通过在两个单独的终端窗口中执行以下命令来启动两个程序，分别为它们分配两个GPU：

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1 python3 program_ 1.py
# and in another terminal:
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

这样程序1仅看到分别名为/gpu：0和/gpu：1的GPU卡0和1，而程序2仅看到分别名为/gpu：1和/gpu：0的GPU卡2和3（请注意顺序）。一切都会正常进行（见图19-12）。当然，你还可以在Python中通过设置os.environ[“CUDA_DEVICE_ORDER”]和os.environ[“CUDA_VISIBLE_DEVICES”]来定义这些环境变量，只要在使用TensorFlow之前就可以了。

另一个选择是告诉TensorFlow仅获取特定数量的GPU内存。导入TensorFlow之后必须立即执行此操作。例如要使TensorFlow在每个GPU

上仅获取2GiB的内存，你必须为每个物理GPU设备创建一个虚拟GPU设备（也称为逻辑GPU设备），并将其内存限制设置为2GiB（即2048MiB）：

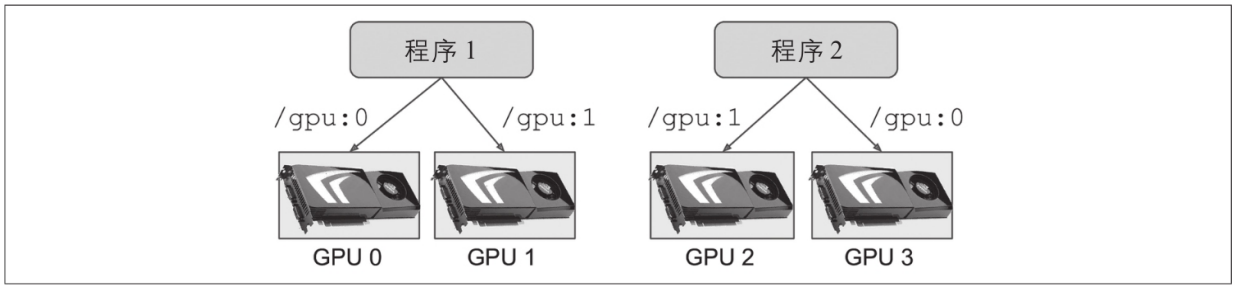


图19-12：每个程序有两个GPU

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_virtual_device_configuration(
        gpu,
        [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

现在（假设你有四个GPU，每个GPU至少具有4 GiB的内存），可以像这样运行两个程序，每个程序都使用所有四个GPU卡（见图19-13）。

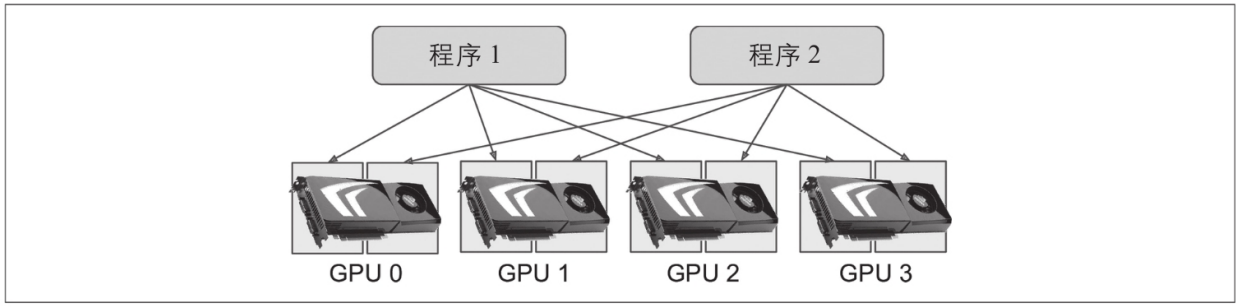


图19-13：每个程序都使用全部四个GPU，但每个GPU上只使用2 GiB RAM

如果在两个程序同时运行时运行nvidia-smi命令，则应该看到每个进程在每个卡上都有2 GiB RAM：

```
$ nvidia-smi
[...]
+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      2373      C   /usr/bin/python3                           2241MiB |
|    0      2533      C   /usr/bin/python3                           2241MiB |
|    1      2373      C   /usr/bin/python3                           2241MiB |
|    1      2533      C   /usr/bin/python3                           2241MiB |
[...]
```

还有一种选择是告诉TensorFlow仅在需要时才获取内存（这也必须在导入TensorFlow之后立即完成）：

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_memory_growth(gpu, True)
```

执行此操作的另一种方法是将TF_FORCE_GPU_ALLOW_GROWTH环境变量设置为true。使用此选项，TensorFlow一旦获取到内存就不会释放内存（避免内存碎片），当然程序结束时除外。使用此选项可能很难保证确定性的行为（例如一个程序可能会崩溃，因为另一个程序的内存使用量达到顶峰），因此在生产环境中，你可能需要坚持使用以前的选项之一。但是在某些情况下它非常有用：例如，当你使用计算机运行多个Jupyter notebook时，其中一些notebook使用TensorFlow。这就是在Colab运行时中将TF_FORCE_GPU_ALLOW_GROWTH环境变量设置为true的原因。

最后，在某些情况下，你可能希望将一个GPU分成两个或多个虚拟GPU，例如，如果你要测试一种分布算法（这是一种尝试本章其余部分代码示例的简便方法，即使你只有一个GPU，例如在Colab运行时中）。以下代码将第一个GPU分为两个虚拟设备，每个虚拟设备有2 GiB的RAM（同样必须在导入TensorFlow之后立即完成）：

```
physical_gpus = tf.config.experimental.list_physical_devices("GPU")
tf.config.experimental.set_virtual_device_configuration(
    physical_gpus[0],
    [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048),
     tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

这两个虚拟设备称为/gpu：0和/gpu：1，你可以在每个虚拟设备上放置操作和变量，就好像它们实际上是两个独立的GPU一样。现在让我们看一下TensorFlow如何决定应在哪些设备上放置变量并执行操作。

19.3.5　在设备上放置操作和变量

TensorFlow白皮书[4]提出了一种友好的动态放置算法，该算法可在所有可用设备之间自动分配操作，并会考虑到诸如图形在先前运行中测得的计算时间，估计每个操作的输入和输出张量大小，每个设备中可用的RAM数量，将数据传入和传出设备时的通信延迟以及用户的提示和约束。实际上，该算法的效率比用户指定的一小组放置规则的效率低，因此TensorFlow团队最终放弃了动态放置器。

也就是说，tf.keras和tf.data通常可以很好地将操作和变量放在它们所属的位置（例如在GPU上进行大量计算，在CPU上进行数据预处理）。但是如果你需要更多控制，也可以在每个设备上手动放置操作和变量：

• 如前所述，你希望将数据预处理操作放置在CPU上，并将神经网络操作放置在GPU上。

• GPU通常具有相当有限的通信带宽，因此避免不必要的数据传入和传出GPU非常重要。

• 在计算机上添加更多的CPU内存很简单且相当便宜，因此通常会有大量内存，而GPU内存被嵌入在GPU中：这是昂贵并且有限的资源，因此如果变量不是在接下来的几个训练步骤中需要使用时，应该将其放置

在CPU上（数据集通常属于CPU）。默认情况下，所有变量和所有操作都将放置在第一个GPU（名为/gpu：0）上，除了没有GPU内核的变量和操作之外[5]：这些变量和操作均放置在CPU（名为/cpu：0）。张量或变量的device属性会告诉你其放置在哪个设备上[6]：

```
>>> a = tf.Variable(42.0)
>>> a.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> b = tf.Variable(42)
>>> b.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

你可以放心地忽略前缀/job：localhost/replica：0/task：0（使用TensorFlow集群时，它允许你在其他计算机上进行操作，我们将在本章后面讨论作业、副本和任务）。如你所见，第一个变量位于默认设备GPU 0上。但是第二个变量放置在CPU上：这是因为没有整数变量（或涉及整数张量的操作）的GPU内核，因此TensorFlow退回到了CPU。如果要在默认设备以外的设备上进行其他操作，请使用tf.device（）上下文：

```
>>> with tf.device("/cpu:0"):
...     c = tf.Variable(42.0)
...
>>> c.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

 即使你的计算机具有多个CPU内核，也始终将CPU视为单个设备（/cpu：0）。如果CPU具有多线程内核，则对CPU进行的任何操作都可以在多个内核上并行运行。

如果你明确要将操作或变量放置在不存在或没有内核的设备上，则会出现异常。但是在某些情况下，你可能更愿意退回到CPU。例如如果

你的程序可以同时在只有CPU的计算机和GPU机器上运行，则你可能希望TensorFlow在只有CPU的计算机上忽略tf.device（"/gpu：*"）。为此，你可以在导入TensorFlow之后立即调用tf.config.set_soft_device_placement（True）。

当放置请求失败时，TensorFlow将退回到其默认的放置规则（即如果存在且有GPU内核，则默认为GPU 0，否则为CPU 0）。

现在，TensorFlow将如何在多个设备上执行所有这些操作呢？

19.3.6 跨多个设备并行执行

正如我们在第12章中看到的那样，使用TF函数的好处之一就是并行性。让我们更仔细地看一下。当TensorFlow运行TF函数时，它首先分析计算图以查找需要评估的操作列表，然后计算每个操作具有多少依赖关系。然后TensorFlow将具有零相关性的每个操作（即每个源操作）添加到该操作的设备的评估队列中（见图19-14）。一旦评估了一个操作，依赖于它的每个操作的依赖计数器就会递减。一旦操作的依赖计数器达到零，它就被推入其设备的评估队列。一旦评估了TensorFlow所需的所有节点，它就会返回其输出。

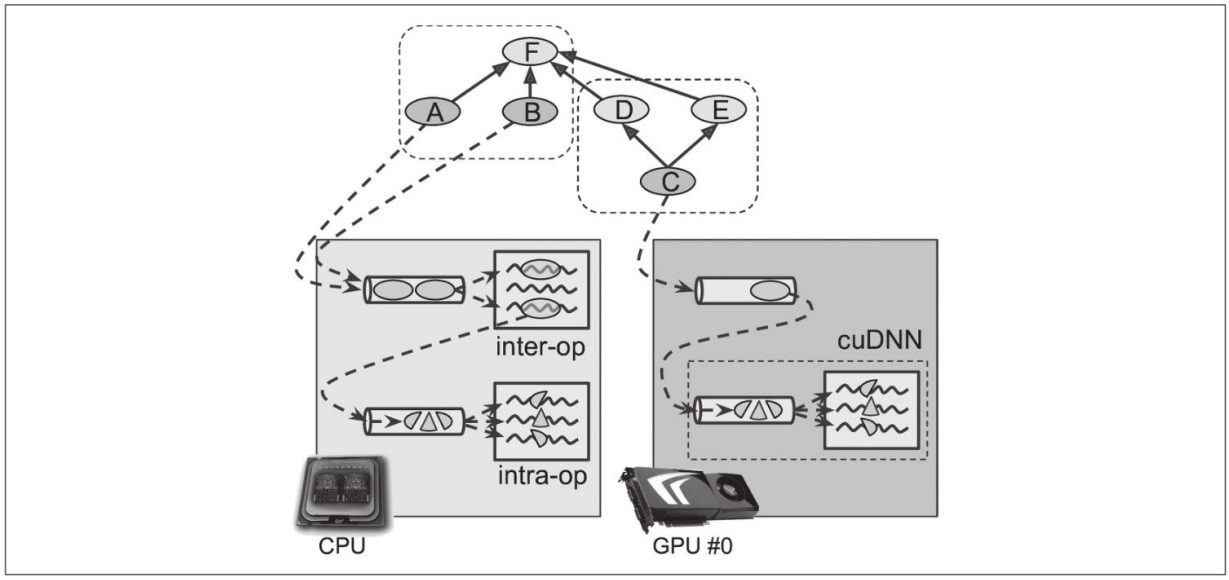


图19-14：TensorFlow图的并行执行

CPU评估队列中的操作被分派到一个称为互操作线程池（inter-op thread pool）的线程池。如果CPU具有多个内核，它将有效地并行评估这些操作。某些操作具有多线程CPU内核：这些内核将其任务分成多个子操作，这些子操作被放置在另一个评估队列中，并分派给第二个线程池，称为内操作线程池（intra-op thread pool）（由所有多线程CPU内核共享）。简而言之，可以在不同的CPU内核上并行评估多个操作和子操作。

对于GPU，事情要简单一些。只需要对GPU评估队列中的操作进行顺序评估。但是，大多数操作都具有多线程GPU内核，通常由TensorFlow依赖的库（例如CUDA和cuDNN）实现。这些实现都有自己的线程池，它们通常会尽可能多地利用GPU线程（这就是GPU中不需要互操作线程池的原因：每个操作都已利用了大多数GPU线程）。例如，在图19-14中，操作A、B和C是源操作，因此可以立即对其进行评估。操作A和B被放置在CPU上，因此它们被发送到CPU的评估队列，然后被分派到互操作线程池并立即进行并行评估。操作A碰巧有一个多线程内核。它的计算分为三部分，由内操作线程池以并行方式执行。操作C进入GPU 0的评估队列，在此示例中，其GPU内核恰好使用cuDNN，该cuDNN管理其自己的内操作线程池并跨多个GPU线程并行运行该操作。假设C首先完成。D和E的依赖计数器递减，它们达到零，因此这两个操作都被推入GPU 0的评估队列，并依次执行。请注意即使D和E都依赖C，C只会被评估一次。假设B接下来完成。F的依赖计数器从4减少到3，并且由于它不是0，所以它还不会运行。一旦A、D和E完成，则F的依赖计数器达到0，则将其推入CPU的评估队列并进行评估。最后，TensorFlow返回请求的输出。

当TF函数修改有状态资源（例如变量）时，TensorFlow会执行额外的操作：即使语句之间没有明显的依赖关系，它也可以确保执行顺序与代码中的顺序匹配。例如，如果你的TF函数包含v.assign_add（1），再跟一条v.assign（v*2），TensorFlow将确保按照这些顺序执行这些操作。

 你可以通过调用tf.config.threading.set_inter_op_parallelism_threads（）来控制互操作线程池中的线程数。要设置内操作线程的数量，请使用tf.config.threading.set_intra_op_parallelism_threads（）。如果你不想让TensorFlow使用所有CPU内核，或者希望它是单线程的，这很有用[7]。

这样，你就可以在任何设备上运行任何操作，并利用GPU的强大功能！以下是你可以执行的一些操作：

• 你可以并行训练多个模型，每个模型都运行在其自己的GPU上：只需为每个模型编写训练脚本并行运行它们，设置CUDA_DEVICE_ORDE R和CUDA_VISIB L E_D E V I C E S，这样每个脚本只能看到一个GPU设备。这对于超参数调整非常有用，因为你可以并行训练具有不同超参数的多个模型。如果你有一台带有两个GPU的机器，并且在一个GPU上训练一个模型需要一个小时，那么并行训练两个模型（每个模型都在自己的专用GPU上）将只需要一个小时。简单！

• 你可以在单个GPU上训练模型并在CPU上并行执行所有的预处理，使用数据集的prefetch（）方法[8]提前准备接下来的几批数据，以便在GPU需要它们时就准备好了（见第13章）。

• 如果你的模型需要两个图像作为输入，那在合并它们的输出之前可以使用两个CNN对其进行处理，如果将每个CNN放在不同的GPU上，它的运行速度可能会更快。

• 你可以创建一个高效的集成学习：在每个GPU上放置一个经过训练的模型，这样你就可以更快地获得所有预测，从而生成集成学习的最终预测。

但是如果你想在多个GPU之间训练一个模型，该怎么办？

[1] 请检查文档以获取详细和最新的安装说明，因为它们经常更改。
[2] 本章中的许多代码示例都使用实验性API。它们很可能会在将来的版本中移至核心API。因此如果实验函数失败，请尝试删除“experimental”一词，并希望它会起作用。如果没有，那么API可能有所改变。请检查Jupyter notebook，我会确保它包含正确的代码。
[3] 这些配额大概是为了阻止可能使用失窃信用卡来利用GCP开采加密货币的坏人。
[4] Martín Abadi et al.，“TensorFlow：Large-Scale Machine Learning on Heterogeneous Distributed Systems”，Google Research whitepaper（2015）。
[5] 正如在第12章中看到的那样，内核是针对特定数据类型和设备类型的变量或操作的实现。例如，对于float32tf.matmul（）操作有一个GPU内核，但是对于int32tf.matmul（）没有GPU内核（只有一个CPU内核）。
[6] 你也可以使用tf.debugging.set_log_device_placement（True）来记录所有设备的放置。
[7] 如我在视频中所述，基于TF 1，如果你想保证完美的可重复性，这会很有用。
[8] 在撰写本书时，它仅将数据预取到CPU内存，但是你可以使用tf.data.experimental.prefetch_to_device（）使其预取数据并将其推送到你选择的设备，这样GPU就不会浪费时间等待数据传输。

19.4 跨多个设备的训练模型

在多个设备上训练单个模型的主要方法有两种：模型并行（模型在设备之间划分）和数据并行（模型在每个设备上复制，每个副本在数据子集上训练）。在多个GPU上训练模型之前，让我们仔细看一下这两个选项。

19.4.1 模型并行

到目前为止，我们已经在单个设备上训练了每个神经网络。如果想跨多个设备训练单个神经网络怎么办？这需要将模型分成单独的块，并在不同的设备上运行每个块。

不幸的是，这种模型并行性非常棘手，它实际上取决于神经网络的架构。对于全连接网络，通常无法从这种方法中获得太多好处（见图19-15）。直观地看，拆分模型的一种简单方法似乎是将每一层都放置在不同的设备上，但这是行不通的，因为每一层都需要等待上一层的输出才能执行任何操作。也许你可以垂直划分呢？例如每层的左半部分在一个设备上，而右部分在另一设备上？稍好一点，因为每一层的两个半部确实可以并行工作，但是问题在于，下一层的每个半部都需要前一层两个半部的输出，因此会有很多跨设备通信（虚线箭头表示）。由于跨设备通信速度很慢（尤其是当设备位于不同的计算机上时），这很可能会完全抵消并行计算的好处。

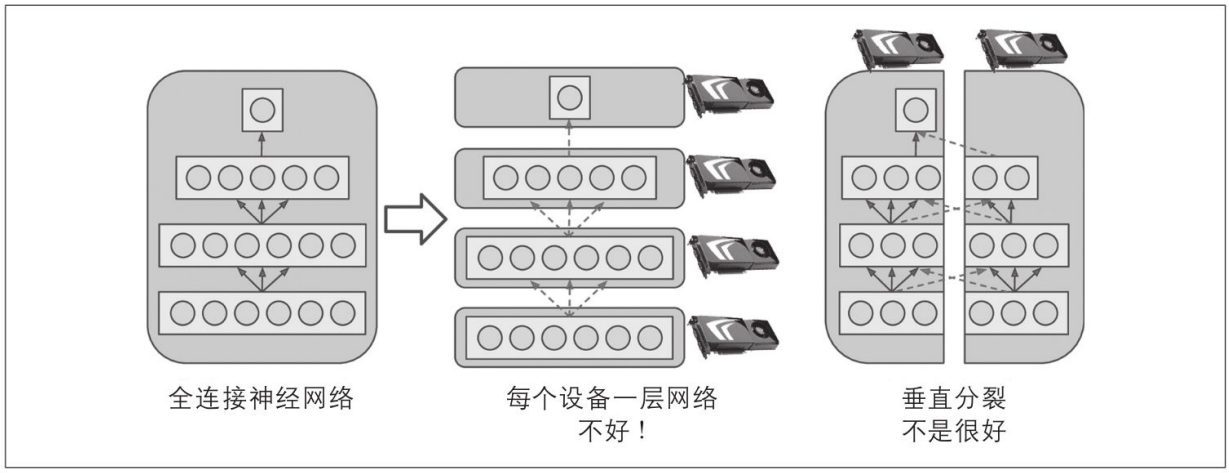


图19-15：划分一个完全连接的神经网络

一些神经网络结构，例如卷积神经网络（见第14章），包含的层仅部分连接到较低层，因此有效地在设备之间分配块要容易得多（见图19-16）。

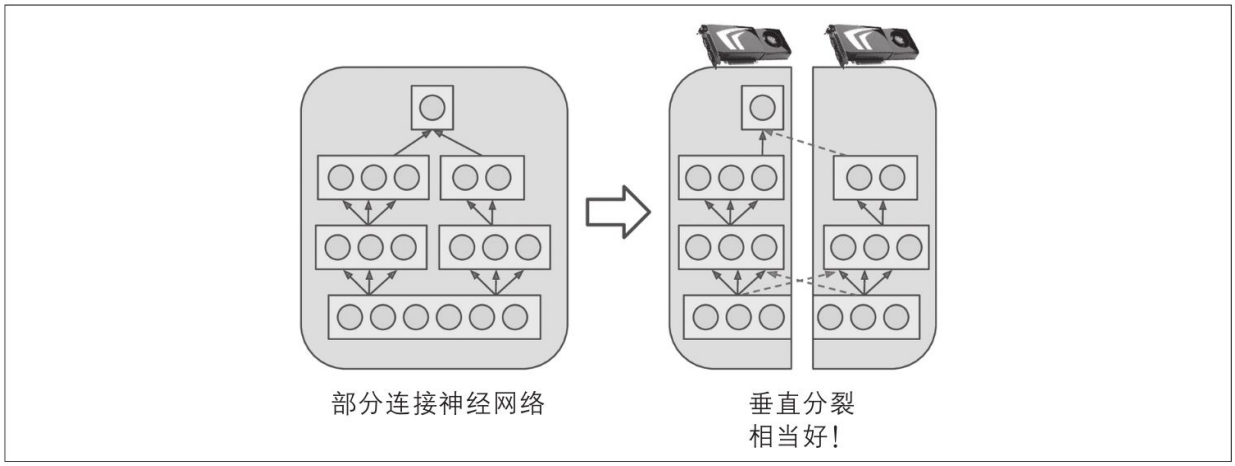


图19-16：划分一个部分连接的神经网络

深度循环神经网络（见第15章）可以在多个GPU之间更有效地拆分。如果你通过将每一层放置在不同的设备上来水平划分网络，使用输入序列来馈送网络来进行处理，则在第一时间步骤中，只有一个设备将处于活动状态（使用该序列的第一个值），在第二步中，两个将处于活动状态（第二层将处理第一层的输出以获得第一个值，而第一层将处理第二个值），并将信号传播到输出层，所有设备将同时处于活动状态

（见图19-17）。虽然仍然有很多跨设备通信正在进行，但是由于每个单元可能相当复杂，因此从理论上讲，并行运行多个单元的好处可能大于通信传输的代价。实际上，在单个GPU上运行的常规LSTM层堆栈运行得更快。

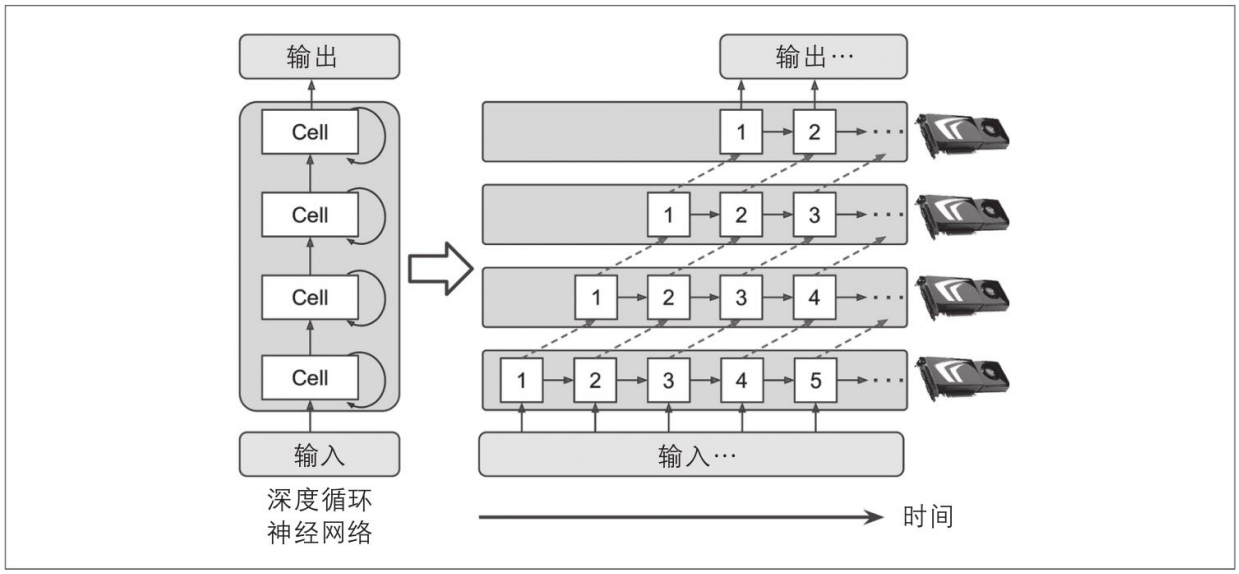


图19-17：划分深度递归神经网络

简而言之，模型并行化可以加快运行或训练某些类型的神经网络的速度，但不是全部，需要特别注意和调整，例如确保需要通信最多的设备在同一台机器上运行[1]。我们来看一个更简单、更有效的选择：数据并行。

19.4.2 数据并行

并行训练神经网络的另一种方法是在每个设备上复制它，并在所有副本上同时运行每个训练步骤，每个副本使用不同的批量数据。然后将每个副本计算出的梯度取平均值，将结果用于更新模型参数。这称为数据并行。这个方法有很多变体，让我们看一下最重要的一个。

使用镜像策略的数据并行

可以说最简单的方法是在所有GPU上完全镜像所有模型参数，并始终在每个GPU上使用完全相同的参数更新。这样所有副本始终保持完全相同。这称为镜像策略，事实证明它非常有效，尤其是在使用单台计算机时（见图19-18）。

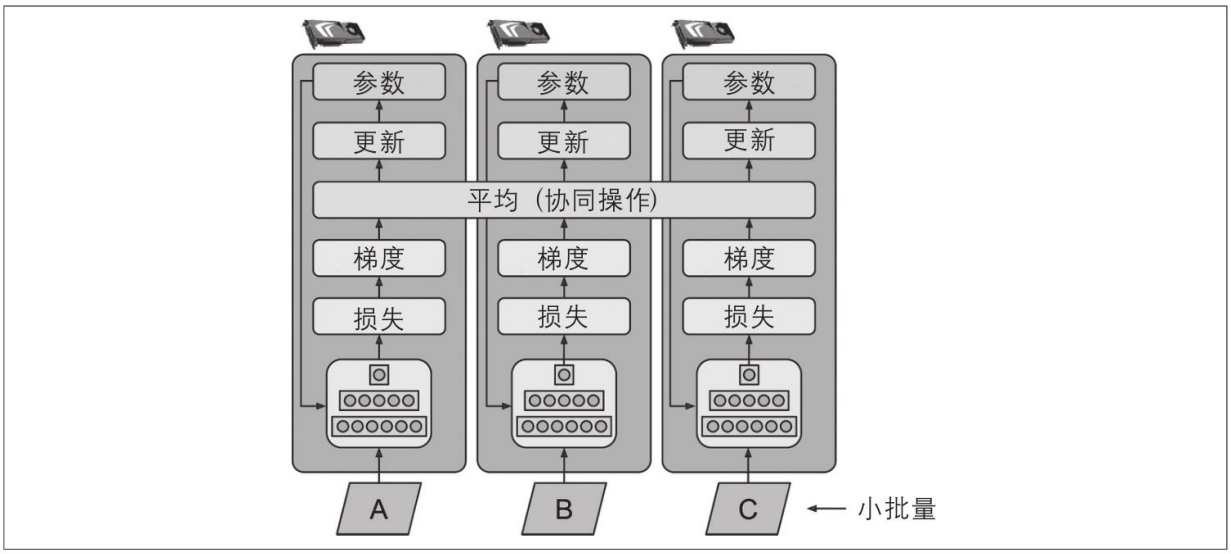


图19-18：使用镜像策略的数据并行

使用此方法时，棘手的部分是从所有GPU有效地计算所有梯度的平均值，并将结果分布到所有GPU上。这可以使用AllReduce算法完成，该算法是多个节点协作有效执行归约运算（例如计算均值、总和和最大值）的一类算法，同时确保所有节点都获得相同的最终结果。幸运的是，此类算法已有一些现成的实现。

具有集中参数的数据并行

另一种方法是将模型参数存储在执行计算（称为工作程序）的GPU设备外部，例如存储在CPU上（请参见图19-19）。在分布式设置中，你将所有参数放在一个或多个称为参数服务器的只有的CPU的服务器上，它们的唯一作用是保存和更新参数。

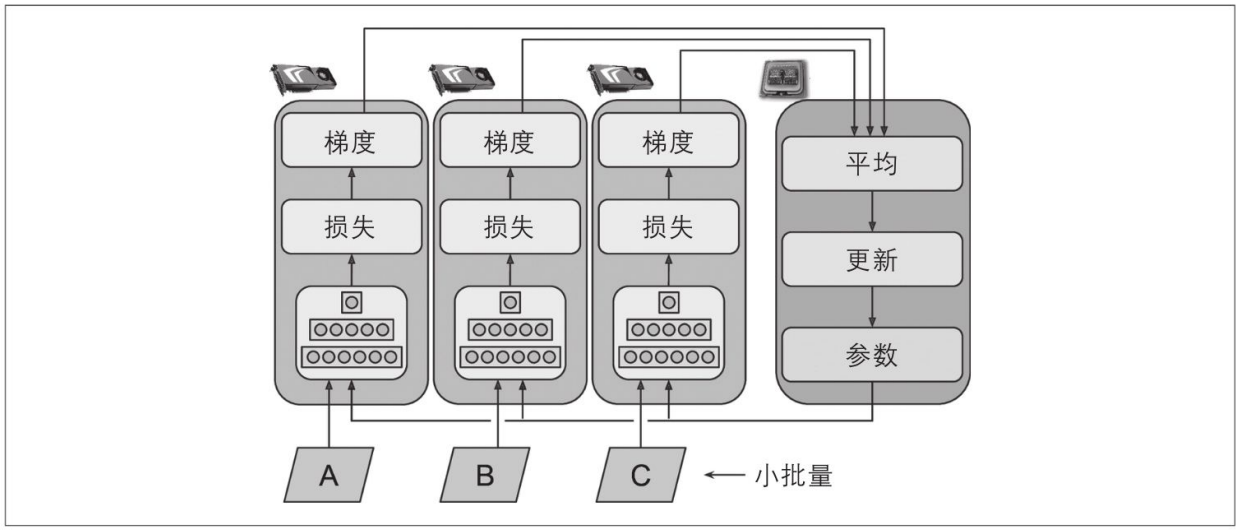


图19-19：具有集中参数的数据并行

镜像策略在所有GPU上强制进行同步权重更新，这种集中式方法允许同步或异步更新。让我们来看看这两种选择的利弊。

同步更新。通过同步更新，聚合器要等待所有梯度计算完成，然后再计算平均梯度并将其传递给优化器，优化器再更新模型参数。副本完成梯度计算后，必须等待参数更新，然后才能继续进行下一个批量处理。缺点是某些设备可能比其他设备慢，因此所有其他设备不得不在每个步骤中等待它们。此外参数将几乎同时（在应用梯度后立即）复制到每个设备，这可能会使参数服务器的带宽饱和。

 为了减少每个步骤的等待时间，你可以忽略最慢的几个副本（通常为忽略总数的10%）中的梯度。例如你可以运行20个副本，但每个步骤仅聚合来自最快18个副本的梯度，而忽略最后2个副本的梯度。参数更新后，前18个副本可以立即重新开始工作，而不必等待2个最慢的副本。通常将这种设置称为具有18个副本加上2个备用副本[2]。

异步更新。对于异步更新，只要一个副本完成梯度计算，它就会立即使用它们来更新模型参数。没有聚合（它消除了图19-19中的“平均”步骤）并且没有同步。副本独立于其他副本工作。由于无须等待其

他副本，此方法每分钟能运行更多的训练步骤。尽管仍然需要在每个步骤将参数复制到每个设备，但对于每个副本是在不同的时间发生，因此降低了带宽饱和的风险。

简单而且没有同步延迟，能更好地利用带宽的优势，使得异步更新的数据并行是一个有吸引力的选择。而且它在实践中运行得相当好，效果完全使人惊讶！确实，当副本完成基于某些参数值的梯度计算时，这些参数将被其他副本更新数次（如果有N个副本，则平均N-1次），并且不能保证计算出的梯度仍指向正确的方向（见图19-20）。当梯度严重过时时，它们被称为陈旧的梯度：它们可以减慢收敛速度，引入噪声和摆动效果（学习曲线可能包含暂时的振荡），甚至可以使训练算法的算法发散。

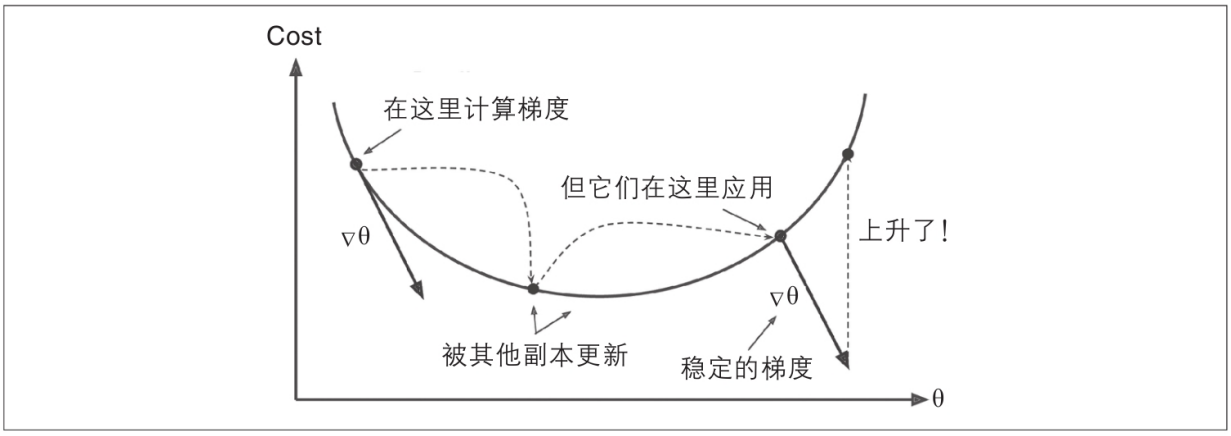


图19-20：使用异步更新时的陈旧的梯度

有几种方法可以减少陈旧的梯度的影响：

- 降低学习率。
- 删除陈旧的梯度或缩小比例。
- 调整小批量的大小。

• 在开始的前几个轮次中仅使用一个副本（这称为预热阶段）。在训练开始时，当梯度通常较大且参数尚未稳定到成本函数的谷底时，陈旧的梯度往往更具破坏性，因此不同的副本可能会将参数推向完全不同的方向。

Google Brain团队在2016年发表的一篇论文[3]对各种方法进行了基准测试，发现使用带有几个副本的同步更新比使用异步更新更有效，不仅收敛更快，而且生成更好的模型。但是这仍然是一个活跃的研究领域，因此你不应把异步更新排除在外。

带宽饱和

无论你使用同步更新还是异步更新，具有集中参数的数据并行仍然需要在每个训练步骤的开始将模型参数从参数服务器传递到每个副本，并在每个训练步骤结束时将梯度传递到另一个方向。类似地，当使用镜像策略时，每个GPU产生的梯度需要与其他所有GPU共享。不幸的是，总会出现这样的情况：添加额外的GPU根本不会提高性能，因为将数据移入和移出GPU内存（以及在分布式设置中跨网络）所花费的时间将超过通过拆分计算负载获得的加速时间。在这一点上，添加更多GPU只会加剧带宽饱和并实际上减慢训练速度。

 对于某些模型（通常相对较小并且在非常大的训练集上进行训练）而言，通常最好在具有单个强大GPU和大内存带宽的单台机器上训练模型。

对于大型密集模型，带宽饱和更为严重，因为它们有大量要传递的参数和梯度。对于较小的模型（但并行的好处有局限）和对于较大的稀疏模型而言，它的严重性较轻，在这些稀疏模型中，梯度通常大多为零，因此可以有效地进行通信。Google Brain项目的发起人和负责人Jeff Dean报告说，在密集型模型的50个GPU上分配计算时，典型的加速

比为25－40×，而在500个GPU上训练的稀疏模型，加速比为300×。如你所见，稀疏模型确实有更好的扩展性。以下是一些具体示例：

- 神经机器翻译：在8个GPU上加速6倍。
- Inception/ImageNet：在50个GPU上加速32倍。
- RankBrain：在500个GPU上加速300倍。

除了用于密集模型的数十个GPU或用于稀疏模型的数百个GPU之外，饱和现象不断出现，性能下降。为了解决这个问题，有大量研究正在进行（探索点对点架构而不是集中式参数服务器，使用有损模型压缩，优化副本何时以及需要传达什么内容，等等），因此在未来几年中，并行神经网络可能会取得很多进展。

同时，为了减少饱和度问题，你可能要使用一些功能强大的GPU，而不要使用大量功能较弱的GPU，并且还应将GPU在很少且互联良好的服务器上分组。你也可以将浮点精度从32位（tf.float32）降至16位（tf.bfloat16）。这会减少一半的数据传输，通常不会对收敛速度或模型的性能产生太大影响。最后如果你使用集中式参数，则可以在多个参数服务器之间分片（分割）参数：添加更多参数服务器会减少每台服务器上的网络负载并降低带宽饱和的风险。

好的，现在让我们跨多个GPU训练模型！

19.4.3　使用分布式策略API进行大规模训练

许多模型可以在单个GPU甚至CPU上进行很好的训练。但是如果培训太慢，则可以尝试将其分布在同一台计算机上的多个GPU之间。如果仍然太慢，请尝试使用功能更强大的GPU，或在计算机上添加更多GPU。如果你的模型是计算密集型（例如大型矩阵乘法），那么它在功能强大的GPU上会运行得更快，甚至可以尝试在Google Cloud AI Platform上使

用TPU，对于这些模型，TPU的运行速度甚至更快。但是如果你不能在同一台计算机上安装更多的GPU，或者TPU不适合你（例如，你的模型可能不会从TPU中获得太多收益，或者你想使用自己的硬件基础架构），那么你可以尝试在多台服务器上对其进行训练，每台服务器都具有多个GPU（如果仍然不够，可以尝试添加一些模型并行性，但这需要付出更多努力）。在本节中，我们将看到如何大规模训练模型，从同一台机器（或TPU）上的多个GPU开始，然后转移到跨多台机器的多个GPU。

幸运的是，TensorFlow附带了一个非常简单的API，它为你解决了所有这些复杂问题：分布式策略API。要使用数据并行和镜像策略在所有可用GPU上（目前仅在一台机器上）训练Keras模型，请创建MirroredStrategy对象，调用其scope（）方法得到分布式上下文，并在这个上下文中包装创建和编译模型。然后正常调用模型的fit（）方法：

```
distribution = tf.distribute.MirroredStrategy()

with distribution.scope():
    mirrored_model = keras.models.Sequential([...])
    mirrored_model.compile([...])

batch_size = 100 # must be divisible by the number of replicas
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

在后台，tf.keras是可以知道分布式的，因此在MirroredStrategy上下文中，它知道必须在所有可用的GPU设备上复制所有变量和操作。注意，fit（）方法会在所有副本上自动分割每个训练批量数据，因此，批量大小可被副本数量整除很重要。通常训练将比使用单个设备快得多，并且代码更改也很小。

训练完模型后，你可以使用它来高效地进行预测：调用predict（）方法，它将自动在所有副本之间分割批量数据，并行进行预测（同样批量大小必须是被可分割的副本数整除）。如果调用模型的save（）方法，它将被保存为常规模型，而不是具有多个副本的镜像模

型。在加载时，它会像常规模型一样在单个设备上运行（默认情况下为GPU 0，如果没有GPU，则为CPU）。如果要加载模型并在所有可用设备上运行它，你必须在分布式上下文中调用keras.models.load_model（）：

```
with distribution.scope():
    mirrored_model = keras.models.load_model("my_mnist_model.h5")
```

如果你只想使用所有可用GPU设备的子集，则可以将列表传递给MirroredStrategy的构造函数：

```
distribution = tf.distribute.MirroredStrategy(["/gpu:0", "/gpu:1"])
```

默认情况下，MirroredStrategy类对AllReduce的均值操作使用NVIDIA Collective Communications Library（NCCL），但是你可以通过将cross_device_ops参数设置为tf.distribute.HierarchicalCopyAllReduce类的实例或tf.distribute.Redu ctionToOneDevice的实例。默认的NCCL选项基于tf.distribute.NcclAllReduce类，该类通常更快，但这取决于GPU的数量和类型，因此你可能需要尝试一下替代方案[4]。

如果要尝试将数据并行与集中式参数一起使用，请将MirroredStrategy替换为CentralStorageStrategy：

```
distribution = tf.distribute.experimental.CentralStorageStrategy()
```

你可以选择设置compute_devices参数以指定要用作计算的设备列表（默认情况下，它将使用所有可用的GPU），还可以选择将parameter_device参数设置为指定要在其上存储参数的设备（默认情况下它将使用CPU或GPU，如果只有一个GPU的话）。现在让我们看看如何在TensorFlow服务器集群上训练模型！

19.4.4 在TensorFlow集群上训练模型

TensorFlow集群是一组并行运行的TensorFlow进程，通常在不同的机器上运行，并且互相通讯以完成一些工作——例如训练或执行神经网络。群集中的每个TF进程都称为任务或TF服务器。它具有IP地址、端口和类型（也称为其角色或工作）。类型可以是"worker""chief""ps"（参数服务器）或"evaluator"：

- 每个工人通常在具有一个或多个GPU的计算机上执行计算。

- 负责人也执行计算（这是一个工人），但它还处理额外的工作，例如编写TensorBoard日志或保存检查点。集群中只有一个负责人。如果未指定负责人，则第一个工人为负责人。

- 参数服务器仅跟踪变量值，通常位于只有CPU的计算机上。此类任务仅与ParameterServerStrategy一起使用。

- 评估者显然负责评估。

要启动TensorFlow集群，必须首先指定它。这意味着要定义每个任务的IP地址、TCP端口和类型。例如，以下集群规范定义了一个包含三个任务的集群（两个工人和一个参数服务器，见图19-21）。群集规范是一个字典，每个作业一个键，其值是任务地址列表（IP：端口）：

```
cluster_spec = {
    "worker": [
```

```
            "machine-a.example.com:2222",  # /job:worker/task:0
            "machine-b.example.com:2222"   # /job:worker/task:1
        ],
        "ps": ["machine-a.example.com:2221"] # /job:ps/task:0
    }
```

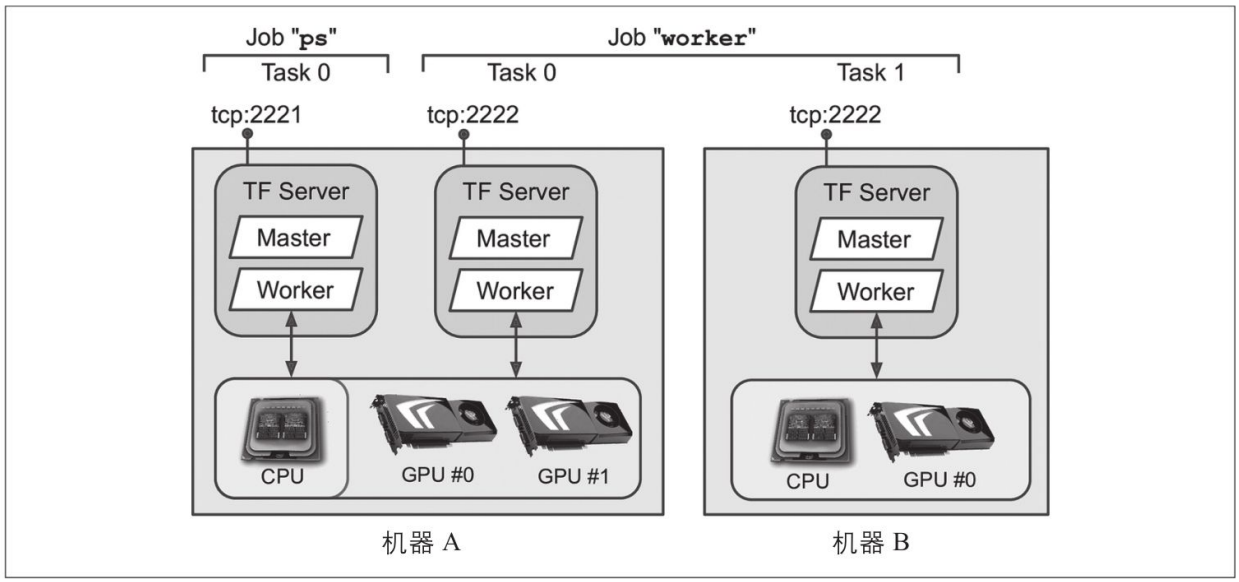


图19-21：TensorFlow集群

通常，每台计算机上只有一个任务，但是如本示例所示，你可以根据需要在同一台计算机上配置多个任务（如果它们共享相同的GPU，请确保内存被适当地分割）。

 默认情况下，群集中的每个任务都可能与其他任务进行通讯，请确保配置防火墙以授权这些端口上这些计算机之间的所有通信（如果在每台计算机上使用相同的端口，通常会更简单一些）。

启动任务时，必须给它一个集群规范，还必须告诉它类型和索引是什么（例如，工人0）。一次指定所有内容（集群规范以及当前任务的类型和索引）的最简单方法是在启动TensorFlow之前设置TF_CONFIG环境变量。它必须是JSON编码的字典，其中包含集群规范（在"cluster"键下）以及当前任务的类型和索引（在"task"键下）。

例如，以下TF_CON FIG环境变量使用我们刚刚定义的集群，并指定要开始的任务是第一个工人：

```
import os
import json

os.environ["TF_CONFIG"] = json.dumps({
    "cluster": cluster_spec,
    "task": {"type": "worker", "index": 0}
})
```

 通常，你想在Python之外定义TF_CONFIG环境变量，因此代码不需要包括当前任务的类型和索引（这使得在所有工人上使用相同的代码成为可能）。

现在让我们在集群上训练模型！从镜像策略开始，这非常简单！首先你需要为每个任务适当地设置TF_CONFIG环境变量。不应该有参数服务器（删除集群规格中的“ps”键），通常每台计算机你需要一个工人。请确保为每个任务设置不同的任务索引。最后，在每个工人上运行以下训练代码：

```
distribution = tf.distribute.experimental.MultiWorkerMirroredStrategy()

with distribution.scope():
    mirrored_model = keras.models.Sequential([...])
    mirrored_model.compile([...])

batch_size = 100 # must be divisible by the number of replicas
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

是的，这与我们之前使用的代码完全相同，只是这次我们使用的是MultiWorker MirroredStrategy（在将来的版本中，MirroredStrategy可能会处理单机和多机情况）。当你在第一个工人上启动此脚本时，它们将在AllReduce步骤中保持阻塞状态，但是当最后一个工人启动时，

训练将开始，并且你会看到它们以完全相同的速率前进（因为它们在每个步骤都同步）。

你可以从两种AllReduce实现中选择一种用于此分布式策略：基于gRPC的环形AllReduce算法用于网络通信以及NCCL的实现。要使用的最佳算法取决于工人的数量-GPU的数量和类型以及网络。默认情况下，TensorFlow会使用启发式方法为你选择合适的算法，但是如果你要强制使用一种算法，请将CollectiveCommunication.RING或CollectiveCommunication.NCCL（来自tf.distribute.experimental）传递给该策略的构造函数。

如果你希望实现带有参数服务器的异步数据并行，请将策略更改为ParameterServerStrategy，添加一个或多个参数服务器，并为每个任务适当地配置TF_CONFIG。请注意尽管工人是异步工作的，但每个工人上的副本会同步工作。

最后，如果你可以访问Google Cloud上的TPU，则可以这样创建TPUStrategy（然后像其他策略一样使用它）：

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()
tf.tpu.experimental.initialize_tpu_system(resolver)
tpu_strategy = tf.distribute.experimental.TPUStrategy(resolver)
```

 如果你是研究人员，则可能有资格免费使用TPU，可访问https://tensorflow.org/tfrc了解更多详情。

你现在可以在多个GPU和多个服务器上训练模型：赞扬一下自己！如果要训练大型模型，则需要在许多服务器上使用许多GPU，这需要购买大量硬件或管理大量云虚拟机。在许多情况下，使用云服务可以在你

需要时为你配置和管理所有这些基础结构，从而省事又省钱。让我们看看在GCP上需要如何做。

19.4.5 在Google Cloud AI平台上运行大型训练作业

如果你决定使用Google AI平台，可以使用与在自己的TF集群上运行的训练代码相同的代码来部署训练作业，该平台会负责根据你的需要来配置尽可能多的GPU VM（在你的配额之内）。

开始工作之前，你需要gcloud命令行工具，该工具是Google Cloud SDK的一部分。你既可以在自己的计算机上安装SDK，也可以在GCP上使用Google Cloud Shell。这是一个可以直接在Web浏览器中使用的终端。它可以在免费的Linux VM（Debian）上运行，并且已经为你安装并预先配置了SDK。在GCP中的任何地方都可以使用Cloud Shell：只需单击页面右上方的“Activate Cloud Shell”图标（见图19-22）。

图19-22：激活Google Cloud Shell

如果你希望在你的计算机上安装SDK，则在安装后，需要通过运行gcloud init对其进行初始化：你需要登录GCP并授予对GCP资源的访问权限，然后选择你想使用的GCP项目（如果有多个），以及要运行任务的地区。gcloud命令可让你访问所有GCP功能，包括我们先前使用的功能。你不必每次都通过Web界面。你可以编写脚本来启动或停止VM、部署模型或执行任何其他GCP操作。

在运行训练任务之前，你需要编写训练代码，就像你之前为分布式设置所做的一样（例如使用ParameterServerStrategy）。AI平台会在

每个VM上为你设置TF_CONFIG。完成此操作后，你可以使用以下命令行将其部署并在TF群集上运行：

```
$ gcloud ai-platform jobs submit training my_job_20190531_164700 \
    --region asia-southeast1 \
    --scale-tier PREMIUM_1 \
    --runtime-version 2.0 \
    --python-version 3.5 \
    --package-path /my_project/src/trainer \
    --module-name trainer.task \
    --staging-bucket gs://my-staging-bucket \
    --job-dir gs://my-mnist-model-bucket/trained_model \
    --
    --my-extra-argument1 foo --my-extra-argument2 bar
```

让我们来看看这些选项。该命令将使用PREMIUM_1扩展等级在asia-southeast1区域中启动名为my_job_20190531_164700的训练任务：这对应于20个工人（包括一个负责人）和11个参数服务器。所有这些VM都基于AI平台的2.0运行时（包含TensorFlow 2.0和许多其他软件包的VM配置）[5]和Python 3.5。训练代码位于/my_project/src/trainer目录中，并且gcloud命令会自动将其捆绑到一个pip包中，并将其上传到gs：//my-staging-bucket上。接下来，AI平台将启动多个VM，并在其上部署程序包，然后运行trainer.task模块。最后，--job-dir参数和其他参数（即位--分隔之后的所有参数）将被传递到训练程序：主要任务通常是使用--job-dir参数来查找将最终模型保存在GCS上的位置，在本例中为gs：//my-mnist-model-bucket/trained_model。在GCP控制台中，可以打开导航菜单，向下滚动到“Artificial Intelligence”部分，然后打开AI Platform→Jobs。你应该看到任务正在运行，如果单击它，你会看到显示每个任务的CPU、GPU和RAM利用率的图表。你可以单击“View Logs”来使用Stackdriver查看详细日志。

 如果你将训练数据放在GCS上，则可以创建一个tf.data.TextLineDataset或tf.data.TFRecordDataset来访问它：只需使用GCS路径作为文件名即可（例如，gs：//my-data-bucket/my_data_001.csv）。这些数据集依赖于tf.io.gfile包来访问文件：它支持本地文件和GCS文件（但请确保你使用的服务账户可以访问GCS）。

你如果要探索一些超参数值，则可以简单地运行多个任务并使用任务的额外参数指定超参数值。但是如果要有效地探索许多超参数，最好使用AI平台的超参数调整服务。

19.4.6 AI平台上的黑箱超参数调整

AI平台提供了一个功能强大的贝叶斯优化超参数调整服务，称为Google Vizier[6]。要使用它，你需要在创建作业时传递YAML配置文件（--config tuning.yaml）。它可能看起来像这样：

```
trainingInput:
  hyperparameters:
    goal: MAXIMIZE
    hyperparameterMetricTag: accuracy
    maxTrials: 10
    maxParallelTrials: 2
    params:
      - parameterName: n_layers
        type: INTEGER
        minValue: 10
        maxValue: 100
        scaleType: UNIT_LINEAR_SCALE
      - parameterName: momentum
        type: DOUBLE
        minValue: 0.1
        maxValue: 1.0
        scaleType: UNIT_LOG_SCALE
```

这告诉AI平台我们要最大化叫"accuracy"的指标，该任务最多将运行10次试验（每个试验都从头运行我们的训练代码来训练模型），最多并行2个试验。我们希望它调整两个超参数：超参数n_layers（10到100

之间的整数）和超参数momentum（0.1到1.0之间的浮点数）。参数scaleType指定超参数值的先验值：UNIT_LINEAR_SCALE表示平均先验值（即没有先验偏好），而UNIT_LOG_SCALE说我们先验地认为最优值更接近最大值（当我们认为最佳值接近最小值时，另一个可能的先验是UNIT_REVERSE_LOG_SCALE）。

参数n_layers和momentum将作为命令行参数传递给训练代码，我们当然可以使用它们。问题是训练代码如何将指标传递回AI平台，以便它可以决定在下一个试验期间使用哪个超参数值？好的，AI平台会监视输出目录（通过--job-dir指定）中是否有任何事件文件（见第10章），该文件中包含名为“accuracy”（或者任何被hyperparameterMetricTag指定的指标）的指标的摘要，并读取这些值。因此你的训练代码只需使用TensorBoard（）回调（无论如何你都希望执行此回调来进行监控），你就可以开始了！

任务完成后，每个试验中使用的所有超参数值以及由此产生的准确率在任务输出中可得（可通过AI Platform→Jobs页获得）。

 AI Platform作业还可以用于有效地对大量数据执行你的模型：每个工人都可以从GCS读取部分数据、做出预测并将其保存到GCS。

现在，你已经拥有创建最先进的神经网络架构，并使用各种分布式策略在自己的硬件基础架构或云上进行大规模训练所需的所有工具和知识，甚至可以执行强大的贝叶斯优化来微调超参数！

[1] 如果你有兴趣进一步研究模型并行性，请查看Mesh TensorFlow。
[2] 这个名字有点让人困惑，因为听起来有些复制品很特殊，什么也没做。实际上所有副本都是等效的：它们在每个训练步骤中都努力成为最快的那个副本，而失败者在每个步骤中都会有所不同（除非某些设备确实比其他设备慢）。这意味着如果服务器崩溃，训练将继续进行。

[3] Jianmin Chen et al.，“Revisiting Distributed Synchronous SGD”，arXiv preprint arXiv：1604.00981（2016）.

[4] 有关AllReduce算法的更多详细信息，请阅读Yuichiro Ueno撰写的精彩文章，以及有关scaling with NCCL的页面。

[5] 在撰写本书时，2.0运行时尚不可用，但在你阅读本书时应已准备就绪。

[6] Daniel Golovin et al.，“Google Vizier：A Service for Black-Box Optimization”，Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining（2017）：1487－1495.

19.5 练习题

1.SavedModel包含什么？如何检查其内容？

2.何时应使用TF Serving？它的主要特点是什么？可以使用哪些工具进行部署？

3.如何跨多个TF Serving实例部署一个模型？

4.什么时候应该使用gRPC API而不是REST API来查询一个被TF Serving服务的模型？5.TFLite减少模型大小以使其在移动端或嵌入式设备上运行的不同方式是什么？

6.什么是量化意识训练，为什么你需要它？

7.什么是模型并行和数据并行？为什么通常建议使用后者？

8.在多台服务器上训练模型时，可以使用什么分布式策略？如何选择使用哪一个？

9.训练模型（你喜欢的任何模型）并将其部署到TF Serving或Google Cloud AI平台。编写客户端代码并使用REST API或gRPC对其进行查询。更新模型并部署新版本。你的客户代码现在会查询新版本。回滚到第一个版本。

10.使用镜像策略（Mirrored Strategy）在同一台计算机上的多个GPU上训练任何模型（如果你没有GPU，你可以用带有GPU Runtime的Colaboratory来创建两个虚拟GPU）。使用CentralStorageStrategy再次训练模型并比较训练时间。

11.使用黑箱超参数调整在Google Cloud AI平台上训练一个小模型。

19.6 致谢

在结束本书的最后一章之前，感谢你阅读到本书的最后一段。我真诚地希望你阅读本书时和我写作本书时一样快乐，并希望本书对你的项目（无论大小）都有用。

如果发现错误，请发送反馈。期待与大家交流，所以请随时通过O’Reilly、ageron/handson-ml2 GitHub项目或通过Twitter@aureliengeron与我联系。

展望未来，我对你最好的建议是练习再练习：尝试完成所有练习（如果尚未完成），玩Jupyter notebook，加入Kaggle.com或其他一些ML社区，观看ML课程，阅读论文，参加会议并与专家会面。无论是工作还是娱乐（最好两者兼具），有一个具体的项目都可以极大地帮助你，因此，如果你有梦想做的一些事，不妨尝试一下！增量地进行工作，不要想着一步登天，而要专注于你的项目并逐步实现它。这需要耐心和恒心，但是当你拥有步行机器人、聊天机器人或你想建造的其他任何东西时，回报将是巨大的。

我最大的希望是本书会激发你创建一个使我们所有人都受益的出色的ML应用程序！那会是什么呢？

——Aurélien Géron，2019年6月17日

附录A　课后练习题解答

 有关代码练习的解答，请参见https://github.com/ageron/handson-ml2上的在线Jupyter notebook。

第1章：机器学习概览

1.机器学习是关于构建可以从数据中学习的系统。学习意味着在一定的性能指标下，在某些任务上会变得越来越好。

2.机器学习非常适合没有算法解答的复杂问题，它可以替代一系列需要手动调整的规则，来构建适应不断变化的环境的系统并最终帮助人类（例如，数据挖掘）。

3.带标签的训练集是一个包含每个实例所需解决方案（也称为标签）的训练集。

4.两个最常见的有监督任务是回归和分类。

5.常见的无监督任务包括聚类、可视化、降维和关联规则学习。

6.如果我们想要机器人在各种未知的地形中学习行走，则强化学习可能会表现最好，因为这通常是强化学习要解决的典型问题。也可以将强化学习问题表示为有监督学习或半监督学习问题，但这种情况不是很自然的想法。

7.如果你不知道如何定义组，则可以使用聚类算法（无监督学习）将客户划分为相似客户集群。但是，如果你知道你想要拥有哪些组，那

么可以将每个组的许多实例提供给分类算法（有监督学习），并将所有客户分类到这些组中。

8.垃圾邮件检测是一个典型的有监督学习问题：向该算法提供许多电子邮件及其标签（垃圾邮件或非垃圾邮件）。

9.与批量学习系统相反，在线学习系统能够进行增量学习。这使得它能够快速适应不断变化的数据和自动系统，并能够处理大量数据。

10.核外算法可以处理无法容纳在计算机主内存中的大量数据。核外学习算法将数据分成小批量，并使用在线学习技术从这些小批量数据中学习。

11.基于实例的学习系统努力通过死记硬背来学习训练数据。然后，当给定一个新的实例时，它将使用相似性度量来查找最相似的实例，并利用它们来进行预测。

12.一个模型具有一个或多个模型参数，这些参数确定在给定一个新实例的情况下该模型将预测什么（例如，线性模型的斜率）。一种学习算法试图找到这些参数的最优值，以使该模型能很好地泛化到新实例。超参数是学习算法本身的参数，而不是模型的参数（例如，要应用正则化的数量）。

13.基于模型的学习算法搜索模型参数的最优值，以便模型可以很好地泛化到新实例。我们通常通过最小化成本函数来训练这样的系统，该函数测量系统对训练数据进行预测时有多不准确，如果对模型进行了正则化则对模型复杂性要加上惩罚。为了进行预测，我们使用学习算法找到的模型参数值，再将新实例的特征输入到模型的预测函数中。

14.机器学习中的一些主要挑战是数据的缺乏、数据质量差、数据的代表性不足、信息量不足、模型过于简单而欠拟合训练数据以及模型过于复杂而过拟合数据。

15. 如果模型在训练数据上表现出色，但在新实例上的泛化效果很差，则该模型可能会过拟合训练数据（或者我们在训练数据上非常幸运）。过拟合的可能解决方法是获取更多数据、简化模型（选择更简单的算法，减少使用的参数或特征的数量，或对模型进行正则化）或减少训练数据中的噪声。

16. 测试数据集是用于在启动生产环境之前，估计模型在新实例上产生的泛化误差。

17. 验证集是用于比较模型。这样就可以选择最佳模型并调整超参数。

18. 当训练数据集与验证数据集和测试数据集中使用的数据之间不匹配时，可以使用train-dev集（该数据集应始终与模型投入生产环境后使用的数据尽可能接近）。train-dev集是训练集的一部分（模型未在其上训练过）。该模型在训练集的其他部分上进行训练，并在train-dev集和验证集上进行评估。如果模型在训练集上表现良好，但在train-dev集上表现不佳，则该模型可能过拟合训练集。如果它在训练集和train-dev集上均表现良好，但在验证集上却表现不佳，那么训练数据与验证数据和测试数据之间可能存在明显的数据不匹配，你应该尝试改善训练数据，使其看起来更像验证数据和测试数据。

19. 如果使用测试集来调整超参数，则可能会过拟合测试集，而且所测得的泛化误差会过于乐观（你可能会得到一个性能比预期差的模型）。

第2章：端到端的机器学习项目

请参阅https://github.com/ageron/handson-ml2上的Jupyter notebook。

第3章：分类

请参阅https://github.com/ageron/handson-ml2上的Jupyter notebook。

第4章：训练模型

1.如果你的训练集具有数百万个特征，则可以使用随机梯度下降或小批量梯度下降。如果训练集适合容纳于内存，则可以使用批量梯度下降。但是你不能使用标准方程法或SVD方法，因为随着特征数量的增加，计算复杂度会快速增长（超过二次方）。

2.如果你的训练集中的特征具有不同的尺寸比例，则成本函数具有细长碗的形状，因此梯度下降算法需要很长时间才能收敛。为了解决这个问题，你应该在训练模型之前缩放数据。请注意，标准方程法或SVD方法无须缩放即可正常工作。此外，如果特征未按比例缩放，则正则化模型可能会收敛至次优解：由于正则化会惩罚较大的权重，因此与具有较大值的特征相比，具有较小值的特征往往会被忽略。

3.训练逻辑回归模型时，梯度下降不会陷入局部最小值，因为成本函数是凸函数[1]。

4.如果优化问题是凸的（例如线性回归或逻辑回归），并且假设学习率不是太高，那么所有梯度下降算法都将接近全局最优并最终产生很相似的模型。但是，除非逐步降低学习率，否则随机梯度下降和小批量梯度下降将永远不会真正收敛。相反，它们会一直围绕全局最优值来回跳跃。这意味着即使你让它们运行很长时间，这些梯度下降算法也会产生略微不同的模型。

5.如果验证错误在每个轮次后持续上升，则一种可能性是学习率过高并且算法在发散。如果训练错误也增加了，那么这显然是问题所在，你应该降低学习率。但是，如果训练错误没有增加，则你的模型已经过拟合训练集，则应该停止训练。

6.由于随机性，随机梯度下降和小批量梯度下降都不能保证在每次训练迭代中都取得进展。因此，如果在验证错误上升时立即停止训练，则可能在达到最优值之前就停止太早了。更好的选择是按照一定的间隔时间保存模型。然后，当它很长时间没有改善（意味着它可能永远不会超过最优值）时，你可以恢复到保存的最佳模型。

7.随机梯度下降法具有最快的训练迭代速度，因为它一次只考虑一个训练实例，因此它通常是第一个到达全局最优值附近的（或是很小批量大小的小批量梯度下降）。但是，给定足够的训练时间，实际上只有批量梯度下降会收敛。如前所述，随机梯度下降和小批量梯度下降会在最优值附近反弹，除非你逐渐降低学习率。

8.如果验证误差远高于训练误差，则可能是因为模型过拟合了训练集。解决此问题的一种方法是降低多项式阶数：较小自由度的模型不太可能过拟合。另一种方法是对模型进行正则化，例如，将ℓ_2（Ridge）或ℓ_1（Lasso）惩罚添加到成本函数。这也会降低模型的自由度。最后，你可以尝试增加训练集的大小。

9.如果训练误差和验证误差几乎相等且相当高，则该模型很可能欠拟合训练集，这意味着它具有很高的偏差。你应该尝试减少正则化超参数α。

10.让我们来看看：

•具有某些正则化的模型通常比没有任何正则化的模型要好，因此，你通常应优先选择岭回归而不是简单的线性回归。

•Lasso回归使用ℓ_1惩罚，这通常会将权重降低为零。这将导致稀疏模型，其中除了最重要的权重之外，所有权重均为零。这是一种自动进行特征选择的方法，如果你怀疑实际上只有很少的特征很重要，那么这是一种很好的方法。如果你不确定，则应首选岭回归。

• 与Lasso相比，弹性网络通常更受青睐，因为Lasso在某些情况下可能产生异常（当几个特征强相关或当特征比训练实例更多时）。但是，它确实增加了额外需要进行调整的超参数。如果你希望Lasso没有不稳定的行为，则可以仅使用l1_ratio接近1的弹性网络。

11. 如果你要将图片分类为室外/室内和白天/夜间，因为它们不是排他的类（即所有四种组合都是可能的），则应训练两个逻辑回归分类器。

12. 请参阅https://github.com/ageron/handson-ml2上的Jupyter notebooks。

第5章：支持向量机

1. 支持向量机的基本思想是拟合类别之间可能的、最宽的“街道”。换言之，它的目的是使决策边界之间的间隔最大化，该决策边界分隔两个类别和训练实例。SVM执行软间隔分类时，实际上是在完美分隔两个类和拥有尽可能最宽的街道之间寻找折中方法（也就是允许少数实例最终还是落在街道上）。还有一个关键点是在训练非线性数据集时，记得使用核函数。

2. 支持向量机的训练完成后，位于“街道”（参考上一个答案）之上的实例被称为支持向量，这也包括处于边界上的实例。决策边界完全由支持向量决定。非支持向量的实例（也就是街道之外的实例）完全没有任何影响。你可以选择删除它们然后添加更多的实例，或者将它们移开，只要一直在街道之外，它们就不会对决策边界产生任何影响。计算预测结果只会涉及支持向量，而不涉及整个训练集。

3. 支持向量机拟合类别之间可能的、最宽的“街道”（参考第1题答案），所以如果训练集不经缩放，SVM将趋于忽略值较小的特征（见图5-2）。

4.支持向量机分类器能够输出测试实例与决策边界之间的距离，你可以将其用作信心分数。但是这个分数不能直接转化成类别概率的估算。如果创建SVM时，在Scikit-Learn中设置probability=True，那么训练完成后，算法将使用逻辑回归对SVM分数进行校准（对训练数据额外进行5-折交叉验证的训练），从而得到概率值。这会给SVM添加predict_proba（）和predict_log_proba（）两种方法。5.这个问题仅适用于线性支持向量机，因为核SVM只能使用对偶问题。对于SVM问题来说，原始形式的计算复杂度与训练实例m的数量成正比，而其对偶形式的计算复杂度与某个介于m2和m3之间的数量成正比。所以如果实例的数量以百万计，一定要使用原始问题，因为对偶问题会非常慢。

6.如果一个使用RBF核训练的支持向量机对欠拟合训练集，可能是由于过度正则化导致的。你需要提升gamma或C（或同时提升二者）来降低正则化。

7.我们把硬间隔问题的QP参数定义为H'、f'、A'及b'（见5.4.3节）。软间隔问题的QP参数还包括m个额外参数（n_p=n+1+m）及m个额外约束（n_c=2m）。它们可以这样定义：

H等于在H'右侧和底部分别加上m列和m行个0：

$$\boldsymbol{H} = \begin{pmatrix} \boldsymbol{H'} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{pmatrix}$$

f等于有m个附加元素的f'，全部等于超参数C的值。

b等于有m个附加元素的b'，全部等于0。

A等于在A'的右侧添加一个m×m的单位矩阵I_m，在这个单位矩阵的正下方再添加单位矩阵-*I*$_m$，剩余部分为0：

$$A = \begin{pmatrix} A' & I_m \\ 0 & -I_m \end{pmatrix}$$

对于练习8～10的解答，请参见Jupyter notebooks：https://github.com/ageron/handsonml2的Jupyter notbook。

第6章：决策树

1.一个包含m个叶节点的均衡二叉树的深度等于$\log_2$（m）（注：$\log_2$是基2对数，$\log_2$（m）=log（m）/log（2）。），取整。通常来说，二元决策树（只做二元决策的树，就像Scikit-Learn中的所有树一样）训练到最后大体都是平衡的，如果不加以限制，最后平均每个叶节点一个实例。因此，如果训练集包含100万个实例，那么决策树的深度为$\log_2$（10^6）≈20层（实际上会更多一些，因为决策树通常不可能完美平衡）。

2.一个节点的基尼不纯度通常比其父节点低。这是由于CART训练算法的成本函数。该算法分裂每个节点的方法，就是使其子节点的基尼不纯度的加权之和最小。但是，如果一个子节点的不纯度远小于另一个，那么也有可能使子节点的基尼不纯度比其父节点高，只要那个不纯度更低的子节点能够抵偿这个增加即可。举例来说，假设一个节点包含4个A类的实例和1个B类的实例，其基尼不纯度等于$1-\left(\frac{1}{5}\right)^2-\left(\frac{4}{5}\right)^2=032$。现在我们假设数据集是一维的，并且实例的排列顺序如下：A，B，A，A，A。你可以验证算法将在第二个实例后拆分该节点，从而生成两个子节点，分别包含的实例为A，B和A，A，A。第一个子节点的基尼不纯度为1-（1/2）^{2}-（1/2）^{2}=0.5，比其父节点要高。这是因为第二个子节点

是纯的，所以总的加权基尼不纯度等于$\frac{2}{5}\times 0.5+\frac{3}{5}\times 0=0.2$，低于父节点的基尼不纯度。

3.如果决策树过拟合训练集，降低max_depth可能是一个好主意，因为这会限制模型，使其正则化。

4.决策树的优点之一就是它们不关心训练数据是缩放还是集中，所以如果决策树不适合训练集，缩放输入特征不过是浪费时间罢了。

5.决策树的训练复杂度为O（n×mlog（m））。所以，如果将训练集大小乘以10，训练时间将乘以K=（n×10m×log（10m））/（n×m×log（m））=10×log（10m）/log（m）。如果$m=10^6$，那么K≈11.7，所以训练1000万个实例大约需要11.7小时。

6.只有当数据集小于数千个实例时，预处理训练集才可以加速训练。如果包含100 000个实例，设置presort=True会显著减慢训练。

对于练习7和练习8的解答，参见Jupyter notebooks：https://github.com/ageron/handson-ml2的Jupyter notebook。

第7章：集成学习和随机森林

1.如果你已经训练了5个不同的模型，并且都达到了95%的精度，则可以尝试将它们组合成一个投票集成，这通常会带来更好的结果。如果模型之间非常不同（例如，一个SVM分类器、一个决策树分类器，以及一个Logistic回归分类器等），则效果更优。如果它们是在不同的训练实例（这是bagging和pasting集成的关键点）上完成训练，那就更好了，但如果不是，只要模型非常不同，这个集成仍然有效。

2.硬投票分类器只是统计每个分类器的投票，然后挑选出得票最多的类。软投票分类器计算出每个类的平均估算概率，然后选出概率最高

的类别。它比硬投票法的表现更优，因为它给予那些高度自信的投票更高的权重。但是它要求每个分类器都能够估算出类别概率才可以正常工作（例如，Scikit-Learn中的SVM分类器必须要设置probability=True）。

3.对于bagging集成来说，将其分布在多个服务器上能够有效加速训练过程，因为集成中的每个预测器都是独立工作的。同理，对于pasting集成和随机森林来说也是如此。但是，boosting集成的每个预测器都是基于其前序的结果，因此训练过程必须是有序的，将其分布在多个服务器上毫无意义。对于stacking集成来说，某个指定层的预测器之间彼此独立，因而可以在多台服务器上并行训练，但是，某一层的预测器只能在其前一层的预测器全部训练完成之后才能开始训练。

4.包外评估可以对bagging集成中的每个预测器使用其未经训练的实例（它们是被保留的）进行评估。不需要额外的验证集，就可以对集成实施相当公正的评估。所以，如果训练使用的实例越多，集成的性能可以略有提升。

5.随机森林在生长过程中，每个节点的分裂仅考虑到了特征的一个随机子集。极限随机树也是如此，它甚至走得更远：常规决策树会搜索出特征的最佳阈值，极端随机树直接对每个特征使用随机阈值。这种极端随机性就像是一种正则化的形式：如果随机森林过拟合训练数据，那么极端随机树可能执行效果更好。而且，由于极端随机树不需要计算最佳阈值，因此它训练起来比随机森林快得多。但是，在做预测的时候，相比随机森林它不快也不慢。

6.如果你的AdaBoost集成欠拟合训练集，可以尝试提升估算器的数量或是降低基础估算器的正则化超参数。你也可以尝试略微提升学习率。

7.如果你的梯度提升集成过拟合训练集，你应该试着降低学习率，也可以通过提前停止法来寻找合适的预测器数量（可能是因为预测器太

多）。

对于练习8和练习9的解答，参见https://github.com/ageron/handson-ml2的Jupyter notebook。

第8章：降维

1.降维的主要动机是：

• 为了加速后续的训练算法（在某些情况下，也可能为了消除噪声和冗余特征，使训练算法性能更好）。

• 为了将数据可视化，并从中获得洞见，了解最重要的特征。

• 为了节省空间（压缩）。

主要的弊端是：

• 丢失部分信息，可能使后续训练算法的性能降低。

• 可能是计算密集型的。

• 为机器学习流水线增添了些许复杂度。

• 转换后的特征往往难以解释。

2.维度的诅咒是指许多在低维空间中不存在的问题，在高维空间中发生。在机器学习领域，一个常见的现象是随机抽样的高维向量通常非常稀疏，提升了过拟合的风险，同时也使得在没有充足训练数据的情况下，要识别数据中的模式非常困难。

3.一旦使用我们讨论的任意算法减少了数据集的维度，就几乎不可能再将操作完美地逆转，因为在降维过程中必然丢失了一部分信息。此外，虽然有一些算法（例如PCA）拥有简单的逆转换过程，可以重建出与原始数据集相似的数据集，但是也有一些算法不能实现逆转（例如T-SNE）。

4.对大多数数据集来说，PCA可以用来进行显著降维，即便是高度非线性的数据集，因为它至少可以消除无用的维度。但是如果不存在无用的维度（例如瑞士卷），那么使用PCA降维将会损失太多信息。你希望的是将瑞士卷展开，而不是将其压扁。

5.这是个不好回答的问题，它取决于数据集。我们来看看两个极端的示例。首先，假设数据集是由几乎完全对齐的点组成的，在这种情况下，PCA可以将数据集降至一维，同时保留95%的方差。现在，试想数据集由完全随机的点组成，分散在1000个维度上，在这种情况下，需要在950个维度上保留95%的方差。所以，这个问题的答案是：取决于数据集，它可能是1到950之间的任何数字。将解释方差绘制成关于维度数量的函数，可以对数据集的内在维度获得一个粗略的概念。

6.常规PCA是默认选择，但是它仅适用于内存足够处理训练集的时候。增量PCA对于内存无法支持的大型数据集非常有用，但是它比常规PCA要慢一些，所以如果内存能够支持，还是应该使用常规PCA。当你需要随时应用PCA来处理每次新增的实例时，增量PCA对于在线任务同样有用。当你想大大降低维度数量，并且内存能够支持数据集时，使用随机PCA非常有效，它比常规PCA快得多。最后，对于非线性数据集，使用核化PCA非常有效。

7.直观来说，如果降维算法能够消除许多维度并且不会丢失太多信息，那么这就算一个好的降维算法。进行衡量的方法之一是应用逆转换然后测量重建误差。然而并不是所有的降维算法都提供了逆转换。还有另一种选择，如果你将降维当作一个预处理过程，用在其他机器学习算法（比如随机森林分类器）之前，那么可以通过简单测量第二个算法的

性能来进行评估。如果降维过程没有损失太多信息，那么第二个算法的性能应该跟使用原始数据集一样好。

8.链接两个不同的降维算法绝对是有意义的。常见的示例是使用PCA快速去除大量无用的维度，然后应用另一种更慢的降维算法，如LLE。这种两步走的策略产生的结果可能与仅使用LLE相同，但是时间要短得多。

有关练习9和10的解答，请参见https://github.com/ageron/handson-ml2上提供的Jupyter notebook。

第9章：无监督学习技术

1.在机器学习中，聚类是将相似的实例组合在一起的无监督任务。相似性的概念取决于你手头的任务：例如，在某些情况下，两个附近的实例将被认为是相似的，而在另一些情况下，只要它们属于同一密度组，则相似的实例可能相距甚远。流行的聚类算法包括K-Means、DBSCAN、聚集聚类、BIRCH、均值平移、亲和度传播和光谱聚类。

2.聚类算法的主要应用包括数据分析、客户分组、推荐系统、搜索引擎、图像分割、半监督学习、降维、异常检测和新颖性检测。

3.肘部法则是一种在使用K-Means时选择集群数的简单技术：将惯量（从每个实例到其最近的中心点的均方距离）作为集群数量的函数绘制出来，并找到曲线中惯量停止快速下降的点（“肘”）。另一种方法是将轮廓分数作为集群数量的函数绘制出来。通常最佳集群数是在一个高峰的附近。轮廓分数是所有实例上的平均轮廓系数。对于位于集群内且与其他集群相距甚远的实例，该系数为+1；对于与另一集群非常接近的实例，该系数为-1。你也可以绘制轮廓图并进行更细致的分析。

4.标记数据集既昂贵又费时。因此，通常有很多未标记的实例，很少有标记的实例。标签传播是一种技术，该技术包括将部分（或全部）标签从已标记的实例复制到相似的未标记实例。这可以大大增加标记实例的数量，从而使监督算法达到更好的性能（这是半监督学习的一种形式）。一种方法是在所有实例上使用诸如K-Means之类的聚类算法，然后为每个集群找到最常见的标签或最具代表性的实例（即最接近中心点的实例）的标签并将其传播到同一集群中未标记的实例。

5.K均值和BIRCH可以很好地扩展到大数据集。DBSCAN和Mean-Shift寻找高密度区域。

6.当你有大量未标记的实例而做标记非常昂贵时，主动学习就非常有用。在这种情况下（非常常见），与其随机选择实例来做标记，不如进行主动学习，这通常是更可取的一种方法，人类专家可以与算法进行交互，并在算法有需要时为特定实例提供标签。常见的方法是不确定性采样（见9.1.5节的“主动学习”）。

7.许多人把术语异常检测和新颖性检测互换，但是它们并不完全相同。在异常检测中，算法对可能包含异常值的数据集进行训练，目标通常是识别这些异常值（在训练集中）以及新实例中的异常值。在新颖性检测中，该算法在假定为“干净”的数据集上进行训练，其目的是严格在新实例中检测新颖性。某些算法最适合异常检测（例如隔离森林），而其他算法更适合新颖性检测（例如单类SVM）。

8.高斯混合模型（GMM）是一种概率模型，它假定实例是由参数未知的多个高斯分布的混合生成的。换句话说，我们假设数据可以分为有限数量的集群，每个集群具有椭圆的形状（但是集群可能具有不同的椭圆形状、大小、方向和密度），而我们不知道每个实例属于哪个簇。该模型可用于密度估计、聚类和异常检测。

9.使用高斯混合模型时，找到正确数量的集群的一种方法是将贝叶斯信息准则（BIC）或赤池信息准则（AIC）作为集群数量的函数绘制出

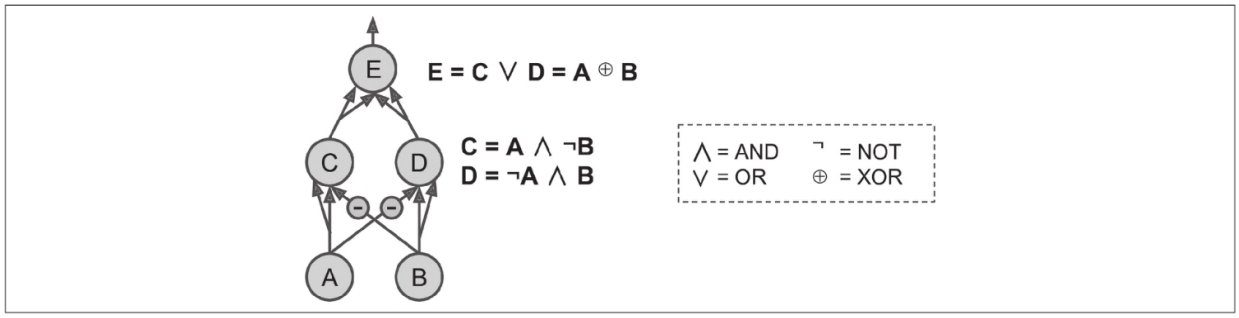
来，然后选择使BIC或AIC最小化的集群数量。另一种技术是使用贝叶斯高斯混合模型，该模型可以自动选择集群数。

有关练习10～13的解答，请参见https://github.com/ageron/handson-ml2上提供的Jupyter notebook。

第10章：Keras人工神经网络简介

1.如本练习所述，访问TensorFlow Playground并使用它。

2.这是基于原生人造神经元来计算（表示计算异或）的神经网络，A⊕B=（A∧¬B）∨（¬A∧B）。当然还有其他做法，比如使用A⊕B=（A∨B）∧¬（A∧B），或者A⊕B=（A∨B）∧（¬A∧B），等等。



3.经典的感知器只有在数据集是线性可分的情况下才会收敛，并且不能估计分类的概率。相反，逻辑回归分类器即使在数据集不是线性可分的情况下也可以很好地收敛，而且还能输出分类的概率。如果你将感知器的激活函数修改为逻辑激活函数（或者如果有多个神经元的时候，采用softmax激活函数），然后使用梯度下降训练它（或者使成本函数最小化的一些其他优化算法，通常是交叉熵法），那么它就会变为一个逻辑回归分类器了。

4.逻辑激活函数是训练第一个MLP的关键因素，因为它的导数总是非零的，所以梯度下降总是可以持续的。当激活功能是一个阶梯函数时，渐变下降就不能再持续了，因为这时候根本没有斜率。

5.常用的激活函数包括阶跃函数、逻辑（sigmoid）函数、双曲正切（tanh）函数和线性整流（ReLU）函数（见图10-8）。有关其他示例，请参见第11章，例如ELU和ReLU函数的各种变体。

6.考虑问题中描述的MLP：你有这样一个MLP，其输入层由10个透传神经元组成，隐藏层有50个人工神经元，输出层有3个人工神经元。所有的人工神经元都使用ReLU激活函数。输入矩阵X的形状是m×10，其中m代表训练批次的大小。

a.隐藏层的权重向量W_h的形状为10×50，其偏置向量b_h的长度为50。

b.输出层的权重向量W_o的形状为50×3，其偏置向量b_o的长度为3。

c.输出矩阵Y的形状是m×3。

d.$Y^*=ReLU(ReLU(X\ W_h+b_h)\ W_o+b_o)$。回想一下ReLU函数只是把矩阵中的每个负数都设置为零。注意，当你将一个偏置向量添加到一个矩阵时，它将被添加到矩阵中的每一行，这就是所谓的广播。

7.要将电子邮件分类为垃圾邮件和正常邮件，你只需要在神经网络的输出层中使用一个神经元——例如，指出电子邮件是垃圾邮件的概率。估算概率时，通常会在输出层使用逻辑激活函数。如果你想要解决MNIST问题，则需要输出层中有10个神经元，并且必须用可以处理多个分类的softmax激活函数替换逻辑函数，为每个分类输出一个概率。如果你想让你的神经网络像第2章那样预测房价，则需要一个输出神经元，而在输出层则无须使用激活函数[2]。

8.反向传播是一种用于训练人工神经网络的技术。它首先计算关于每个模型参数（所有的权重和偏差）的成本函数的梯度，然后使用这些梯度执行梯度下降。这种反向传播步骤通常执行数千次或数百万次，并需要多个训练批次，直到模型参数收敛到最小化成本函数的值为止。为

了计算梯度，反向传播使用反向模式autodiff（尽管在反向传播被发明的时候还不叫autodiff，事实上autodiff的概念已经被重新发明了多次）。反向模式autodiff会先在计算图上正向执行一次，计算当前训练批次的每个节点的值，然后反向执行一次，一次性计算所有梯度（详见附录D）。那和反向传播有什么区别呢？反向传播是指使用多个反向传播步骤来训练人工神经网络的全部过程，每个步骤计算梯度并使用它们执行梯度下降过程。相反，反向模式autodiff只是一种有效地计算梯度的技术，只是恰好被反向传播使用了而已。

9.这里列出了所有可以在基本MLP中调整的超参数：隐藏层的数量、每个隐藏层中神经元的数量，以及每个隐藏层和输出层中使用的激活函数[3]。一般情况下，ReLU激活函数（或其中的一个变体，见第11章）是隐藏层的一个很好的默认值。对于输出层，通常需要二元分类的逻辑激活函数、多类分类的softmax激活函数，在做回归时则无须任何激活函数。

如果MLP过拟合训练数据，可以尝试减少隐藏层的数量，并减少每个隐藏层的神经元数量。

10.请参阅位于https://github.com/ageron/handson-ml2的Jupyter notebook。

第11章：训练深度神经网络

1.不，所有权重需要独立处理，它们不可以初始化为统一值。随机取样权重一个重要的目的是破坏对称性：如果所有的权重具有相同的初始值，即使该值不为0，对称性也无法被破坏（即一层中所有神经元都是一样的），并且反向传播将无法破坏它。具体来说，这就意味着一层中所有神经元始终保持相同的权重。就像每层只有一个神经元，而且要慢得多。这样的配置是无法收敛到一个好的解决方案的。

2.当然可以设置为0。有些人喜欢像初始化权重一样处理偏差项，这样也是可以的，没有太大的区别。

3.SELU函数相对于ReLU函数的一些优点是：

•它可以使用负值，所以相比使用ReLU激活方程（从不输出负值），某一给定层的神经元输出平均值理论上更容易接近0。这样有助于缓解梯度消失问题。

•它总是有一个非零的导数，可以避免影响ReLU单元的单元消失问题。

•当条件合适时（即如果模型是顺序的，权重使用LeCun初始化进行初始化，输入被标准化，并且不存在不兼容的层或正则化，例如dropout或ℓ_1正则化），然后SELU激活函数可确保模型进行自归一化，从而解决了梯度爆炸与梯度消失问题。

4.SELU激活函数是一个不错的默认选择。如果对神经网络的速度要求很高，可以用leaky ReLU的一个变体代替（例如，使用默认超参数值的leaky ReLU）。这是因为ReLU激活函数简单方便，所以很多人会将其作为首选，即使输出表现会被ELU和leaky ReLU超过。但是，某些情况下，ReLU激活函数精确输出0的能力是有用的（见第17章）。此外，它有时可以从优化的实现和硬件加速中获益。如果你需要输出一个介于-1和1之间的数，tanh在输出层会比较有效，但是现在在隐藏层的使用频度并不高（除了经常使用的网络）。在你需要评估可能性（比如二进制分类）时，逻辑激活函数在输出层比较有效，但是同样在隐藏层中很少使用（也有例外，比如，变分自动编码器的编码层，见第17章）。最后softmax激活函数在输出层输出互斥类的概率是有效的，但是除了（或者曾经）隐藏层以外基本不使用。

5.如果使用SGD优化器时，将momentum超参数设置得太接近1（比如，0.99999），算法就会提速很高，偏向全局最小值，但是其动量使

其刚好超过最小值。之后就会慢慢降速回落，再加速，再超调，循环往复。这种方式在收敛前会振荡好几次，所以总体来说，收敛速度会比用小动量慢。

6.一种实现稀疏模型（即大多数权重等于0）的方法是正常训练一个模型，然后将小权重设置为0。为了更稀疏，可以在训练过程中应用ℓ_1正则化，这样可以促使优化器更加稀疏。第三种方式是使用TensorFlow的模型优化工具箱。

7.是的，dropout确实会减慢训练速度，通常大约会减少到$\frac{1}{2}$。但是，它仅在训练期间打开，因此对推理速度没有影响。MC Dropout与训练期间的dropout完全一样，但是在推理过程中它仍然处于运行状态，因此每次推理速度都会稍微放慢。更重要的是，使用MC Dropout时，你通常希望运行推理能快10倍或更多倍以获得更好的预测结果。这意味着进行预测的速度大约降低到$\frac{1}{10}$。

有关练习8、9和10的解决方案，请参见https://github.com/ageron/handson-ml2上提供的Jupyter notebook。

第12章：使用TensorFlow自定义模型和训练

1.TensorFlow是一个用于数值计算的开源库，特别适合大规模机器学习并对其进行微调。它的核心类似于NumPy，但是它还支持GPU、分布式计算、计算图分析和优化功能（具有可移植图格式，允许你在一个环境中训练TensorFlow模型并在另一个环境中运行）。基于反向模式autodiff的优化API，以及一些强大的API，例如tf.keras、tf.data、tf.image、tf.signal等。其他受欢迎的深度学习库包括PyTorch、MXNet、微软的Cognitive工具包、Theano、Caffe2和Chainer。

2.尽管TensorFlow提供了NumPy提供的大多数功能，但由于某些原因，它不是简单替代品。首先，函数的名称是并不总是相同的（例如，tf.reduce_sum（）与np.sum（））。其次，某些函数的行为并不完全相同（例如，tf.transpose（）创建一个张量的转置副本，而NumPy的T属性创建一个转置视图，而没有复制任何数据）。最后，NumPy数组是可变的，而TensorFlow张量是不可变的（但如果你需要一个可变对象，则可以使用tf.Variable）。

3.tf.range（10）和tf.constant（np.arange（10））都返回包含整数0到9的一维张量。但是，前者使用32位整数，而后者使用64位整数。实际上，TensorFlow默认为32位，而NumPy默认为64位。

4.除了常规张量之外，TensorFlow还提供其他几种数据结构，包括稀疏张量、张量数组、参差不齐的张量、队列、字符串张量和集合。最后两个实际上表示为常规张量，但是TensorFlow提供了特殊的函数来操作它们（在tf.strings和tf.sets中）。

5.当你想自定义一个损失函数时，通常可以将其实现为常规Python函数。但是，如果自定义损失函数必须支持某些超参数（或任何其他状态），则应继承keras.losses.Loss类，并实现__init__（）方法和call（）方法。如果你想和模型一起保存损失函数的超参数，则还必须实现get_config（）方法。

6.与自定义损失函数很像，大多数指标可以定义为常规Python函数。但是，如果你希望自定义指标支持某些超参数（或任何其他状态），则应继承keras.metrics.Metric类。此外，如果在整个轮次内计算指标不等同于计算该轮次内所有批量的均值指标（例如，精度和召回率指标），则应继承keras.metrics.Metric类和实现__init__（）方法、update_state（）方法和result（）方法来跟踪每个轮次中运行的指标。你还应该实现reset_states（）方法，除非将所有变量都重置为0.0。如果你想把状态与模型一起保存，则还应该实现get_config（）方法。

7.你应该将模型的内部组件（即层或可重复使用的层块）与模型本身（即要训练的对象）区分开。前者应继承keras.layers.Layer类，而后者应继承keras.models.Model类。

8.编写你自己的自定义训练循环是相当高级的，因此只有在确实需要时才应该这样做。Keras提供了多种工具来自定义训练，而不必编写自定义训练循环：回调、自定义正则化、自定义约束、自定义损失等。你应该尽可能使用这些而不是编写自定义训练循环：编写自定义训练循环更容易出错，并且很难重用你编写的自定义代码。但是，在某些情况下有必要编写自定义训练循环——例如，如果你想对神经网络的不同部分使用不同的优化器，例如在“宽与深”论文中。自定义训练循环在调试或试图理解训练的工作原理时也很有用。

9.自定义Keras组件应转换为TF函数，这意味着它们应尽可能遵循TF操作并遵守12.4.2节中列出的所有规则。如果你需要在定制组件中包括任意Python代码，可以将其包装在tf.py_function（）操作中（但这会降低性能并限制模型的可移植性），或者在创建定制层或模型时设置dynamic=True（或在调用模型的compile（）方法时设置run_eagerly=True）。

10.有关创建TF函数时要遵循的规则列表，请参阅12.4.2节。

11.创建动态Keras模型对于调试很有用，因为它不会将任何自定义组件编译为TF函数，并且你可以使用任何Python调试器来调试代码。如果你想在模型（或训练代码）中包括任意Python代码（包括对外部库的调用），它也很有用。要使模型是动态的，必须在创建模型时设置dynamic=True。或者在调用模型的compile（）方法时设置run_eagerly=True。动态创建模型会阻止Keras使用TensorFlow的任何图特征，因此会减低训练和推理的速度，并且你将无法导出计算图，从而限制模型的可移植性。

有关练习12和13的解答，请参见https://github.com/ageron/handson-ml2上提供的Jupyter notebook。

第13章：使用TensorFlow加载和预处理数据

1.获取大数据集并对其进行有效预处理可能是一个复杂的工程挑战。数据API简化了这个过程。它提供了许多功能，包括从各种来源（例如文本或二进制文件）加载数据，从多个来源并行读取数据，对其进行转换，对记录进行交织，打乱数据的次序，批处理和预取。

2.将大型数据集拆分为多个文件后，可以在使用乱序缓冲区将其次序打乱为更细的级别之前，先对其进行粗略的乱序。它还使处理单个计算机上无法容纳的庞大数据集成为可能。操作成千上万个小文件也比操作一个大文件更容易。例如，很容易将数据拆分为多个子集。最后，如果将数据拆分为多个文件，而这些文件分布在多个服务器上，则可以同时在不同的服务器上下载文件，从而提高了带宽利用率。

3.你可以使用TensorBoard来可视化数据分析：如果GPU没有完全利用，则你的输入流水线可能会成为瓶颈。你可以并行读取和预处理多个线程中的数据，并确保它预取一些批次来改正它。如果这还不能使你的GPU在训练时达到100%使用率，请确保优化了预处理代码。你也可以将数据集保存到多个TFRecord文件中，并在必要时提前执行一些预处理，以便在训练过程中无须即时进行处理（TF Transform可以帮助你做这些）。如有必要，请使用有更多CPU和RAM的计算机，并确保GPU带宽足够大。

4.TFRecord文件由一系列任意二进制记录组成：你可以在每个记录中存储所需的任何二进制数据。但是，实际上，大多数TFRecord文件都包含序列化协议缓冲区的序列。这可以利用协议缓冲区的优势，例如，它们可以在多种平台和语言之间轻松读取，并且以后可以以向后兼容的方式更新其定义。

5.Example protobuf格式的优点是TensorFlow提供了一些操作来解析它（tf.io.parse*example（）函数），而无须定义自己的格式。它足够灵活，可以代表大多数数据集中的实例。但是，如果它不能满足你的要求，则你可以定义自己的协议缓冲区，使用protoc进行编译（设置--descriptor_set_out和--include_imports参数以导出protobuf描述符），然后使用tf.io.decode_proto（）函数来解析序列化的protobuf（有关示例，请参见notebook的“Custom protobuf”部分）。它比较复杂，需要将描述符与模型一起部署，但是可以做到。

6.使用TFRecords时，如果训练脚本需要下载TFRecord文件，通常会希望压缩文件，因为压缩会使文件更小，从而减少下载时间。但是，如果文件与训练脚本位于同一台机器上，通常最好不要压缩，以避免浪费CPU进行解压缩。

7.让我们看一下每个预处理选项的优缺点：

•如果在创建数据文件时对数据进行预处理，则训练脚本将运行得更快，因为它不必即时执行预处理。在某些情况下，预处理后的数据也会比原始数据小得多，因此可以节省一些空间并加快下载速度。这也可能有利于预处理的数据，例如进行检查或存档。但是，这种方法有一些缺点。首先，如果需要为每个想法生成预处理的数据集，则实验各种预处理逻辑并不容易。其次，如果要执行数据扩充，则必须实现数据集的许多变体，这将占用大量磁盘空间并花费大量时间来生成。最后，训练的模型需要有预处理的数据，因此你必须在应用程序调用模型之前添加预处理代码。

•如果使用tf.data流水线对数据进行预处理，则调整预处理逻辑和应用数据扩充要容易得多。同样，tf.data可以轻松构建高效的预处理流水线（例如，使用多线程和预取）。但是，以这种方式预处理数据会减慢训练速度。此外，每个训练实例将在每个轮次进行一次预处理，而不是在创建数据文件时对数据进行一次预处理。

最后，训练后的模型还是需要有预处理的数据。

• 如果将预处理层添加到模型中，则只需编写一次预处理代码即可进行训练和推理。如果你的模型需要部署到许多不同的平台，则无须多次编写预处理代码。另外，你不会有模型使用了错误的预处理逻辑的风险，因为它将成为模型的一部分。不利的一面是，对数据进行预处理将减慢训练速度，并且每个训练实例会在每个轮次进行一次预处理。此外，默认情况下，预处理操作将在GPU上针对当前轮次运行（你无法从CPU上的并行预处理和预取中受益）。幸运的是，新版本的Keras预处理层能够从预处理层中进行预处理操作，并将其作为tf.data流水线的一部分运行，因此你将受益于CPU的多线程执行和预取。

• 最后，使用TF Transform进行预处理可为你提供先前选项的许多好处：已实现了预处理的数据，每个实例仅进行了一次预处理（加快了训练速度），并且预处理层自动生成，因此你只需要编写一次预处理代码即可。主要缺点是你需要学习如何使用此工具。

8.让我们看一下如何对分类特征和文本进行编码：

• 要对诸如电影等级（例如，“坏”“一般”“好”）这样具有自然顺序的分类特征进行编码，最简单的选择是使用顺序编码：按照其自然顺序对类别进行排序并将每个类别映射到其顺序（例如，“坏”映射为0，“一般”映射为1，“好”映射为2）。但是，大多数分类特征没有这种自然顺序。例如，没有职业或国家的自然秩序。在这种情况下，你可以使用独热编码，如果类别很多，也可以使用嵌入。

• 对于文本，一种选择是使用词袋表示：一个句子可以表示为每个可能单词计数的向量。由于普通单词通常不是很重要，因此你需要使用TF-IDF来减轻其权重。除了单词计数，n-gram计数也很常见，n-gram是n个连续单词的序列，既好又简洁。另外，你可以使用可能经过预训练的词嵌入对每个词进行编码。除了对单词进行编码外，还可以对每个字

母或子单词令牌进行编码（例如，将“最聪明”分为“聪明”和“最”）。在第16章中讨论了这最后两个选项。

有关练习9和10的解答，请参见https://github.com/ageron/handson-ml2上提供的Jupyter notebook。

第14章：使用卷积神经网络的深度计算机视觉

1.与全连接的DNN相比，CNN的主要优势在于图像分类：

- 因为连续的层仅部分连接并且由于其大量复用权重，所以CNN的参数比全连接的DNN少得多，这使其训练速度快得多，降低了过拟合的风险，并且需要的训练数据也少得多。

- CNN学会了可以检测到特定特征的内核后，便可以在图像中的任何位置检测到该特征。相反，当DNN在一个位置学习某个特征时，它只能在该特定位置检测到它。由于图像通常具有非常重复的特征，因此对于CNN而言，使用较少的训练实例，可以比DNN更好地泛化图像处理任务（例如分类）。

- 最后，DNN没有像素的排列方式的先验知识。它不知道附近的像素很近。CNN的架构嵌入了此先验知识。较低的层通常在图像的较小区域中标识特征，而较高的层将较低层的特征组合为较大的特征。这对大多数自然图像都能很好地工作，从而使CNN与DNN相比具有领先优势。

2.让我们计算一下CNN有多少个参数。由于其第一个卷积层具有3×3内核，并且输入具有3个通道（红色、绿色和蓝色），每个特征图具有3×3×3权重以及一个偏置项。每个特征图有28个参数。由于该第一个卷积层具有100个特征图，因此共有2800个参数。第二个卷积层具有3×3内核，其输入是前一层的100个特征图的集合，因此每个特征图具有3×3×100=900权重，加上一个偏置项。由于它具有200个特征图，

因此该层具有901×200=180 200个参数。最后，第三个和最后一个卷积层也具有3×3内核，其输入是前一层的200个特征图的集合，因此每个特征图具有3×3×200=1800权重，再加上偏置项。由于它具有400个特征图，因此该层总共具有1801×400=720 400个参数。总而言之，CNN的参数为2800+180 200+720 400=903 400。

现在让我们计算一下对单个实例进行预测时，该神经网络至少需要多少RAM。首先让我们计算每层的特征图大小。由于我们使用的步幅为2且填充为"same"，因此特征图的水平和垂直尺寸在每一层均被2除（必要时向上取整）。因此，当输入通道为200×300像素时，第一层的特征图为100×150，第二层的特征图为50×75，第三层的特征图是25×38。由于32位是4个字节，并且第一个卷积层有100个特征图，因此该第一层占用4×100×150×100=6百万个字节（6 MB）。第二层占用4×50×75×200=3百万个字节（3 MB）。最后，第三层占用4×25×38×400=1 520 000字节（约1.5 MB）。但是，一旦计算出一层，就可以释放上一层所占用的内存，因此，如果一切都进行了优化，则仅需6+3=9百万字节（9 MB）的RAM（当第二层刚刚进行了计算，但尚未释放第一层占用的内存）。但是，等等，你还需要添加CNN参数占用的内存！我们之前计算过，它有903 400个参数，每个参数占用4个字节，因此增加了3 613 600字节（约3.6 MB）。因此，所需的总RAM为（至少）12 613 600字节（约12.6 MB）。

最后，让我们计算在50个图像的小批量上训练CNN时所需的最小RAM量。在训练期间，TensorFlow使用反向传播，这需要保持在正向传递过程中计算出的所有值，直到反向传播开始。因此，我们必须为单个实例计算出所有层所需的总RAM，并将其乘以50。在这一点上，让我们开始以兆字节而不是字节为单位进行计数。之前计算得出，每个实例的三层分别需要6 MB、3 MB和1.5 MB。每个实例总共10.5 MB，因此对于50个实例，所需的总RAM为525 MB。加上输入图像所需的RAM，即50×4×200×300×3=3600万字节（36 MB），再加上模型参数所需的RAM，大约为3.6 MB（较早计算得出），加上一些用于梯度的RAM（我们将忽略这一点，因为随着反向传播沿着层向下传播，它可以逐渐释

放）。我们总共大约需要525+36+3.6=564.6 MB，这是一个乐观的最低要求。

3.如果训练CNN时GPU内存不足，可以尝试以下5种方法来解决（除了购买具有更多RAM的GPU以外）：

• 减小小批量的大小。

• 在一层或多层中使用较大的步幅来降低维度。

• 去除一层或多层。

• 使用16位浮点数而不是32位浮点数。

• 在多个设备上分布CNN。

4.最大池化层根本没有任何参数，而卷积层有很多（请参阅前面的问题）。

5.局部响应归一化层使最强烈激活的神经元抑制在相同位置但在相邻特征图中的神经元，从而使不同的特征图有针对性并相互远离，使它们探索更广泛的特征。它通常用于较低的层，以拥有更大的低层特征池（较高层可以在其上构建）。

6.与LeNet-5相比，AlexNet的主要创新之处在于它更大、更深，并且将卷积层直接堆叠在彼此之上，而不是将池化层堆叠在每个卷积层之上。GoogLeNet的主要创新之处在于引入了inception模块，与以前的CNN架构相比，它具有更少的参数，从而可以有更深的网络。ResNet的主要创新是引入了连接跳跃，这使超过100层成为可能。可以说，其简单性和一致性也颇具创新性。SENet的主要创新是在inception网络中的每个inception模块或ResNet中的每个残差单元之后使用SE块（两层密

集网络）来重新校准特征图的相对重要性。最后，Xception的主要创新是使用了深度可分离卷积层，它们可以各自识别空间模式和深度模式。

7.完全卷积网络是仅由卷积层和池化层组成的神经网络。FCN可以有效地处理任何宽度和高度（至少大于最小尺寸）的图像。它们对于物体检测和语义分割最有用，因为它们只需要查看一次图像（而不必在图像的不同部分多次运行CNN）。如果你有CNN（其上有一些密集层），则可以将这些密集层转换为卷积层来创建FCN：只需用内核大小等于该层输入大小的卷积层替换最低密集层，在密集层中每个神经元使用一个过滤器，并使用"valid"填充。通常，步幅应为1，但你可以根据需要将其设置为更高的值。激活函数应与密集层的相同。其他密集层应以相同的方式转换，但要使用1×1的内核。实际上可以通过适当地重排密集层的权重矩阵这种方式来转换经过训练的CNN。

8.语义分割的主要技术难点是，当信息经过每一层时（尤其是在池化层和步幅大于1的层中）许多空间信息会在CNN中丢失，这个空间信息需要被恢复才能准确地预测出每个像素的类别。

有关练习9～12的解答，请参见https://github.com/ageron/handson-ml2上提供的Jupyter notebook。

第15章：使用RNN和CNN处理序列

1.以下是一些RNN的应用：

• 对于序列到序列的RNN：预测天气（或任何其他时间序列）、机器翻译（使用编码器-解码器架构）、视频字幕、语音到文本、音乐生成（或其他序列生成）、识别歌曲的和弦。

• 对于从序列到向量的RNN：按音乐流派对音乐样本进行分类、分析书评的情绪、根据脑植入物的读数预测失语症患者正在思考的单词、

根据用户的观看记录来预测他们观看一个电影的概率（这是推荐系统协作过滤的许多可能实现之一）。

• 对于向量到序列的RNN：图像字幕、基于当前艺术家来创建音乐播放列表、基于一组参数生成旋律、在图片中定位行人（例如，来自自动驾驶汽车的摄像头的视频帧）。

2.RNN层必须具有三维输入：第一维是批量处理的维度（其大小是批处理大小），第二维表示时间（其大小是时间步长数），第三维表示在每个时间步长的输入（其大小是每个时间步长的输入特征的数量）。例如，如果要处理包含5个时间序列（每个10个时间步长）的批处理，每个时间步长有2个值（例如，温度和风速），则维度将为[5、10、2]。输出也是三维的，前两个维相同，但最后一个维度等于神经元的数量。例如，如果具有32个神经元的RNN层处理了刚才讨论的批处理，则输出的维度将为[5，10，32]。

3.要使用Keras来构建深度序列到序列的RNN，必须为所有RNN层设置return_sequences=True。要构建从序列到向量的RNN，必须为所有RNN层设置return_sequences=True，但最顶层RNN层必须具有return_sequences=False（或不设置此参数，因为默认为False）。

4.如果你有日常不变的时间序列，并且要预测接下来的7天，则可以使用的最简单的RNN架构是堆叠RNN层（除了顶层RNN层以外，其他所有层的设置均为return_sequences=True），在输出RNN层中使用7个神经元。然后，你可以使用时间序列中的随机窗口来训练该模型（例如，连续30天的序列作为输入，而包含接下来7天的值的向量作为目标）。这是从序列到向量的RNN。另外，你可以在所有RNN层中设置return_sequences=True来创建序列到序列的RNN。你可以使用时间序列中的随机窗口来训练该模型，作为目标序列的长度与输入长度相同。每个目标序列的每个时间步长应具有7个值（例如，对于时间步长t，目标应为包含时间步长t+1至t+7的值的向量）。

5.训练RNN的两个主要困难是不稳定的梯度（爆炸或消失）和非常有限的短期记忆。当处理长序列时，这些问题都会变得更糟。为了解决不稳定的梯度问题，可以使用较小的学习率，使用饱和的激活函数（例如双曲线正切，默认设置），并可以在每个时间步长中使用梯度修剪、层归一化或dropout。要解决有限的短期记忆问题，可以使用LSTM或GRU层（这也有助于解决不稳定梯度的问题）。

6.LSTM单元的架构看起来很复杂，但是如果你了解底层逻辑，实际上并不太难。该单元具有短期状态向量和长期状态向量。在每个时间步长，输入和先前的短期状态被馈送到一个简单的RNN单元和三个门：遗忘门决定从长期状态中删除什么，输入门决定简单RNN单元的输出的哪一部分应该添加到长期状态，然后输出门决定长期状态的哪一部分应该输出（经过tanh激活函数之后）。新的短期状态等于单元的输出。见图15-9。

7.一个RNN层基本上是顺序的：为了在时间步长t计算输出，它必须首先在所有较早的时间步长计算输出。这使得不可能并行计算。另一方面，一维卷积层很适合并行化，因为它不保持时间步长之间的状态。换句话说，它没有内存：在任何时间步长的输出都仅基于输入值的一小窗口进行计算，而不必知道所有过去的值。此外，由于一维卷积层不是递归的，因此受不稳定梯度的影响较小。RNN中的一个或多个一维卷积层可用于有效地预处理输入，例如降低其时间分辨率（下采样），从而帮助RNN层检测长期模式。实际上，通过构建WaveNet架构，可以仅使用卷积层。

8.为了根据视频的视觉内容对视频进行分类，一种可能的架构是每秒取一帧，然后通过相同的卷积神经网络运行每一帧（例如，预先训练的Xception模型，如果你的数据集不大，可能不需要修改），将CNN的输出序列提供给序列到向量的RNN，最后通过softmax层运行，输出所有类的概率。进行训练时，你可以将交叉熵用作成本函数。如果你还想用音频进行分类，则可以使用堆叠的一维卷积层将时间分辨率从每秒数千

个音频帧降低到每秒一个（以匹配每秒的图像数），并将输出序列连接到序列到向量的RNN的输入（最后一个维度）。

有关练习9和10的解答，请参见https://github.com/ageron/handson-ml2上提供的Jupyter notebook。

第16章：使用RNN和注意力机制进行自然语言处理

1.无状态RNN只能识别长度小于或等于在其上训练的RNN的窗口大小的模式。相反，有状态RNN可以识别长期模式。但是，实现有状态RNN很困难，尤其是要正确准备数据集。而且有状态RNN并非总是能更好地工作，部分原因是连续的批量数据不是独立的且不均匀分布（IID）。梯度下降不适合非IID数据集。

2.通常，如果逐词翻译一个句子，结果将很糟糕。例如，法语句子“Je vous en prie”的意思是“不客气”，但是如果一次翻译一个单词，就会得到“我你在祈祷”。最好先阅读整个句子然后再翻译。普通序列到序列RNN在读取第一个单词后将立即开始翻译句子，而编码器-解码器RNN将首先读取整个句子然后进行翻译。就是说，你可以想象一个普通的序列到序列RNN，当不确定接下来要说什么时，它将输出静音（就像人工翻译在直播节目时做的一样）。

3.可变长度的输入序列可以通过填充较短的序列来处理，使一批量中的所有序列具有相同的长度，及使用掩码来确保RNN忽略填充令牌。为了获得更好的性能，你可能还想创建包含相似大小序列的批量。大小不一的张量可以容纳可变长度的序列，而tf.keras可能最终会支持它们，这将大大简化对可变长度输入序列的处理（在撰写本文时，情况尚不如此）。对于可变长度输出序列，如果预先知道输出序列的长度（例如，如果你知道它与输入序列相同），则只需配置损失函数，这样就可以在序列末尾忽略令牌。类似地，使用模型的代码应在序列结束之后忽

略令牌。但是通常提前不知道输出序列的长度，因此解决方案是训练模型，以使其在每个序列的末尾输出序列结束令牌。

4.集束搜索是一种用于提高训练的编码器-解码器模型的性能的技术，例如在神经机器翻译系统中。该算法会跟踪k个最有前途的输出情感的简短列表（例如，前三个），并在每个解码器步骤中尝试扩展一个单词。然后它只保留k个最有可能的句子。参数k称为集束宽度：它越大，将使用的CPU和RAM越多，但系统也越精确。与其在每个步骤上都贪婪地选择最可能的下一个单词来扩展单个句子，该技术允许系统同时探索几个有希望的句子。而且，这种技术手段很适合并行化。你可以使用TensorFlow插件轻松实现集束搜索。

5.注意力机制是最初在编码器-解码器模块中使用的一种技术，它使解码器可以更直接地访问输入序列，从而使其能够处理更长的输入序列。在每个解码器时间步长，当前的解码器状态和编码器的全部输出由对齐模型处理，该对齐模型输出每个输入时间步长的对齐分数。该分数指示输入的哪一部分与当前解码器时间步长最相关。然后，将编码器输出的加权总和（通过其对齐分数加权）输入到解码器，该解码器将生成下一个解码器状态和该时间步长的输出。使用注意力机制的主要好处是，编码器-解码器模型可以成功处理更长的输入序列。另一个好处是，对齐分数使模型更易于调试和解释：例如，如果模型出错，则可以查看它关注的输入部分，这可以帮助诊断问题。注意机制也是Multi-Head Attention层中Transformer架构的核心。请参阅习题6。

6.Transformer架构中最重要的层是Multi-Head Attention层（原始Transformer架构包含18个层，其中包括6个Masked Multi-Head Attention层）。它是语言模型（如BERT和GPT-2）的核心。其目的是使模型能够识别出哪些单词彼此最对齐，然后使用这些上下文线索来改善每个单词的表示。

7.当有很多类（例如，数千个）时，在训练分类模型时会使用采样的softmax。它基于模型为正确类别预测的对数和不正确单词样本的预

测对数，计算交叉熵损失的近似值。相较于计算所有logit的softmax，然后估计交叉熵损失，这大大提高了训练速度。训练后可以正常使用模型，使用常规softmax函数根据所有逻辑计算所有类别概率。有关练习8～11的解答，请参见https://github.com/ageron/handson-ml2上提供的Jupyter notebook。

第17章：使用自动编码器和GAN的表征学习和生成学习

1.以下是使用自动编码器的一些主要任务：

- 特征提取
- 无监督预训练
- 降维
- 生成模型
- 异常检测（自动编码器通常无法重建异常值）

2.如果你想训练一个分类器并且拥有大量未标记的训练数据，但只有数千个有标记的实例，那么可以先在完整的数据集（有标记的加上未标记的）上训练一个深度自动编码器，然后对分类器重用其下半部分（即重用包括编码层在内的上部所有层），并使用有标记的数据训练分类器。如果有标记数据很少，则你在训练分类器时可能要冻结重用的层。

3.自动编码器可以完美地重构其输入并不一定意味着它是一个很好的自动编码器。也许这仅仅是一个过于完整的自动编码器，它学会了将其输入复制到编码层，然后复制到输出层。实际上，即使编码层只包含单个神经元，非常深的自动编码器也有可能学习到把每个训练实例映射到不同的编码（例如，第一个实例可以映射到0.001，第二个实例可以

映射到0.002，第三个到0.003，以此类推），它可以用心地学习为每个编码重建正确的训练实例。它可以完美地重构其输入，而无须真正学习数据中的任何有用模式。实际上，这种映射不太可能发生，但它说明了一个事实，即完美的重构并不能保证自动编码器会学到任何有用的信息。但是，如果产生了非常差的重构，则几乎可以保证它是一个差的自动编码器。为了评估自动编码器的性能，一种选择是测量重建损失（例如，计算MSE或输出减去输入的均方根）。同样，高的重建损失是一个坏自动编码器的迹象，但是低的重建损失并不能保证它是好的。你还应该根据其用途来评估自动编码器。例如，如果你将其用于分类器的无监督预训练，则还应该评估分类器的性能。

4.一个不完整的自动编码器，其编码层小于输入层和输出层。如果它很大，则它是一个完整的自动编码器。过于不完整的自动编码器的主要风险是可能无法重建输入。过于完整的自动编码器的主要风险在于，它可能只是将输入复制到输出，而没有学习到任何有用的特征。

5.要将编码层及其相应的解码层的权重绑定在一起，只需使解码器权重等于编码器权重的转置即可。这会使模型中的参数数量减少一半，通常使训练收敛速度更快，训练数据更少，并减少了过拟合训练集的风险。

6.生成模型是一种能够随机生成类似于训练实例的输出的模型。例如，一旦成功地在MNIST数据集上进行了训练，生成模型就可以随机生成数字的逼真图像。输出分布通常类似于训练数据。例如，由于MNIST包含每个数字的许多图像，因此生成模型会输出大致相同数量的每个数字的图像。某些生成模型可以被参数化——例如，仅生成某些类型的输出。生成自动编码器的一个示例是变分自动编码器。

7.生成式对抗网络是一种神经网络架构，由具有相反目的的两个部分组成：生成器和判别器。生成器的目的是生成类似于训练集中的实例来欺骗判别器。判别器必须将实际实例与生成的实例区分开。在每次训练迭代中，像正常的二元分类器一样训练判别器，然后训练生成器使判

别器的误差最大。GAN用于高级图像处理任务，例如超分辨率、彩色化、图像编辑（将对象替换为逼真的背景），将简单的草图转换为逼真的图像或预测视频中的下一帧。它们还用于扩充数据集（来训练其他模型），生成其他类型的数据（例如文本、音频和时间序列），识别其他模型中的弱点并加以弥补。8.由于生成器和判别器之间复杂的动态关系，训练GAN极其困难。最大的困难是模式崩溃，生成器产生的输出几乎没有多样性。而且，训练可能非常不稳定：它可能开始时很好，然后突然开始振荡或发散，而没有任何明显的原因。GAN对选择超参数也非常敏感。

有关练习9、10和11的解答，请参阅https://github.com/ageron/handson-ml2上提供的Jupyter notebook。

第18章：强化学习

1.强化学习是机器学习的一个领域，旨在创建能够在环境中采取行动的智能体，从而使奖励随着时间的推移而最大化。RL与常规有监督学习和无监督学习之间有很多差异。下面是一些差异：

• 在有监督学习和无监督学习中，目标通常是在数据中找到模式并使用它们来进行预测。在强化学习中，目标是找到一个好的策略。

• 与有监督学习不同，强化学习没有明确为智能体给出“正确”的答案。它必须通过反复试验错误来学习。

• 与无监督学习不同，有一种通过奖励的有监督形式。我们不告诉智能体如何执行任务，但是会告诉智能体其成功或失败。

• 强化学习代理需要在探索环境、寻找获得奖励的新方法以及利用已经知道的奖励来源之间找到适当的平衡。相反，有监督学习和无监督学习系统通常不需要担心探索。它们只是根据给定的训练数据。

• 在有监督学习和无监督学习中，训练实例通常是独立的（实际上，它们通常是随机混洗的）。在强化学习中，连续观察通常不是独立的。智能体在继续前进之前可能会在环境的同一区域中停留一段时间，因此连续的观察结果将非常相关。在某些情况下，重播存储（缓冲区）用于确保训练算法能得到相当独立的观察结果。2.除了第18章中提到的，这里还有一些强化学习的可能应用：

音乐个性化

环境是用户的个性化网络广播。智能体是决定该用户接下来要播放什么歌曲的软件。它可能的操作是播放目录中的任何歌曲（必须选择用户喜欢的歌曲）或播放广告（必须选择会引起用户兴趣的广告）。每次用户听一首歌曲，它都会得到很小的奖励；每次用户收听广告，它会得到更大的奖励；当用户跳过歌曲或广告时，它会得到负面奖励；如果用户离开，则得到更大的负面奖励。

市场营销

环境是你公司的营销部门。智能体是一种软件，它根据给定的个人资料和历史购买记录来定义应将邮件发送给哪个客户（对于每个客户，它有两个可能的操作：发送或不发送）。它为邮件的成本给予负面奖励，为该活动产生的估计收入给予正面奖励。

产品交付

让智能体控制一批货车，确定它们应该在仓库取什么货，应该去的地方，应该卸什么货，等等。对于按时交付的每种产品，它将获得正面奖励；而对于延迟交付的产品，它将获得负面奖励。

3.估计动作的值时，强化学习算法通常会汇总该动作带来的所有奖励，将更多的权重分配给即时奖励，将较少的权重分配给以后的奖励（考虑到某项动作对近期未来的影响大于遥远未来的影响）。为了对此

建模，通常在每个时间步长应用折扣因子。例如，在折扣因子为0.9的情况下，当你估算操作的值时，在两个时间步长之后收到的100的奖励仅计为0.92×100=81。你可以将折扣因子视为衡量相对于当前的未来价值的量度：如果它非常接近1，则未来的值几乎与现在的值相同；如果接近0，则仅是立即获得的奖励很重要。当然，这会对最优策略产生巨大影响：如果你看重未来，你可能愿意为最终回报的前景承担很多当即的痛苦；如果你不看重未来，则只会抓住你可以找到的任何可以立即获得的回报，从不对未来进行投资。

4.要衡量强化学习智能体的性能，你可以简单地汇总它所获得的奖励。在模拟环境中，你可以运行许多个回合，查看其平均获得的总奖励（可以查看最小值、最大值、标准差等）。

5.信用分配问题是这样的事实：当强化学习智能体收到奖励时，它没有直接的方法来知道其先前的哪些行为促成了该奖励。这通常在动作和所得奖励之间存在较大延迟时发生（例如，在Atari的Pong游戏中，从智能体击球到获胜之间可能有几十个时间步长）。解决它的一种方法是在可能的情况下为智能体提供短期奖励。这通常需要有关任务的先验知识。例如，如果我们想建立一个会下棋的智能体，不是仅仅在赢得比赛时才给予奖励，我们可以在每次吃掉对手的一个棋子时给予奖励。

6.智能体通常可以在其环境的同一区域中停留一段时间，因此在这段时间内，其所有的经历都非常相似。这可能会在学习算法中引入一些偏差。它可能会针对此环境区域调整其策略，但是一旦移出该区域，它的性能会不好。要解决此问题，你可以使用重播存储。智能体不使用最近的学习经历，而将基于过去的经历的缓冲来学习（也许这就是我们晚上做梦的原因：重播我们白天的经历并更好地学习？）。

7.异策略RL算法学习最佳策略的值（即如果智能体采取最佳行动，则每个状态可以预期的折扣奖励总和），而智能体遵循不同的策略。Q学习是这种算法的一个很好的示例。相反，同策略的算法学习智能体实际执行的策略的值，包括探索和利用。

有关练习8、9和10的解答，请参见https://github.com/ageron/handson-ml2上提供的Jupyter notebook。

第19章：大规模训练和部署TensorFlow模型

1.SavedModel包含一个TensorFlow模型，包括其架构（计算图）及其权重。它保存在一个目录中，该目录包含一个save_model.pb文件和一个variables子目录——该文件定义了计算图（表示为序列化的协议缓冲区），该子目录包含变量值。对于包含大量权重的模型，可以将这些变量值拆分为多个文件。SavedModel还包括assets子目录，该子目录可能包含其他数据，例如词汇表文件、类名或该模型的某些实例。准确地说，SavedModel可以包含一个或多个元图。元图是一个计算图，外加一些函数签名定义（包括它们的输入名称输出名称、类型和形状）。每个元数据由一组标签来标识。要查看SavedModel，你可以使用命令行工具save_model_cli或使用tf.saved_model.load（）来加载它并在Python中对其进行查看。

2.TF Serving允许你部署多个TensorFlow模型（或同一模型的多个版本），使其可以通过REST API或gRPC API轻松地被所有应用程序访问。直接在你的应用程序中使用模型会使在所有应用程序中部署模型的新版本变得更加困难。实现你自己的微服务来包装TF模型需要额外的工作，并且很难匹配TF Serving的特性。TF Serving具有许多特性：它可以监视目录并自动部署放置在其中的模型，并且你不必更改甚至重新启动任何应用程序就可以从新的模型版本中受益。它的速度很快、很好测试并且扩展性非常好。它支持对实验模型进行A/B测试，并仅向部分用户部署新的模型版本（在这种情况下，该模型称为金丝雀）。TF Serving还能够将单个请求分组，可以使它们在GPU上一起运行。要部署TF Serving，你可以从源代码安装它，但是使用Docker映像安装要简单得多。要部署TF Serving Docker映像集群，你可以使用编排工具（例如Kubernetes）也可以使用完全托管的解决方案（例如Google Cloud AI平台）。

3.要跨多个TF Serving实例来部署模型，你需要做的就是配置这些TF Serving实例来监视相同的models目录，然后将新模型作为SavedModel导出到子目录中。

4.gRPC API比REST API更有效。但是，它的客户端库不那么广泛，如果在使用REST API时使用压缩，则你可以获得几乎相同的性能。因此，当你需要尽可能高的性能并且客户端不仅限于REST API时，gRPC API最为有用。

5.为了减小模型的大小，使其可以在移动端或嵌入式设备上运行，TFLite使用了以下几种技术：

•它提供了一个可以优化SavedModel的转换器：它可以缩小模型并减少其延迟。为此，它修剪进行预测不需要的所有操作（例如训练操作），并在可能的情况下优化和融合操作。

•转换器还可以执行训练后的量化：该技术极大地减小了模型大小，因此下载和存储速度更快。

•它使用FlatBuffer格式来保存优化的模型，该格式可以直接加载到RAM中，而无须进行解析。这样可以减少加载时间和占用的内存。

6.有量化意识的训练包括在训练过程中向模型添加伪造的量化操作。这使模型能够学会忽略量化噪声。最终的权重会对量化更加稳健。

7.模型并行化意味着将模型分成多个部分，在多个设备上并行运行它们，希望在训练或推理期间加快模型的运行速度。数据并行化意味着创建模型的多个精确副本，并将其部署在多个设备上。在训练过程中的每次迭代中，每个副本会获得不同批次的数据，并且它会根据模型参数来计算损失的梯度。在同步数据并行处理中，汇总所有副本的梯度，然后优化器执行“梯度下降”步骤。可以将参数集中（例如，在参数服务器上），或者在所有副本之间复制参数，并使用AllReduce使其保持同

步。在异步数据并行处理中，参数是集中的，副本彼此独立运行，每个副本都在每次训练迭代结束时直接更新中心参数，而不必等待其他副本。为了加快训练速度，通常来说，数据并行性要比模型并行性更好。这主要是因为它需要较少的跨设备通信。此外，它更容易实现，并且对任何模型都以相同的方式工作，而模型并行性则需要分析模型来确定将其分成模块的最佳方法。

8.在跨多台服务器上训练模型时，可以使用以下分配策略：

• MultiWorkerMirroredStrategy并行执行镜像的数据。该模型在所有可用的服务器和设备上复制，每个副本在每次训练迭代时获取不同批次的数据，并计算其自己的梯度。使用分布式AllReduce实现（默认情况下为NCCL）来计算并在所有副本之间共享梯度的平均值，所有副本都执行相同的梯度下降步骤。由于所有服务器和设备的处理方式都完全相同，因此该策略使用起来最简单，性能良好。通常，你应该使用此策略。它的主要局限是它要求模型适合每个副本中的RAM。

• ParameterServerStrategy执行异步数据并行。该模型在所有工人上的所有设备上复制，并且参数在所有参数服务器上分片。每个工人都有自己的训练循环，与其他工人异步运行。在每次训练迭代中，每个工人都会获取自己的数据批次，从参数服务器中获取模型参数的最新版本，然后针对这些参数计算损失的梯度，将其发送到参数服务器。最后，参数服务器使用这些梯度执行“梯度下降”步骤。此策略通常比以前的策略慢，而且部署起来有点困难，因为它需要管理参数服务器。但是，训练不适合GPU RAM的大型模型很有用。

有关练习9～11的解答，请参阅https://github.com/ageron/handson-ml2上提供的Jupyter notebook。

[1] 如果在曲线上的任意两点之间绘制一条直线，则该线永远不会穿过曲线。

[2] 当要预测的值有很多数量级的变动时，你可能需要预测目标值的对数，而不是直接预测目标值。只需计算神经网络输出的指数，即可得出估计值（因为exp（log v）=v）。

[3] 在第11章中，我们讨论了许多介绍其他超参数的技术：权重初始化的类型、激活函数超参数（例如，leaky ReLU中的泄漏量）、梯度裁剪的阈值、优化器的类型及其超参数（例如，使用MomentumOptimizer时的动量超参数）、每层的正则化类型和正则化超参数（例如，使用dropout时的dropout率），等等。

附录B　机器学习项目清单

该清单可以指导你完成机器学习项目。主要有8个步骤：

1. 框出问题并看整体。

2. 获取数据。

3. 研究数据以获得深刻见解。

4. 准备数据以便更好地将潜在的数据模式提供给机器学习算法。

5. 探索许多不同的模型，并列出最佳模型。

6. 微调你的模型，并将它们组合成一个很好的解决方案。

7. 演示你的解决方案。

8. 启动、监视和维护你的系统。

显然，你应该根据自己的需要随意调整此清单。

B.1　框出问题并看整体

1. 用业务术语定义目标。

2. 你的解决方案将如何使用？

3. 当前有什么解决方案/解决方法（如果有）？

4. 你应该如何阐述这个问题（有监督/无监督，在线/离线等）？

5. 应该如何衡量性能？

6. 性能指标是否符合业务目标？

7. 达到业务目标所需的最低性能是多少？

8. 有没有一些相似的问题？你可以重用经验或工具吗？

9. 有没有相关有经验的人？

10. 你会如何手动解决问题？

11. 列出你（或其他人）到目前为止所做的假设。

12. 如果可能，请验证假设。

B.2 获取数据

注意：尽可能地自动化，以便你可以轻松地获取新数据。

1. 列出所需的数据以及你需要多少数据。

2. 查找并记录可从何处获取该数据。

3. 检查将占用多少空间。

4. 检查法律义务，并在必要时获得授权。

5. 获取访问授权。

6. 创建一个工作空间（具有足够的存储空间）。

7. 获取数据。

8. 将数据转换为可以轻松操作的格式（无须更改数据本身）。

9. 确保敏感信息被删除或受保护（例如匿名）。

10. 检查数据的大小和类型（时间序列、样本、地理等）。

11. 抽样一个测试集，将其放在一边，再也不要看它（无数据监听！）。

B.3 研究数据

注意：请尝试从现场专家那里获取有关这些步骤的见解。

1. 创建数据副本来进行研究（必要时将其采样到可以管理的大小）。

2. 创建Jupyter notebook以记录你的数据研究。

3. 研究每个属性及其特征：

- 名称
- 类型（分类、整数/浮点型、有界/无界、文本、结构化等）
- 缺失值的百分比
- 噪声和噪声类型（随机、异常值、舍入误差等）

- 任务的实用性
- 分布类型（高斯分布、均匀分布、对数分布等）

4. 对于有监督学习任务，请确定目标属性。

5. 可视化数据。

6. 研究属性之间的相关性。

7. 研究如何手动解决问题。

8. 确定你可能希望使用的转变。

9. 确定有用的额外数据。

10. 记录所学的知识。

B.4 准备数据

注意：

- 在数据副本上工作（保持原始数据集完整）。
- 为你应用的所有数据转换编写函数，原因有5个：
- 下次获取新的数据集时，你可以轻松准备数据。
- 可以在未来的项目中应用这些转换。
- 清理并准备测试集。

• 解决方案上线后清理并准备新的数据实例。

• 使你可以轻松地将准备选择视为超参数。

1. 数据清理：

• 修复或删除异常值（可选）。

• 填写缺失值（例如，零、均值、中位数）或删除其行（或列）。

2. 特征选择（可选）：

• 删除没有为任务提供有用信息的属性。

3. 特征工程（如果适用）：

• 离散化连续特征。

• 分解特征（例如分类、日期/时间等）。

• 添加有希望的特征转换（例如log（x）、sqrt（x）、x2等）。

• 将特征聚合为有希望的新特征。

4. 特征缩放：

• 标准化或归一化特征。

B.5 列出有前途的模型

注意：

• 如果数据量巨大，则可能需要采样为较小的训练集，以便可以在合理的时间内训练许多不同的模型（请注意，这会对诸如大型神经网络或随机森林之类的复杂模型造成不利影响）。

• 尽可能自动化地执行这些步骤。

1. 使用标准参数训练来自不同类别（例如线性、朴素贝叶斯、SVM、随机森林、神经网络等）的许多快速和粗糙的模型。

2. 衡量并比较其性能。

• 对于每个模型，使用N折交叉验证，在N折上计算性能度量的均值和标准差。

3. 分析每种算法的最重要的变量。

4. 分析模型所犯错误的类型。

• 人类将使用什么数据来避免这些错误？

5. 快速进行特征选择和特征工程。

6. 在前面5个步骤中执行一两个以上的快速迭代。

7. 筛选出前三到五个最有希望的模型，优先选择会产生不同类型错误的模型。

B.6 微调系统

注意：

• 你将需要在此步骤中使用尽可能多的数据，尤其是在微调结束时。

• 与往常一样，尽可能做到自动化。

1. 使用交叉验证微调超参数：

• 将你的数据转换选择视为超参数，尤其是当你对它们不确定时（例如，如果不确定是否用零或中位数替换缺失值，或者只是删除行）。

• 除非要研究的超参数值很少，否则应优先选择随机搜索而不是网格搜索。如果训练时间很长，你可能更喜欢贝叶斯优化方法（如Jasper Snoek等人所述使用高斯过程先验）[1]。

2. 尝试使用集成方法。组合最好的模型通常会比单独运行有更好的性能。

3. 一旦对最终模型有信心，就可以在测试集中测量其性能，以估计泛化误差。

 在测量了泛化误差之后，请不要对模型进行调整：否则你会开始过拟合测试集。

B.7 演示你的解决方案

1. 记录你所做的事情。

2. 创建一个不错的演示文稿。

• 确保先突出大的蓝图。

3. 说明你的解决方案为何可以实现业务目标。

4. 别忘了介绍你一路上注意到的有趣观点。

• 描述什么有效，什么无效。

• 列出你的假设和系统的局限性。

5. 确保通过精美的可视化效果或易于记忆的陈述来传达你的主要发现（例如，“中等收入是房价的第一大预测指标”）。

B.8 启动！

1. 使你的解决方案准备投入生产环境（插入生产数据输入、编写单元测试等）。

2. 编写监控代码，以定期检查系统的实时性能，并在系统故障时触发警报。

• 当心缓慢的退化：随着数据的发展，模型往往会“腐烂”。

• 评估性能可能需要人工流水线（例如通过众包服务）。

• 监视你的输入的质量（例如，传感器出现故障，发送了随机值，或者另一个团队的输出变得过时）。这对于在线学习系统尤其重要。

3. 定期根据新数据重新训练模型（尽可能自动进行）。

[1] Jasper Snoek等人，“Practical Bayesian Optimization of Machine Learning Algorithms”，Proceedings of the 25th International Conference on Neural Information Processing Systems 2（2012）：2951－2959。

附录C　SVM对偶问题

要了解对偶性，你首先需要了解拉格朗日乘数方法。一般的想法是通过将约束移到目标函数中，将一个约束的优化目标转换为无约束的目标。让我们看一个简单的示例。假设你想找到x和y的值，使函数f（x，y）=x^2+2y最小化，并受一个等式约束：3x+2y+1=0。使用拉格朗日乘数方法，我们首先定义一个称为拉格朗日（或Lagrange函数）的新函数：g（x，y，α）=f（x，y）-α（3x+2y+1）。从原始目标中减去每个约束（在这个示例中，只有一个），然后乘以一个新的称为拉格朗日乘数的变量。

Joseph-Louis Lagrange证明，如果$(\hat{x}, \hat{y})$是有约束优化的一个解，那么肯定存在一个α，使得$(\hat{x}, \hat{y}, \hat{\alpha})$是一个拉格朗日稳定点（稳定点是所有偏导都等于零的点）。换句话说，我们可以计算g（x，y，α）对于x、y和α的偏导。我们可以找到使这些偏导为零的取值，约束优化问题（如果存在的话）的解必须在这些稳定点之间。

在此示例中，偏导数为：

$$\begin{cases} \dfrac{\partial}{\partial x} g(x,y,\alpha) = 2x - 3\alpha \\ \dfrac{\partial}{\partial y} g(x,y,\alpha) = 2 - 2\alpha \\ \dfrac{\partial}{\partial \alpha} g(x,y,\alpha) = -3x - 2y - 1 \end{cases}$$

当所有这些偏导数都等于0时，我们发现 $2\hat{x}-3\hat{\alpha}=2-2\hat{\alpha}=-3\hat{x}-2\hat{y}-1=0$，从中我们可__以很容易地找到 $\hat{x}=\frac{3}{2},\hat{y}=-\frac{11}{4}$，且 $\hat{\alpha}=1$。这是唯一的稳定点，并且考虑到约束，它必须是约束优化问题的解。

但是，此方法仅适用于相等约束。幸运的是，在某些正则性条件下（SVM目标已满足），该方法也可以推广到不等式约束（例如3x+2y+1≥0）。公式C-1给出了硬边界问题的广义拉格朗日公式，其中α（i）变量称为Karush－Kuhn－Tucker（KKT）乘数，并且必须大于或等于零。

公式C-1：硬边距问题的广义拉格朗日

$$\mathcal{L}(\boldsymbol{w},b,\alpha)=\frac{1}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w}-\sum_{i=1}^{m}\alpha^{(i)}(t^{(i)}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}^{(i)}+b)-1)$$

$$\text{其中}\quad \alpha^{(i)}\geqslant 0,\quad i=1,2,\cdots,m$$

就像使用拉格朗日乘法方法一样，你可以计算偏导数并定位稳定点。如果有解决方案，它肯定是满足KKT条件的平衡点 $(\hat{\boldsymbol{w}},\hat{b},\hat{\alpha})$：

- 满足问题约束： $t^{(i)}(\hat{\boldsymbol{w}}^{\mathrm{T}}\boldsymbol{x}^{(i)}+\hat{b})\geqslant 1,\ i=1,2,\cdots,m$
- 验证 $\hat{\alpha}^{(i)}\geqslant 0$，其中i=1，2，…，m
- 要么 $\hat{\alpha}^{(i)}=0$，要么第i个约束必须是一个主动约束，这意味着必须： $t^{(i)}(\hat{\boldsymbol{w}}^{\mathrm{T}}\boldsymbol{x}^{(i)}+\hat{b})=1$。这种情况被称为互补松弛条件。意味着 $\hat{\alpha}^{(i)}=0$或第i个实例在边界上（它是一个支持向量）。

请注意，KKT条件是稳定点成为约束优化问题解的必要条件。在某些条件下，它们也是充分条件。幸运的是，SVM优化问题恰好满足了这些条件，因此，保证了任何满足KKT条件的稳定点都可以作为约束优化问题的解。

我们可以用公式C-2计算关于w和b的广义拉格朗日偏导数。

公式C-2：广义拉格朗日的偏导数

$$\nabla_w \mathcal{L}(\boldsymbol{w}, b, \alpha) = \boldsymbol{w} - \sum_{i=1}^{m} \alpha^{(i)} t^{(i)} \boldsymbol{x}^{(i)}$$

$$\frac{\partial}{\partial b} \mathcal{L}(\boldsymbol{w}, b, \alpha) = -\sum_{i=1}^{m} \alpha^{(i)} t^{(i)}$$

当这些偏导数等于零时，我们得到方程C-3。

公式C-3：稳定点的性质

$$\hat{\boldsymbol{w}} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \boldsymbol{x}^{(i)}$$

$$\sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} = 0$$

如果将这些结果带入广义拉格朗日的定义中，有些项会消失，得到公式C-4。

公式C-4：SVM问题的对偶形式

$$\mathcal{L}(\hat{\boldsymbol{w}}, \hat{b}, \alpha) = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}\boldsymbol{x}^{(i)\mathrm{T}}\boldsymbol{x}^{(j)} - \sum_{i=1}^{m}\alpha^{(i)}$$

$$\alpha^{(i)} \geqslant 0 \quad \text{其中} i = 1, 2, \cdots, m$$

现在的目标是找到使该函数最小化的向量$\hat{\boldsymbol{\alpha}}$，所有情况下$\hat{\alpha}^{(i)} \geqslant 0$。这个受约束的优化问题是我们一直在寻找的对偶问题。

一旦找到最优解$\hat{\boldsymbol{\alpha}}$，就可以使用公式C-3的第一行来计算$\hat{\boldsymbol{w}}$。要计算$\hat{b}$，你可以使用以下事实：支持向量必须验证$t^{(i)}(\hat{\boldsymbol{w}}^{\mathrm{T}}\boldsymbol{x}^{(i)} + \hat{b}) = 1$，因此，如果第k个实例是支持向量（即$\hat{\alpha}^{(k)} > 0$），你可以使用它来计算$\hat{b} = t^{(k)} - \hat{\boldsymbol{w}}^{\mathrm{T}}\boldsymbol{x}^{(k)}$。但是，通常最好计算所有支持向量的平均值，以获得更稳定和精确的值，如公式C-5所示。

公式C-5：使用对偶形式的偏差估计

$$\hat{b} = \frac{1}{n_s}\sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m}\left[t^{(i)} - \hat{\boldsymbol{w}}^{\mathrm{T}}\boldsymbol{x}^{(i)}\right]$$

附录D　自动微分

本附录介绍了TensorFlow的自动微分特性如何工作，以及与其他解决方案的比较。

假设你定义了函数f（x，y）=x^2y+y+2，你需要它的偏导$\frac{\partial f}{\partial x}$和$\frac{\partial f}{\partial y}$，通常需要执行梯度下降（或其他优化算法）。你的主要选择有手动微分、有限差分近似、前向模式自动微分（autodiff）和反向模式自动微分。TensorFlow实现了最后一个选项。我们来逐一看看这些选项。

D.1　手动微分

第一种方式是拿出纸和笔，用微积分知识手动推导出偏导。对刚刚定义的方程f（x，y），推出偏导很容易。需要使用以下5个规则：

- 常数的导数为0。
- λx的导数是λ（其中λ是一个常数）。
- x^{λ}的导数是$\lambda x^{\lambda-1}$，因此x^2的导数是2x。
- 函数和的导数是这些函数的导数的和。
- 函数的λ倍的导数是函数导数的λ倍。

从这些规则，你可以得出公式D-1。

公式D-1：f（x，y）的偏导数

$$\frac{\partial f}{\partial x}=\frac{\partial(x^2y)}{\partial x}+\frac{\partial y}{\partial x}+\frac{\partial 2}{\partial x}=y\frac{\partial(x^2)}{\partial x}+0+0=2xy$$

$$\frac{\partial f}{\partial y}=\frac{\partial(x^2y)}{\partial y}+\frac{\partial y}{\partial y}+\frac{\partial 2}{\partial y}=x^2+1+0=x^2+1$$

该方法对于复杂函数就会变得特别烦琐，且容易出错。好消息是，我们还可以选择有限差分近似。

D.2　有限差分近似

回想一下，函数h（x）在点x_0处的导数h’（x_0）是该点处函数的斜率。更精确地说，导数的定义为当x无限接近x_0时通过x_0点和函数上另一个点x的直线的斜率的极限（见公式D-2）。

公式D-2：点x_0处函数h（x）的导数的定义

$$h'(x_0)=\lim_{x\to x_0}\frac{h(x)-h(x_0)}{x-x_0}$$

$$=\lim_{\varepsilon\to 0}\frac{h(x_0+\varepsilon)-h(x_0)}{\varepsilon}$$

因此，如果我们要计算在x=3和y=4时关于x的f（x，y）的偏导数，则可以计算f（3+ε，4）-f（3，4）并将结果除以ε（使用极小的ε值）。这种类型的导数数值逼近称为有限差分近似，这种特定的方程式称为牛顿差商。如以下代码所示：

```
def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0, 0.00001)
```

不幸的是，结果是不精确的（对于更复杂的函数，它会变得更糟）。正确的结果分别是24和10，但是我们得到：

```
>>> print(df_dx)
24.000039999805264
>>> print(df_dy)
10.000000000331966
```

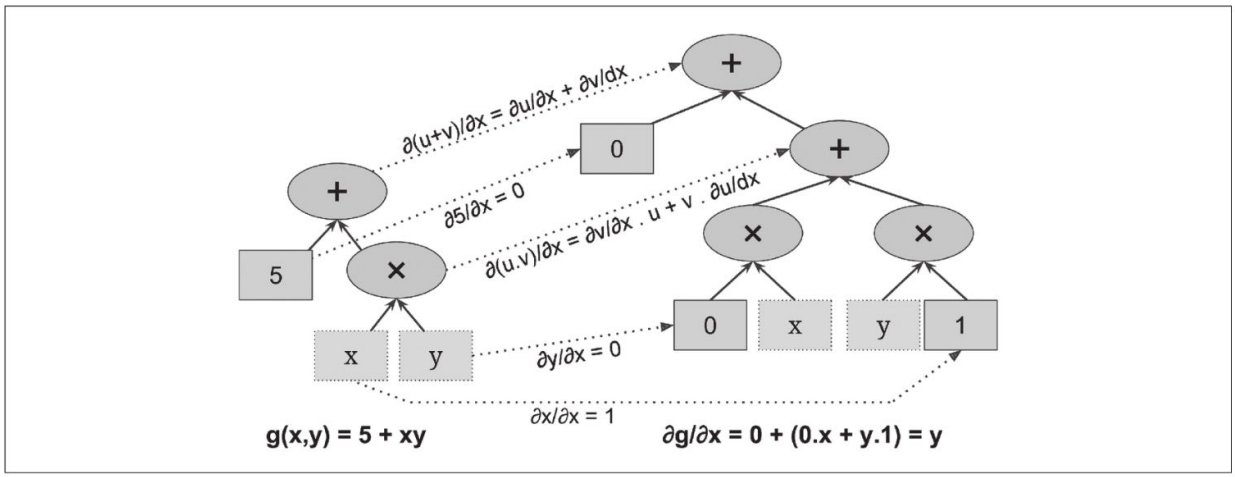
注意，为了计算两个偏导，至少需要调用函数f（）三次（代码中调用了四次，但是可以优化）。如果它有1000个参数，则至少调用1001次。当处理大型神经网络时，这会使有限差分近似的效率过低。

然而，该方法非常容易实现，它是检查其他方法是否正确实现的好工具。例如，如果它的计算结果与你手动推导的不同，那么你的计算肯定有问题。

到目前为止，我们已经考虑了两种计算梯度的方法：手动微分和有限差分近似。不幸的是，两者都存在训练大规模神经网络的致命缺陷。因此，让我们转向前向模式自动微分。

D.3　前向模式自动微分

图D-1显示了前向模式自动微分如何在更简单的函数g（x，y）=5+xy上工作。该函数的图形显示在左侧。经过前向模式自动微分后，我们得到了右侧的图，该图表示了偏导数 $\frac{\partial y}{\partial x} = 0 + (0 \times x + y \times 1) = y$ （我们可以类似地获得关于y的偏导数）。



图D-1：前向模式自动微分

该算法从输入到输出遍历计算图（因此称为“前向模式”）。首先从获取叶节点的偏导数开始。常数节点（5）返回常数0，因为常数的导数始终为0。变量x由于∂x/∂x=1而返回常数1，变量y由于∂y/∂x=0返回常数0（如果我们寻找关于y的偏导数，则正好相反）。

现在我们拥有将图上移到函数g中的乘法节点所需的全部。微积分告诉我们，两个函数u和v的乘积的导数为∂（u×v）/∂x=（∂v/∂x）u+v（∂u/∂x）。因此，我们可以在构造右侧图的很大一部分，表示0×x+y×1。

最后，我们可以转到函数g中的加法节点。如前所述，函数总和的导数就是这些函数的导数的总和。因此，我们只需要创建一个加法节点并将其连接到我们已经计算的图的各部分。我们得到正确的偏导数：∂g/∂x=0+（0×x+y×1）。

但是，该方程式可以简化（很多）。可以将一些修剪步骤应用于计算图，以消除所有不必要的操作，我们得到的图要小得多，只有一个节点：∂g/∂x=y。在这种情况下，简化是相当容易的，但是对于更复杂的函数，前向模式自动微分可能会产生巨大的图形，可能难以简化并导致性能欠佳。

请注意，我们从一个计算图开始，而前向模式自动微分产生了另一个计算图。这被称为符号微分，它有两个不错的功能：首先，一旦生成了导数的计算图，我们就可以使用任意多次来计算给定函数的x和x的任意值；其次，如果需要，我们可以在结果图上再次运行前向模式自动微分来获得二阶导数（即导数的导数）。我们甚至可以计算三阶导数，以此类推。

但是，也可以仅通过即时计算中间结果而无须构造图（即在数字上而不是在符号上）来运行前向模式自动微分。一种方法是使用二元数，它是奇怪但引人入胜的数，形式为a+b ε ，其中a和b是实数，而 ε 是无穷小数，使得 $\varepsilon^2=0$（但 $\varepsilon \neq 0$）。你可以将二元数42+24 ε 视为类似于42.0000…000024，无穷个0（但是，简化的目的只是为了让你了解什么是二元数）。一个二元数在内存中表示为一对浮点数。例如，42+24 ε 由（42.0，24.0）对表示。

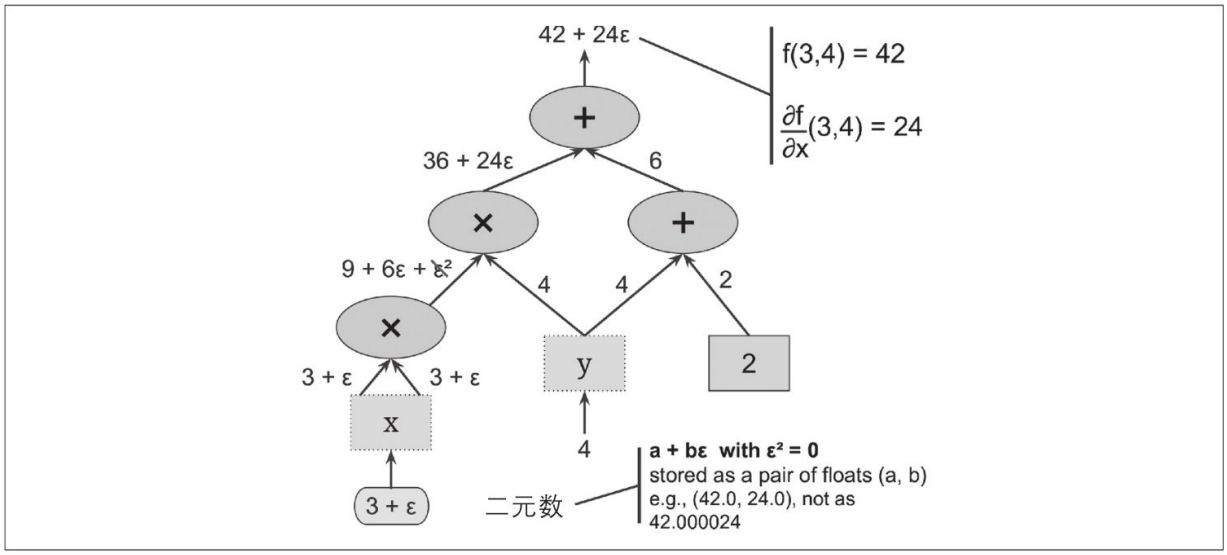
如公式D-3所示，可以对二元数进行加法、乘积等操作。

公式D-3：一些二元数运算

$$\lambda(a+b\varepsilon)=\lambda a+\lambda b\varepsilon$$
$$(a+b\varepsilon)+(c+d\varepsilon)=(a+c)+(b+d)\varepsilon$$
$$(a+b\varepsilon)\times(c+d\varepsilon)=ac+(ad+bc)\varepsilon+(bd)\varepsilon^2=ac+(ad+bc)\varepsilon$$

最重要的是，可以证明h（a+b ε ）=h（a）+b×h′（a） ε ，因此计算h（a+ ε ）既可以得到h（a），也可以得到导数h’（a）。图D-2显

示了f（x，y）关于x在x=3和y=4（写为$\frac{\partial f}{\partial x}(3,4)$）时的偏导数可以使用二元数计算。我们要做的就是计算f（3+ ε，4），这将输出一个二元数，其第一个分量等于f（3，4），第二个分量等于$\frac{\partial f}{\partial x}(3,4)$。



图D-2：使用二元数的前向模式自动微分

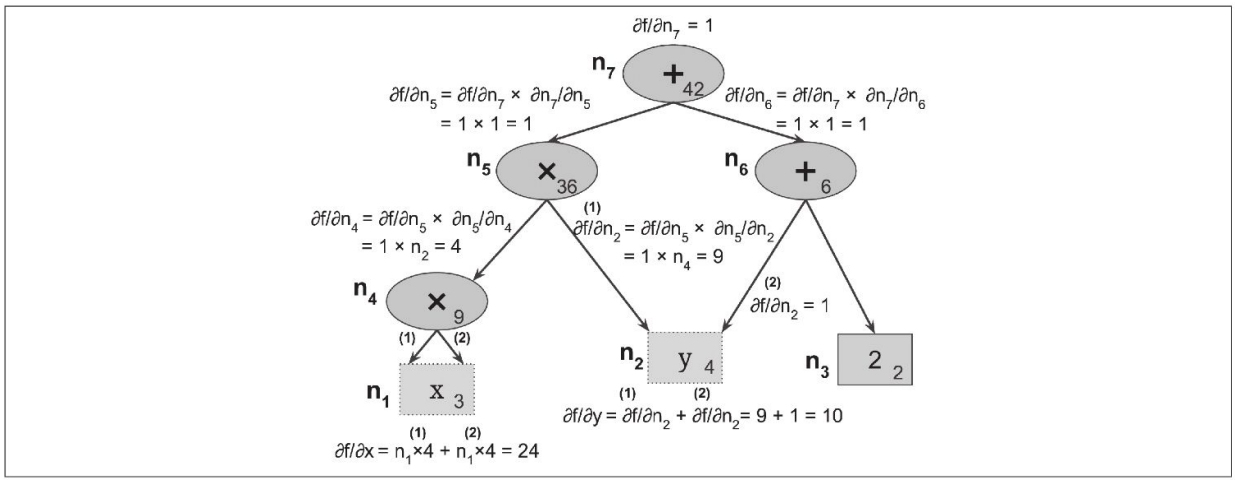
为了计算$\frac{\partial f}{\partial x}(3,4)$，我们将不得不再次遍历该图，但这一次x=3且y=4+ ε 。

因此，前向模式自动微分比有限差分近似要精确得多，但它也有相同的缺陷，至少在输入多而输出少的情况下（例如在处理神经网络时）：如果有1000个参数，则需要遍历1000次图形才能计算所有的偏导数。这就是反向模式自动微分的亮点：它只需遍历两次图形就可以计算所有参数。让我们看看它是如何做的。

D.4　反向模式自动微分

反向模式自动微分是TensorFlow实现的解决方案。它首先以向前的方向（即从输入到输出）遍历图形来计算每个节点的值。然后，它进行第二次遍历，这一次反向（即从输出到输入）计算所有的偏导数。名称

“反向模式”来自此图的第二次遍历，梯度沿相反方向流动。图D-3表示第二次遍历。在第一次遍历中，从x=3和y=4开始计算所有节点值。你可以在每个节点的右下角看到这些值（例如x×x=9）。为了清楚起见，将节点标记为n_1至n_7。输出节点为n_7：f（3，4）=n_7=42。



图D-3：反向模式自动微分

这个想法是逐步的下降图，计算关于每个连续节点的f（x，y）的偏导数，直到到达变量节点为止。为此，反向模式自动微分严重依赖于链式规则，如公式D-4所示。

公式D-4：链式规则

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

因为n_7是输出节点，$f=n_7$，所以$\frac{\partial f}{\partial n_7} = 1$。

让我们继续下降图到n_5：当n_5变化时，f变化了多少？答案是$\frac{\partial f}{\partial n_5} = \frac{\partial f}{\partial n_7} \times \frac{\partial n_7}{\partial n_5}$。我们已经知道$\frac{\partial f}{\partial n_7} = 1$，所以我们只需要$\frac{\partial n_7}{\partial n_5}$。因为$n_7$是$n_5$+$n_6$

的和，我们发现$\frac{\partial n_7}{\partial n_5}=1$，因此$\frac{\partial f}{\partial n_5}=1\times 1=1$。

现在我们进入节点n_4：当n_4变化时，f变化了多少？答案是$\frac{\partial f}{\partial n_4}=\frac{\partial f}{\partial n_5}\times\frac{\partial n_5}{\partial n_4}$。因为$n_5=n_4\times n_2$，我们发现$\frac{\partial n_5}{\partial n_4}=n_2$，所以$\frac{\partial f}{\partial n_4}=1\times n_2=4$。

这个过程一直持续到图的底部。到那时，我们在点x=3和y=4处计算f（x，y）的所有偏__导数。在此示例中，我们发现$\frac{\partial f}{\partial x}=24$且$\frac{\partial f}{\partial y}=10$。听起来不错！

反向模式自动微分是一种非常强大且准确的技术，尤其是在输入很多而输出很少的情况下，因为它仅需要一个前向加上每个输出一个反向，来计算所有输出对所有输入的所有偏导数。在训练神经网络时，我们通常希望将损失最小化，只有一个输出（损失），因此只需要通过两次图遍历即可计算梯度。反向模式自动微分也可以处理不是完全可微的函数，只要它在可微点上计算偏导数即可。在图D-3中，数值结果是在每个节点上即时计算的。但是，这并不完全是TensorFlow所做的：相反，它创建了一个新的计算图。换句话说，它实现了符号反向模式自动微分。这样，只需要生成一次用于计算与神经网络中所有参数有关的损失梯度的计算图，就可以在优化器需要计算梯度时一次又一次地执行该计算图。此外，如果需要，还可以计算高阶导数。

 如果你想在C++中实现一种新的底层TensorFlow操作，并且希望使其与自动微分兼容，那么你将需要提供一个函数，该函数返回对其输入的输出的偏导数。例如，假设你实现了一个计算输入的平方的函数：f（x）=x^2。在这种情况下，你将需要提供相应的导数函数：f’（x）=2x。

附录E　其他流行的人工神经网络架构

在本附录中，我将快速概述一些历史上重要的神经网络架构，这些架构的使用如今比深度多层感知器（见第10章），卷积神经网络（见第14章），循环神经网络（见第15章）或自动编码器要少得多（见第17章）。它们在论文中经常被提及，有些仍然在多种应用中使用，因此有必要了解它们。此外，我们将讨论深度置信网络，这是深度学习直到21世纪10年代初期的最新技术。它们仍然是非常活跃的研究主题，很可能在不久的将来重新流行。

E.1　Hopfield网络

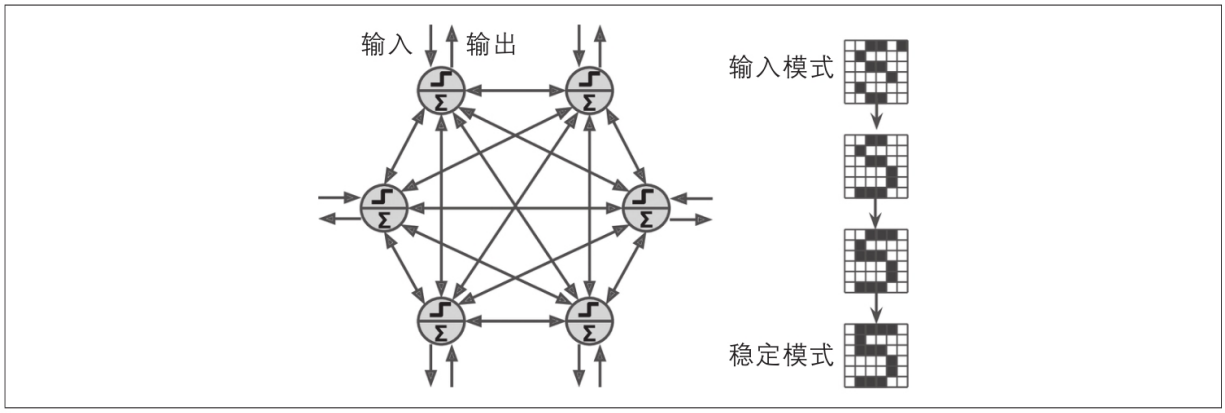
Hopfield网络于1974年第一次被W.A.Little提出，随后由J.Hopfield在1982年推广开来。它们是联想记忆网络：首先教它们一些模式，然后当它们遇见新模式时就会输出最接近该模式的已学习模式。在被其他方法超过之前，该方法对特征识别特别有用：首先通过显示特征图像示例来训练网络（每个二进制像素映射到一个神经元），然后当显示一个新特征图像后，经过几轮迭代，它会输出最接近的已学习特征图像。

它们是全连接图（见图E-1）；每个神经元连接到其他神经元。注意，图中的图像是6×6像素，所以左边的神经网络应该包含36个神经元（及630个连接），但是为了视觉上更加清晰，这里表示为一个小得多的网络。

训练算法使用Hebb规则工作（见10.1.3节）：对每个训练图像，如果响应的像素都开启或关闭，则两个神经元之间的权重增加，但是如果一个像素开启而另一个关闭，则两个神经元之间的权重减小。

要向网络显示一个新图像，激活与激活像素对应的神经元即可。然后，网络会计算每个神经元的输出，并给出一个新图像。接着，可以使

用这个新图像，并重复整个过程。一段时间后，网络进入一个新的稳定状态。一般来说，这对应于最接近输入图像的训练图像。



图E-1：Hopfield网络

能量函数与Hopfield网络相关联。在每个迭代中，能量减小，所以网络保证最终稳定在低能量状态。训练算法以减小训练模式的能量等级的方式来调整权重，所以网络稳定在这些低能量结构之一中。不幸的是，一些不在训练集中的模式也以低能量模式终止，所以网络有时候会稳定在没有学到东西的结构中。这被称为虚假模式。

Hopfield网络另一个主要缺陷是它不能很好地被扩展——它们的内存容量大约等于神经元数量的14%。例如，要分类28×28个图像，需要一个有784个全连接神经元和306 936个权重的Hopfield网络。该网络仅能学习110个不同特征（784的14%）。对如此小的内存来说，其参数量很多。

E.2　玻尔兹曼机

玻尔兹曼机是Geoffrey Hinton和Terrence Sejnowski在1985年发明的。与Hopfield网络类似，它们是全连接的ANN，但是基于随机神经元：这些神经元以某些概率输出1，其他输出0，而不是使用一个确定的步骤函数决定输出值。这些ANN使用的概率函数基于玻尔兹曼分布（用于统计力学），并以此得名。公式E-1给出了特定神经元输出1的概率。

公式E-1：第i个神经元输出1的概率

$$p(s_i^{(next\ step)} = 1) = \sigma\left(\frac{\sum_{j=1}^{N} w_{i,j}s_j + b_i}{T}\right)$$

• s_j是第j个神经元的状态（0或1）。

• $w_{i,j}$是第i和第j个神经元之间的连接权重。注意$w_{i,i}=0$。

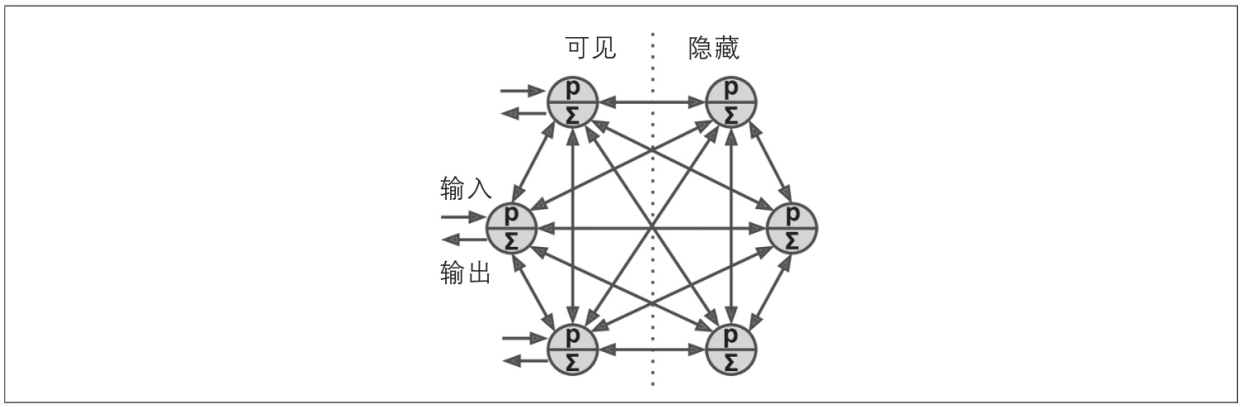
• b_i是第i个神经元的偏置项。我们可以通过向网络添加偏置神经元来实现。

• N是网络中神经元的数量。

• T是一个数字，称为网络温度。温度越高，输出的随机性就越高（即接近50%的可能性越大）。

• σ 是逻辑函数。

玻尔兹曼机里的神经元被分为两组：可见单元和隐藏单元（见图E-2）。所有的神经元都以相同的随机方式工作，但是可见单元是接收输入并从中读取输出的单元。



图E-2：玻尔兹曼机

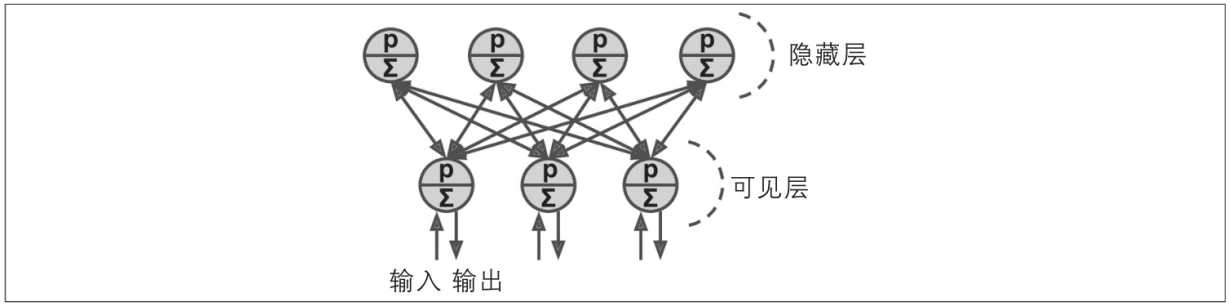
由于其随机性，玻尔兹曼机永远不会稳定在一个固定结构，而是在许多结构之间不断切换。如果运行了足够长的时间，观察特定结构的概率将仅是连接权重和偏置项的函数，而不是原始结构的函数（类似地，将一副纸牌洗牌足够长时间后，则这副牌的顺序将与原始状态无关）。当网络到达一个原始结构“忘记”了的状态，就处于热平衡状态（尽管其结构仍然在改变）。通过适当地设置网络参数，使网络达到热平衡状态，然后观察状态，就可以模拟各种概率分布。这被称为生成模型。

训练玻尔兹曼机意味着找到使得网络接近训练集合概率分布的参数。例如，如果有三个可见神经元，且训练集包含75%的（0，1，1）三元组、10%的（0，0，1）三元组，以及15%的（1，1，1）三元组，在训练玻尔兹曼机后，可以用它来生成具有大致相同的概率分布的随机二进制三元组。例如，大约75%的时间会输出（0，1，1）三元组。这种生成模型可以以各种方式使用。例如，如果用于训练图像，并向网络提供不完整图像或有噪声图像，则会以合理的方式自动“修复”图像。也可以使用生成模型进行分类。只需添加一些可见神经元来编码训练图像的类（例如，当训练图像表示5时，添加10个可见神经元，并打开第5个神经元）。然后，当给出一个新图像时，网络会自动打开适当的可见神经元，表示图像的类（例如，如果图像表示5，则打开第5个可见神经元）。

不幸的是，没有有效的技术来训练玻尔兹曼机。然而，已经开发出相对有效的算法来训练受限玻尔兹曼机（Restricted Boltzmann Machine，RBM）。

E.3　受限玻尔兹曼机

RBM是简化的玻尔兹曼机，其可见单元或隐藏单元之间没有连接，只在可见单元和隐藏单元之间有连接。例如，图E-3表示一个有三个可见单元和四个隐藏单元的RBM。



图E-3：受限玻尔兹曼机

一个被称为对比散度（Contrastive Divergence）的有效训练算法于2005年被MiguelÁ.Carreira-Perpiñán和Geoffrey Hinton.提出[1]。以下是它的工作原理：对每个训练实例x，算法首先将其可见单元的状态设置为x_1，x_2，…，x_n，然后将其传递到网络。之后通过应用随机方程（见公式E-1）来计算隐藏单元的状态。得到隐藏向量h（hi等于第i个单元的状态）。接下来通过应用相同的随机方程计算可见单元的状态。得出向量x′ 。然后再次计算隐藏层的状态，得出向量h’。现在，可以通过公式E-2中的规则更新每个连接权重，其中 η 是学习率。

公式E-2：对比散度权重更新

$$w_{i,j} \leftarrow w_{i,j} + \eta(\boldsymbol{x}\boldsymbol{h}^{\mathrm{T}} - \boldsymbol{x}'\boldsymbol{h}'^{\mathrm{T}})$$

该算法最大的好处是不需要等待网络到达热平衡：它只前进，后退，再前进。这使得它比之前的算法更加高效，它是基于多个栈式RBM的深度学习第一次成功的关键因素。

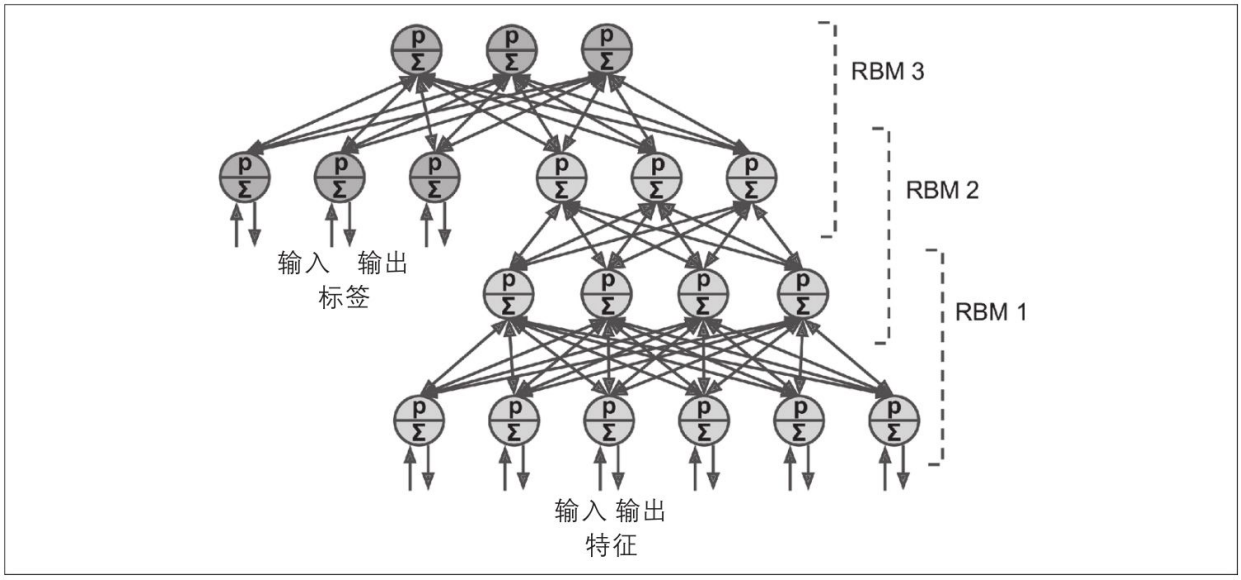
E.4　深度置信网

RBM的多层可以堆叠。第一层RBM的隐藏单元可以是第二层RBM的可见单元，等等。这种RBM堆叠被称为深度置信网（Deep Belief Net，DBN）。

Geoffrey Hinton的学生之一Yee-Whye Teh观察到，有可能使用对比散度一次训练DBN的一层，从较低的层开始，然后逐渐向上移到最高的层。这导致了一篇开创性的文章，在2006年2月引发了深度学习海啸[2]。

与RBM相似，DBN学会在没有任何监督的情况下，重现其输入的分布概率。然而，DBN的性能更好，原因同样是深度神经网络比浅的更强大：真实世界中的数据通常是以层次模式组织的，DBN利用了这一点。它们的低层学习输入数据中的低层特征，同时高层学习高层特征。

与RBM类似，DBN基本上是无监督的，但是仍然可以通过添加一些可见单元代表标签来以有监督的方式训练它们。此外，DBN的一个重要特征是它们可以以半监督的方式进行训练。图E-4显示了为半监督学习配置的DBN。



图E-4：为半监督学习配置的深度置信网络

首先，RBM 1在无监督的情况下被训练。它学习训练数据中的低层特征。然后RBM 2使用RBM 1的隐藏单元作为输入进行训练，还是无监督的：它学习高层特征（注意，RBM 2的隐藏单元仅包括最右边的三个单元，没有标记单元）。还有几个RBM以这种方式被堆叠，你已经知道方法，此处不再赘述。到目前为止，训练是100%无监督的。最后，RBM 3使用RBM 2的隐藏单元作为输入，额外的可见单元表示目标标签（例如，独热向量代表实例类）。它学习使用训练标签关联高层特征。该步骤是有监督的。

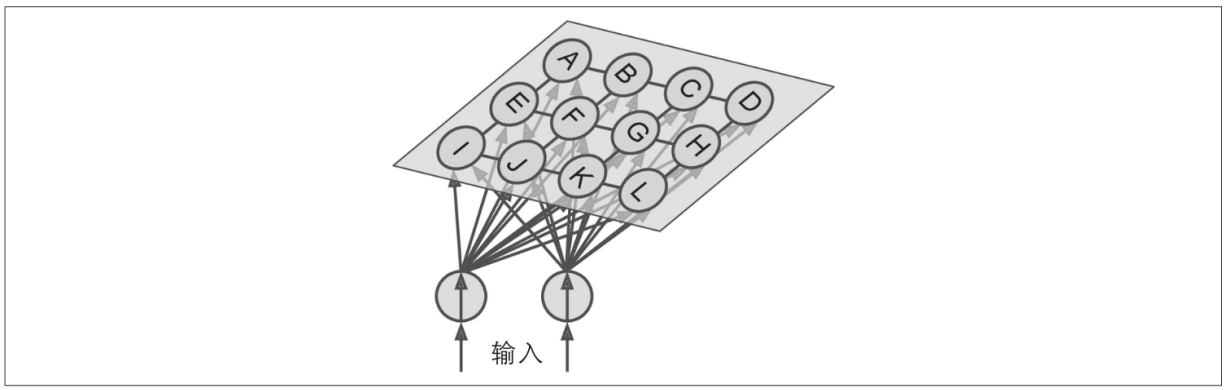
在训练结束时，如果将RBM 1传到一个新实例，信号将传播到RBM 2，然后再到顶层的RBM 3，最后回到标签单元。如果顺利的话，适当的标签会被点亮。这是如何将DBN用于分类的示例。

这种半监督方式的很大好处是不需要太多标记的训练数据。如果无监督的RBM做得足够好，那么每个类只需要少量标记的训练数据。类似地，婴儿在无监督情况下学习识别物体，所以当你指着椅子说“椅子”时，婴儿可以把“椅子”这个词和他已经学会识别的物体联系起来。你不需要指着每个椅子说它是“椅子”；只需要几个示例就够了（只要婴儿可以确定你指的是椅子，而不是椅子的颜色或椅子的一部分）。

令人惊讶的是，DBN也可以反向工作。如果激活其中一个标签单元，信号将传播到RBM 3的隐藏单元，然后到RBM 2，最后到RBM 1，并且RBM 1的可见单元会输出一个新实例。该新实例通常看起来和激活的标签单元类似。DBN的这种生成能力是十分强大的。例如，它已经被用于自动生成图像字幕，反之亦然：首先，DBN被训练（无监督）来学习图像特征，另一个DBN被训练来学习字幕中的特征（例如，“car”通常和“automobile”一起出现）。然后，一个RBM堆叠在两个DBN的顶部，并用一组图像及其字幕进行训练。它学习将图像中的高层特征与字幕中的高层特征相关联。接下来，如果传给图像DNB一个汽车图像，信号将通过网络传播，直到最高的RBM，并返回底部的字幕DBN，产生字幕。由于RBM和DBN的随机性，字幕会随机变化，但一般都适用于图像。如果生成数百个字幕，那么生成最频繁的字幕可能是图像较好的描述[3]。

E.5 自组织映射

自组织映射（SOM）与迄今为止我们所讨论的所有其他类型的神经网络都有很大不同。它们用于生产高维度据集的低维表示，通常用于可视化、聚类或分类。如图E-5所示，神经元分布在一张二维图上（通常二维用于可视化，但可以是任意数量的维度），并且每个神经元对每个输入都有一个加权连接（注意图中只显示两个输入，但通常输入数量很大，因为SOM的所有点都用于降低维度）。



图E-5：自组织映射

一旦网络被训练，可以传给它一个新实例，这将只激活一个神经元（即图上的一个点）：其权重向量最接近输入的神经元。一般情况下，原始输入空间附近的实例将激活图附近的神经元。这使得SOM不仅可以用于可视化（特别是可以轻松识别图上的集群），还可以用于语音识别等应用。例如，如果每个实例表示人类发音的音频记录，则元音“a”的不同发音将激活图中相同区域的神经元，而元音“e”的发音将激活另一区域的神经元，中间声音通常激活图中间的神经元。

 与第8章讨论的其他降维技术不同，它的所有实例都映射到低维空间中离散的点（每个神经元一个点）。当神经元很少时，该技术更应该称为聚类而不是降维。

训练算法是无监督的。它通过使所有神经元相互竞争而起作用。首先，所有权重被随机初始化。然后随机挑选一个训练实例，并传到网络。所有神经元计算它们的权重向量和输入向量之间的距离（这与我们迄今为止看到的人造神经元非常不同）。与其距离最小的神经元胜出，并调整其权重向量使其更接近输入向量，使其能够赢得未来类似于该输入的其他输入。它也拉拢其他相邻神经元，更新它们的权重向量使其稍微接近输入向量（但是不如胜者那样更新它们的权重多）。然后算法选择另一个训练实例，并不断重复该过程。该算法倾向使附近神经元逐渐具有相似的输入[4]。

[1] MiguelÁ.Carreira-Perpiñán and Geoffrey E.Hinton，“On Contrastive Divergence Learning”，Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics（2005）：59－66。

[2] Geoffrey E.Hinton et al.，“A Fast Learning Algorithm for Deep Belief Nets”，Neural Computation 18（2006）：1527－1554。

[3] 有关更多详细信息和演示，请参见Geoffrey Hinton的这段视频：https://homl.info/137。

[4] 想象一个技能大致相同的幼儿班。一个孩子在篮球上好一点。这促使他们去做更多的练习，尤其是和朋友们一起练习。一段时间以后，这群小朋友都变得擅长篮球，远胜于其他小朋友。但是没关系，其他小朋友有其他的兴趣。这样，一段时间后，这个班级就充满了各种兴趣小组。

附录F　特殊数据结构

在本附录中，我们将快速浏览TensorFlow支持的数据结构，而不仅仅是常规的浮点或整数张量。这包括字符串、不规则张量、稀疏张量、张量数组、集合和队列。

F.1　字符串

张量可以容纳字节串，这对于自然语言处理特别有用（见第16章）：

```
>>> tf.constant(b"hello world")
<tf.Tensor: id=149, shape=(), dtype=string, numpy=b'hello world'>
如果你使用Unicode字符串来构建一个张量，则TensorFlow
```

如果你使用Unicode字符串来构建一个张量，则TensorFlow会自动将其编码为UTF-8：

```
>>> tf.constant("café")
<tf.Tensor: id=138, shape=(), dtype=string, numpy=b'caf\xc3\xa9'>
```

也可以创建表示Unicode字符串的张量。只需创建一个32位整数的数组，每个整数代表一个Unicode代码点[1]：

```
>>> tf.constant([ord(c) for c in "café"])
<tf.Tensor: id=211, shape=(4,), dtype=int32,
            numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

 在类型为tf.string的张量中，字符串长度不是张量形状的一部分。换句话说，字符串被视为原子值。但是，在Unicode字符串张量（即int32张量）中，字符串的长度是张量形状的一部分。

tf.strings包包含几个函数来操作字符串张量，例如length（）用来计算字节字符串中的字节数（如果设置unit="UTF8_CHAR"，则是计算代码点数），unicode_encode（）将Unicode字符串张量（即int32张量）转换为字节字符串张量，并使用unicode_decode（）进行反操作：

```
>>> b = tf.strings.unicode_encode(u, "UTF-8")
>>> tf.strings.length(b, unit="UTF8_CHAR")
<tf.Tensor: id=386, shape=(), dtype=int32, numpy=4>
>>> tf.strings.unicode_decode(b, "UTF-8")
<tf.Tensor: id=393, shape=(4,), dtype=int32,
            numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

你还可以操作包含多个字符串的张量：

```
>>> p = tf.constant(["Café", "Coffee", "caffè", "咖啡"])
>>> tf.strings.length(p, unit="UTF8_CHAR")
<tf.Tensor: id=299, shape=(4,), dtype=int32,
            numpy=array([4, 6, 5, 2], dtype=int32)>
>>> r = tf.strings.unicode_decode(p, "UTF8")
>>> r
tf.RaggedTensor(values=tf.Tensor(
[   67    97   102   233    67   111   102   102   101   101    99    97
   102   102   232 21654 21857], shape=(17,), dtype=int32),
   row_splits=tf.Tensor([ 0  4 10 15 17], shape=(5,), dtype=int64))
>>> print(r)
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101],
                  [99, 97, 102, 102, 232], [21654, 21857]]>
```

请注意，解码后的字符串存储在RaggedTensor中。那是什么？

F.2　不规则张量

不规则张量是一种特殊的张量，代表不同大小的数组列表。更一般而言，它是具有一个或多个参差不齐尺寸的张量，这意味着其切片可能具有不同长度的尺寸。在不规则张量r中，第二个维度是不规则的维度。在所有不规则张量中，第一个维度始终是规则的维度（也称为统一维度）。

不规则张量r的所有元素都是规则张量。例如，让我们看一下不规则张量的第二个元素：

```
>>> print(r[1])
tf.Tensor([ 67 111 102 102 101 101], shape=(6,), dtype=int32)
```

tf.ragged包包含几个函数来创建和操作不规则张量。让我们使用tf.ragged.constant（）创建第二个不规则张量，并沿轴0和第一个不规则张量进行合并：

```
>>> r2 = tf.ragged.constant([[65, 66], [], [67]])
>>> print(tf.concat([r, r2], axis=0))
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101], [99, 97,
102, 102, 232], [21654, 21857], [65, 66], [], [67]]>
```

结果并不令人惊讶：r2中的张量被附加到沿轴0的r中的张量。但是，如果我们将r和另一个不规则张量沿轴1合并会怎么样？

```
>>> r3 = tf.ragged.constant([[68, 69, 70], [71], [], [72, 73]])
>>> print(tf.concat([r, r3], axis=1))
<tf.RaggedTensor [[67, 97, 102, 233, 68, 69, 70], [67, 111, 102, 102, 101, 101,
71], [99, 97, 102, 102, 232], [21654, 21857, 72, 73]]>
```

这次，请注意r中的第i个张量和r3中的第i个张量是合并的。现在，这很不寻常了，因为所有这些张量具有不同的长度。

如果你调用to_tensor（）方法，它将转换为常规张量，用零填充较短的张量以获得相等长度的张量（你可以通过设置default_value参数来更改默认值）：

```
>>> r.to_tensor()
<tf.Tensor: id=1056, shape=(4, 6), dtype=int32, numpy=
array([[   67,    97,   102,   233,     0,     0],
       [   67,   111,   102,   102,   101,   101],
       [   99,    97,   102,   102,   232,     0],
       [21654, 21857,     0,     0,     0,     0]], dtype=int32)>
```

许多TF操作支持不规则张量。有关完整列表，请参见tf.RaggedTensor类的文档。

F.3　稀疏张量

TensorFlow还可以有效地表示稀疏张量（即主要包含零的张量）。只需创建一个tf.SparseTensor，指定非零元素的索引和值以及张量的形状即可。索引必须以“读取顺序”列出（从左到右，从上到下）。如果不确定，只需使用tf.sparse.reorder（）。你可以使用tf.sparse.to_dense（）将稀疏张量转换为密集张量（即规则张量）：

```
>>> s = tf.SparseTensor(indices=[[0, 1], [1, 0], [2, 3]],
                        values=[1., 2., 3.],
                        dense_shape=[3, 4])
>>> tf.sparse.to_dense(s)
<tf.Tensor: id=1074, shape=(3, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [2., 0., 0., 0.],
       [0., 0., 0., 3.]], dtype=float32)>
```

请注意，稀疏张量不支持与密集张量一样多的操作。例如，你可以将一个稀疏张量乘以任何标量值，然后得到一个新的稀疏张量，但是你不能将标量值加到稀疏张量，这不会返回一个稀疏张量：

```
>>> s * 3.14
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x13205d470>
>>> s + 42.0
[...] TypeError: unsupported operand type(s) for +: 'SparseTensor' and 'float'
```

F.4 张量数组

tf.TensorArray表示张量的一个列表。这在包含循环的动态模型中很方便，可以累加结果并稍后计算一些统计信息。你可以在数组中的任何位置读取或写入张量：

```
array = tf.TensorArray(dtype=tf.float32, size=3)
array = array.write(0, tf.constant([1., 2.]))
array = array.write(1, tf.constant([3., 10.]))
array = array.write(2, tf.constant([5., 7.]))
tensor1 = array.read(1) # => returns (and pops!) tf.constant([3., 10.])
```

请注意，读取一个元素会从数组中弹出该元素，并用相同形状的零张量替换它。

 当你写入数组时，必须将输出分配回数组，如代码示例所示。如果不这样，尽管你的代码在eager模式下可以正常工作，但在图模式下会中断（这些模式在第12章中介绍了）。

创建TensorArray时，你必须提供其size，但在图模式下除外。另外，你可以不设置size，而是设置dynamic_size=True，但这会影响性

能。因此，如果你事先知道size，则应该进行设置。你还必须指定dtype，所有元素的形状必须与写入数组的第一个元素相同。

你可以通过调用stack（）方法将所有元素堆叠到一个常规张量中：

```
>>> array.stack()
<tf.Tensor: id=2110875, shape=(3, 2), dtype=float32, numpy=
array([[1., 2.],
       [0., 0.],
       [5., 7.]], dtype=float32)>
```

F.5 集合

TensorFlow支持整数或字符串集合（但不支持浮点数）。它使用常规张量表示它们。例如，集合{1，5，9}仅表示为张量[[1，5，9]]。请注意，张量必须至少具有两个维度，并且集合必须位于最后一个维度。例如，[[1，5，9]，[2，5，11]]是一个张量，包含两个独立的集合：{1，5，9}和{2，5，11}。如果某些集合比其他集合短，则必须使用填充值来填充它们（默认为0，但可以使用你喜欢的任何其他值）。

tf.sets包包含几个操作集合的函数。例如，让我们创建两个集合并计算它们的并集（结果是稀疏张量，因此我们调用to_dense（）来显示它）：

```
>>> a = tf.constant([[1, 5, 9]])
>>> b = tf.constant([[5, 6, 9, 11]])
>>> u = tf.sets.union(a, b)
>>> u
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x132b60d30>
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11]], dtype=int32)>
```

你还可以同时计算多对集合的并集：

```
>>> a = tf.constant([[1, 5, 9], [10, 0, 0]])
>>> b = tf.constant([[5, 6, 9, 11], [13, 0, 0, 0, 0]])
>>> u = tf.sets.union(a, b)
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
                               [ 0, 10, 13,  0,  0]], dtype=int32)>
```

如果你希望使用不同的填充值，则在调用to_dense（）时必须设置default_value：

```
>>> tf.sparse.to_dense(u, default_value=-1)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
                               [ 0, 10, 13, -1, -1]], dtype=int32)>
```

 默认的default_value为0，因此在处理字符串集合时，必须设置default_value（例如，设置为空字符串）。

tf.sets中可用的其他函数包括difference（）、intersection（）和size（），这是不言自明的。如果要检查集合是否包含某些给定值，可以计算该集合和这些值的交集。如果要向集合中添加一些值，则可以计算集合和值的并集。

F.6　队列

队列是一种数据结构，你可以将数据记录推送到该数据结构中，然后再将其取出。TensorFlow在tf.queue包中实现了几种类型的队列。在实现有效的数据加载和预处理流水线时，它们曾经非常重要，但是tf.data API实际上使它们没有用处（在某些罕见情况下可能除

外），因为它使用起来更简单并且提供了你需要建立有效的流水线的所有工具。为了完整起见，让我们快速看一下它们。

最简单的队列是先进先出（FIFO）队列。要构建它，你需要指定它可以包含的最大记录数。此外，每个记录都是张量的元组，因此你必须指定每个张量的类型以及可选的形状。例如，以下代码示例创建一个FIFO队列，该队列最多包含三个记录，每个记录包含一个具有32位整数和一个字符串的元组。然后，它向其中推送两个记录，查看其大小（此时为2），之后取出一条记录：

```
>>> q = tf.queue.FIFOQueue(3, [tf.int32, tf.string], shapes=[(), ()])
>>> q.enqueue([10, b"windy"])
>>> q.enqueue([15, b"sunny"])
>>> q.size()
<tf.Tensor: id=62, shape=(), dtype=int32, numpy=2>
>>> q.dequeue()
[<tf.Tensor: id=6, shape=(), dtype=int32, numpy=10>,
 <tf.Tensor: id=7, shape=(), dtype=string, numpy=b'windy'>]
```

也可以一次使多个记录入队列和出队列（后者在创建队列时需要指定形状）：

```
>>> q.enqueue_many([[13, 16], [b'cloudy', b'rainy']])
>>> q.dequeue_many(3)
[<tf.Tensor: [...] numpy=array([15, 13, 16], dtype=int32)>,
 <tf.Tensor: [...] numpy=array([b'sunny', b'cloudy', b'rainy'], dtype= object)>]
```

其他队列类型包括：PaddingFIFOQueue与FIFOQueue相同，但其dequeue_many（）方法支持不同形状的多个记录出队列。它会自动填充最短的记录以确保批次中的所有记录具有相同的形状。

PriorityQueue

以优先顺序使记录出队的队列。优先级必须是包含在每个记录的第一个元素中的64位整数。令人惊讶的是，优先级较低的记录将首先出队。具有相同优先级的记录将按FIFO顺序出队。

RandomShuffleQueue

一个队列，其记录以随机顺序出队。这对于在tf.data存在之前实现混洗缓冲区很有用。

如果队列已满，并且你尝试使另一个记录入队，则enqueue*（）方法将冻结，直到一个线程从队列中取出一个记录。同样地，如果队列为空，并且你尝试使记录出队，则dequeue*（）方法将冻结，直到另一个线程将记录推入队列。

[1] 如果你不熟悉Unicode代码点，请查看https://homl.info/unicode。

附录G　TensorFlow图

在本附录中，我们将探讨由TF函数生成的图（见第12章）。

G.1　TF函数和具体函数

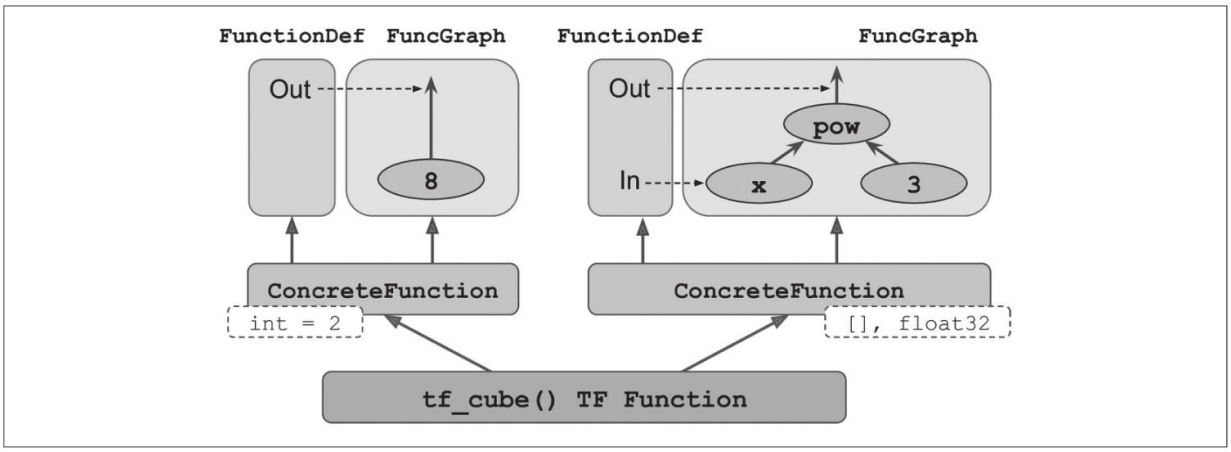
TF函数是多态的，这意味着它们支持不同类型（和形状）的输入。例如，考虑以下tf_cube（）函数：

```
@tf.function
def tf_cube(x):
    return x ** 3
```

每次使用一个输入类型或形状的新组合来调用TF函数时，它都会生成一个新的具体函数，并带有针对该特定组合的专用图形。参数类型和形状的这种组合称为输入签名。如果你使用之前已经看到过的输入签名来调用TF函数，它将重用之前生成的具体函数。例如，如果你调用tf_cube（tf.constant（3.0））， 则TF函数将重用与tf_cube（tf.constant（2.0））（用于float32标量张量）相同的具体函数。但是，如果调用tf_cube（tf.constant（[2.0]））或者tf_cube（tf.constant（[3.0]））（对于形状为[1]的float32张量），或者tf_cube（tf.constant（[[1.0，2.0]，[3.0，4.0]]））（对于形状为[2，2]的float32张量），它将生成一个新的具体函数。通过调用TF函数的get_concrete_function（）方法，可以获得特定输入组合的具体函数。然后可以像常规函数一样调用它，但是它仅支持一个输入签名（在本示例中为float32标量张量）：

```
>>> concrete_function = tf_cube.get_concrete_function(tf.constant(2.0))
>>> concrete_function
<tensorflow.python.eager.function.ConcreteFunction at 0x155c29240>
>>> concrete_function(tf.constant(2.0))
<tf.Tensor: id=19068249, shape=(), dtype=float32, numpy=8.0>
```

图G-1显示了tf_cube（）TF函数，在我们分别调用了tf_cube（2）和tf_cube（tf.constant（2.0））之后，生成了两个具体函数，每个签名都有一个，每个函数都有自己优化的函数图（FuncGraph），以及它自己的函数定义（FunctionDef）。函数定义指向图形中与函数的输入和输出相对应的部分。在每个FuncGraph中，节点（椭圆形）表示操作（例如x等参数的幂、常数或占位符），而边（操作之间的实线箭头）表示将流过图形的张量。左侧的具体函数专用于x=2，因此TensorFlow设法简化了它，使其始终仅输出8（请注意，函数定义甚至没有输入）。右边的具体函数专用于float32标量张量，因此无法简化。如果调用tf_cube（tf.constant（5.0））， 则将调用第二个具体函数，x的占位符操作将输出5.0，然后幂运算将计算5.0**3，因此输出将为125.0。



图G-1：tf_cube（）TF函数及其具体函数和它们的函数图

这些图中的张量是符号张量，这意味着它们没有实际值，只是数据类型、形状和名称。它们表示一旦将实际值输入到占位符x并执行图形，张量将流过图形。使用符号张量可以提前指定如何连接操作，给定其输入的数据类型和形状，它们还允许TensorFlow递归推断所有张量的数据类型和形状。

现在让我们继续看看内幕，看看如何访问函数定义和函数图，以及探索图的操作和张量。

G.2 探索函数定义和图

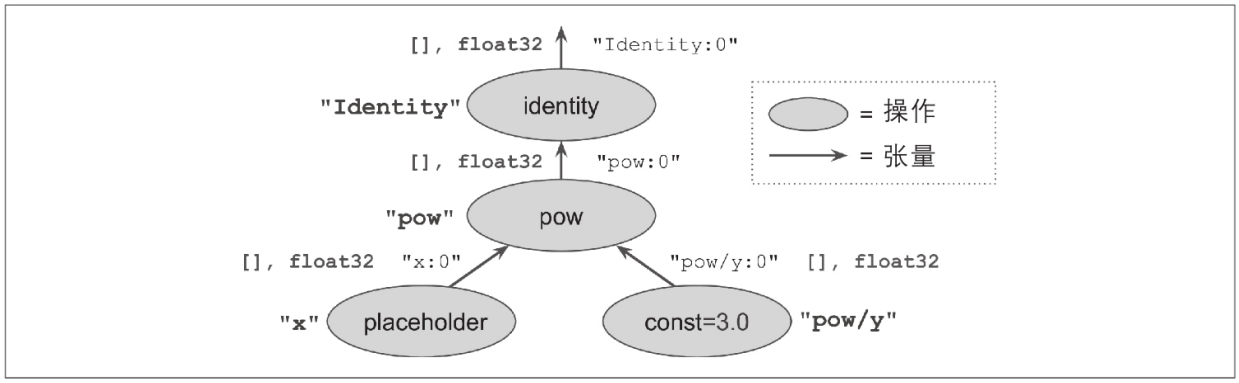
你可以使用graph属性来访问具体函数的计算图，并调用该图的get_operations（）方法来得到其操作列表：

```
>>> concrete_function.graph
<tensorflow.python.framework.func_graph.FuncGraph at 0x14db5ef98>
>>> ops = concrete_function.graph.get_operations()
>>> ops
[<tf.Operation 'x' type=Placeholder>,
 <tf.Operation 'pow/y' type=Const>,
 <tf.Operation 'pow' type=Pow>,
 <tf.Operation 'Identity' type=Identity>]
```

在此示例中，第一个操作代表输入参数x（称为占位符），第二个“操作”代表常数3，第三个操作代表幂运算（**），最后一个操作代表此函数的输出（这是一个标识操作，这意味着它仅会复制加法操作的输出[1]）。每个操作都有一个输入和输出张量列表，你可以使用该操作的inputs和outputs属性轻松访问它们。例如，让我们得到幂操作的输入和输出列表：

```
>>> pow_op = ops[2]
>>> list(pow_op.inputs)
[<tf.Tensor 'x:0' shape=() dtype=float32>,
 <tf.Tensor 'pow/y:0' shape=() dtype=float32>]
>>> pow_op.outputs
[<tf.Tensor 'pow:0' shape=() dtype=float32>]
```

该计算图如图G-2所示。



图G-2：计算图示例

请注意，每个操作都有一个名字。它默认为操作的名称（例如"pow"），但是你可以在调用操作时手动定义它（例如tf.pow（x，3，name="other_name"））。如果名称已经存在，TensorFlow会自动添加一个唯一索引（例如"pow_1""pow_2"等）。每个张量还具有唯一的名称：它始终是输出此张量的操作的名称，如果是操作的第一个输出，则加：0，如果是第二个输出，则加：1，以此类推。你可以使用图的get_operation_by_name（）或get_tensor_by_name（）方法按名称来得到操作或张量：

```
>>> concrete_function.graph.get_operation_by_name('x')
<tf.Operation 'x' type=Placeholder>
>>> concrete_function.graph.get_tensor_by_name('Identity:0')
<tf.Tensor 'Identity:0' shape=() dtype=float32>
```

具体函数还包含函数定义（表示为协议缓冲区[2]），其中包含函数的签名。通过此签名，具体函数可以知道哪些占位符与输入值会被提供，以及返回哪些张量：

```
>>> concrete_function.function_def.signature
name: "__inference_cube_19068241"
input_arg {
  name: "x"
  type: DT_FLOAT
}
output_arg {
```

```
  name: "identity"
  type: DT_FLOAT
}
```

现在让我们更仔细地研究跟踪。

G.3 仔细查看跟踪

让我们调整tf_cube（）函数来打印其输入：

```
@tf.function
def tf_cube(x):
    print("x =", x)
    return x ** 3
```

现在让我们调用它：

```
>>> result = tf_cube(tf.constant(2.0))
x = Tensor("x:0", shape=(), dtype=float32)
>>> result
<tf.Tensor: id=19068290, shape=(), dtype=float32, numpy=8.0>
```

result看起来不错，但看看打印的内容：x是一个符号张量！它有形状和数据类型，但没有值。另外，它还有一个名称（"x：0"）。这是因为print（）函数不是TensorFlow操作，所以它仅仅在跟踪Python函数时运行，该函数在图模式下发生，并且用符号张量替换参数（相同的类型和形状，但没有值）。由于未将print（）函数捕获到图中，因此下一次我们使用float32标量张量调用tf_cube（）时，不会打印任何内容：

```
>>> result = tf_cube(tf.constant(3.0))
>>> result = tf_cube(tf.constant(4.0))
```

但是，如果我们使用具有不同类型或形状的张量或使用新的Python值调用tf_cube（），则会再次跟踪该函数，因此print（）函数会被调用：

```
>>> result = tf_cube(2) # new Python value: trace!
x = 2
>>> result = tf_cube(3) # new Python value: trace!
x = 3
>>> result = tf_cube(tf.constant([[1., 2.]])) # New shape: trace!
x = Tensor("x:0", shape=(1, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[3., 4.], [5., 6.]])) # New shape: trace!
x = Tensor("x:0", shape=(None, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[7., 8.], [9., 10.]])) # Same shape: no trace
```

 如果你的函数有Python副作用（例如，它将一些日志保存到磁盘），请注意，只有在跟踪函数时（即每次使用新的输入签名调用TF函数时），此代码才会运行。最好假定每次调用TF函数时都可以跟踪（或不跟踪）该函数。在某些情况下，你可能希望将TF函数限制为特定的输入签名。例如，假设你知道你将只调用带有28×28个像素图像批次的TF函数，但是这些批次具有不同的大小。你可能不希望TensorFlow为每个批次生成不同的具体函数，也不希望依靠它自己找出何时使用None的情况。在这种情况下，可以这样指定输入签名：

```
@tf.function(input_signature=[tf.TensorSpec([None, 28, 28], tf.float32)])
def shrink(images):
    return images[:, ::2, ::2] # drop half the rows and columns
```

此TF函数将接受任何形状为[*，28，28]的float32张量，并且每次都将重用相同的具体函数：

```
img_batch_1 = tf.random.uniform(shape=[100, 28, 28])
img_batch_2 = tf.random.uniform(shape=[50, 28, 28])
```

```
preprocessed_images = shrink(img_batch_1) # Works fine. Traces the function.
preprocessed_images = shrink(img_batch_2) # Works fine. Same concrete function.
```

但是，如果你尝试使用Python值或非预期的数据类型或形状的张量调用此TF函数，则会出现异常：

```
img_batch_3 = tf.random.uniform(shape=[2, 2, 2])
preprocessed_images = shrink(img_batch_3)  # ValueError! Unexpected signature.
```

G.4 使用AutoGraph捕获控制流

如果你的函数包含一个简单的for循环，你期望会发生什么？例如，让我们编写一个函数，将其输入加10，只需将1加10次：

```
@tf.function
def add_10(x):
    for i in range(10):
        x += 1
    return x
```

它工作正常，但是当我们查看其图形时，发现它不包含循环：它仅包含10个加法运算！

```
>>> add_10(tf.constant(0))
<tf.Tensor: id=19280066, shape=(), dtype=int32, numpy=10>
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'add' type=Add>, [...],
 <tf.Operation 'add_1' type=Add>, [...],
 <tf.Operation 'add_2' type=Add>, [...],
 [...]
 <tf.Operation 'add_9' type=Add>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

这实际上是有道理的：当跟踪函数时，循环运行了10次，因此x+=1操作运行了10次，并且由于它处于图模式，在图中记录了10次该操作。你可以将此for循环视为创建图形时展开的“静态”循环。

如果要让图包含一个“动态”循环（即执行该图形时运行的循环），则可以使用tf.while_loop（）操作来手动创建一个，但这不是很直观（有关示例，请参见第12章notebook的“Using AutoGraph to Capture Control Flow”部分）。相反，使用TensorFlow的AutoGraph特性要简单得多，该功能在第12章讨论过。AutoGraph实际上是默认激活的（如果你需要将其关闭，则可以将autograph=False传递给tf.function（））。因此，如果激活了，为什么它没有在add_10（）函数中捕获for循环呢？好吧，它只捕获在tf.range（）而不是range（）上迭代的循环。以下是你的选择：

- 如果使用ra nge（），则for循环将是静态的，这意味着仅在跟踪函数时才会执行。如我们所见，对于每次迭代，循环将“展开”为一组操作。

- 如果使用tf.ra nge（），则循环将是动态的，这意味着它将包含在图本身中（但不会在跟踪过程中运行）。

让我们看一下在add_10（）函数中将range（）替换为tf.range（）时生成的图形：

```
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'range' type=Range>, [...],
 <tf.Operation 'while' type=While>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

如你所见，该图现在包含While循环操作，就像调用了tf.while_loop（）函数一样。

G.5 在TF函数中处理变量和其他资源

在TensorFlow中，变量和其他有状态对象（例如队列或数据集）称为资源。TF函数会特别谨慎地对待它们：任何读取或更新资源的操作都被认为是有状态的，并且TF函数确保有状态操作按其出现的顺序执行（与无状态操作相反，无状态操作可能并行运行，因此它们无法保证执行顺序）。此外，当你将资源作为参数传递给TF函数时，它会通过引用进行传递，因此函数可以对其进行修改。例如：

```
counter = tf.Variable(0)

@tf.function
def increment(counter, c=1):
    return counter.assign_add(c)

increment(counter) # counter is now equal to 1
increment(counter) # counter is now equal to 2
```

如果你浏览一下函数定义，则第一个参数被标记为资源：

```
>>> function_def = increment.get_concrete_function(counter).function_def
>>> function_def.signature.input_arg[0]
name: "counter"
type: DT_RESOURCE
```

也可以在函数外部使用tf.Variable定义，而无须将其作为参数显式传递：

```
counter = tf.Variable(0)

@tf.function
def increment(c=1):
    return counter.assign_add(c)
```

TF函数会将其视为隐式的第一个参数，因此实际上它会得到相同的签名（参数名称除外）。但是，使用全局变量会变得混乱，因此通常应将变量（和其他资源）包装在类中。好消息是@tf.function也可以很好地与方法配合使用：

```
class Counter:
    def __init__(self):
        self.counter = tf.Variable(0)

    @tf.function
    def increment(self, c=1):
        return self.counter.assign_add(c)
```

 请勿将=、+=、-=或任何其他带有TF变量的Python赋值运算符一起使用。相反，你必须使用assign（）、assign_add（）或assign_sub（）方法。如果尝试使用Python赋值运算符，则会在调用该方法时出现异常。

当然，这种面向对象方法的一个很好的示例是tf.keras。让我们看看如何在tf.keras中使用TF函数。

G.6　将TF函数与tf.keras一起使用（或不使用）

默认情况下，你与tf.keras一起使用的任何自定义函数、层或模型都将自动转换为TF函数。你根本不需要做任何事情！但是，在某些情况下，你可能想取消此自动转换，例如，如果你的自定义代码无法转换为TF函数，或者你仅想调试代码，在eager模式下这会容易得多。为此，你可以在创建模型或其他任何层时简单地传递dynamic=True：

```
model = MyModel(dynamic=True)
```

如果你的自定义模型或层始终是动态的，则可以改为使用dynamic=True来调用基类的构造函数：

```
class MyLayer(keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__(dynamic=True, **kwargs)
        [...]
```

或者，可以在调用compile（）方法时传递run_eagerly=True：

```
model.compile(loss=my_mse, optimizer="nadam", metrics=[my_mae],
              run_eagerly=True)
```

至此你已经知道TF函数如何处理多态（具有多个具体函数），如何使用AutoGraph和跟踪自动生成图，图是什么样，如何探索其符号运算和张量，如何处理变量和资源以及如何使用带有tf.keras的TF函数。

[1] 你可以放心地忽略它——只是出于技术原因，以确保TF函数不会泄漏内部结构。
[2] 第13章中讨论了一种流行的二进制格式。

作者介绍

Aurélien Géron是一名机器学习顾问和讲师。他是前Google员工，于2013年至2016年领导YouTube的视频分类团队。他是多家公司的创始人兼CTO，这些公司包括：Wifirst（法国领先的无线ISP）、Polyconseil（一家专注于电信、媒体和战略的咨询公司）、Kiwisoft（一家专注于机器学习和数据隐私的咨询公司）。

在此之前，他曾在多个领域担任工程师：金融（JP Morgan和SociétéGénérale）、国防（加拿大的DOD）和医疗保健（输血）。他还出版了一些技术书籍（有关C++、WiFi和互联网架构），并在法国一家工程院校教授计算机科学。

一些有趣的事实：他教他的三个孩子用手指来进行二进制计数（最高1023），在进入软件工程行业之前，他学习了微生物学和进化遗传学，并且他的降落伞在第二跳时没有打开。

封面介绍

本书封面的动物是火蜥蜴（Salamandra salamandra），这是一种在欧洲大部分地区都能发现的两栖动物。它有黑色而光滑的皮肤，头部和背部均有很大的黄色斑点，这说明它的身上存在生物碱毒素。这可能是这种两栖动物名字的来源：与这些毒素接触（它们也可以近距离喷射）会引起抽搐和过度换气。火蜥蜴有毒而湿润的皮肤，使人们误以为这些生物不仅可以在火中生存，还可以将火扑灭。

火蜥蜴生活在阴凉的森林中，躲在潮湿的缝隙里、靠近水池或其他便于繁殖的淡水水体附近的原木下。尽管它们大部分时间都生活在陆地上，但它们是在水中繁殖后代的。它们主要以昆虫、蜘蛛、蛞蝓和蠕虫为食。火蜥蜴可以长到一英尺（约0.305米）长，人工饲养的火蜥蜴可以活到50岁。

火蜥蜴的数量由于其森林栖息地的破坏和宠物贸易的捕获而减少了，但是它们面临的最大威胁是其透湿性皮肤对污染物和微生物的敏感性。自2014年以来，由于感染了真菌，它们已在荷兰和比利时的部分地区灭绝。

O’Reilly封面上的许多动物都濒临灭绝，所有这些动物对世界都很重要。

封面插图由Karen Montgomery基于版画Wood’s Illustrated Natural History绘制而成。