

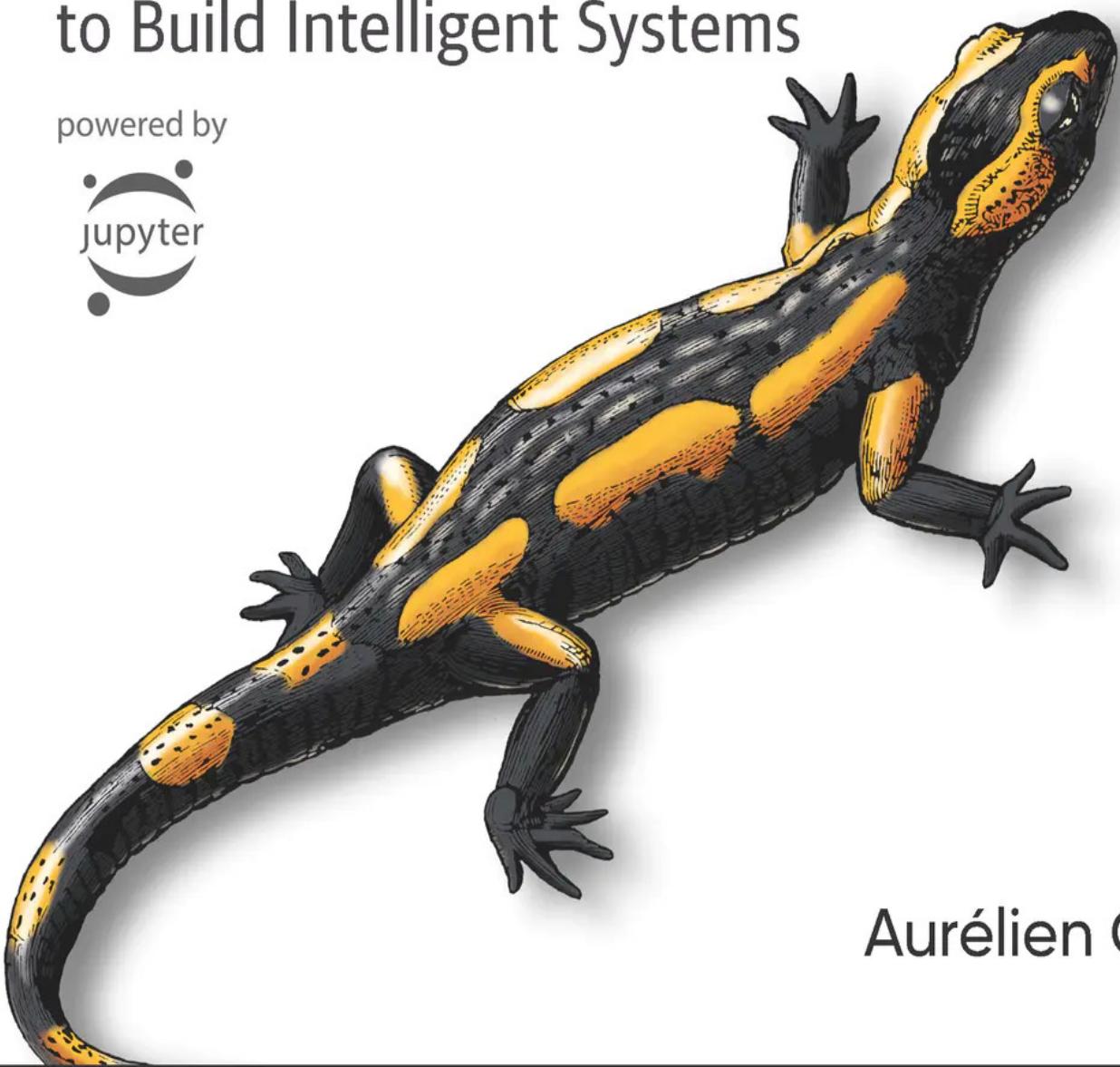
O'REILLY®

2nd Edition  
Updated for  
TensorFlow 2

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

Concepts, Tools, and Techniques  
to Build Intelligent Systems

powered by



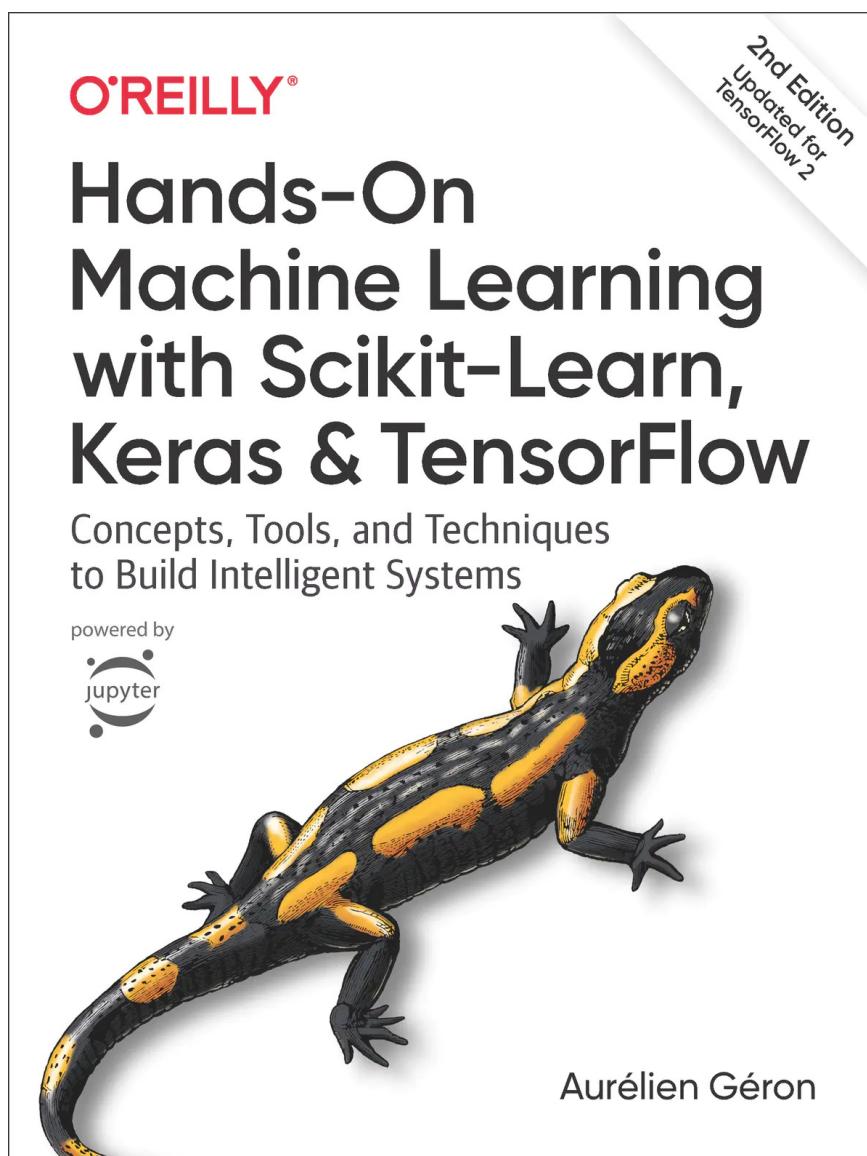
Aurélien Géron

## Table of Contents

---

Introduction	1.1
零、前言	1.2
一、机器学习概览	1.3
二、端到端的机器学习项目	1.4
三、分类	1.5
四、训练模型	1.6
五、支持向量机	1.7
六、决策树	1.8
七、集成学习和随机森林	1.9
八、降维	1.10
十、使用 Keras 搭建人工神经网络	1.11
十一、训练深度神经网络	1.12
十二、使用 TensorFlow 自定义模型并训练	1.13
十三、使用 TensorFlow 加载和预处理数据	1.14
十四、使用卷积神经网络实现深度计算机视觉	1.15
十五、使用 RNN 和 CNN 处理序列	1.16
十六、使用 RNN 和注意力机制进行自然语言处理	1.17
十七、使用自编码器和 GAN 做表征学习和生成式学习	1.18
十八、强化学习	1.19
十九、规模化训练和部署 TensorFlow 模型	1.20

# Sklearn 与 TensorFlow 机器学习实用指南 第二版



协议 : CC BY-NC-SA 4.0

懦夫才用磁带备份，真男人把重要的东西传到 FTP，然后世界会帮他备份。  
——林纳斯·托瓦兹

- [在线阅读](#)
- [ApacheCN 面试求职交流群 724187166](#)
- [ApacheCN 学习资源](#)
- [利用 Python 进行数据分析 第二版](#)

编译

```
npm install -g gitbook-cli          # 安装 gitbook  
gitbook fetch 3.2.3                 # 安装 gitbook 子版本  
gitbook install                     # 安装必要的插件  
gitbook <build|pdf|epub|mobi>      # 编译 HTML/PDF/EPUB/MOBI
```

## 下载

### Docker

```
docker pull apachecn0/hands-on-ml-2e-zh  
docker run -tid -p <port>:80 apachecn0/hands-on-ml-2e-zh  
# 访问 http://localhost:{port} 查看文档
```

### PYPI

```
pip install hands-on-ml-2e-zh  
hands-on-ml-2e-zh <port>  
# 访问 http://localhost:{port} 查看文档
```

### NPM

```
npm install -g handson-ml-2e-zh  
handson-ml-2e-zh <port>  
# 访问 http://localhost:{port} 查看文档
```

# 零、前言

译者：[@小瑶](#)

校对者：[@小瑶](#)

## 1、机器学习海啸

2006 年，Geoffrey Hinton 等人发表了一篇论文，展示了如何训练能够识别具有最新精度 ( $> 98\%$ ) 的手写数字的深度神经网络。他们称这种技术为“Deep Learning”。当时，深度神经网络的训练被广泛认为是不可能的，并且大多数研究人员自 20 世纪 90 年代以来就放弃了这个想法。这篇论文重新激起了科学界的兴趣，不久之后，许多新发表的论文表明，深度学习不仅是可能的，而且能够取得其他的 Machine Learning 技术都难以匹配的令人兴奋的成就（借助巨大的计算能力和大量的数据）。这种热情很快扩展到机器学习的许多的其他领域。

Deep Learning 快速发展的 10 年间和机器学习已经征服了这个行业：它现在成为了当今高科技产品中的许多黑科技的核心，比如，为您的网络搜索结果排名，为智能手机的语音识别提供支持，为您推荐您喜欢的视频，在围棋游戏中击败世界冠军。在你知道之前，它都可能会驾驶您的汽车。

## 2、您项目中的机器学习

现在你是不是对机器学习感到兴奋，并且很乐意加入到这个阵营中？也许你希望给自己制造的机器人赋予一个自己的大脑？让它可以面部识别？还是学会到处走走？

也许你的公司有大量的数据（用户日志，财务数据，生产数据，机器传感器数据，热线统计数据，人力资源报告等），如果你知道在哪方面观察，你可能会发现一些隐藏着的瑰宝。例如：

- 细分客户，为每个团队找到最佳的营销策略
- 根据类似客户购买的产品为每个客户推荐产品
- 检测哪些交易可能是欺诈行为
- 预测下一年的收入
- 更多应用

无论什么原因，你决定开始学习机器学习，并在你的项目中实施，这是一个好主意！

## 3、目标和方法

本书假定你对机器学习几乎一无所知。它的目标是给你实际实现能够从数据中学习的程序所需的概念，直觉和工具。

我们将介绍大量的技术，从最简单的和最常用的（如线性回归）到一些定期赢得比赛的深度学习技术。

我们将使用现成的 Python 框架，而不是实现我们自己的每个算法的玩具版本：

- Scikit-learn 非常易于使用，并且实现了许多有效的机器学习算法，因此它为学习机器学习提供了一个很好的切入点。
- TensorFlow 是使用数据流图进行分布式数值计算的更复杂的库。它通过在潜在的数千个多 GPU 服务器上分布式计算，可以高效地训练和运行非常大的神经网络。TensorFlow 是被 Google 创造的，支持其大型机器学习应用程序。于 2015 年 11 月开源。

本书倾向于实际操作的方法，通过具体的实例和一点理论来增加对机器学习的直观理解。虽然你可以在不拿笔记本电脑的情况下阅读此书，但是我们强烈建议你通过 <https://github.com/ageron/handson-ml> 在线实现 Jupyter 笔记本上的代码示例。

## 4、准备条件

本书假定您有一些 Python 编程经验，并且比较熟悉 Python 的主要科学库，特别是 NumPy, Pandas 和 Matplotlib。

另外，如果你关心的是底层实现/原理，你应该对大学水平的数学（微积分，线性代数，概率和统计学）有一些了解。

如果你还不了解 Python, <http://learnpython.org/> 是你学习使用 Python 的好地方。python.org 官方教程也是相当不错的。

如果你从未使用过 Jupyter，第 2 章将指导你完成安装和基本操作：它是你工具箱中的一个很好的工具。

如果你不熟悉 Python 的科学库，提供的一些 Jupyter 笔记本包括了一些教程。还有一个线性代数的快速数学教程。

## 5、路线图

这本书分为两个部分。

第一部分，机器学习的基础知识，涵盖以下主题：

- 什么是机器学习？它被试图用来解决什么问题？机器学习系统的主要类别和基本概念是什么？
- 典型的机器学习项目中的主要步骤。
- 通过拟合数据来学习模型。
- 优化成本函数 (cost function)。
- 处理，清洗和准备数据。
- 选择和设计特征。
- 使用交叉验证选择一个模型并调整超参数。
- 机器学习的主要挑战，特别是欠拟合和过拟合（偏差和方差权衡）。
- 对训练数据进行降维以对抗 the curse of dimensionality（维度诅咒）
- 最常见的学习算法：线性和多项式回归，Logistic 回归，k-最近邻，支持向量机，决策树，随机森林和集成方法。

第二部分，神经网络和深度学习，包括以下主题：

- 什么是神经网络？它们有啥优势？
- 使用 TensorFlow 构建和训练神经网络。

- 最重要的神经网络架构：前馈神经网络，卷积网络，递归网络，长期短期记忆网络（LSTM）和自动编码器。
- 训练深度神经网络的技巧。
- 对于大数据集缩放神经网络。
- 强化学习。

第一部分主要基于 scikit-learn，而第二部分则使用 TensorFlow。

注意：不要太急于深入学习到核心知识：深度学习无疑是机器学习中最令人兴奋的领域之一，但是你应该首先掌握基础知识。而且，大多数问题可以用较简单的技术很好地解决（而不需要深度学习），比如随机森林和集成方法（我们会在第一部分进行讨论）。如果你拥有足够的数据，计算能力和耐心，深度学习是最适合复杂的问题的，如图像识别，语音识别或自然语言处理。

## 6、其他资源

有许多资源可用于了解机器学习。Andrew Ng 在 Coursera 上的 [ML 课程](#) 和 Geoffrey Hinton 关于 [神经网络和深度学习](#) 的课程都是非常棒的，尽管这些课程需要大量的时间投入（大概是几个月）。

还有许多关于机器学习的比较有趣的网站，当然还包括 scikit-learn 出色的 [用户指南](#)。你可能会喜欢上 [Dataquest](#)，它提供了一个非常好的交互式教程，还有 ML 博客，比如那些在 [Quora](#) 上列出来的博客。最后，[Deep Learning 网站](#)有一个很好的资源列表来学习更多。

当然，还有很多关于机器学习的其他介绍性书籍，特别是：

- Joel Grus, Data Science from Scratch (O'Reilly). 这本书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（从名字上看就可以知道，从头开始）。
- Stephen Marsland, Machine Learning: An Algorithmic Perspective (Chapman and Hall). 这本书对机器学习有一个很好的介绍，涵盖了广泛的主题，Python 中的代码示例（也是从零开始，但是使用 NumPy）。
- Sebastian Raschka, Python Machine Learning (Packt Publishing). 本书也对机器学习有一个很好的介绍，但是利用了 Python 的开源库（Pylearn 2 和 Theano）。
- Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin, Learning from Data (MLBook). 对 ML 有一个相对理论化的介绍，这本书提供了比较深刻的见解，特别是 bias/variance tradeoff（偏差/方差 权衡）（见第 4 章）。
- Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach, 3rd Edition (Pearson). 这是一本很好的（并且很大）的书，涵盖了包括机器学习在内的大量主题。这有助于更加深刻地理解 ML。

最后，一个很好的学习方法就是加入 ML 竞赛网站，例如 kaggle.com，这样可以让你在现实世界的问题上锻炼自己的技能，并从一些最好的 ML 专业人士那里获得帮助和见解。

## 7、本书中的一些约定

本书使用以下印刷约定：

- 斜体 —— 指示新术语，网址，电子邮件地址，文件名和文件扩展名。
- 等宽 —— 用于程序清单，以及段落内用于引用程序元素，如变量或函数名称，数据库，数据类型，环境变量，语句和关键字。
- 等宽粗体 —— 显示应由用户逐字输入的命令或其他文本。
- 等宽斜体 —— 显示应由用户提供或由上下文确定的值替换的文本。
- 小松鼠图标 —— 此元素表示一个小提示或建议。
- 小乌鸦图标 —— 此元素表示一个普通的说明。
- 小蝎子图标 —— 此元素表示一个警告和注意。

## 8、使用代码示例

补充材料（代码示例，练习题等）可以从 <https://github.com/ageron/handson-ml> 下载。

这本书是为了帮助你完成工作。一般来说，如果本书提供了示例代码，则可以在程序和文档中使用它。除非你复制了大部分代码，否则你无需联系我们获得许可。例如，编写使用本书中几个代码块的程序不需要许可。销售或者分发 O'Reilly 书籍的 CD-ROM 例子需要获得许可。

通过引用本书和使用示例代码来回答问题并不需要获得许可。将大量来自本书的示例代码整合到产品文档中并不需要获得许可。

我们感谢，但是并不要求，贡献。贡献通常包括标题，作者，出版商和 ISBN。例如：“Hands-On Machine Learning with Scikit-Learn and TensorFlow by Aurélien Géron (O'Reilly). Copyright 2017 Aurélien Géron, 978-1-491-96229-9.”

如果您觉得您对代码示例的使用超出了合理使用范围或上述权限，请随时联系我们：[permissions@oreilly.com](mailto:permissions@oreilly.com)。

## 9、O'Reilly Safari

Safari（以前被称为 Safari Books Online）是一个针对企业，政府，教育工作者和个人的基于会员的培训和参考平台。

会员可以访问 250 多家发布商的数千本图书，培训视频，学习路径，互动教程和策划播放列表，其中包括 O'Reilly Media，哈佛商业评论，Prentice Hall 专业人员，Addison-Wesley 专业人员，Microsoft Press，Sams，Que，Peachpit Press，Adobe，Focal Press，Cisco Press 等。想要了解更多信息，请访问 <http://oreilly.com/safari>。

## 10、如何联系我们

请向出版商发表有关本书的评论和问题：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (在美国或者加拿大)

707-829-0515 (国际或地区)

707-829-0104 (传真)

我们有一个这本书的网页，在这里我们列出了勘误表，例子和任何额外的信息。你可以访问这个网页 <http://bit.ly/hands-on-machine-learning-with-scikit-learn-and-tensorflow>

要评论或者询问有关本书的技术问题，请发送电子邮件到  
[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)。

有关我们的书籍，课程，会议和新闻的更多信息，请访问我们的网站  
<http://www.oreilly.com>。

在 facebook 上找到我们：<http://facebook.com/oreilly>

在 Twitter 上关注我们：<http://twitter.com/oreillymedia>

在 Youtube 上观看我们的视频：<http://www.youtube.com/oreillymedia>

## 11、致谢

我要感谢我的 Google 同事，特别是 Youtube 视频分类小组，教给我很多关于机器学习的知识。没有他们，我永远无法开始这个项目。特别感谢我的个人 ML 专家：Clément Courbet, Julien Dubois, Mathias Kende, Daniel Kitachewsky, James Pack, Alexander Pak, Anosh Raj, Vitor Sessak, Wiktor Tomczak, Ingrid von Glehn, Rich Washington, 以及 Youtube Paris 的所有人。

我非常感谢所有那些从繁忙的生活中抽出时间来仔细阅读我的书的人。感谢 Pete Warden 回答了我所有的 TensorFlow 的问题，回顾第二部分，提供了许多有趣的见解，当然也成为了 TensorFlow 核心团队的一员。你一定想要看看他的[博客](#)！非常感谢 Lukas Biewald 对第二部分的非常全面的审查：他毫不留情地尝试了所有的代码（并且发现了一些错误），做出了许多伟大的建议，而且他的热情是具有感染力的。你应该看看他的博客，和他的超酷的机器人！感谢 Justin Francis，他也非常全面地审查了第二部分，特别是在第 16 章提到了错误并提供了很好的见解。你可以在 TensorFlow 上看到他的帖子！

也非常感谢 David Andrzejewski，他审查了第一部分，提供了非常有用的反馈意见，确定了不明确的部分并提出了改进建议。查看一下他的网页吧。感谢 Grégoire Mesnil，他审查了第二部分，并提供了非常有趣的关于神经网络的实用建议。感谢 Eddy Hung, Salim Sémaoune, Karim Matrah, Ingrid von Glehn, Iain Smears, 和 Vincent Guilbeau 对第一部分的审查和建议。我还要感谢我的岳父，前数学老师 Michel Tessier，现在是 Anton Chekhov 的一名优秀翻译，帮助我在本书中提供了一些非常好的数学和符号，并且审查了线性代数 Jupyter 笔记本。

当然，对我亲爱的弟弟说一个巨大的“谢谢”，他测试了每一行代码，几乎在每个部分都提供了反馈，并鼓励我从第一行到最后一行。爱你，我的兄弟。

非常感谢 O'Reilly 出色的员工，特别是 Nicole Tache，他给出了深刻的反馈，并且总是开朗，鼓舞和乐于助人的。还要感谢 Marie Beaugureau, Ben Lorica, Mike Loukides, 和 Laurel Ruma 相信这个项目并帮助我确定其范围。感谢 Matt Hacker 和所有的 Atlasteam 回答了关于格式化，asciidoc 和 LaTeX 的所有技术团队问题，也感谢 Rachel Monaghan, Nick Adams, 和所有的制作团队进行了最终的审查和数百次的修正。

最后但也很重要的一点，我非常感谢我的爱妻 Emmanuelle 和三个非常棒的孩子，Alexandre, Rémi, 和 Gabrielle，在这本书中写了很多，问了很多问题（谁说不能教 7 岁的孩子神经网络？），甚至帮我送饼干和咖啡。你还梦想得到什么呢？

## 一、机器学习概览

译者：[@SeanCheney](#)

校对者：[@Lisanaaaa](#)、[@飞龙](#)、[@yanmengk](#)、[@Liu Shangfeng](#)

大多数人听到“机器学习”，往往会在脑海中勾勒出一个机器人：一个可靠的管家，或是一个可怕的终结者，这取决于你问的是谁。但是机器学习并不是未来的幻想，它已经来到我们身边了。事实上，一些特定领域已经应用机器学习几十年了，比如光学字符识别（Optical Character Recognition, OCR）。但是直到 1990 年代，第一个影响了数亿人的机器学习应用才真正成熟，它就是垃圾邮件过滤器（spam filter）。虽然并不是一个有自我意识的天网系统（Skynet），垃圾邮件过滤器从技术上是符合机器学习的（它可以很好地进行学习，用户几乎不用再标记某个邮件为垃圾邮件）。后来出现了更多的数以百计的机器学习产品，支撑了更多你经常使用的产品和功能，从推荐系统到语音识别。

机器学习的起点和终点分别是什么呢？确切的讲，机器进行学习是什么意思？如果我下载了一份维基百科的拷贝，我的电脑就真的学会了什么吗？它马上就变聪明了吗？在本章中，我们首先会澄清机器学习到底是什么，以及为什么你要使用它。

然后，在我们出发去探索机器学习新大陆之前，我们要观察下地图，以便知道这片大陆上的主要地区和最明显的地标：监督学习 vs 非监督学习，在线学习 vs 批量学习，基于实例 vs 基于模型学习。然后，我们会学习一个典型的机器学习项目的工作流程，讨论可能碰到的难点，以及如何评估和微调一个机器学习系统。

这一章介绍了大量每个数据科学家需要牢记在心的基础概念（和习语）。第一章只是概览（唯一不含有代码的一章），相当简单，但你要确保每一点都搞明白了，再继续进行学习本书其余章节。端起一杯咖啡，开始学习吧！

提示：如果你已经知道了机器学习的所有基础概念，可以直接翻到第 2 章。

如果你不确认，可以尝试回答本章末尾列出的问题，然后再继续。

## 什么是机器学习？

机器学习是通过编程让计算机从数据中进行学习的科学（和艺术）。

下面是一个更广义的概念：

机器学习是让计算机具有学习的能力，无需进行明确编程。—— 亚瑟·萨缪尔，1959

和一个工程性的概念：

计算机程序利用经验  $E$  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，1997

例如，你的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。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

这个例子中，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。

如果你下载了一份维基百科的拷贝，你的电脑虽然有了很多数据，但不会马上变得聪明起来。因此，这不是机器学习。

## 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（图 1-1）：

1. 你先观察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，也许还注意到发件人名字、邮件正文的格式，等等。
2. 你为观察到的规律写了一个检测算法，如果检测到了这些规律，程序就会标记邮件为垃圾邮件。
3. 测试程序，重复第 1 步和第 2 步，直到满足要求。

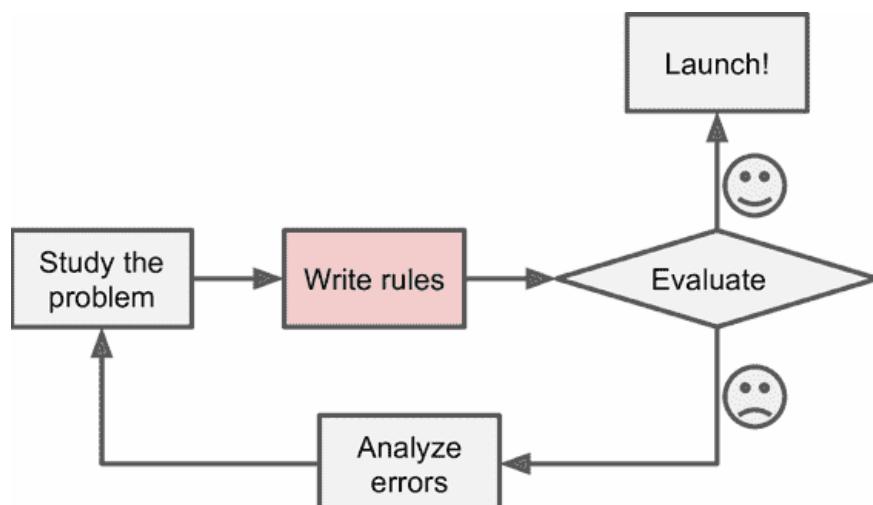


图 1-1 传统方法

这个问题并不简单，你的程序很可能会变成一长串复杂的规则——这样就会很难维护。

相反的，基于机器学习技术的垃圾邮件过滤器会自动学习哪个词和短语是垃圾邮件的预测值，通过与普通邮件比较，检测垃圾邮件中反常频次的词语格式（图 1-2）。这个程序短得多，更易维护，也更精确。

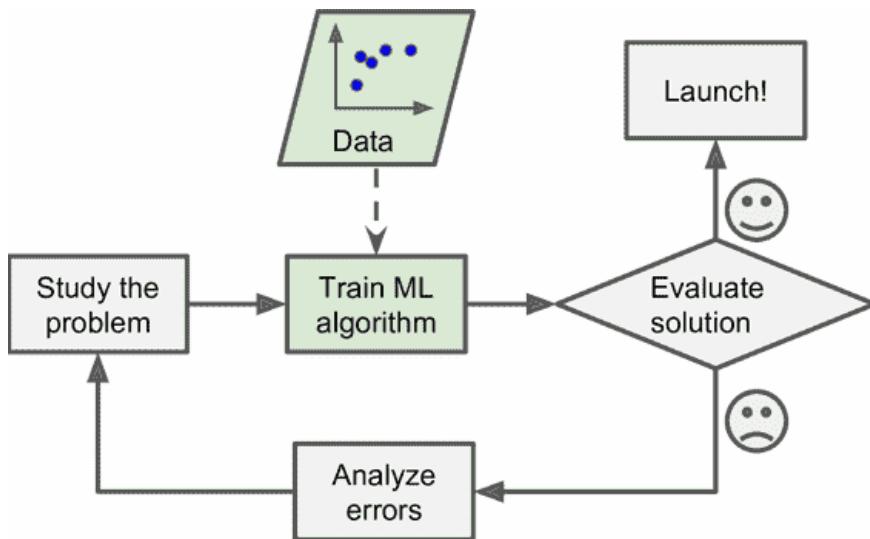


图 1-2 机器学习方法

进而，如果发送垃圾邮件的人发现所有包含 `4u` 的邮件都被屏蔽了，可能会转而使用 `For U`。使用传统方法的垃圾邮件过滤器需要更新以标记 `For U`。如果发送垃圾邮件的人持续更改，你就需要被动地不停地写入新规则。

相反的，基于机器学习的垃圾邮件过滤器会自动注意到 `For U` 在用户手动标记垃圾邮件中的反常频繁性，然后就能自动标记垃圾邮件而无需干预了（图 1-3）。

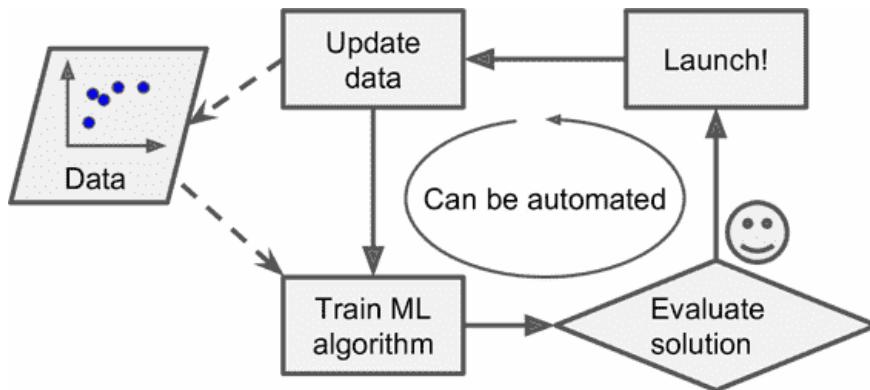


图 1-3 自动适应改变

机器学习的另一个优点是善于处理对于传统方法太复杂或是没有已知算法的问题。例如，对于语言识别：假如想写一个可以识别 `one` 和 `two` 的简单程序。你可能注意到 `two` 起始是一个高音（`t`），所以可以写一个可以测量高音强度的算法，用它区分 `one` 和 `two`。很明显，这个方法不能推广到嘈杂环境下的数百万人的数千词汇、数十种语言。（现在）最佳的方法是根据大量单词的录音，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（图 1-4）：可以检查机器学习算法已经掌握了什么（尽管对于某些算法，这样做会有点麻烦）。例如，当垃圾邮件过滤器被训练了足够多的垃圾邮件，就可以用它列出垃圾邮件预测值的单词和单词组合列表。有时，可能会发现不引人关注的关联或新趋势，有助于对问题更好的理解。

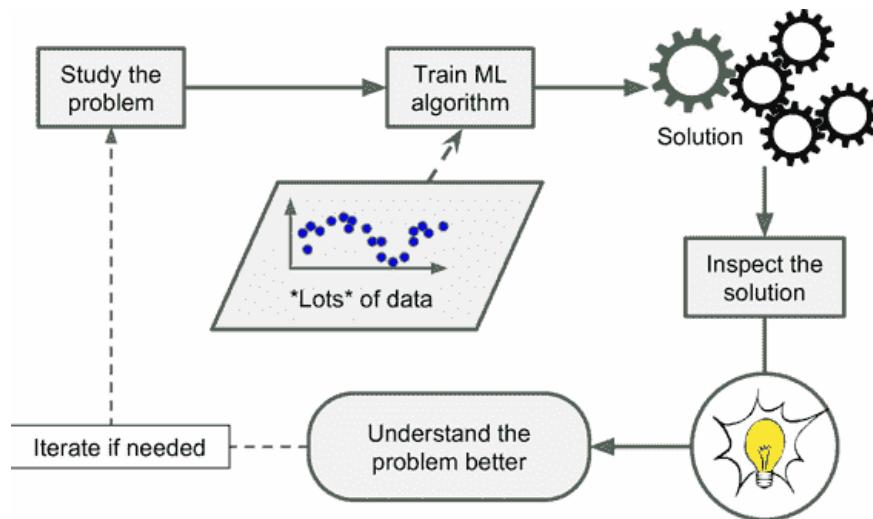


图 1-4 机器学习可以帮助人类学习

使用机器学习方法挖掘大量数据，可以发现并不显著的规律。这称作数据挖掘。

总结一下，机器学习善于：

- 需要进行大量手工调整或需要拥有长串规则才能解决的问题：机器学习算法通常可以简化代码、提高性能。
- 问题复杂，传统方法难以解决：最好的机器学习方法可以找到解决方案。
- 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。

## 机器学习系统的类型

机器学习有多种类型，可以根据如下规则进行分类：

- 是否在人类监督下进行训练（监督，非监督，半监督和强化学习）
- 是否可以动态渐进学习（在线学习 vs 批量学习）
- 它们是否只是通过简单地比较新的数据点和已知的数据点，还是在训练数据中进行模式识别，以建立一个预测模型，就像科学家所做的那样（基于实例学习 vs 基于模型学习）

规则并不仅限于以上的，你可以将他们进行组合。例如，一个先进的垃圾邮件过滤器可以使用神经网络模型动态进行学习，用垃圾邮件和普通邮件进行训练。这就让它成了一个在线、基于模型、监督学习系统。

下面更仔细地学习这些规则。

## 监督/非监督学习

机器学习可以根据训练时监督的量和类型进行分类。主要有四类：监督学习、非监督学习、半监督学习和强化学习。

### 监督学习

在监督学习中，用来训练算法的训练数据包含了答案，称为标签（图 1-5）。

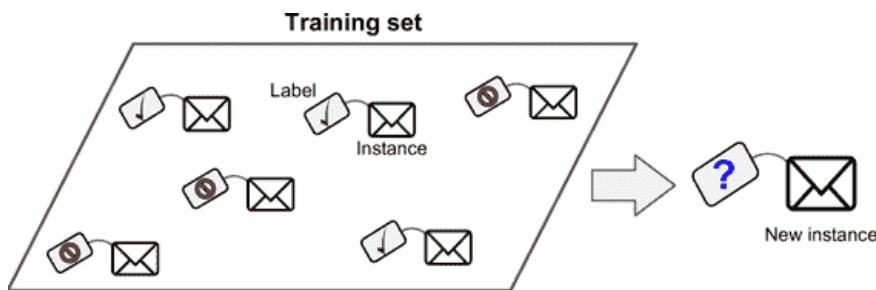


图 1-5 用于监督学习（比如垃圾邮件分类）的加了标签的训练集

一个典型的监督学习任务是分类。垃圾邮件过滤器就是一个很好的例子：用许多带有归类（垃圾邮件或普通邮件）的邮件样本进行训练，过滤器必须还能对新邮件进行分类。

另一个典型任务是预测目标数值，例如给出一些特征（里程数、车龄、品牌等等）称作预测值，来预测一辆汽车的价格。这类任务称作回归（图 1-6）。要训练这个系统，你需要给出大量汽车样本，包括它们的预测值和标签（即，它们的价格）。

注解：在机器学习中，一个属性就是一个数据类型（例如，“里程数”），取决于具体问题一个特征会有多个含义，但通常是属性加上它的值（例如，“里程数 =15000 ”）。许多人是不区分地使用属性和特征。

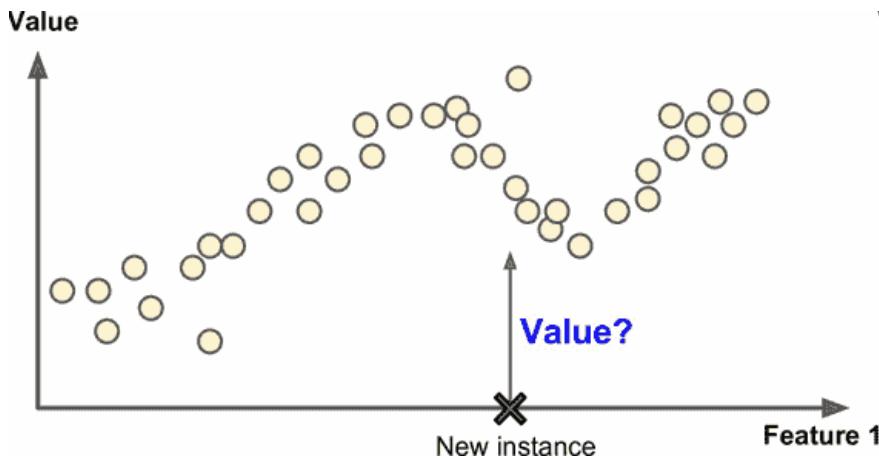


图 1-6 回归

注意，一些回归算法也可以用来进行分类，反之亦然。例如，逻辑回归通常用来进行分类，它可以生成一个归属某一类的可能性的值（例如，20% 几率为垃圾邮件）。

下面是一些重要的监督学习算法（本书都有介绍）：

- 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

## 非监督学习

在非监督学习中，你可能猜到了，训练数据是没有加标签的（图 1-7）。系统在没有老师的条件下进行学习。

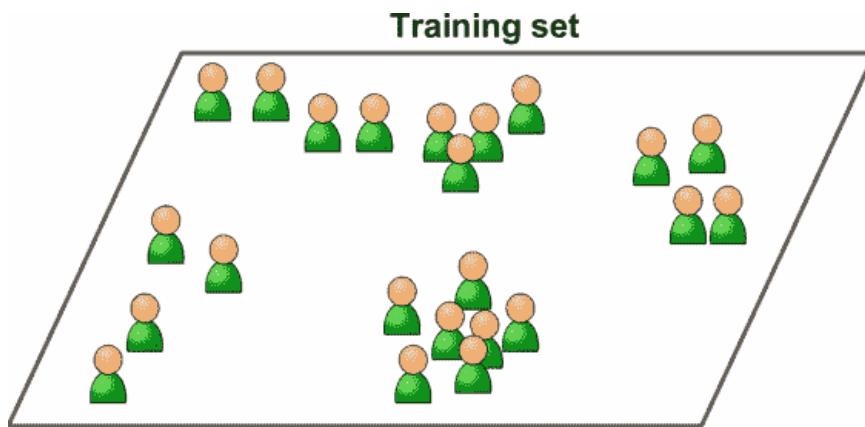


图 1-7 非监督学习的一个不加标签的训练集

下面是一些最重要的非监督学习算法（我们会在第 8 章介绍降维）：

- 聚类
  - K 均值
  - 层次聚类分析 (Hierarchical Cluster Analysis, HCA)
  - 期望最大值
- 可视化和降维
  - 主成分分析 (Principal Component Analysis, PCA)
  - 核主成分分析
  - 局部线性嵌入 (Locally-Linear Embedding, LLE)
  - t-分布邻域嵌入算法 (t-distributed Stochastic Neighbor Embedding, t-SNE)
- 关联性规则学习
  - Apriori 算法
  - Eclat 算法

例如，假设你有一份关于你的博客访客的大量数据。你想运行一个聚类算法，检测相似访客的分组（图 1-8）。你不会告诉算法某个访客属于哪一类：它会自己找出关系，无需帮助。例如，算法可能注意到 40% 的访客是喜欢漫画书的男性，通常是晚上访问，20% 是科幻爱好者，他们是在周末访问等等。如果你使用层次聚类分析，它可能还会细分每个分组为更小的组。这可以帮助你为每个分组定位博文。

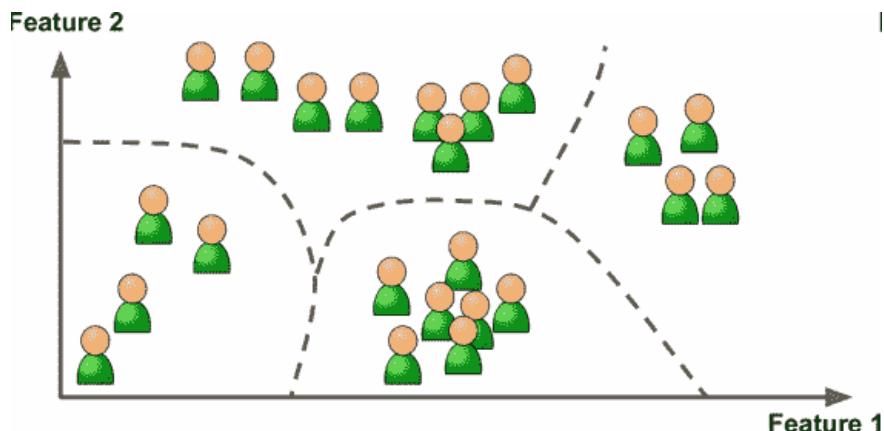


图 1-8 聚类

可视化算法也是极佳的非监督学习案例：给算法大量复杂的且不加标签的数据，算法输出数据的 2D 或 3D 图像（图 1-9）。算法会试图保留数据的结构（即尝试保留输入的独立聚类，避免在图像中重叠），这样就可以明白数据是如何组织起来的，也许还能发现隐藏的规律。

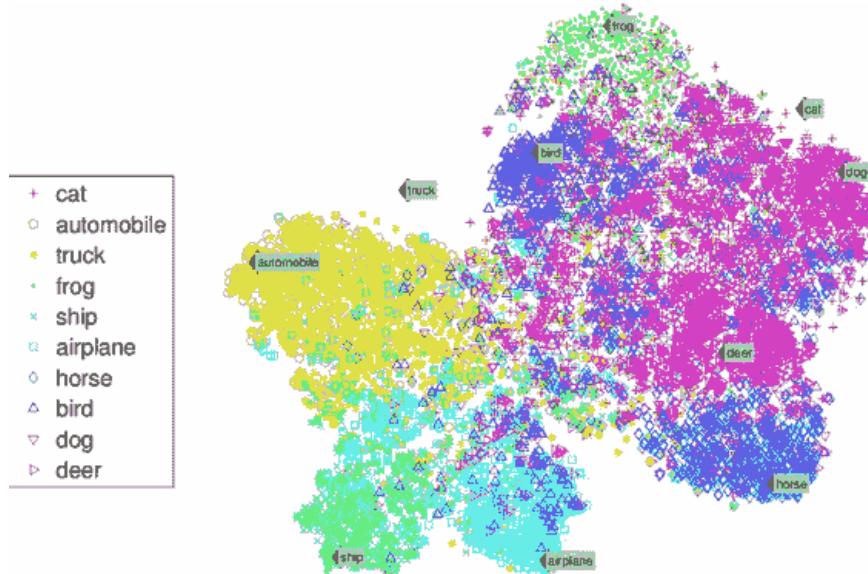


图 1-9 t-SNE 可视化案例，突出了聚类（注：注意动物是与汽车分开的，马和鹿很近、与鸟距离远，以此类推）

与此有关联的任务是降维，降维的目的是简化数据、但是不能失去大部分信息。做法之一是合并若干相关的特征。例如，汽车的里程数与车龄高度相关，降维算法就会将它们合并成一个，表示汽车的磨损。这叫做特征提取。

**提示：**在用训练集训练机器学习算法（比如监督学习算法）时，最好对训练集进行降维。这样可以运行的更快，占用的硬盘和内存空间更少，有些情况下性能也更好。

另一个重要的非监督任务是异常检测（anomaly detection）——例如，检测异常的信用卡转账以防欺诈，检测制造缺陷，或者在训练之前自动从训练数据集去除异常值。异常检测的系统使用正常值训练的，当它碰到一个新实例，它可以判断这个新实例是像正常值还是异常值（图 1-10）。



图 1-10 异常检测

最后，另一个常见的非监督任务是关联规则学习，它的目标是挖掘大量数据以发现属性间有趣的关系。例如，假设你拥有一个超市。在销售日志上运行关联规则，可能发现买了烧烤酱和薯片的人也会买牛排。因此，你可以将这些商品放在一起。

## 半监督学习

一些算法可以处理部分带标签的训练数据，通常是大量不带标签数据加上小部分带标签数据。这称作半监督学习（图 1-11）。

一些图片存储服务，比如 Google Photos，是半监督学习的好例子。一旦你上传了所有家庭相片，它就能自动识别到人物 A 出现在了相片 1、5、11 中，另一个人 B 出现在了相片 2、5、7 中。这是算法的非监督部分（聚类）。现在系统需要的就是你告诉它这两个人是谁。只要给每个人一个标签，算法就可以命名每张照片中的每个人，特别适合搜索照片。

Feature 2

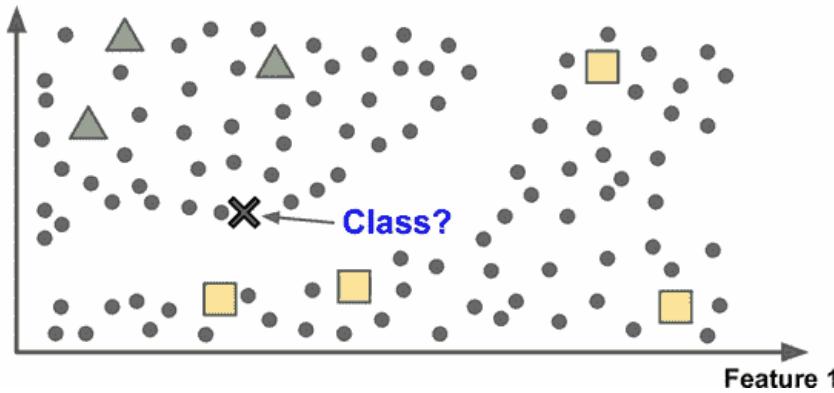


图 1-11 半监督学习

多数半监督学习算法是非监督和监督算法的结合。例如，深度信念网络（deep belief networks）是基于被称为互相叠加的受限玻尔兹曼机（restricted Boltzmann machines, RBM）的非监督组件。RBM 是先用非监督方法进行训练，再用监督学习方法对整个系统进行微调。

## 强化学习

强化学习非常不同。学习系统在这里被称为智能体（agent），可以对环境进行观察、选择和执行动作，并获得奖励作为回报（负奖励是惩罚，见图 1-12）。然后它必须自己学习哪个是最佳方法（称为策略，policy），以得到长久的最大奖励。策略决定了智能体在给定情况下应该采取的行动。

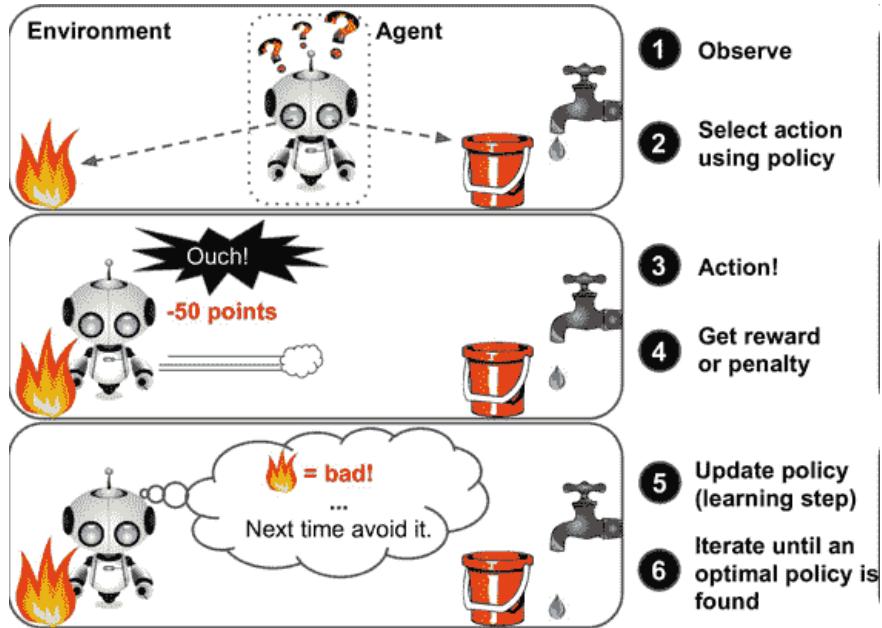


图 1-12 强化学习

例如，许多机器人运行强化学习算法以学习如何行走。DeepMind 的 AlphaGo 也是强化学习的例子：它在 2016 年三月击败了世界围棋冠军李世石（译者注：2017 年五月，AlphaGo 又击败了世界排名第一的柯洁）。它是通过分析数百万盘棋局学习制胜策略，然后自己和自己下棋。要注意，在比赛中机器学习是关闭的；AlphaGo 只是使用它学会的策略。

## 批量和在线学习

另一个用来分类机器学习的准则是，它是否能从导入的数据流进行持续学习。

### 批量学习

在批量学习中，系统不能进行持续学习：必须用所有可用数据进行训练。这通常会占用大量时间和计算资源，所以一般是线下做的。首先是进行训练，然后部署在生产环境且停止学习，它只是使用已经学到的策略。这称为离线学习。

如果你想让一个批量学习系统明白新数据（例如垃圾邮件的新类型），就需要从头训练一个系统的新版本，使用全部数据集（不仅有新数据也有老数据），然后停掉老系统，换上新系统。

幸运的是，训练、评估、部署一套机器学习的系统的整个过程可以自动进行（见图 1-3），所以即便是批量学习也可以适应改变。只要有需要，就可以方便地更新数据、训练一个新版本。

这个方法很简单，通常可以满足需求，但是用全部数据集进行训练会花费大量时间，所以一般是每 24 小时或每周训练一个新系统。如果系统需要快速适应变化的数据（比如，预测股价变化），就需要一个响应更及时的方案。

另外，用全部数据训练需要大量计算资源（CPU、内存空间、磁盘空间、磁盘 I/O、网络 I/O 等等）。如果你有大量数据，并让系统每天自动从头开始训练，就会开销很大。如果数据量巨大，甚至无法使用批量学习算法。

最后，如果你的系统需要自动学习，但是资源有限（比如，一台智能手机或火星车），携带大量训练数据、每天花费数小时的大量资源进行训练是不实际的。

幸运的是，对于上面这些情况，还有一个更佳的方案可以进行持续学习。

## 在线学习

在线学习中，是用数据实例持续地进行训练，可以一次一个或一次几个实例（称为小批量）。每个学习步骤都很快且廉价，所以系统可以动态地学习收到的最新数据（见图 1-13）。

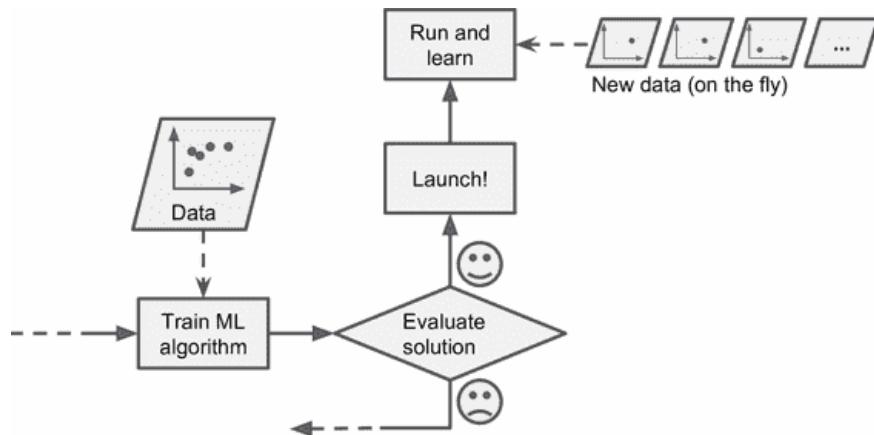


图 1-13 在线学习

在线学习很适合系统接收连续流的数据（比如，股票价格），且需要自动对改变作出调整。如果计算资源有限，在线学习是一个不错的方案：一旦在线学习系统学习了新的数据实例，它就不再需要这些数据了，所以扔掉这些数据（除非你想滚回到之前的一个状态，再次使用数据）。这样可以节省大量的空间。

在线学习算法也适用于在超大数据集（一台计算机不足以用于存储它）上训练系统（这称作核外学习，*out-of-core learning*）。算法每次只加载部分数据，用这些数据进行训练，然后重复这个过程，直到使用完所有数据（见图 1-14）。

**警告：**这个整个过程通常是离线完成的（即，不在部署的系统上），所以在线学习这个名字会让人疑惑。可以把它想成持续学习。

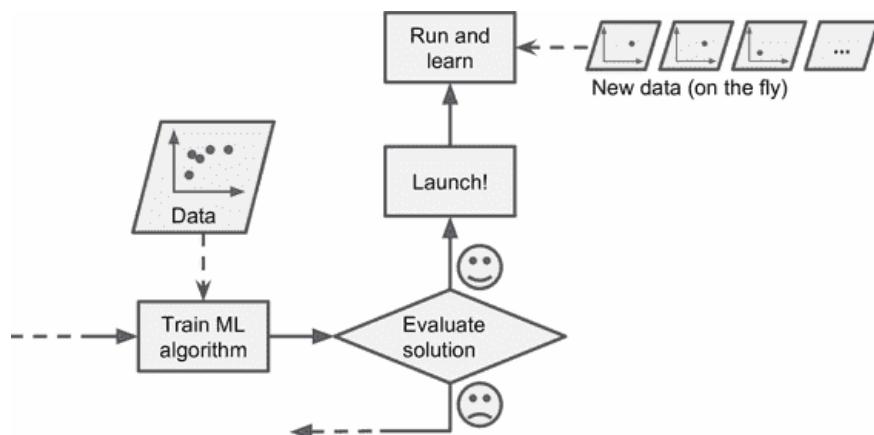


图 1-14 使用在线学习处理大量数据集

在线学习系统的一个重要参数是，它们可以多快地适应数据的改变：这被称为学习速率。如果你设定一个高学习速率，系统就可以快速适应新数据，但是也会快速忘记老数据（你可不想让垃圾邮件过滤器只标记最新的垃圾邮件种类）。相反的，如果你设定的学习速率低，系统的惰性就会强：即，它学的更慢，但对新数据中的噪声或没有代表性的数据点结果不那么敏感。

在线学习的挑战之一是，如果坏数据被用来进行训练，系统的性能就会逐渐下滑。如果这是一个部署的系统，用户就会注意到。例如，坏数据可能来自失灵的传感器或机器人，或某人向搜索引擎传入垃圾信息以提高搜索排名。要减小这种风险，你需要密集监测，如果检测到性能下降，要快速关闭（或是滚回到一个之前的状况）。你可能还要监测输入数据，对反常数据做出反应（比如，使用异常检测算法）。

## 基于实例 vs 基于模型学习

另一种分类机器学习的方法是判断它们是如何进行归纳推广的。大多机器学习任务是关于预测的。这意味着给定一定数量的训练样本，系统需要能推广到之前没见到过的样本。对训练数据集有很好的性能还不够，真正的目标是对新实例预测的能力。

有两种主要的归纳方法：基于实例学习和基于模型学习。

### 基于实例学习

也许最简单的学习形式就是用记忆学习。如果用这种方法做一个垃圾邮件检测器，只需标记所有和用户标记的垃圾邮件相同的邮件——这个方法不差，但肯定不是最好的。

不仅能标记和已知的垃圾邮件相同的邮件，你的垃圾邮件过滤器也要能标记类似垃圾邮件的邮件。这就需要测量两封邮件的相似性。一个（简单的）相似度测量方法是统计两封邮件包含的相同单词的数量。如果一封邮件含有许多垃圾邮件中的词，就会被标记为垃圾邮件。

这被称作基于实例学习：系统先用记忆学习案例，然后使用相似度测量推广到新的例子（图 1-15）。

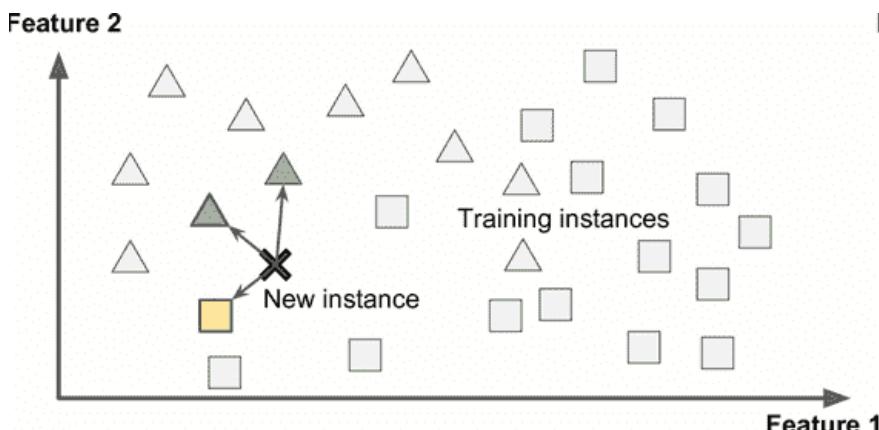


图 1-15 基于实例学习

## 基于模型学习

另一种从样本集进行归纳的方法是建立这些样本的模型，然后使用这个模型进行预测。这称作基于模型学习（图 1-16）。

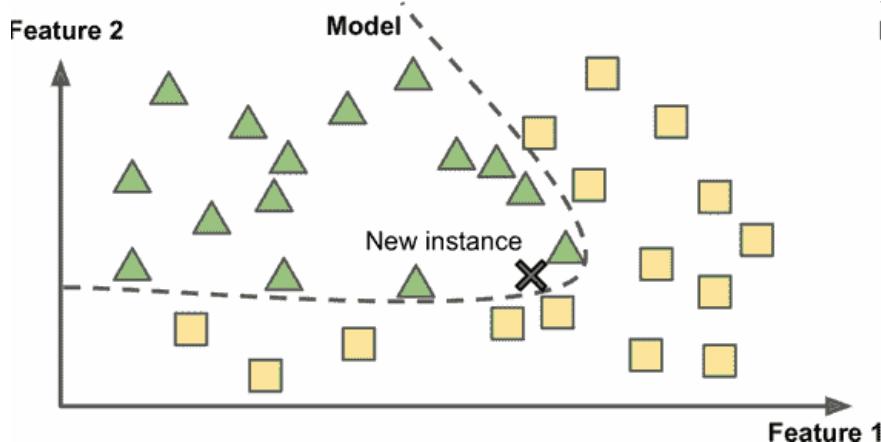


图 1-16 基于模型学习

例如，你想知道钱是否能让人快乐，你从 [OECD 网站](#) 下载了 Better Life Index 指数数据，还从 [IMF](#) 下载了人均 GDP 数据。表 1-1 展示了摘要。

国家	人均 GDP (美元)	生活满意度
匈牙利	12240	4.9
韩国	27195	5.8
法国	37675	6.5
澳大利亚	50962	7.3
美国	55805	7.2

表 1-1 钱会使人幸福吗？

用一些国家的数据画图（图 1-17）。

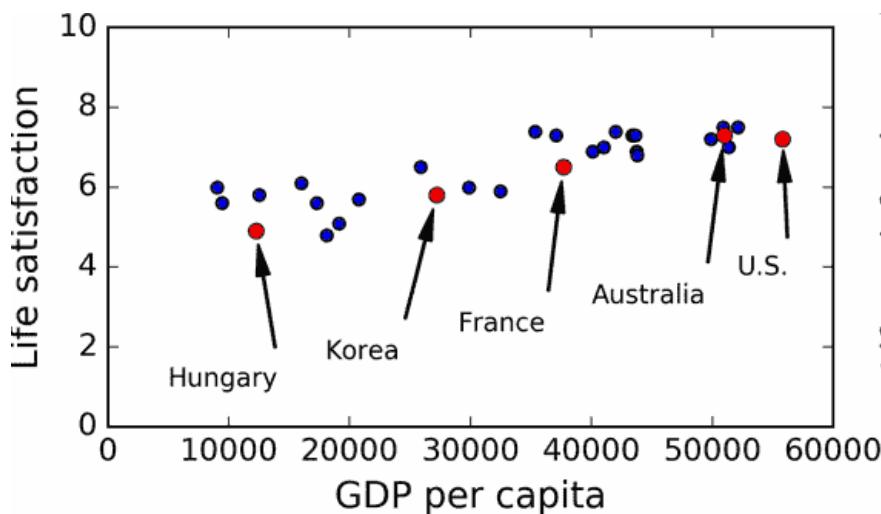


图 1-17 你看到趋势了吗？

确实能看到趋势！尽管数据有噪声（即，部分随机），看起来生活满意度是随着人均 GDP 的增长线性提高的。所以，你决定生活满意度建模为人均 GDP 的线性函数。这一步称作模型选择：你选一个生活满意度的线性模型，只有一个属性，人均 GDP（公式 1-1）。

$$\text{life\_satisfaction} = \theta_0 + \theta_1 \times \text{GDP\_per\_capita}$$

公式 1-1 一个简单的线性模型

这个模型有两个参数  $\theta_0$  和  $\theta_1$ 。通过调整这两个参数，你可以使你的模型表示任何线性函数，见图 1-18。

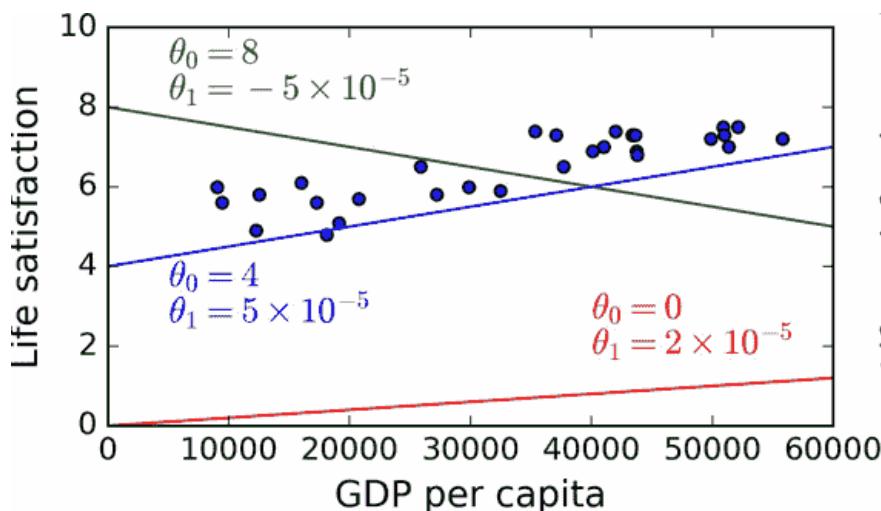


图 1-18 几个可能的线性模型

在使用模型之前，你需要确定  $\theta_0$  和  $\theta_1$ 。如何能知道哪个值可以使模型的性能最佳呢？要回答这个问题，你需要指定性能的量度。你可以定义一个实用函数（或拟合函数）用来测量模型是否够好，或者你可以定义一个代价函数来测量模型有多差。对于线性回归问题，人们一般是用代价函数测量线性模型的预测值和训练样本之间的距离差，目标是使距离差最小。

接下来就是线性回归算法，你用训练样本训练算法，算法找到使线性模型最拟合数据的参数。这称作模型训练。在我们的例子中，算法得到的参数值是  $\theta_0=4.85$  和  $\theta_1=4.91 \times 10^{-5}$ 。

现在模型已经最紧密地拟合到训练数据了，见图 1-19。

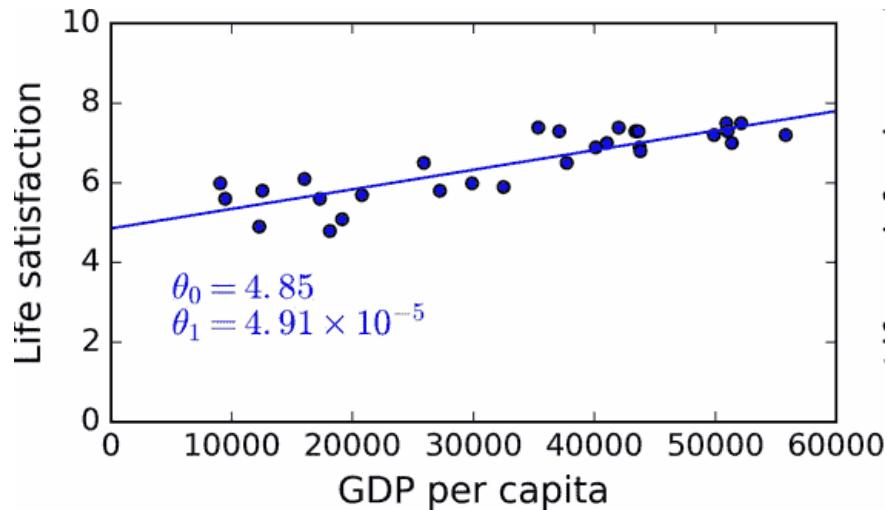


图 1-19 最佳拟合训练数据的线性模型

最后，可以准备运行模型进行预测了。例如，假如你想知道塞浦路斯人有多幸福，但 OECD 没有它的数据。幸运的是，你可以用模型进行预测：查询塞浦路斯的人均 GDP，为 22587 美元，然后应用模型得到生活满意度，后者的值在  $4.85 + 22,587 \times 4.91 \times 10^{-5} = 5.96$  左右。

为了激起你的兴趣，案例 1-1 展示了加载数据、准备、创建散点图的 Python 代码，然后训练线性模型并进行预测。

案例 1-1，使用 Scikit-Learn 训练并运行线性模型。

```

import matplotlib
import matplotlib.pyplot as plt
import numpy as np

import pandas as pd
import sklearn

# 加载数据
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")

# 准备数据
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"]]

# 可视化数据
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()

# 选择线性模型
lin_reg_model = sklearn.linear_model.LinearRegression()

# 训练模型
lin_reg_model.fit(X, y)

# 对塞浦路斯进行预测
X_new = [[22587]] # 塞浦路斯的人均 GDP
print(lin_reg_model.predict(X_new)) # outputs [[ 5.96242338]]

```

注解：如果你之前接触过基于实例学习算法，你会发现斯洛文尼亚的人均 GDP（20732 美元）和塞浦路斯差距很小，OECD 数据上斯洛文尼亚的生活满意度是 5.7，就可以预测塞浦路斯的生活满意度也是 5.7。如果放大一下范围，看一下接下来两个临近的国家，你会发现葡萄牙和西班牙的生活满意度分别是 5.1 和 6.5。对这三个值进行平均得到 5.77，就和基于模型的预测值很接近。这个简单的算法叫做 k 近邻回归（这个例子中，`k=3`）。

在前面的代码中替换线性回归模型为 K 近邻模型，只需更换下面一行：

```
clf = sklearn.linear_model.LinearRegression()
```

为：

```
clf = sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)
```

如果一切顺利，你的模型就可以作出好的预测。如果不能，你可能需要使用更多的属性（就业率、健康、空气污染等等），获取更多更好的训练数据，或选择一个更好的模型（比如，多项式回归模型）。

总结一下：

- 研究数据
- 选择模型
- 用训练数据进行训练（即，学习算法搜寻模型参数值，使代价函数最小）
- 最后，使用模型对新案例进行预测（这称作推断），但愿这个模型推广效果不差

这就是一个典型的机器学习项目。在第 2 章中，你会第一手地接触一个完整的项目。

我们已经学习了许多关于基础的内容：你现在知道了机器学习是关于什么的、为什么它这么有用、最常见的机器学习的分类、典型的项目工作流程。现在，让我们看一看学习中会发生什么错误，导致不能做出准确的预测。

## 机器学习的主要挑战

简而言之，因为你的主要任务是选择一个学习算法并用一些数据进行训练，会导致错误的两件事就是“错误的算法”和“错误的数据”。我们从错误的数据开始。

### 训练数据量不足

要让一个蹒跚学步的孩子知道什么是苹果，需要做的就是指着一个苹果说“苹果”（可能需要重复这个过程几次）。现在这个孩子就能认识所有形状和颜色的苹果。真是个天才！

机器学习还达不到这个程度；需要大量数据，才能让多数机器学习算法正常工作。即便对于非常简单的问题，一般也需要数千的样本，对于复杂的问题，比如图像或语音识别，你可能需要数百万的样本（除非你能重复使用部分存在的模型）。

### 数据的不可思议的有效性

在一篇 2001 年发表的[著名论文](#)中，微软研究员 Michele Banko 和 Eric Brill 展示了不同的机器学习算法，包括非常简单的算法，一旦有了大量数据进行训练，在进行去除语言歧义的测试中几乎有相同的性能（见图 1-20）。

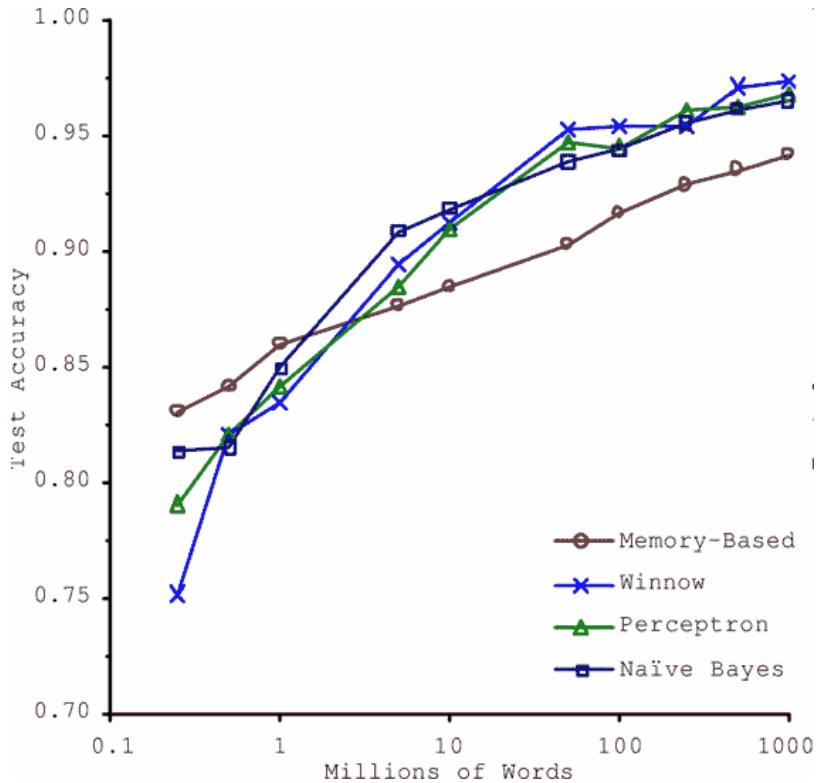


图 1-20 数据和算法的重要性对比

论文作者说：“结果说明，我们可能需要重新考虑在算法开发 vs 语料库发展上花费时间和金钱的取舍。”

对于复杂问题，数据比算法更重要的主张在 2009 年由 Norvig 发表的论文[《数据的不合理有效性》](#)得到了进一步的推广。但是，应该注意到，小型和中型的数据集仍然是非常常见的，获得额外的训练数据并不总是轻易和廉价的，所以不要抛弃算法。

## 没有代表性的训练数据

为了更好地进行归纳推广，让训练数据对新数据具有代表性是非常重要的。无论你用的是基于实例学习或基于模型学习，这点都很重要。

例如，我们之前用来训练线性模型的国家集合不够具有代表性：缺少了一些国家。

图 1-21 展示了添加这些缺失国家之后的数据。

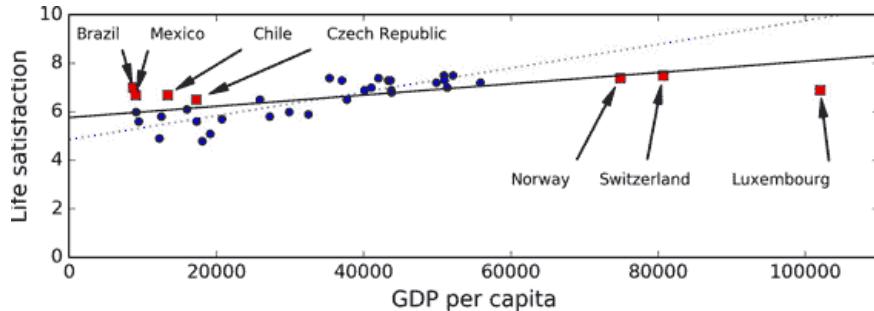


图 1-21 一个更具代表性的训练样本

如果你用这份数据训练线性模型，得到的是实线，旧模型用虚线表示。可以看到，添加几个国家不仅可以显著地改变模型，它还说明如此简单的线性模型可能永远不会达到很好的性能。貌似非常富裕的国家没有中等富裕的国家快乐（事实上，非常富裕的国家看起来更不快乐），相反的，一些贫穷的国家看上去比富裕的国家还幸福。

使用了没有代表性的数据集，我们训练了一个不可能得到准确预测的模型，特别是对于非常贫穷和非常富裕的国家。

使用具有代表性的训练集对于推广到新案例是非常重要的。但是做起来比说起来要难：如果样本太小，就会有样本噪声（即，会有一定概率包含没有代表性的数据），但是即使是非常大的样本也可能没有代表性，如果取样方法错误的话。这叫做样本偏差。

#### 一个样本偏差的著名案例

也许关于样本偏差最有名的案例发生在 1936 年兰登和罗斯福的美国大选：《文学文摘》做了一个非常大的民调，给 1000 万人邮寄了调查信。得到了 240 万回信，非常有信心地预测兰登会以 57% 赢得大选。然而，罗斯福赢得了 62% 的选票。错误发生在《文学文摘》的取样方法：

- 首先，为了获取发信地址，《文学文摘》使用了电话黄页、杂志订阅用户、俱乐部会员等相似的列表。所有这些列表都偏向于富裕人群，他们都倾向于投票给共和党（即兰登）。
- 第二，只有 25% 的回答了调研。这就又一次引入了样本偏差，它排除了不关心政治的人、不喜欢《文学文摘》的人，和其它关键人群。这种特殊的样本偏差称作无应答偏差。

下面是另一个例子：假如你想创建一个能识别放克音乐（Funk Music, 别名骚乐）视频的系统。建立训练集的方法之一是在 YouTube 上搜索“放克音乐”，使用搜索到的视频。但是这样就假定了 YouTube 的搜索引擎返回的视频集，是对 YouTube 上的所有放克音乐有代表性的。事实上，搜索结果可能更偏向于流行歌手（如果你居住在巴西，你会得到许多“funk carioca”视频，它们和 James Brown 的截然不同）。从另一方面来讲，你还能怎么得到一个大的训练集呢？

## 低质量数据

很明显，如果训练集中的错误、异常值和噪声（错误测量引入的）太多，系统检测出潜在规律的难度就会变大，性能就会降低。花费时间对训练数据进行清理是十分重要的。事实上，大多数据科学家的大部分时间是做清洗工作的。例如：

- 如果一些实例是明显的异常值，最好删掉它们或尝试手工修改错误；
- 如果一些实例缺少特征（比如，你的 5% 的顾客没有说明年龄），你必须决定是否忽略这个属性、忽略这些实例、填入缺失值（比如，年龄中位数），或者训练一个含有这个特征的模型和一个不含有这个特征的模型，等等。

## 不相关的特征

俗语说：如果进来的是垃圾，那么出去的也是垃圾。你的系统只有在训练数据包含足够相关特征、非相关特征不多的情况下，才能进行学习。机器学习项目成功的关键之一是用好的特征进行训练。这个过程称作特征工程，包括：

- 特征选择：在所有存在的特征中选取最有用的特征进行训练。
- 特征提取：组合存在的特征，生成一个更有用的特征（如前面看到的，可以使用降维算法）。
- 收集新数据创建新特征。

现在，我们已经看过了许多坏数据的例子，接下来看几个坏算法的例子。

## 过拟合训练数据

如果你在外国游玩，当地的出租车司机多收了你的钱。你可能会说这个国家所有的出租车司机都是小偷。过度归纳是我们人类经常做的，如果我们不小心，机器也会犯同样的错误。在机器学习中，这称作过拟合：意思是说，模型在训练数据上表现很好，但是推广效果不好。

图 1-22 展示了一个高阶多项式生活满意度模型，它大大过拟合了训练数据。即使它比简单线性模型在训练数据上表现更好，你会相信它的预测吗？

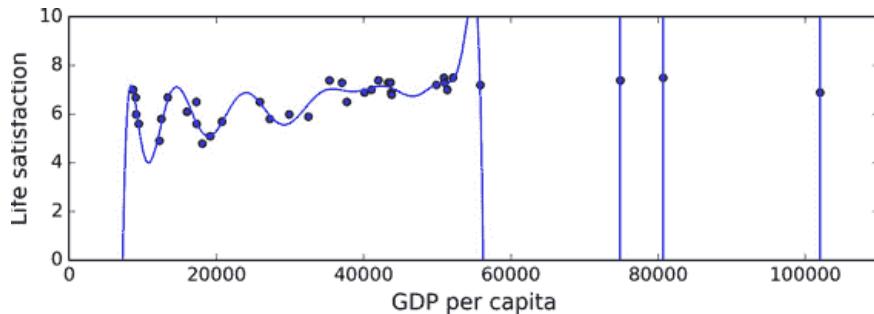


图 1-22 过拟合训练数据

复杂的模型，比如深度神经网络，可以检测数据中的细微规律，但是如果训练集有噪声，或者训练集太小（太小会引入样本噪声），模型就会去检测噪声本身的规律。很明显，这些规律不能推广到新实例。例如，假如你用更多的属性训练生活满意度模型，包括不包含信息的属性，比如国家的名字。如此一来，复杂的模型可能会检测出训练集中名字有 w 字母的国家的生活满意度大于 7：新西兰 (7.3)，挪威 (7.4)，瑞典 (7.2) 和瑞士 (7.5)。你能相信这个 W-满意度法则推广到卢旺达和津巴布韦吗？很明显，这个规律只是训练集数据中偶然出现的，但是模型不能判断这个规律是真实的、还是噪声的结果。

警告：过拟合发生在相对于训练数据的量和噪声，模型过于复杂的情况。可能的解决方案有：

- 简化模型，可以通过选择一个参数更少的模型（比如使用线性模型，而不是高阶多项式模型）、减少训练数据的属性数、或限制一下模型
- 收集更多的训练数据
- 减小训练数据的噪声（比如，修改数据错误和去除异常值）

限定一个模型以让它更简单并且降低过拟合的风险被称作正则化（regularization）。例如，我们之前定义的线性模型有两个参数， $\theta_0$  和  $\theta_1$ 。它给了学习算法两个自由度以让模型适应训练数据：可以调整截距  $\theta_0$  和斜率  $\theta_1$ 。如果强制  $\theta_1=0$ ，算法就只剩一个自由度，拟合数据就会更为困难：它所能做的只是将拟合曲线上下移动去尽可能地靠近训练实例，结果会在平均值附近。这就是一个非常简单的模型！如果我们允许算法可以修改  $\theta_1$ ，但是只能在一个很小的范围内修改，算法的自由度就会介于 1 和 2 之间。它要比两个自由度的模型简单，比 1 个自由度的模型要复杂。你的目标是在完美拟合数据和保持模型简单性上找到平衡，确保算法的推广效果。

图 1-23 展示了三个模型：虚线表示用一些缺失国家的数据训练的原始模型，短划线是我们的第二个用所有国家训练的模型，实线模型的训练数据和第一个相同，但进行了正则化限制。你可以看到正则化强制模型有一个小的斜率，它对训练数据的拟合不是那么好，但是对新样本的推广效果好。

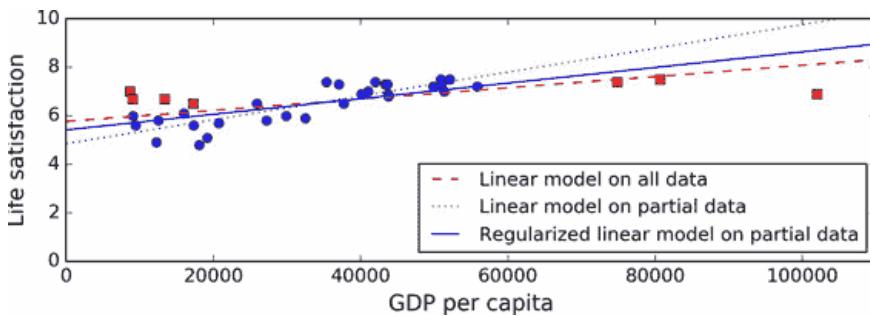


图 1-23 正则化降低了过度拟合的风险

正则化的度可以用一个超参数（hyperparameter）控制。超参数是一个学习算法的参数（而不是模型的）。这样，它是不会被学习算法本身影响的，它优于训练，在训练中是保持不变的。如果你设定的超参数非常大，就会得到一个几乎是平的模型（斜率接近于 0）；这种学习算法几乎肯定不会过拟合训练数据，但是也很难得到一个好的解。调节超参数是创建机器学习算法非常重要的一部分（下一章你会看到一个详细的例子）。

## 欠拟合训练数据

你可能猜到了，欠拟合是和过拟合相对的：当你的模型过于简单时就会发生。例如，生活满意度的线性模型倾向于欠拟合；现实要比这个模型复杂的多，所以预测很难准确，即使在训练样本上也很难准确。

解决这个问题的选项包括：

- 选择一个更强大的模型，带有更多参数
- 用更好的特征训练学习算法（特征工程）
- 减小对模型的限制（比如，减小正则化超参数）

## 回顾

现在，你已经知道了很多关于机器学习的知识。然而，学过了这么多概念，你可能会感到有些迷失，所以让我们退回去，回顾一下重要的：

- 机器学习是让机器通过学习数据对某些任务做得更好，而不使用确定的代码规则。
- 有许多不同类型的机器学习系统：监督或非监督，批量或在线，基于实例或基于模型，等等。
- 在机器学习项目中，我们从训练集中收集数据，然后对学习算法进行训练。如果算法是基于模型的，就调节一些参数，让模型拟合到训练集（即，对训练集本身作出好的预测），然后希望它对新样本也能有好预测。如果算法是基于实例的，就是用记忆学习样本，然后用相似度推广到新实例。
- 如果训练集太小、数据没有代表性、含有噪声、或掺有不相关的特征（垃圾进，垃圾出），系统的性能不会好。最后，模型不能太简单（会发生欠拟合）或太复杂（会发生过拟合）。

还差最后一个主题要学习：训练完了一个模型，你不只希望将它推广到新样本。如果你想评估它，那么还需要作出必要的微调。一起来看一看。

## 测试和确认

---

要知道一个模型推广到新样本的效果，唯一的方法就是真正的进行试验。一种方法是将模型部署到生产环境，观察它的性能。这么做可以，但是如果模型的性能很差，就会引起用户抱怨——这不是最好的方法。

更好的选项是将你的数据分成两个集合：训练集和测试集。正如它们的名字，用训练集进行训练，用测试集进行测试。对新样本的错误率称作推广错误（或样本外错误），通过模型对测试集的评估，你可以预估这个错误。这个值可以告诉你，你的模型对新样本的性能。

如果训练错误率低（即，你的模型在训练集上错误不多），但是推广错误率高，意味着模型对训练数据过拟合。

**提示：**一般使用 80% 的数据进行训练，保留 20% 用于测试。

因此，评估一个模型很简单：只要使用测试集。现在假设你在两个模型之间犹豫不决（比如一个线性模型和一个多项式模型）：如何做决定呢？一种方法是两个都训练，，然后比较在测试集上的效果。

现在假设线性模型的效果更好，但是你想做一些正则化以避免过拟合。问题是：如何选择正则化超参数的值？一种选项是用 100 个不同的超参数训练 100 个不同的模型。假设你发现最佳的超参数的推广错误率最低，比如只有 5%。然后就选用这个模型作为生产环境，但是实际中性能不佳，误差率达到了 15%。发生了什么呢？

答案在于，你在测试集上多次测量了推广误差率，调整了模型和超参数，以使模型最适合这个集合。这意味着模型对新数据的性能不会高。

这个问题通常的解决方案是，再保留一个集合，称作验证集合。用训练集和多个超参数训练多个模型，选择在验证集上有最佳性能的模型和超参数。当你对模型满意时，用测试集再做最后一次测试，以得到推广误差率的预估。

为了避免“浪费”过多训练数据在验证集上，通常的办法是使用交叉验证：训练集分成互补的子集，每个模型用不同的子集训练，再用剩下的子集验证。一旦确定模型类型和超参数，最终的模型使用这些超参数和全部的训练集进行训练，用测试集得到推广误差率。

### 没有免费午餐公理

模型是观察的简化版本。简化意味着舍弃无法进行推广的表面细节。但是，要确定舍弃什么数据、保留什么数据，必须要做假设。例如，线性模型的假设是数据基本上是线性的，实例和模型直线间的距离只是噪音，可以放心忽略。

在一篇 1996 年的[著名论文](#)中，David Wolpert 证明，如果完全不对数据做假设，就没有理由选择一个模型而不选另一个。这称作没有免费午餐 (NFL) 公理。对于一些数据集，最佳模型是线性模型，而对其他数据集是神经网络。没有一个模型可以保证效果更好（如这个公理的名字所示）。确信的唯一方法就是测试所有的模型。因为这是不可能的，实际中就必须做一些对数据合理的假设，只评估几个合理的模型。例如，对于简单任务，你可能是用不同程度的正则化评估线性模型，对于复杂问题，你可能要评估几个神经网络模型。

## 练习

---

本章中，我们学习了一些机器学习中最为重要的概念。下一章，我们会更加深入，并写一些代码。开始下章之前，确保你能回答下面的问题：

1. 如何定义机器学习？
2. 机器学习可以解决的四类问题？
3. 什么是带标签的训练集？
4. 最常见的两个监督任务是什么？
5. 指出四个常见的非监督任务？
6. 要让一个机器人能在各种未知地形行走，你会采用什么机器学习算法？
7. 要对你的顾客进行分组，你会采用哪类算法？
8. 垃圾邮件检测是监督学习问题，还是非监督学习问题？
9. 什么是在线学习系统？
10. 什么是核外学习？
11. 什么学习算法是用相似度做预测？
12. 模型参数和学习算法的超参数的区别是什么？
13. 基于模型学习的算法搜寻的是什么？最成功的策略是什么？基于模型学习如何做预测？
14. 机器学习的四个主要挑战是什么？
15. 如果模型在训练集上表现好，但推广到新实例表现差，问题是什么？给出三个可能的解决方案。
16. 什么是测试集，为什么要使用它？
17. 验证集的目的是什么？
18. 如果用测试集调节超参数，会发生什么？
19. 什么是交叉验证，为什么它比验证集好？

练习答案见附录 A。

## 二、端到端的机器学习项目

译者：[@SeanCheney](#)

校对者：[@Lisanaaa](#)、[@飞龙](#)、[@PeterHo](#)、[@ZhengqiJiang](#)、[@tabeworks](#)

本章中，你会假装作为被一家地产公司刚刚雇佣的数据科学家，完整地学习一个案例项目。下面是主要步骤：

1. 项目概述。
2. 获取数据。
3. 发现并可视化数据，发现规律。
4. 为机器学习算法准备数据。
5. 选择模型，进行训练。
6. 微调模型。
7. 给出解决方案。
8. 部署、监控、维护系统。

### 使用真实数据

学习机器学习时，最好使用真实数据，而不是人工数据集。幸运的是，有上千个开源数据集可以进行选择，涵盖多个领域。以下是一些可以查找的数据的地方：

- 流行的开源数据仓库：
  - [UC Irvine Machine Learning Repository](#)
  - [Kaggle datasets](#)
  - [Amazon's AWS datasets](#)
- 准入口（提供开源数据列表）
  - <http://dataportals.org/>
  - <http://opendatamonitor.eu/>
  - <http://quandl.com/>
- 其它列出流行开源数据仓库的网页：
  - [Wikipedia's list of Machine Learning datasets](#)
  - [Quora.com question](#)
  - [Datasets subreddit](#)

本章，我们选择的是 StatLib 的加州房产价格数据集（见图 2-1）。这个数据集是基于 1990 年加州普查的数据。数据已经有点老（1990 年还能买一个湾区不错的房子），但是它有许多优点，利于学习，所以假设这个数据为最近的。为了便于教学，我们添加了一个类别属性，并除去了一些。

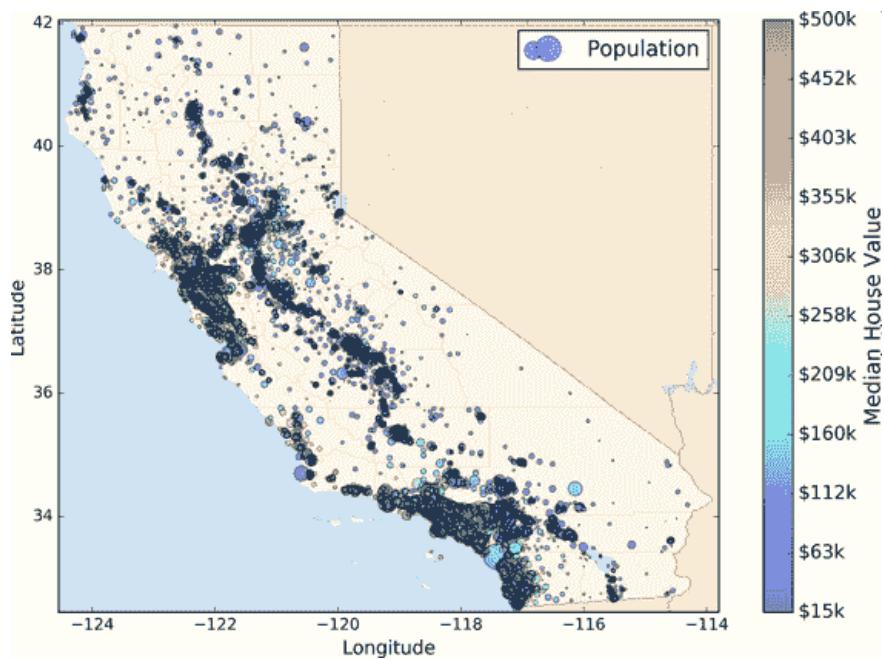


图 2-1 加州房产价格

## 项目概览

欢迎来到机器学习房地产公司！你的第一个任务是利用加州普查数据，建立一个加州房价模型。这个数据包含每个街区组的人口、收入中位数、房价中位数等指标。

街区组是美国调查局发布样本数据的最小地理单位（一个街区通常有 600 到 3000 人）。我们将其简称为“街区”。

你的模型要利用这个数据进行学习，然后根据其它指标，预测任何街区的房价中位数。

**提示：**你是一个有条理的数据科学家，你要做的第一件事是拿出你的机器学习项目清单。你可以使用附录 B 中的清单；这个清单适用于大多数的机器学习项目，但是你还是要确认它是否满足需求。在本章中，我们会检查许多清单上的项目，但是也会跳过一些简单的，有些会在后面的章节再讨论。

## 划定问题

问老板的第一个问题应该是商业目标是什么？建立模型可能不是最终目标。公司要如何使用、并从模型受益？这非常重要，因为它决定了如何划定问题，要选择什么算法，评估模型性能的指标是什么，要花多少精力进行微调。

老板告诉你你的模型的输出（一个区的房价中位数）会传给另一个机器学习系统（见图 2-2），也有其它信号会传入后面的系统。这一整套系统可以确定某个区进行投资值不值。确定值不值得投资非常重要，它直接影响利润。

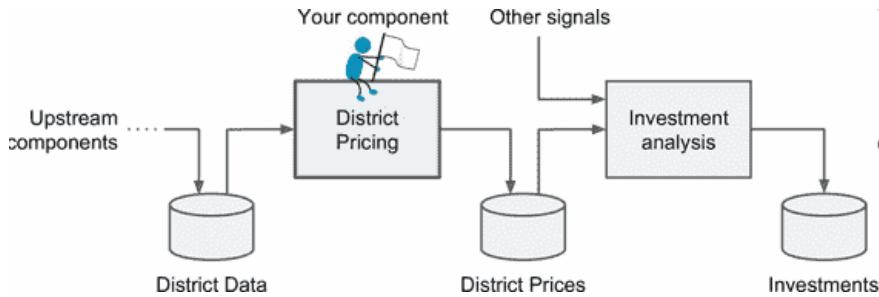


图 2-2 房地产投资的机器学习流水线

## 流水线

一系列的数据处理组件被称为数据流水线。流水线在机器学习系统中很常见，因为有许多数据要处理和转换。

组件通常是异步运行的。每个组件吸纳进大量数据，进行处理，然后将数据传输到另一个数据容器中，而后流水线中的另一个组件收入这个数据，然后输出，这个过程依次进行下去。每个组件都是独立的：组件间的接口只是数据容器。这样可以让系统更便于理解（记住数据流的图），不同的项目组可以关注于不同的组件。进而，如果一个组件失效了，下游的组件使用失效组件最后生产的数据，通常可以正常运行（一段时间）。这样就使整个架构相当健壮。

另一方面，如果没有监控，失效的组件会在不被注意的情况下运行一段时间。数据会受到污染，整个系统的性能就会下降。

下一个要问的问题是，现在的解决方案效果如何。老板通常会给一个参考性能，以及如何解决问题。老板说，现在街区的房价是靠专家手工估计的，专家队伍收集最新的关于一个区的信息（不包括房价中位数），他们使用复杂的规则进行估计。这种方法费钱费时间，而且估计结果不理想，误差率大概有 15%。

OK，有了这些信息，你就可以开始设计系统了。首先，你需要划定问题：监督或非监督，还是强化学习？这是个分类任务、回归任务，还是其它的？要使用批量学习还是线上学习？继续阅读之前，请暂停一下，尝试自己回答下这些问题。

你能回答出来吗？一起看下答案：很明显，这是一个典型的监督学习任务，因为你要使用的是有标签的训练样本（每个实例都有预定的产出，即街区的房价中位数）。并且，这是一个典型的回归任务，因为你要预测一个值。讲的更细些，这是一个多变量回归问题，因为系统要使用多个变量进行预测（要使用街区的人口，收入中位数等等）。在第一章中，你只是根据人均 GDP 来预测生活满意度，因此这是一个单变量回归问题。最后，没有连续的数据流进入系统，没有特别需求需要对数据变动作出快速适应。数据量不大可以放到内存中，因此批量学习就够了。

**提示：**如果数据量很大，你可以要在多个服务器上对批量学习做拆分（使用 MapReduce 技术，后面会看到），或是使用线上学习。

## 选择性能指标

下一步是选择性能指标。回归问题的典型指标是均方根误差（RMSE）。均方根误差测量的是系统预测误差的标准差。例如，RMSE 等于 50000，意味着，68% 的系统预测值位于实际值的 50000 美元以内，95% 的预测值位于实际值的 100000

美元以内（一个特征通常都符合高斯分布，即满足“68-95-99.7”规则：大约 68% 的值落在  $1\sigma$  内，95% 的值落在  $2\sigma$  内，99.7% 的值落在  $3\sigma$  内，这里的  $\sigma$  等于 50000）。公式 2-1 展示了计算 RMSE 的方法。

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(\mathbf{x}^{(i)}) - y^{(i)})^2}$$

公式 2-1 均方根误差 (RMSE)

## 符号的含义

这个方程引入了一些常见的贯穿本书的机器学习符号：

- $m$  是测量 RMSE 的数据集中的实例数量。  
例如，如果用一个含有 2000 个街区的验证集求 RMSE，则  $m = 2000$ 。
- $x^{(i)}$  是数据集第  $i$  个实例的所有特征值（不包含标签）的向量， $y^{(i)}$  是它的标签（这个实例的输出值）。

例如，如果数据集中的第一个街区位于经度  $-118.29^\circ$ ，纬度  $33.91^\circ$ ，有 1416 名居民，收入中位数是 38372 美元，房价中位数是 156400 美元（忽略掉其它的特征），则有：

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1,416 \\ 38,372 \end{pmatrix}$$

和，

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

- $x$  是包含数据集中所有实例的所有特征值（不包含标签）的矩阵。每一行是一个实例，第  $i$  行是  $x^{(i)}$  的转置，记为  $x^{(i)T}$ 。

例如，仍然是前面提到的第一区，矩阵  $x$  就是：

$$\mathbf{x} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- $h$  是系统的预测函数，也称为假设（hypothesis）。当系统收到一个实例的特征向量  $x^{(i)}$ ，就会输出这个实例的一个预测值  $y_{\text{hat}} = h(x^{(i)})$ （ $y_{\text{hat}}$  读作  $y$ -hat）。

例如，如果系统预测第一区的房价中位数是 158400 美元，  
则  $y_{\text{hat}}(1) = h(x(1)) = 158400$ 。预测误差是  
 $y_{\text{hat}}(1) - y(1) = 2000$ 。

- RMSE( $X, h$ ) 是使用假设  $h$  在样本集上测量的损失函数。

我们使用小写斜体表示标量值（例如  $m$  或  $y^{(i)}$ ）和函数名（例如  $h$ ），  
小写粗体表示向量（例如  $x^{(i)}$ ），大写粗体表示矩阵（例如  $X$ ）。

虽然大多数时候 RMSE 是回归任务可靠的性能指标，在有些情况下，你可能需要另外的函数。例如，假设存在许多异常的街区。此时，你可能需要使用平均绝对误差（Mean Absolute Error，也称作平均绝对偏差），见公式 2-2：

$$\text{MAE}(X, h) = \frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}|$$

公式 2-2 平均绝对误差

RMSE 和 MAE 都是测量预测值和目标值两个向量距离的方法。有多种测量距离的方法，或范数：

- 计算对应欧几里得范数的平方和的根（RMSE）：这个距离介绍过。它也称作  $\ell_2$  范数，标记为  $\|\cdot\|_2$ （或只是  $\|\cdot\|$ ）。
- 计算对应于  $\ell_1$ （标记为  $\|\cdot\|_1$ ）范数的绝对值和（MAE）。有时，也称其为曼哈顿范数，因为它测量了城市中的两点，沿着矩形的边行走的距离。
- 更一般的，包含  $n$  个元素的向量  $v$  的  $\ell_k$  范数（ $K$  阶闵氏范数），定义成

$$\|v\|_k = (\|v_0\|^k + \|v_1\|^k + \dots + \|v_n\|^k)^{\frac{1}{k}}$$

$\ell_0$ （汉明范数）只显示了这个向量的基数（即，非零元素的个数）， $\ell_\infty$ （切比雪夫范数）是向量中最大的绝对值。

- 范数的指数越高，就越关注大的值而忽略小的值。这就是为什么 RMSE 比 MAE 对异常值更敏感。但是当异常值是指数分布的（类似正态曲线），RMSE 就会表现很好。

## 核实假设

最后，最好列出并核对迄今（你或其他人）作出的假设，这样可以尽早发现严重的问题。例如，你的系统输出的街区房价，会传入到下游的机器学习系统，我们假设这些价格确实会被当做街区房价使用。但是如果下游系统实际上将价格转化成了分类（例如，便宜、中等、昂贵），然后使用这些分类，而不是使用价格。这样的话，获得准确的价格就不那么重要了，你只需要得到合适的分类。问题相应地就变成了一个分类问题，而不是回归任务。你可不想在一个回归系统上工作了数月，最后才发现真相。

幸运的是，在与下游系统主管探讨之后，你很确信他们需要的就是实际的价格，而不是分类。很好！整装待发，可以开始写代码了。

## 获取数据

开始动手。最后用 Jupyter 笔记本完整地敲一遍示例代码。完整的代码位于 <https://github.com/ageron/handson-ml>。

## 创建工作空间

首先，你需要安装 Python。可能已经安装过了，没有的话，可以从官网下载 <https://www.python.org/>。

接下来，需要为你的机器学习代码和数据集创建工作空间目录。打开一个终端，输入以下命令（在提示符 \$ 之后）：

```
$ export ML_PATH="$HOME/ml"      # 可以更改路径  
$ mkdir -p $ML_PATH
```

还需要一些 Python 模块：Jupyter、NumPy、Pandas、Matplotlib 和 Scikit-Learn。如果所有这些模块都已经在 Jupyter 中运行了，你可以直接跳到下一节“下载数据”。如果还没安装，有多种方法可以进行安装（包括它们的依赖）。你可以使用系统的包管理系统（比如 Ubuntu 上的 apt-get，或 macOS 上的 MacPorts 或 HomeBrew），安装一个 Python 科学计算环境比如 Anaconda，使用 Anaconda 的包管理系统，或者使用 Python 自己的包管理器 pip，它是 Python 安装包（自从 2.7.9 版本）自带的。可以用下面的命令检测是否安装 pip：

```
$ pip3 --version  
pip 9.0.1 from [...]/lib/python3.5/site-packages (python 3.5)
```

你需要保证 pip 是近期的版本，至少高于 1.4，以保障二进制模块文件的安装（也称为 wheel）。要升级 pip，可以使用下面的命令：

```
$ pip3 install --upgrade pip  
Collecting pip  
[...]  
Successfully installed pip-9.0.1
```

### 创建独立环境

如果你希望在一个独立环境中工作（强烈推荐这么做，不同项目的库的版本不会冲突），用下面的 pip 命令安装 virtualenv：

```
$ pip3 install --user --upgrade virtualenv
Collecting virtualenv
[...]
Successfully installed virtualenv
```

现在可以通过下面命令创建一个独立的 Python 环境：

```
$ cd $ML_PATH
$ virtualenv env
Using base prefix '[...]'
New python executable in [...]/ml/env/bin/python3.5
Also creating executable in [...]/ml/env/bin/python
Installing setuptools, pip, wheel...done.
```

以后每次想要激活这个环境，只需打开一个终端然后输入：

```
$ cd $ML_PATH
$ source env/bin/activate
```

启动该环境时，使用 pip 安装的任何包都只安装于这个独立环境中，Python 指挥访问这些包（如果你希望 Python 能访问系统的包，创建环境时要使用包选项 `--system-site`）。更多信息，请查看 `virtualenv` 文档。

现在，你可以使用 pip 命令安装所有必需的模块和它们的依赖：

```
$ pip3 install --upgrade jupyter matplotlib numpy pandas scipy scikit-learn
Collecting jupyter
  Downloading jupyter-1.0.0-py2.py3-none-any.whl
Collecting matplotlib
[...]
```

要检查安装，可以用下面的命令引入每个模块：

```
$ python3 -c "import jupyter, matplotlib, numpy, pandas, scipy, sklearn"
```

这个命令不应该有任何输出和错误。现在你可以用下面的命令打开 Jupyter：

```
$ jupyter notebook
[I 15:24 NotebookApp] Serving notebooks from local directory: [...]/ml
[I 15:24 NotebookApp] 0 active kernels
[I 15:24 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888
[I 15:24 NotebookApp] Use Control-C to stop this server and shut down all
kernels (twice to skip confirmation).
```

Jupyter 服务器现在运行在终端上，监听 8888 端口。你可以用浏览器打开 `http://localhost:8888/`，以访问这个服务器（服务器启动时，通常就自动打开了）。你可以看到一个空的工作空间目录（如果按照先前的 `virtualenv` 步骤，只包含 `env` 目录）。

现在点击按钮 New 创建一个新的 Python 注本，选择合适的 Python 版本（见图 2-3）。

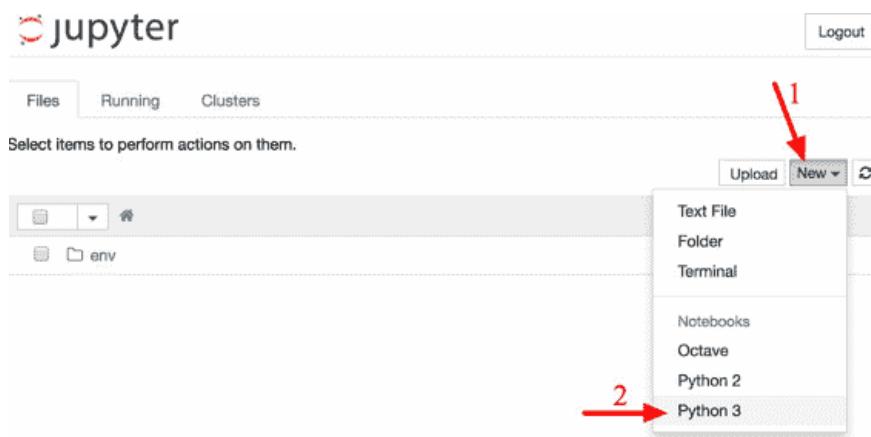


图 2-3 Jupyter 的工作空间

这一步做了三件事：首先，在工作空间中创建了一个新的笔记本文件 `Untitled.ipynb`；第二，它启动了一个 Jupyter 的 Python 内核来运行这个笔记本；第三，在一个新栏中打开这个笔记本。接下来，点击 `Untitled`，将这个笔记本重命名为 `Housing`（这会将 `ipynb` 文件自动命名为 `Housing.ipynb`）。

笔记本包含一组代码框。每个代码框可以放入可执行代码或格式化文本。现在，笔记本只有一个空的代码框，标签是 `In [1]:`。在框中输入 `print("Hello world!")`，点击运行按钮（见图 2-4）或按 `Shift+Enter`。这会将当前的代码框发送到 Python 内核，运行之后会返回输出。结果显示在代码框下方。由于抵达了笔记本的底部，一个新的代码框会被自动创建出来。从 Jupyter 的 Help 菜单中的 User Interface Tour，可以学习 Jupyter 的基本操作。

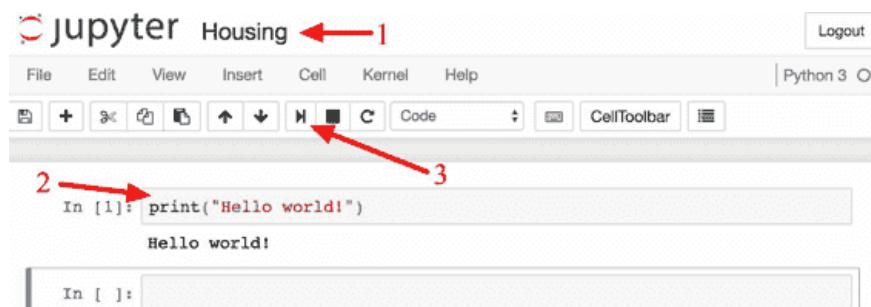


图 2-4 在笔记本中打印 Hello world!

## 下载数据

一般情况下，数据是存储于关系型数据库（或其它常见数据库）中的多个表、文档、文件。要访问数据，你首先要有密码和登录权限，并要了解数据模式。但是在这个项目中，这一切要简单些：只要下载一个压缩文件，`housing.tgz`，它包含一个 CSV 文件 `housing.csv`，含有所有数据。

你可以使用浏览器下载，运行 `tar xzf housing.tgz` 解压出 `csv` 文件，但是更好的办法是写一个小函数来做这件事。如果数据变动频繁，这么做是非常好的，因为可以让你写一个小脚本随时获取最新的数据（或者创建一个定时任务来做）。如果你想在多台机器上安装数据集，获取数据自动化也是非常好的。

下面是获取数据的函数：

```

import os
import tarfile
from six.moves import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
HOUSING_PATH = "datasets/housing"
HOUSING_URL = DOWNLOAD_ROOT + HOUSING_PATH + "/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    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，当你调用 `fetch_housing_data()`，就会在工作空间创建一个 `datasets/housing` 目录，下载 `housing.tgz` 文件，解压出 `housing.csv`。

然后使用 Pandas 加载数据。还是用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。

## 快速查看数据结构

使用 `DataFrame` 的 `head()` 方法查看该数据集的前 5 行（见图 2-5）。

The screenshot shows a Jupyter Notebook interface. In the top cell (In [5]), the code `housing = load_housing_data()` is run, followed by `housing.head()`. In the bottom cell (Out [5]), the resulting DataFrame is displayed as a table with 6 rows and 7 columns. The columns are labeled: longitude, latitude, housing\_median\_age, total\_rooms, total\_bedrooms, population, and income. The first five rows of data are shown:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
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 数据集的前五行

每一行都表示一个街区。共有 10 个属性（截图中可以看到 6 个）：经度、维度、房屋年龄中位数、总房间数、总卧室数、人口数、家庭数、收入中位数、房屋价值中位数、离大海距离。

`info()` 方法可以快速查看数据的描述，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 房屋信息

数据集中共有 20640 个实例，按照机器学习的标准这个数据量很小，但是非常适合入门。我们注意到总卧室数只有 20433 个非空值，这意味着有 207 个街区缺少这个值。我们将在后面对它进行处理。

所有的属性都是数值的，除了离大海距离这项。它的类型是对象，因此可以包含任意 Python 对象，但是因为该项是从 CSV 文件加载的，所以必然是文本类型。在刚才查看数据前五项时，你可能注意到那一列的值是重复的，意味着它可能是一项表示类别的属性。可以使用 `value_counts()` 方法查看该项中都有哪些类别，每个类别中都包含有多少个街区：

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

再来看其它字段。`describe()` 方法展示了数值属性的概括（见图 2-7）。

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000
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	6445.000000

图 2-7 每个数值属性的概括

`count`、`mean`、`min` 和 `max` 几行的意思很明显了。注意，空值被忽略了（所以，卧室总数是 20433 而不是 20640）。`std` 是标准差（揭示数值的分散度）。`25%`、`50%`、`75%` 展示了对应的分位数：每个分位数指明小于这个值，且指定分位数的百分比。例如，`25%` 的街区的房屋年龄中位数小于 18，而 `50%` 的小于 29，`75%` 的小于 37。这些值通常称为第 25 个百分位数（或第一个四分位数），中位数，第 75 个百分位数（第三个四分位数）。

另一种快速了解数据类型的方法是画出每个数值属性的柱状图。柱状图（的纵轴）展示了特定范围的实例的个数。你还可以一次给一个属性画图，或对完整数据集调用 `hist()` 方法，后者会画出每个数值属性的柱状图（见图 2-8）。例如，你可以看到略微超过 800 个街区的 `median_house_value` 值差不多等于 500000 美元。

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

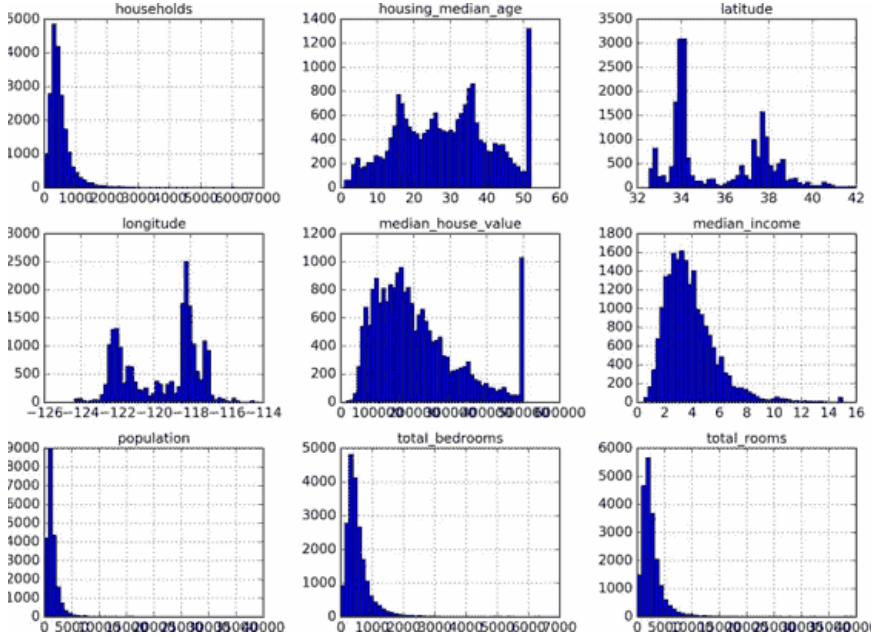


图 2-8 每个数值属性的柱状图

注：`hist()` 方法依赖于 Matplotlib，后者依赖于用户指定的图形后端以打印到屏幕上。因此在画图之前，你要指定 Matplotlib 要使用的后端。最简单的方法是使用 Jupyter 的魔术命令 `%matplotlib inline`。它会告诉 Jupyter 设定好 Matplotlib，以使用 Jupyter 自己的后端。绘图就会在笔记本中渲染了。注意在 Jupyter 中调用 `show()` 不是必要的，因为代码框执行后 Jupyter 会自动展示图像。

注意柱状图中的一些点：

1. 首先，收入中位数貌似不是美元（USD）。与数据采集团队交流之后，你被告知数据是经过缩放调整的，过高收入中位数的会变为 15（实际为 15.0001），过低的会变为 5（实际为 0.4999）。在机器学习中对数据进行预处理很正常，这不一定是个问题，但你要明白数据是如何计算出来的。
2. 房屋年龄中位数和房屋价值中位数也被设了上限。后者可能是个严重的问题，因为它是你的目标属性（你的标签）。你的机器学习算法可能学习到价格不会超出这个界限。你需要与下游团队核实，这是否会成为问题。如果他们告诉你他们需要明确的预测值，即使超过 500000 美元，你则有两个选项：
  - i. 对于设了上限的标签，重新收集合适的标签；
  - ii. 将这些街区从训练集移除（也从测试集移除，因为若房价超出 500000 美元，你的系统就会被差评）。
3. 这些属性值有不同的量度。我们会在本章后面讨论特征缩放。

4. 最后，许多柱状图的尾巴很长：相较于左边，它们在中位数的右边延伸过远。

对于某些机器学习算法，这会使检测规律变得更难些。我们会在后面尝试变换处理这些属性，使其变为正态分布。

希望你现在对要处理的数据有一定了解了。

**警告：**稍等！在你进一步查看数据之前，你需要创建一个测试集，将它放在一旁，千万不要再看它。

## 创建测试集

在这个阶段就分割数据，听起来很奇怪。毕竟，你只是简单快速地查看了数据而已，你需要再仔细调查下数据以决定使用什么算法。这么想是对的，但是人类的大脑是一个神奇的发现规律的系统，这意味着大脑非常容易发生过拟合：如果你查看了测试集，就会不经意地按照测试集中的规律来选择某个特定的机器学习模型。再当你使用测试集来评估误差率时，就会导致评估过于乐观，而实际部署的系统表现就会差。这称为数据透视偏差。

理论上，创建测试集很简单：只要随机挑选一些实例，一般是数据集的 20%，放到一边：

```
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]
```

然后可以像下面这样使用这个函数：

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> print(len(train_set), "train +", len(test_set), "test")
16512 train + 4128 test
```

这个方法可行，但是并不完美：如果再次运行程序，就会产生一个不同的测试集！多次运行之后，你（或你的机器学习算法）就会得到整个数据集，这是需要避免的。

解决的办法之一是保存第一次运行得到的测试集，并在随后的过程加载。另一种方法是在调用 `np.random.permutation()` 之前，设置随机数生成器的种子（比如 `np.random.seed(42)`），以产生总是相同的洗牌指数（shuffled indices）。

但是如果数据集更新，这两个方法都会失效。一个通常的解决办法是使用每个实例的 ID 来判定这个实例是否应该放入测试集（假设每个实例都有唯一并且不变的 ID）。例如，你可以计算出每个实例 ID 的哈希值，只保留其最后一个字节，如果该值小于等于 51（约为 256 的 20%），就将其放入测试集。这样可以保证在多次运行中，测试集保持不变，即使更新了数据集。新的测试集会包含新实例中的 20%，但不会有之前位于训练集的实例。下面是一种可用的方法：

```

import hashlib

def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio

def split_train_test_by_id(data, test_ratio, id_column, hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
    return data.loc[~in_test_set], data.loc[in_test_set]

```

不过，房产数据集没有 ID 这一列。最简单的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识别码，你需要保证新数据都放到现有数据的尾部，且没有行被删除。如果做不到，则可以用最稳定的特征来创建唯一识别码。例如，一个区的维度和经度在几百万年之内是不变的，所以可以将两者结合成一个 ID：

```

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，可以用多种方式将数据集分割成多个子集。最简单的函数是 `train_test_split`，它的作用和之前的函数 `split_train_test` 很像，并带有其它一些功能。首先，它有一个 `random_state` 参数，可以设定前面讲过的随机生成器种子；第二，你可以将种子传递给多个行数相同的数据集，可以在相同的索引上分割数据集（这个功能非常有用，比如你的标签值是放在另一个 `DataFrame` 里的）：

```

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

```

目前为止，我们采用的都是纯随机的取样方法。当你的数据集很大时（尤其是和属性数相比），这通常可行；但如果数据集不大，就会有采样偏差的风险。当一个调查公司想要对 1000 个人进行调查，它们不是在电话亭里随机选 1000 个人出来。调查公司要保证这 1000 个人对人群整体有代表性。例如，美国人口的 51.3% 是女性，48.7% 是男性。所以在美国，严谨的调查需要保证样本也是这个比例：513 名女性，487 名男性。这称作分层采样（stratified sampling）：将人群分成均匀的子分组，称为分层，从每个分层去取合适数量的实例，以保证测试集对总人数有代表性。如果调查公司采用纯随机采样，会有 12% 的概率导致采样偏差：女性人数少于 49%，或多于 54%。不管发生那种情况，调查结果都会严重偏差。

假设专家告诉你，收入中位数是预测房价中位数非常重要的属性。你可能想要保证测试集可以代表整体数据集中的多种收入分类。因为收入中位数是一个连续的数值属性，你首先需要创建一个收入类别属性。再仔细地看一下收入中位数的柱状图（图 2-9）（译注：该图是对收入中位数处理过后的图）：

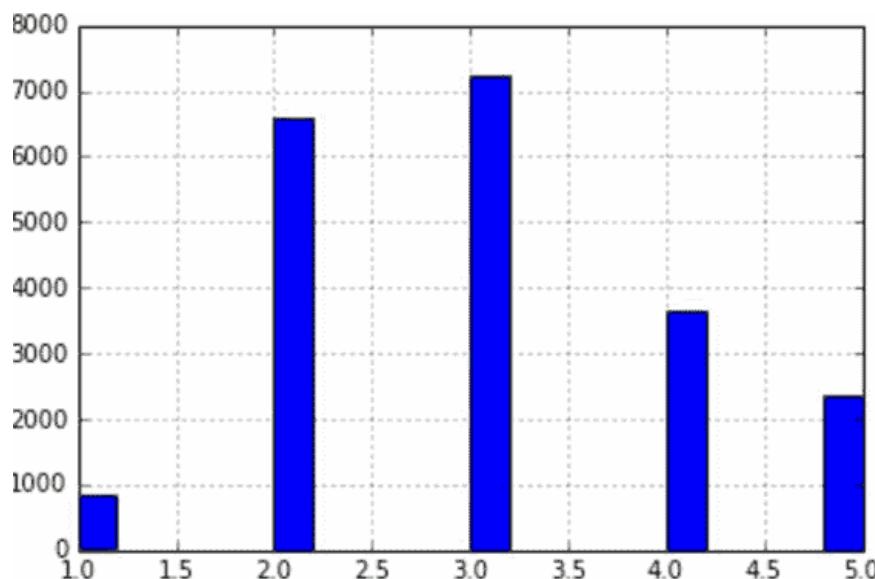


图 2-9 收入分类的柱状图

大多数的收入中位数的值聚集在 2-5（万美元）, 但是一些收入中位数会超过 6。数据集中的每个分层都要有足够的实例位于你的数据中, 这点很重要。否则, 对分层重要性的评估就会有偏差。这意味着, 你不能有过多的分层, 且每个分层都要足够大。后面的代码通过将收入中位数除以 1.5 (以限制收入分类的数量), 创建了一个收入类别属性, 用 `ceil` 对值舍入 (以产生离散的分类), 然后将所有大于 5 的分类归入到分类 5:

```
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)
```

现在, 就可以根据收入分类, 进行分层采样。你可以使用 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]
```

检查下结果是否符合预期。你可以在完整的房产数据集中查看收入分类比例:

```
>>> housing["income_cat"].value_counts() / len(housing)
3.0    0.350581
2.0    0.318847
4.0    0.176308
5.0    0.114438
1.0    0.039826
Name: income_cat, dtype: float64
```

使用相似的代码, 还可以测量测试集中收入分类的比例。图 2-10 对比了总数据集、分层采样的测试集、纯随机采样测试集的收入分类比例。可以看到, 分层采样测试集的收入分类比例与总数据集几乎相同, 而随机采样数据集偏差严重。

	<b>Overall</b>	<b>Random</b>	<b>Stratified</b>	<b>Rand. %error</b>	<b>Strat. %error</b>
<b>1.0</b>	0.039826	0.040213	0.039738	0.973236	-0.219137
<b>2.0</b>	0.318847	0.324370	0.318876	1.732260	0.009032
<b>3.0</b>	0.350581	0.358527	0.350618	2.266446	0.010408
<b>4.0</b>	0.176308	0.167393	0.176399	-5.056334	0.051717
<b>5.0</b>	0.114438	0.109496	0.114369	-4.318374	-0.060464

图 2-10 分层采样和纯随机采样的样本偏差比较

现在，你需要删除 `income_cat` 属性，使数据回到初始状态：

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

我们用了大量时间来生成测试集的原因是：测试集通常被忽略，但实际是机器学习非常重要的一部分。还有，生成测试集过程中的许多思路对于后面的交叉验证讨论是非常有帮助的。接下来进入下一阶段：数据探索。

## 数据探索和可视化、发现规律

目前为止，你只是快速查看了数据，对要处理的数据有了整体了解。现在的目标是更深的探索数据。

首先，保证你将测试集放在了一旁，只是研究训练集。另外，如果训练集非常大，你可能需要再采样一个探索集，保证操作方便快速。在我们的案例中，数据集很小，所以可以在全集上直接工作。创建一个副本，以免损伤训练集：

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

## 地理数据可视化

因为存在地理信息（纬度和经度），创建一个所有街区的散点图来数据可视化是一个不错的主意（图 2-11）：

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

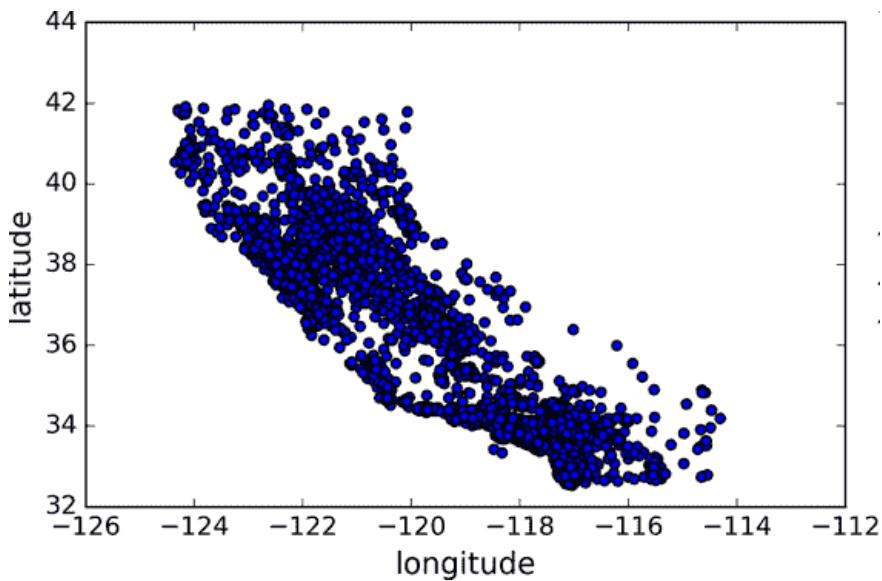


图 2-11 数据的地理信息散点图

这张图看起来很像加州，但是看不出什么特别的规律。将 `alpha` 设为 0.1，可以更容易看出数据点的密度（图 2-12）：

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

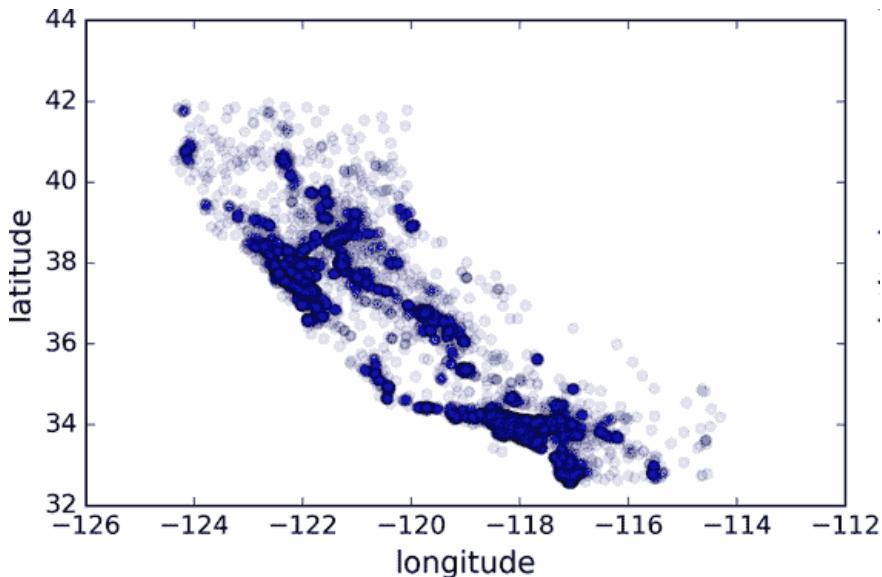


图 2-12 显示高密度区域的散点图

现在看起来好多了：可以非常清楚地看到高密度区域，湾区、洛杉矶和圣迭戈，以及中央谷，特别是从萨克拉门托和弗雷斯诺。

通常来讲，人类的大脑非常善于发现图片中的规律，但是需要调整可视化参数使规律显现出来。

现在来看房价（图 2-13）。每个圈的半径表示街区的人口（选项 `s`），颜色代表价格（选项 `c`）。我们用预先定义的名为 `jet` 的颜色图（选项 `cmap`），它的范围是从蓝色（低价）到红色（高价）：

```

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

```

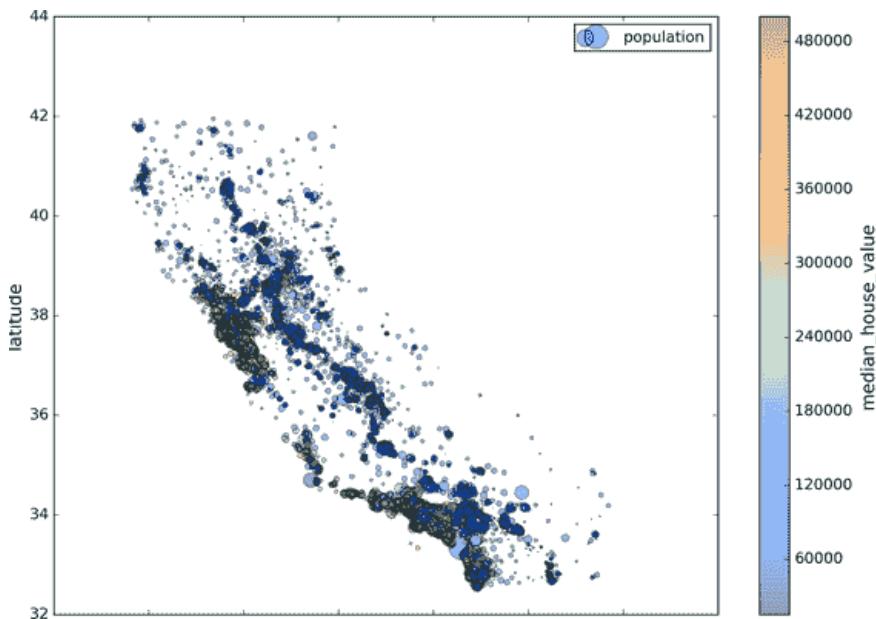


图 2-13 加州房价

这张图说明房价和位置（比如，靠海）和人口密度联系密切，这点你可能早就知道。可以使用聚类算法来检测主要的聚集，用一个新的特征值测量聚集中心的距离。尽管北加州海岸区域的房价不是非常高，但离大海距离属性也可能很有用，所以这不是用一个简单的规则就可以定义的问题。

## 查找关联

因为数据集并不是非常大，你可以很容易地使用 `corr()` 方法计算出每对属性间的标准相关系数（standard correlation coefficient，也称作皮尔逊相关系数）：

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

现在来看下每个属性和房价中位数的关联度：

```

>>> 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

```

相关系数的范围是 -1 到 1。当接近 1 时，意味强正相关；例如，当收入中位数增加时，房价中位数也会增加。当相关系数接近 -1 时，意味强负相关；你可以看到，纬度和房价中位数有轻微的负相关性（即，越往北，房价越可能降低）。最后，相关系数接近 0，意味没有线性相关性。图 2-14 展示了相关系数在横轴和纵轴之间的不同图形。

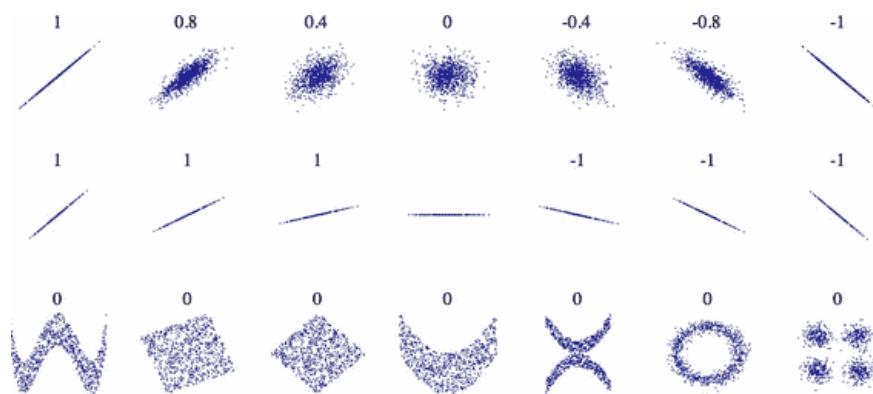


图 2-14 不同数据集的标准相关系数（来源：Wikipedia；公共领域图片）

警告：相关系数只测量线性关系（如果  $x$  上升， $y$  则上升或下降）。相关系数可能会完全忽略非线性关系（例如，如果  $x$  接近 0，则  $y$  值会变高）。在上面图片的最后一行中，他们的相关系数都接近于 0，尽管它们的轴并不独立：这些就是非线性关系的例子。另外，第二行的相关系数等于 1 或 -1；这和斜率没有任何关系。例如，你的身高（单位是英寸）与身高（单位是英尺或纳米）的相关系数就是 1。

另一种检测属性间相关系数的方法是使用 Pandas 的 `scatter_matrix` 函数，它能画出每个数值属性对每个其它数值属性的图。因为现在共有 11 个数值属性，你可以得到  $11 * 2 = 121$  张图，在一页上画不下，所以只关注几个和房价中位数最有可能相关的属性（图 2-15）：

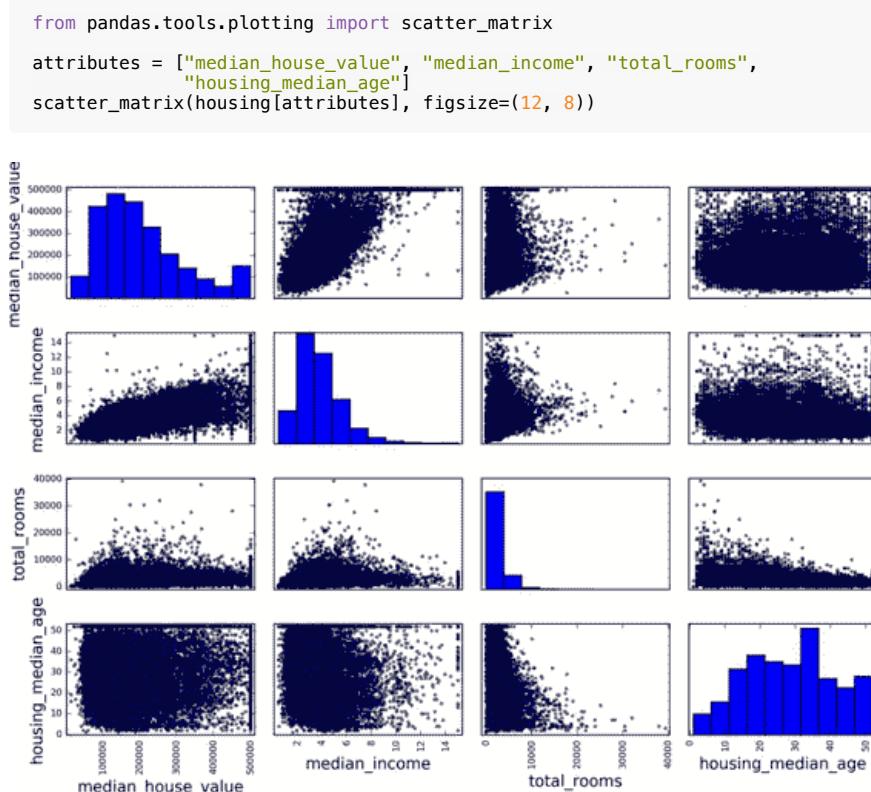


图 2-15 散点矩阵

如果 pandas 将每个变量对自己作图，主对角线（左上到右下）都会是直线图。所以 Pandas 展示的是每个属性的柱状图（也可以是其它的，请参考 Pandas 文档）。

最有希望用来预测房价中位数的属性是收入中位数，因此将这张图放大（图 2-16）：

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

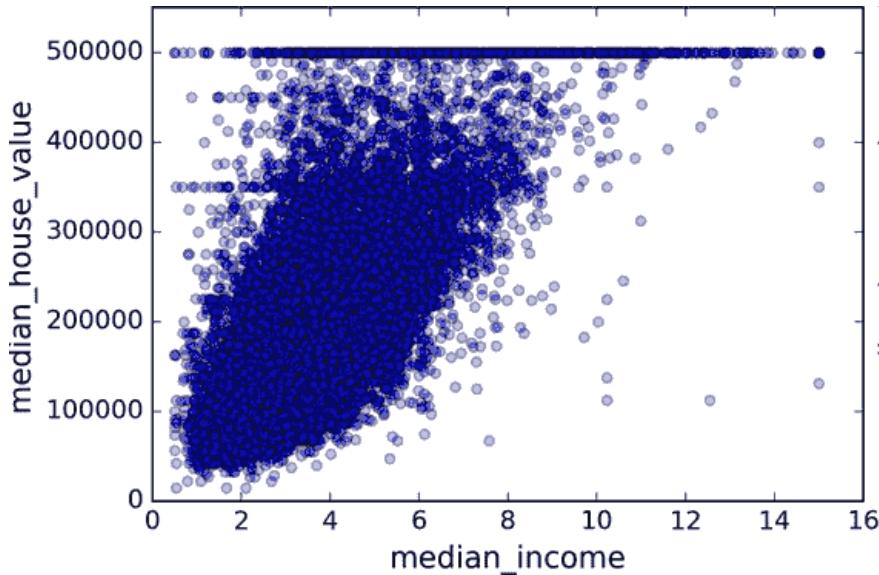


图 2-16 收入中位数 vs 房价中位数

这张图说明了几点。首先，相关性非常高；可以清晰地看到向上的趋势，并且数据点不是非常分散。第二，我们之前看到的最高价，清晰地呈现为一条位于 500000 美元的水平线。这张图也呈现了一些不是那么明显的直线：一条位于 450000 美元的直线，一条位于 350000 美元的直线，一条在 280000 美元的线，和一些更靠下的线。你可能希望去除对应的街区，以防止算法重复这些巧合。

## 属性组合试验

希望前面的一节能教给你一些探索数据、发现规律的方法。你发现了一些数据的巧合，需要在给算法提供数据之前，将其去除。你还发现了一些属性间有趣的关联，特别是目标属性。你还注意到一些属性具有长尾分布，因此你可能要将其进行转换（例如，计算其 log 对数）。当然，不同项目的处理方法各不相同，但大体思路是相似的。

给算法准备数据之前，你需要做的最后一件事是尝试多种属性组合。例如，如果你不知道某个街区有多少户，该街区的总房间数就没什么用。你真正需要的是每户有几个房间。相似的，总卧室数也不重要：你可能需要将其与房间数进行比较。每户的人口数也是一个有趣的属性组合。让我们来创建这些新的属性：

```
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"]
```

现在，再来看相关矩阵：

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value      1.000000
median_income          0.687170
rooms_per_household   0.199343
total_rooms            0.135231
housing_median_age     0.114220
households             0.064702
total_bedrooms          0.047865
population_per_household -0.021984
population              -0.026699
longitude                -0.047279
latitude                 -0.142826
bedrooms_per_room       -0.260070
Name: median_house_value, dtype: float64
```

看起来不错！与总房间数或卧室数相比，新的 `bedrooms_per_room` 属性与房价中位数的关联更强。显然，卧室数/总房间数的比例越低，房价就越高。每户的房间数也比街区的总房间数的更有信息，很明显，房屋越大，房价就越高。

这一步的数据探索不必非常完备，此处的目的是有一个正确的开始，快速发现规律，以得到一个合理的原型。但是这是一个交互过程：一旦你得到了一个原型，并运行起来，你就可以分析它的输出，进而发现更多的规律，然后再回到数据探索这步。

## 为机器学习算法准备数据

现在来为机器学习算法准备数据。不要手工来做，你需要写一些函数，理由如下：

- 函数可以让你在任何数据集上（比如，你下一次获取的是一个新的数据集）方便地进行重复数据转换。
- 你能慢慢建立一个转换函数库，可以在未来的项目中复用。
- 在将数据传给算法之前，你可以在实时系统中使用这些函数。
- 这可以让你方便地尝试多种数据转换，查看哪些转换方法结合起来效果最好。

但是，还是先回到干净的训练集（通过再次复制 `strat_train_set`），将预测量和标签分开，因为我们不想对预测量和目标值应用相同的转换（注意 `drop()` 创建了一份数据的备份，而不影响 `strat_train_set`）：

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

## 数据清洗

大多机器学习算法不能处理缺失的特征，因此先创建一些函数来处理特征缺失的问题。前面，你应该注意到了属性 `total_bedrooms` 有一些缺失值。有三个解决选项：

- 去掉对应的街区；
- 去掉整个属性；
- 进行赋值（0、平均值、中位数等等）。

用 `DataFrame` 的 `dropna()`，`drop()`，和 `fillna()` 方法，可以方便地实现：

```

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

```

如果选择选项 3，你需要计算训练集的中位数，用中位数填充训练集的缺失值，不要忘记保存该中位数。后面用测试集评估系统时，需要替换测试集中的缺失值，也可以用来实时替换新数据中的缺失值。

Scikit-Learn 提供了一个方便的类来处理缺失值：`Imputer`。下面是其使用方法：首先，需要创建一个 `Imputer` 实例，指定用某属性的中位数来替换该属性所有的缺失值：

```

from sklearn.preprocessing import Imputer
imputer = Imputer(strategy="median")

```

因为只有数值属性才能算出中位数，我们需要创建一份不包括文本属性 `ocean_proximity` 的数据副本：

```

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

```

现在，就可以用 `fit()` 方法将 `imputer` 实例拟合到训练数据：

```

imputer.fit(housing_num)

```

`imputer` 计算了每个属性的中位数，并将结果保存在了实例变量 `statistics_` 中。虽然此时只有属性 `total_bedrooms` 存在缺失值，但我们不能确定在以后的新数据中会不会有其他属性也存在缺失值，所以安全的做法是将 `imputer` 应用到每个数值：

```

>>> imputer.statistics_
array([-118.51, 34.26, 29., 2119., 433., 1164., 408., 3.5414])
>>> housing_num.median().values
array([-118.51, 34.26, 29., 2119., 433., 1164., 408., 3.5414])

```

现在，你就可以使用这个“训练过的” `imputer` 来对训练集进行转换，将缺失值替换为中位数：

```

X = imputer.transform(housing_num)

```

结果是一个包含转换后特征的普通的 Numpy 数组。如果你想将其放回到 Pandas DataFrame 中，也很简单：

```

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

```

### Scikit-Learn 设计

Scikit-Learn 设计的 API 设计的非常好。它的主要设计原则是：

- 一致性：所有对象的接口一致且简单：
  - 估计器（estimator）。任何可以基于数据集对一些参数进行估计的对象都被称为估计器（比如，`imputer` 就是个估计器）。估计本身是通过 `fit()` 方法，只需要一个数据集作为参数（对于监督学习算法，需要两个数据集；第二个数据集包含标签）。任何其它用来指导估计过程的参数都被当做超参数（比如 `imputer` 的 `strategy`），并且超参数要被设置成实例变量（通常通过构造器参数设置）。
  - 转换器（transformer）。一些估计器（比如 `imputer`）也可以转换数据集，这些估计器被称为转换器。API 也是相当简单：转换是通过 `transform()` 方法，被转换的数据集作为参数。返回的是经过转换的数据集。转换过程依赖学习到的参数，比如 `imputer` 的例子。所有的转换都有一个便捷的方法 `fit_transform()`，等同于调用 `fit()` 再 `transform()`（但有时 `fit_transform()` 经过优化，运行的更快）。
  - 预测器（predictor）。最后，一些估计器可以根据给出的数据集做预测，这些估计器称为预测器。例如，上一章的 `LinearRegression` 模型就是一个预测器：它根据一个国家的人均 GDP 预测生活满意度。预测器有一个 `predict()` 方法，可以用新实例的数据集做出相应的预测。预测器还有一个 `score()` 方法，可用于评估测试集（如果是监督学习算法的话，还要给出相应的标签）的预测质量。
- 可检验。所有估计器的超参数都可以通过实例的公共变量直接访问（比如，`imputer.strategy`），并且所有估计器学习到的参数也可以通过在实例变量名后加下划线来访问（比如，`imputer.statistics_`）。
- 类不可扩散。数据集被表示成 NumPy 数组或 SciPy 稀疏矩阵，而不是自制的类。超参数只是普通的 Python 字符串或数字。
- 可组合。尽可能使用现存的模块。例如，用任意的转换器序列加上一个估计器，就可以做成一个流水线，后面会看到例子。
- 合理的默认值。Scikit-Learn 给大多数参数提供了合理的默认值，很容易就能创建一个系统。

## 处理文本和类别属性

前面，我们丢弃了类别属性 `ocean_proximity`，因为它是一个文本属性，不能计算出中位数。大多数机器学习算法更喜欢和数字打交道，所以让我们把这些文本标签转换为数字。

Scikit-Learn 为这个任务提供了一个转换器 `LabelEncoder`：

```
>>> from sklearn.preprocessing import LabelEncoder
>>> encoder = LabelEncoder()
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat_encoded = encoder.fit_transform(housing_cat)
>>> housing_cat_encoded
array([1, 1, 4, ..., 1, 0, 3])
```

译注:

在原书中使用 `LabelEncoder` 转换器来转换文本特征列的方式是错误的，该转换器只能用来转换标签（正如其名）。在这里使用 `LabelEncoder` 没有出错的原因是该数据只有一列文本特征值，在有多个文本特征列的时候就会出错。应使用 `factorize()` 方法来进行操作：

```
housing_cat_encoded, housing_categories = housing_cat.factorize()
housing_cat_encoded[:10]
```

好了一些，现在就可以在任何 ML 算法里用这个数值数据了。你可以查看映射表，编码器是通过属性 `classes_` 来学习的（`<1H OCEAN` 被映射为 0，`INLAND` 被映射为 1，等等）：

```
>>> print(encoder.classes_)
['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']
```

这种做法的问题是，ML 算法会认为两个临近的值比两个疏远的值要更相似。显然这样不对（比如，分类 0 和分类 4 就比分类 0 和分类 1 更相似）。要解决这个问题，一个常见的方法是给每个分类创建一个二元属性：当分类是 `<1H OCEAN`，该属性为 1（否则为 0），当分类是 `INLAND`，另一个属性等于 1（否则为 0），以此类推。这称作独热编码（One-Hot Encoding），因为只有一个属性会等于 1（热），其余会是 0（冷）。

Scikit-Learn 提供了一个编码器 `OneHotEncoder`，用于将整数分类值转变为独热向量。注意 `fit_transform()` 用于 2D 数组，而 `housing_cat_encoded` 是一个 1D 数组，所以需要将其变形：

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> encoder = OneHotEncoder()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
>>> housing_cat_1hot
<16513x5 sparse matrix of type '<class 'numpy.float64'>'>
    with 16513 stored elements in Compressed Sparse Row format>
```

注意输出结果是一个 SciPy 稀疏矩阵，而不是 NumPy 数组。当类别属性有数千个分类时，这样非常有用。经过独热编码，我们得到了一个有数千列的矩阵，这个矩阵每行只有一个 1，其余都是 0。使用大量内存来存储这些 0 非常浪费，所以稀疏矩阵只存储非零元素的位置。你可以像一个 2D 数据那样进行使用，但是如果你真的想将其转变成一个（密集的）NumPy 数组，只需调用 `toarray()` 方法：

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

使用类 `LabelBinarizer`，我们可以一步执行这两个转换（从文本分类到整数分类，再从整数分类到独热向量）：

```
>>> from sklearn.preprocessing import LabelBinarizer
>>> encoder = LabelBinarizer()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
array([[0, 1, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 0, 0, 1],
       ...,
       [0, 1, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 0, 0, 1, 0]])
```

注意默认返回的结果是一个密集 NumPy 数组。向构造器 `LabelBinarizer` 传递 `sparse_output=True`，就可以得到一个稀疏矩阵。

译注:

在原书中使用 `LabelBinarizer` 的方式也是错误的，该类也应用于标签列的转换。正确做法是使用 `sklearn` 即将提供的 `CategoricalEncoder` 类。如果在你阅读此文时 `sklearn` 中尚未提供此类，用如下方式代替：（来自 [Pull Request #9151](#)）

```

# Definition of the CategoricalEncoder class, copied from PR #9151.
# Just run this cell, or copy it to your code, do not try to understand

from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.utils import check_array
from sklearn.preprocessing import LabelEncoder
from scipy import sparse

class CategoricalEncoder(BaseEstimator, TransformerMixin):
    """Encode categorical features as a numeric array.
    The input to this transformer should be a matrix of integers or strings
    denoting the values taken on by categorical (discrete) features.
    The features can be encoded using a one-hot aka one-of-K scheme
    (`encoding='onehot'`, the default) or converted to ordinal integer
    (`encoding='ordinal'`).
    This encoding is needed for feeding categorical data to many scikit-
    estimators, notably linear models and SVMs with the standard kernels
    Read more in the :ref:`User Guide <preprocessing_categorical_feature
    Parameters
    -----
    encoding : str, 'onehot', 'onehot-dense' or 'ordinal'
        The type of encoding to use (default is 'onehot'):
        - 'onehot': encode the features using a one-hot aka one-of-K scheme
        (or also called 'dummy' encoding). This creates a binary column
        for each category and returns a sparse matrix.
        - 'onehot-dense': the same as 'onehot' but returns a dense array
        instead of a sparse matrix.
        - 'ordinal': encode the features as ordinal integers. This results
        in a single column of integers (0 to n_categories - 1) per feature
        categories : 'auto' or a list of lists/arrays of values.
        Categories (unique values) per feature:
        - 'auto' : Determine categories automatically from the training
        - list : ``categories[i]`` holds the categories expected in the
        column. The passed categories are sorted before encoding the data
        (used categories can be found in the ``categories_`` attribute
    dtype : number type, default np.float64
        Desired dtype of output.
    handle_unknown : 'error' (default) or 'ignore'
        Whether to raise an error or ignore if a unknown categorical feature
        is present during transform (default is to raise). When this is set to
        'ignore' and an unknown category is encountered during transform,
        the resulting one-hot encoded columns for this feature
        will be all zeros.
        Ignoring unknown categories is not supported for
        ``encoding='ordinal'``.
    Attributes
    -----
    categories_ : list of arrays
        The categories of each feature determined during fitting. When
        categories were specified manually, this holds the sorted categories
        (in order corresponding with output of `transform`).
    Examples
    -----
    Given a dataset with three features and two samples, we let the encoder
    find the maximum value per feature and transform the data to a binary
    one-hot encoding.
    >>> from sklearn.preprocessing import CategoricalEncoder
    >>> enc = CategoricalEncoder(handle_unknown='ignore')
    >>> enc.fit([[0, 0, 3], [1, 1, 0], [0, 2, 1], [1, 0, 2]])
    ... # doctest: +ELLIPSIS
    CategoricalEncoder(categories='auto', dtype=<... 'numpy.float64'>,
                       encoding='onehot', handle_unknown='ignore')
    >>> enc.transform([[0, 1, 1], [1, 0, 4]]).toarray()
    array([[ 1.,  0.,  0.,  1.,  0.,  0.,  1.,  0.,  0.],
           [ 0.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.]])
    See also
    -----
    sklearn.preprocessing.OneHotEncoder : performs a one-hot encoding of
        integer ordinal features. The ``OneHotEncoder assumes`` that input
        features take on values in the range ``[0, max(feature)]`` instead
        using the unique values.
    sklearn.feature_extraction.DictVectorizer : performs a one-hot encoding
        of dictionary items (also handles string-valued features).
    sklearn.feature_extraction.FeatureHasher : performs an approximate one-
        hot encoding of dictionary items or strings.
    ......

    def __init__(self, encoding='onehot', categories='auto', dtype=np.float64,
                 handle_unknown='error'):
        self.encoding = encoding
        self.categories = categories
        self.dtype = dtype
        self.handle_unknown = handle_unknown

    def fit(self, X, y=None):
        """Fit the CategoricalEncoder to X.
        Parameters
        -----
        X : array-like, shape [n_samples, n_features]
            The data to determine the categories of each feature.
        Returns
        -----

```

```

-----
self
"""

if self.encoding not in ['onehot', 'onehot-dense', 'ordinal']:
    template = ("encoding should be either 'onehot', 'onehot-den"
                "or 'ordinal', got %s")
    raise ValueError(template % self.handle_unknown)

if self.handle_unknown not in ['error', 'ignore']:
    template = ("handle_unknown should be either 'error' or "
                "'ignore', got %s")
    raise ValueError(template % self.handle_unknown)

if self.encoding == 'ordinal' and self.handle_unknown == 'ignore':
    raise ValueError("handle_unknown='ignore' is not supported f"
                     "or encoding='ordinal'")

X = check_array(X, dtype=np.object, accept_sparse='csc', copy=True)
n_samples, n_features = X.shape

self._label_encoders_ = [LabelEncoder() for _ in range(n_features)]
for i in range(n_features):
    le = self._label_encoders_[i]
    Xi = X[:, i]
    if self.categories == 'auto':
        le.fit(Xi)
    else:
        valid_mask = np.in1d(Xi, self.categories[i])
        if not np.all(valid_mask):
            if self.handle_unknown == 'error':
                diff = np.unique(Xi[~valid_mask])
                msg = ("Found unknown categories {0} in column {1}"
                       " during fit".format(diff, i))
                raise ValueError(msg)
            le.classes_ = np.array(np.sort(self.categories[i]))
    self.categories_ = [le.classes_ for le in self._label_encoders_]

return self

def transform(self, X):
    """Transform X using one-hot encoding.

    Parameters
    -----
    X : array-like, shape [n_samples, n_features]
        The data to encode.
    Returns
    -----
    X_out : sparse matrix or a 2-d array
        Transformed input.
    """
    X = check_array(X, accept_sparse='csc', dtype=np.object, copy=True)
    n_samples, n_features = X.shape
    X_int = np.zeros_like(X, dtype=np.int)
    X_mask = np.ones_like(X, dtype=np.bool)

    for i in range(n_features):
        valid_mask = np.in1d(X[:, i], self.categories_[i])

        if not np.all(valid_mask):
            if self.handle_unknown == 'error':
                diff = np.unique(X[~valid_mask, i])
                msg = ("Found unknown categories {0} in column {1}"
                       " during transform".format(diff, i))
                raise ValueError(msg)
            else:
                # Set the problematic rows to an acceptable value and
                # continue. The rows are marked `X_mask` and will be
                # removed later.
                X_mask[:, i] = valid_mask
                X[:, i][~valid_mask] = self.categories_[i][0]
        X_int[:, i] = self._label_encoders_[i].transform(X[:, i])

    if self.encoding == 'ordinal':
        return X_int.astype(self.dtype, copy=False)

    mask = X_mask.ravel()
    n_values = [cats.shape[0] for cats in self.categories_]
    n_values = np.array([0] + n_values)
    indices = np.cumsum(n_values)

    column_indices = (X_int + indices[:-1]).ravel()[mask]
    row_indices = np.repeat(np.arange(n_samples, dtype=np.int32),
                           n_features)[mask]
    data = np.ones(n_samples * n_features)[mask]

    out = sparse.csc_matrix((data, (row_indices, column_indices)),
                           shape=(n_samples, indices[-1]),
                           dtype=self.dtype).tocsr()

    if self.encoding == 'onehot-dense':

```

```

        return out.toarray()
else:
    return out

```

转换方法：

```

#from sklearn.preprocessing import CategoricalEncoder # in future version
cat_encoder = CategoricalEncoder()
housing_cat_reshaped = housing_cat.values.reshape(-1, 1)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat_reshaped)
housing_cat_1hot

```

## 自定义转换器

尽管 Scikit-Learn 提供了许多有用的转换器，你还是需要自己动手写转换器执行任务，比如自定义的清理操作，或属性组合。你需要让自制的转换器与 Scikit-Learn 组件（比如流水线）无缝衔接工作，因为 Scikit-Learn 是依赖鸭子类型的（而不是继承），你所需要做的是创建一个类并执行三个方法：`fit()`（返回 `self`），`transform()`，和 `fit_transform()`。通过添加 `TransformerMixin` 作为基类，可以很容易地得到最后一个。另外，如果你添加 `BaseEstimator` 作为基类（且构造器中避免使用 `*args` 和 `**kwargs`），你就能得到两个额外的方法（`get_params()` 和 `set_params()`），二者可以方便地进行超参数自动微调。例如，一个小转换器类添加了上面讨论的属性：

```

from sklearn.base import BaseEstimator, TransformerMixin
rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room=True): # no *args or **kwargs
        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, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_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)

```

在这个例子中，转换器有一个超参数 `add_bedrooms_per_room`，默认设为 `True`（提供一个合理的默认值很有帮助）。这个超参数可以让你方便地发现添加了这个属性是否对机器学习算法有帮助。更一般地，你可以为每个不能完全确保的数据准备步骤添加一个超参数。数据准备步骤越自动化，可以自动化的操作组合就越多，越容易发现更好用的组合（并能节省大量时间）。

## 特征缩放

数据要做的最重要的转换之一是特征缩放。除了个别情况，当输入的数值属性量度不同时，机器学习算法的性能都不会好。这个规律也适用于房产数据：总房间数分布范围是 6 到 39320，而收入中位数只分布在 0 到 15。注意通常情况下我们不需要对目标值进行缩放。

有两种常见的方法可以让所有的属性有相同的量度：线性函数归一化（Min-Max scaling）和标准化（standardization）。

线性函数归一化（许多人称其为归一化（normalization））很简单：值被转变、重新缩放，直到范围变成 0 到 1。我们通过减去最小值，然后再除以最大值与最小值的差值，来进行归一化。Scikit-Learn 提供了一个转换器 `MinMaxScaler` 来实现这个功能。它有一个超参数 `feature_range`，可以让你改变范围，如果不希望范围是 0 到 1。

标准化就很不同：首先减去平均值（所以标准化值的平均值总是 0），然后除以方差，使得得到的分布具有单位方差。与归一化不同，标准化不会限定值到某个特定的范围，这对某些算法可能构成问题（比如，神经网络常需要输入值得范围是 0 到 1）。但是，标准化受到异常值的影响很小。例如，假设一个街区的收入中位数由于某种错误变成了 100，归一化会将其它范围是 0 到 15 的值变为 0-0.15，但是标准化不会受什么影响。Scikit-Learn 提供了一个转换器 `StandardScaler` 来进行标准化。

**警告：**与所有的转换一样，缩放器只能向训练集拟合，而不是向完整的数据集（包括测试集）。只有这样，你才能用缩放器转换训练集和测试集（和新数据）。

## 转换流水线

你已经看到，存在许多数据转换步骤，需要按一定的顺序执行。幸运的是，Scikit-Learn 提供了类 `Pipeline`，来进行这一系列的转换。下面是一个数值属性的小流水线：

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

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

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

`Pipeline` 构造器需要一个定义步骤顺序的名字/估计器对的列表。除了最后一个估计器，其余都要是转换器（即，它们都要有 `fit_transform()` 方法）。名字可以随意起。

当你调用流水线的 `fit()` 方法，就会对所有转换器顺序调用 `fit_transform()` 方法，将每次调用的输出作为参数传递给下一个调用，一直到最后一个估计器，它只执行 `fit()` 方法。

流水线暴露相同的方法作为最终的估计器。在这个例子中，最后的估计器是一个 `StandardScaler`，它是一个转换器，因此这个流水线有一个 `transform()` 方法，可以顺序对数据做所有转换（它还有一个 `fit_transform` 方法可以使用，就不必先调用 `fit()` 再进行 `transform()`）。

如果不需要手动将 Pandas `DataFrame` 中的数值列转成 Numpy 数组的格式，而可以直接将 `DataFrame` 输入 `pipeline` 中进行处理就好了。Scikit-Learn 没有工具来处理 Pandas `DataFrame`，因此我们需要写一个简单的自定义转换器来做这项工作：

```

from sklearn.base import BaseEstimator, TransformerMixin

class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values

```

每个子流水线都以一个选择转换器开始：通过选择对应的属性（数值或分类） 、丢弃其它的，来转换数据，并将输出 DataFrame 转变成一个 NumPy 数组。这样，你就可以很简单的写出一个以 Pandas DataFrame 为输入并且可以处理数值的流水线：该流水线从 DataFrameSelector 开始获取数值属性，前面讨论过的其他数据处理步骤紧随其后。并且你也可以通过使用 DataFrameSelector 选择类别属性并为其写另一个流水线然后应用 LabelBinarizer .

你现在就有了一个对数值的流水线，你还需要对分类值应用 LabelBinarizer : 如何将这些转换写成一个流水线呢？Scikit-Learn 提供了一个类 FeatureUnion 实现这个功能。你给它一列转换器（可以是所有的转换器），当调用它的 transform() 方法，每个转换器的 transform() 会被并行执行，等待输出，然后将输出合并起来，并返回结果（当然，调用它的 fit() 方法就会调用每个转换器的 fit() ）。一个完整的处理数值和类别属性的流水线如下所示：

```

from sklearn.pipeline import FeatureUnion

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

num_pipeline = Pipeline([
    ('selector', DataFrameSelector(num_attribs)),
    ('imputer', Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])

cat_pipeline = Pipeline([
    ('selector', DataFrameSelector(cat_attribs)),
    ('label_binarizer', LabelBinarizer()),
])

full_pipeline = FeatureUnion(transformer_list=[
    ("num_pipeline", num_pipeline),
    ("cat_pipeline", cat_pipeline),
])

```

译注:

如果你在上面代码中的 cat\_pipeline 流水线使用 LabelBinarizer 转换器会导致执行错误，解决方案是用上文提到的 CategoricalEncoder 转换器来代替：

```

cat_pipeline = Pipeline([
    ('selector', DataFrameSelector(cat_attribs)),
    ('cat_encoder', CategoricalEncoder(encoding="onehot-dense")),
])

```

你可以很简单地运行整个流水线：

```

>>> housing_prepared = full_pipeline.fit_transform(housing)
>>> housing_prepared
array([[ 0.73225807, -0.67331551,  0.58426443, ...,  0.,
         0.,      0.], ...,
       [-0.99102923,  1.63234656, -0.92655887, ...,  0.,
         0.,      0.], ...])
>>> housing_prepared.shape
(16513, 17)

```

## 选择并训练模型

可到这一步了！你在前面限定了问题、获得了数据、探索了数据、采样了一个测试集、写了自动化的转换流水线来清理和为算法准备数据。现在，你已经准备好选择并训练一个机器学习模型了。

### 在训练集上训练和评估

好消息是基于前面的工作，接下来要做的比你想的要简单许多。像前一章那样，我们先来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)
```

完毕！你现在就有了一个可用的线性回归模型。用一些训练集中的实例做下验证：

```
>>> some_data = housing.iloc[:5]
>>> some_labels = housing_labels.iloc[:5]
>>> some_data_prepared = full_pipeline.transform(some_data)
>>> print("Predictions:\t", lin_reg.predict(some_data_prepared))
Predictions: [ 303104. 44800. 308928. 294208. 368704.]
>>> print("Labels:\t\t", list(some_labels))
Labels: [359400.0, 69700.0, 302100.0, 301300.0, 351900.0]
```

行的通，尽管预测并不怎么准确（比如，第二个预测偏离了 50%！）。让我们使用 Scikit-Learn 的 `mean_squared_error` 函数，用全部训练集来计算下这个回归模型的 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
68628.413493824875
```

OK，有总比没有强，但显然结果并不好：大多数街区的 `median_housing_values` 位于 120000 到 265000 美元之间，因此预测误差 68628 美元不能让人满意。这是一个模型欠拟合训练数据的例子。当这种情况发生时，意味着特征没有提供足够多的信息来做出一个好的预测，或者模型并不强大。就像前一章看到的，修复欠拟合的主要方法是选择一个更强大的模型，给训练算法提供更好的特征，或去掉模型上的限制。这个模型还没有正则化，所以排除了最后一个选项。你可以尝试添加更多特征（比如，人口的对数值），但是首先让我们尝试一个更为复杂的模型，看看效果。

来训练一个 `DecisionTreeRegressor`。这是一个强大的模型，可以发现数据中复杂的非线性关系（决策树会在第 6 章详细讲解）。代码看起来很熟悉：

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

现在模型就训练好了，用训练集评估下：

```
>>> 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
```

等一下，发生了什么？没有误差？这个模型可能是绝对完美的吗？当然，更大可能性是这个模型严重过拟合数据。如何确定呢？如前所述，直到你准备运行一个具备足够信心的模型，都不要碰测试集，因此你需要使用训练集的部分数据来做训练，用一部分来做模型验证。

## 使用交叉验证做更佳的评估

评估决策树模型的一种方法是用函数 `train_test_split` 来分割训练集，得到一个更小的训练集和一个验证集，然后用更小的训练集来训练模型，用验证集来评估。这需要一定工作量，并不难而且也可行。

另一种更好的方法是使用 Scikit-Learn 的交叉验证功能。下面的代码采用了 K 折交叉验证（K-fold cross-validation）：它随机地将训练集分成十个不同的子集，成为“折”，然后训练评估决策树模型 10 次，每次选一个不用的折来做评估，用其它 9 个来做训练。结果是一个包含 10 个评分的数组：

```
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)
```

警告：Scikit-Learn 交叉验证功能期望的是效用函数（越大越好）而不是损失函数（越低越好），因此得分函数实际上与 MSE 相反（即负值），这就是为什么前面的代码在计算平方根之前先计算 `-scores`。

来看下结果：

```
>>> def display_scores(scores):
...     print("Scores:", scores)
...     print("Mean:", scores.mean())
...     print("Standard deviation:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Scores: [ 74678.4916885   64766.2398337   69632.86942005  69166.67693232
         71486.76507766  73321.65695983  71860.04741226  71086.32691692
         76934.2726093   69060.93319262]
Mean: 71199.4280043
Standard deviation: 3202.70522793
```

现在决策树就不像前面看起来那么好了。实际上，它看起来比线性回归模型还糟！注意到交叉验证不仅可以让你得到模型性能的评估，还能测量评估的准确性（即，它的标准差）。决策树的评分大约是 71200，通常波动有  $\pm 3200$ 。如果只有一个验证集，就得不到这些信息。但是交叉验证的代价是训练了模型多次，不可能总是这样。

让我们计算下线性回归模型的相同分数，以做确保：

```
>>> 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: [ 70423.5893262  65804.84913139  66620.84314068  72510.11362141
          66414.74423281  71958.89083606  67624.90198297  67825.36117664
          72512.36533141  68028.11688067]
Mean: 68972.377566
Standard deviation: 2493.98819069
```

判断没错：决策树模型过拟合很严重，它的性能比线性回归模型还差。

现在再尝试最后一个模型：`RandomForestRegressor`。第 7 章我们会看到，随机森林是通过用特征的随机子集训练许多决策树。在其它多个模型之上建立模型称为集成学习（Ensemble Learning），它是推进 ML 算法的一种好方法。我们会跳过大部分的代码，因为代码本质上和其它模型一样：

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
...
>>> [...]
>>> forest_rmse
22542.396440343684
>>> display_scores(forest_rmse_scores)
Scores: [ 53789.2879722  50256.1980622  52521.55342602  53237.44937943
          52428.82176158  55854.61222549  52158.02291609  50093.66125649
          53240.80406125  52761.50852822]
Mean: 52634.1919593
Standard deviation: 1576.20472269
```

现在好多了：随机森林看起来很有希望。但是，训练集的评分仍然比验证集的评分低很多。解决过拟合可以通过简化模型，给模型加限制（即，规整化），或用更多的训练数据。在深入随机森林之前，你应该尝试下机器学习算法的其它类型模型（不同核心的支持向量机，神经网络，等等），不要在调节超参数上花费太多时间。目标是列出一个可能模型的列表（两到五个）。

提示：你要保存每个试验过的模型，以便后续可以再用。要确保有超参数和训练参数，以及交叉验证评分，和实际的预测值。这可以让你比较不同类型模型的评分，还可以比较误差种类。你可以用 Python 的模块 `pickle`，非常方便地保存 Scikit-Learn 模型，或使用 `sklearn.externals.joblib`，后者序列化大 NumPy 数组更有效率：

```
from sklearn.externals import joblib
joblib.dump(my_model, "my_model.pkl")
# 然后
my_model_loaded = joblib.load("my_model.pkl")
```

## 模型微调

假设你现在有了一个列表，列表里有几个有希望的模型。你现在需要对它们进行微调。让我们来看几种微调的方法。

### 网格搜索

微调的一种方法是手工调整超参数，直到找到一个好的超参数组合。这么做的话会非常冗长，你也可能没有时间探索多种组合。

你应该使用 Scikit-Learn 的 `GridSearchCV` 来做这项搜索工作。你所需要做的是告诉 `GridSearchCV` 要试验有哪些超参数，要试验什么值，`GridSearchCV` 就能用交叉验证试验所有可能超参数值的组合。例如，下面的代码搜索了 `RandomForestRegressor` 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')
grid_search.fit(housing_prepared, housing_labels)
```

当你不能确定超参数该有什么值，一个简单的方法是尝试连续的 10 的幂（如果想要一个粒度更小的搜寻，可以用更小的数，就像在这个例子中对超参数 `n_estimators` 做的）。

`param_grid` 告诉 Scikit-Learn 首先评估所有的列在第一个 `dict` 中的 `n_estimators` 和 `max_features` 的  $3 \times 4 = 12$  种组合（不用担心这些超参数的含义，会在第 7 章中解释）。然后尝试第二个 `dict` 中超参数的  $2 \times 3 = 6$  种组合，这次会将超参数 `bootstrap` 设为 `False` 而不是 `True`（后者是该超参数的默认值）。

总之，网格搜索会探索  $12 + 6 = 18$  种 `RandomForestRegressor` 的超参数组合，会训练每个模型五次（因为用的是五折交叉验证）。换句话说，训练总共  $18 \times 5 = 90$  轮！K 折将要花费大量时间，完成后，你就能获得参数的最佳组合，如下所示：

```
>>> grid_search.best_params_
{'max_features': 6, 'n_estimators': 30}
```

提示：因为 30 是 `n_estimators` 的最大值，你也应该估计更高的值，因为评估的分数可能会随 `n_estimators` 的增大而持续提升。

你还能直接得到最佳的估计器：

```
>>> grid_search.best_estimator_
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
                      max_features=6, max_leaf_nodes=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=30, n_jobs=1, oob_score=False, random_state=None,
                      verbose=0, warm_start=False)
```

注意：如果 `GridSearchCV` 是以（默认值）`refit=True` 开始运行的，则一旦用交叉验证找到了最佳的估计器，就会在整个训练集上重新训练。这是一个好方法，因为用更多数据训练会提高性能。

当然，也可以得到评估得分：

```
>>> cvres = grid_search.cv_results_
... for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
...     print(np.sqrt(-mean_score), params)
...
64912.0351358 {'max_features': 2, 'n_estimators': 3}
55535.2786524 {'max_features': 2, 'n_estimators': 10}
52940.2696165 {'max_features': 2, 'n_estimators': 30}
60384.0908354 {'max_features': 4, 'n_estimators': 3}
52709.9199934 {'max_features': 4, 'n_estimators': 10}
50503.5985321 {'max_features': 4, 'n_estimators': 30}
59058.1153485 {'max_features': 6, 'n_estimators': 3}
52172.0292957 {'max_features': 6, 'n_estimators': 10}
49958.9555932 {'max_features': 6, 'n_estimators': 30}
59122.260006 {'max_features': 8, 'n_estimators': 3}
52441.5896087 {'max_features': 8, 'n_estimators': 10}
50041.4899416 {'max_features': 8, 'n_estimators': 30}
62371.1221202 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54572.2557534 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59634.0533132 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52456.0883904 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
58825.665239 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
52012.9945396 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

在这个例子中，我们通过设定超参数 `max_features` 为 6, `n_estimators` 为 30，得到了最佳方案。对这个组合，RMSE 的值是 49959，这比之前使用默认的超参数的值（52634）要稍微好一些。祝贺你，你成功地微调了最佳模型！

**提示：**不要忘记，你可以像超参数一样处理数据准备的步骤。例如，网格搜索可以自动判断是否添加一个你不确定的特征（比如，使用转换器 `CombinedAttributesAdder` 的超参数 `add_bedrooms_per_room`）。它还能用相似的方法来自动找到处理异常值、缺失特征、特征选择等任务的最佳方法。

## 随机搜索

当探索相对较少的组合时，就像前面的例子，网格搜索还可以。但是当超参数的搜索空间很大时，最好使用 `RandomizedSearchCV`。这个类的使用方法和类 `GridSearchCV` 很相似，但它不是尝试所有可能的组合，而是通过选择每个超参数的一个随机值的特定数量的随机组合。这个方法有两个优点：

- 如果你让随机搜索运行，比如 1000 次，它会探索每个超参数的 1000 个不同的值（而不是像网格搜索那样，只搜索每个超参数的几个值）。
- 你可以方便地通过设定搜索次数，控制超参数搜索的计算量。

## 集成方法

另一种微调系统的方法是将表现最好的模型组合起来。组合（集成）之后的性能通常要比单独的模型要好（就像随机森林要比单独的决策树要好），特别是当单独模型的误差类型不同时。我们会在第 7 章更深入地讲解这点。

## 分析最佳模型和它们的误差

通过分析最佳模型，常常可以获得对问题更深的了解。比如，`RandomForestRegressor` 可以指出每个属性对于做出准确预测的相对重要性：

```
>>> feature_importances = grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([ 7.14156423e-02,   6.76139189e-02,   4.44260894e-02,
       1.66308583e-02,   1.66076861e-02,   1.82402545e-02,
       1.63458761e-02,   3.26497987e-01,   6.04365775e-02,
       1.13055290e-01,   7.79324766e-02,   1.12166442e-02,
       1.53344918e-01,   8.41308969e-05,   2.68483884e-03,
       3.46681181e-03])
```

将重要性分数和属性名放到一起：

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_one_hot_attribs = list(encoder.classes_)
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.32649798665134971, 'median_income'),
 (0.15334491760305854, 'INLAND'),
 (0.11305529021187399, 'pop_per_hhold'),
 (0.07793247662544775, 'bedrooms_per_room'),
 (0.071415642259275158, 'longitude'),
 (0.067613918945568688, 'latitude'),
 (0.060436577499703222, 'rooms_per_hhold'),
 (0.04442608939578685, 'housing_median_age'),
 (0.018240254462909437, 'population'),
 (0.0166308583386218, 'total_rooms'),
 (0.016607680091288865, 'total_bedrooms'),
 (0.016345876147580776, 'households'),
 (0.011216644219017424, '<1H OCEAN'),
 (0.0034668118081117387, 'NEAR OCEAN'),
 (0.0026848388432755429, 'NEAR BAY'),
 (8.4130896890070617e-05, 'ISLAND')]
```

有了这个信息，你就可以丢弃一些不那么重要的特征（比如，显然只要一个 ocean\_proximity 的类型（ ISLAND ）就够了，所以可以丢弃掉其它的）。

你还应该看一下系统犯的误差，搞清为什么会有些误差，以及如何改正问题（添加更多的特征，或相反，去掉没有什么信息的特征，清洗异常值等等）。

## 用测试集评估系统

调节完系统之后，你终于有了一个性能足够好的系统。现在就可以用测试集评估最后的模型了。这个过程没有什么特殊的：从测试集得到预测值和标签，运行 full\_pipeline 转换数据（调用 transform() ，而不是 fit\_transform() ! ），再用测试集评估最终模型：

```
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 48,209.6
```

评估结果通常要比交叉验证的效果差一点，如果你之前做过很多超参数微调（因为你的系统在验证集上微调，得到了不错的性能，通常不会在未知的数据集上有同样好的效果）。这个例子不属于这种情况，但是当发生这种情况时，你一定要忍住不要调节超参数，使测试集的效果变好；这样的提升不能推广到新数据上。

然后就是项目的预上线阶段：你需要展示你的方案（重点说明学到了什么、做了什么、没做什么、做过什么假设、系统的限制是什么，等等），记录下所有事情，用漂亮的图表和容易记住的表达（比如，“收入中位数是房价最重要的预测量”）做一次精彩的展示。

## 启动、监控、维护系统

很好，你被允许启动系统了！你需要为实际生产做好准备，特别是接入输入数据源，并编写测试。

你还需要编写监控代码，以固定间隔检测系统的实时表现，当发生下降时触发报警。这对于捕获突然的系统崩溃和性能下降十分重要。做监控很常见，是因为模型会随着数据的演化而性能下降，除非模型用新数据定期训练。

评估系统的表现需要对预测值采样并进行评估。这通常需要人来分析。分析者可能是领域专家，或者是众包平台（比如 Amazon Mechanical Turk 或 CrowdFlower）的工人。不管采用哪种方法，你都需要将人工评估的流水线植入系统。

你还要评估系统输入数据的质量。有时因为低质量的信号（比如失灵的传感器发送随机值，或另一个团队的输出停滞），系统的表现会逐渐变差，但可能需要一段时间，系统的表现才能下降到一定程度，触发警报。如果监测了系统的输入，你就可能尽量早的发现问题。对于线上学习系统，监测输入数据是非常重要的。

最后，你可能想定期用新数据训练模型。你应该尽可能自动化这个过程。如果不这么做，非常有可能你需要每隔至少六个月更新模型，系统的表现就会产生严重波动。如果你的系统是一个线上学习系统，你需要定期保存系统状态快照，好能方便地回滚到之前的工作状态。

## 实践！

希望这一章除了告诉你机器学习项目是什么样的，你能用学到的工具训练一个好系统。你已经看到，大部分的工作是数据准备步骤、搭建监测工具、建立人为评估的流水线和自动化定期模型训练，当然，最好能了解整个过程、熟悉三或四种算法，而不是在探索高级算法上浪费全部时间，导致在全局上的时间不够。

因此，如果你还没这样做，现在最好拿起台电脑，选择一个感兴趣的数据集，将整个流程从头到尾完成一遍。一个不错的着手开始的地点是竞赛网站，比如 <http://kaggle.com/>：你会得到一个数据集，一个目标，以及分享经验的人。

## 练习

使用本章的房产数据集：

1. 尝试一个支持向量机回归器（`sklearn.svm.SVR`），使用多个超参数，比如 `kernel="linear"`（多个超参数 `c` 值）。现在不用担心这些超参数是什么含义。最佳的 SVR 预测表现如何？
2. 尝试用 `RandomizedSearchCV` 替换 `GridSearchCV`。
3. 尝试在准备流水线中添加一个只选择最重要属性的转换器。
4. 尝试创建一个单独的可以完成数据准备和最终预测的流水线。
5. 使用 `GridSearchCV` 自动探索一些准备过程中的候选项。

练习题答案可以在[线上的 Jupyter 笔记本](#)找到。

## 三、分类

译者：@时间魔术师

校对者：@Lisanaaa、@飞龙、@ZTFrom1994、@XinQiu、@tabeworks、  
@JasonLee、@howie.hu

在第一章我们提到过最常用的监督学习任务是回归（用于预测某个值）和分类（预测某个类别）。在第二章我们探索了一个回归任务：预测房价。我们使用了多种算法，诸如线性回归，决策树，和随机森林（这个将会在后面的章节更详细地讨论）。现在我们将我们的注意力转到分类任务上。

### MNIST

在本章当中，我们将会使用 MNIST 这个数据集，它有着 70000 张规格较小的手写数字图片，由美国的高中生和美国人口调查局的职员手写而成。这相当于机器学习当中的“Hello World”，人们无论什么时候提出一个新的分类算法，都想知道该算法在这个数据集上的表现如何。机器学习的初学者迟早也会处理 MNIST 这个数据集。

Scikit-Learn 提供了许多辅助函数，以便于下载流行的数据集。MNIST 是其中一个。下面的代码获取 MNIST

```
>>> from sklearn.datasets import fetch_mldata
>>> mnist = fetch_mldata('MNIST original')
>>> mnist
{'COL_NAMES': ['label', 'data'],
 'DESCR': 'mldata.org dataset: mnist-original',
 'data': array([[0, 0, 0, ..., 0, 0, 0],
 [0, 0, 0, ..., 0, 0, 0],
 [0, 0, 0, ..., 0, 0, 0],
 ...,
 [0, 0, 0, ..., 0, 0, 0],
 [0, 0, 0, ..., 0, 0, 0],
 [0, 0, 0, ..., 0, 0, 0]], dtype=uint8),
 'target': array([ 0.,  0.,  0., ..., 9.,  9.,  9.])}
```

一般而言，由 sklearn 加载的数据集有着相似的字典结构，这包括：

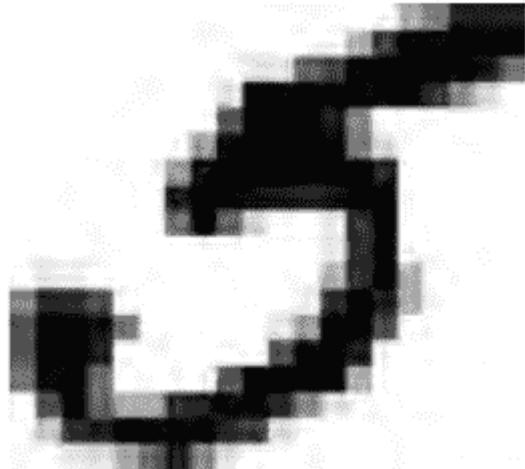
- `DESCR` 键描述数据集
- `data` 键存放一个数组，数组的一行表示一个样例，一列表示一个特征
- `target` 键存放一个标签数组

让我们看一下这些数组

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

MNIST 有 70000 张图片，每张图片有 784 个特征。这是因为每个图片都是 28\*28 像素的，并且每个像素的值介于 0~255 之间。让我们看一看数据集的某一个数字。你只需要将某个实例的特征向量，`reshape` 为 28\*28 的数组，然后使用 Matplotlib 的 `imshow` 函数展示出来。

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
some_digit = X[36000]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap = matplotlib.cm.binary, interpolation="nearest")
plt.axis("off")
plt.show()
```



这看起来像个 5，实际上它的标签告诉我们：

```
>>> y[36000]
5.0
```

图 3-1 展示了一些来自 MNIST 数据集的图片。当你处理更加复杂的分类任务的时候，它会让你更有感觉。



先等一下！你总是应该先创建测试集，并且在验证数据之前先把测试集晾到一边。MNIST 数据集已经事先被分成了一个训练集（前 60000 张图片）和一个测试集（最后 10000 张图片）

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

让我们打乱训练集。这可以保证交叉验证的每一折都是相似（你不会期待某一折缺少某类数字）。而且，一些学习算法对训练样例的顺序敏感，当它们在一列当中得到许多相似的样例，这些算法将会表现得非常差。打乱数据集将保证这种情况不会发生。

```
import numpy as np
shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

## 训练一个二分类器

现在我们简化一下问题，只尝试去识别一个数字，比如说，数字 5。这个“数字 5 检测器”就是一个二分类器，能够识别两类别，“是 5”和“非 5”。让我们为这个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)
```

现在让我们挑选一个分类器去训练它。用随机梯度下降分类器 SGD，是一个不错的开始。使用 Scikit-Learn 的 `SGDClassifier` 类。这个分类器有一个好处是能够高效地处理非常大的数据集。这部分原因在于 SGD 一次只处理一条数据，这也使得 SGD 适合在线学习（online learning）。我们在稍后会看到它。让我们创建一个 `SGDClassifier` 和在整个数据集上训练它。

```
from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

`SGDClassifier` 依赖于训练集的随机程度（所以被命名为 `stochastic`，随机之义）。如果你想重现结果，你应该固定参数 `random_state`

现在你可以用它来查出数字 5 的图片。

```
>>> sgd_clf.predict([some_digit])
array([True], dtype=bool)
```

分类器猜测这个数字代表 5（`True`）。看起来在这个例子当中，它猜对了。现在让我们评估这个模型的性能。

## 对性能的评估

评估一个分类器，通常比评估一个回归器更加玄学。所以我们将会花大量的篇幅在这个话题上。有许多量度性能的方法，所以拿来一杯咖啡和准备学习许多新概念和首字母缩略词吧。

## 使用交叉验证测量准确性

评估一个模型的好方法是使用交叉验证，就像第二章所做的那样。

### 实现交叉验证

在交叉验证过程中，有时候你会需要更多的控制权，相较于函数 `cross_val_score()` 或者其他相似函数所提供的功能。这种情况下，你可以实现你自己版本的交叉验证。事实上它相当简单。以下代码粗略地做了和 `cross_val_score()` 相同的事情，并且输出相同的结果。

```
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
```

`StratifiedKFold` 类实现了分层采样（详见第二章的解释），生成的折（fold）包含了各类相应比例的样例。在每一次迭代，上述代码生成分类器的一个克隆版本，在训练折（training folds）的克隆版本上进行训练，在测试折（test folds）上进行预测。然后它计算出被正确预测的数目和输出正确预测的比例。

让我们使用 `cross_val_score()` 函数来评估 `SGDClassifier` 模型，同时使用 K 折交叉验证，此处让 `k=3`。记住：K 折交叉验证意味着把训练集分成 K 折（此处 3 折），然后使用一个模型对其中一折进行预测，对其他折进行训练。

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.9502, 0.96565, 0.96495])
```

哇！在交叉验证上有大于 95% 的精度（accuracy）？这看起来很令人吃惊。先别高兴，让我们来看一个非常笨的分类器去分类，看看其在“非 5”这个类上的表现。

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

你能猜到这个模型的精度吗？揭晓谜底：

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.909, 0.90715, 0.9128])
```

没错，这个笨的分类器也有 90% 的精度。这是因为只有 10% 的图片是数字 5，所以你总是猜测某张图片不是 5，你也会有 90% 的可能性是对的。

这证明了为什么精度通常来说不是一个好的性能度量指标，特别是当你处理有偏差的数据集，比方说其中一些类比其他类频繁得多。

## 混淆矩阵

对分类器来说，一个好得多的性能评估指标是混淆矩阵。大体思路是：输出类别 A 被分类成类别 B 的次数。举个例子，为了知道分类器将 5 误分为 3 的次数，你需要查看混淆矩阵的第五行第三列。

为了计算混淆矩阵，首先你需要有一系列的预测值，这样才能将预测值与真实值做比较。你或许想在测试集上做预测。但是我们现在先不碰它。（记住，只有当你处于项目的尾声，当你准备上线一个分类器的时候，你才应该使用测试集）。相反，你应该使用 `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)
```

就像 `cross_val_score()`，`cross_val_predict()` 也使用 K 折交叉验证。它不是返回一个评估分数，而是返回基于每一个测试折做出的一个预测值。这意味着，对于每一个训练集的样例，你得到一个干净的预测（“干净”是说一个模型在训练过程当中没有用到测试集的数据）。

现在使用 `confusion_matrix()` 函数，你将会得到一个混淆矩阵。传递目标类 (`y_train_5`) 和预测类 (`y_train_pred`) 给它。

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53272, 1307],
       [ 1077, 4344]])
```

混淆矩阵中的每一行表示一个实际的类，而每一列表示一个预测的类。该矩阵的第一行认为“非 5”（反例）中的 53272 张被正确归类为“非 5”（他们被称为真反例，true negatives），而其余 1307 被错误归类为“是 5”（假正例，false positives）。

第二行认为“是 5”（正例）中的 1077 被错误地归类为“非 5”（假反例，false negatives），其余 4344 正确分类为“是 5”类（真正例，true positives）。一个完美的分类器将只有真反例和真正例，所以混淆矩阵的非零值仅在其主对角线（左上至右下）。

```
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579, 0],
       [ 0, 5421]])
```

混淆矩阵可以提供很多信息。有时候你会想要更加简明的指标。一个有趣的指标是正例预测的精度，也叫做分类器的准确率（precision）。

公式 3-1 准确率

$$\text{precision} = \frac{TP}{TP + FP}$$

其中 `TP` 是真正例的数目，`FP` 是假正例的数目。

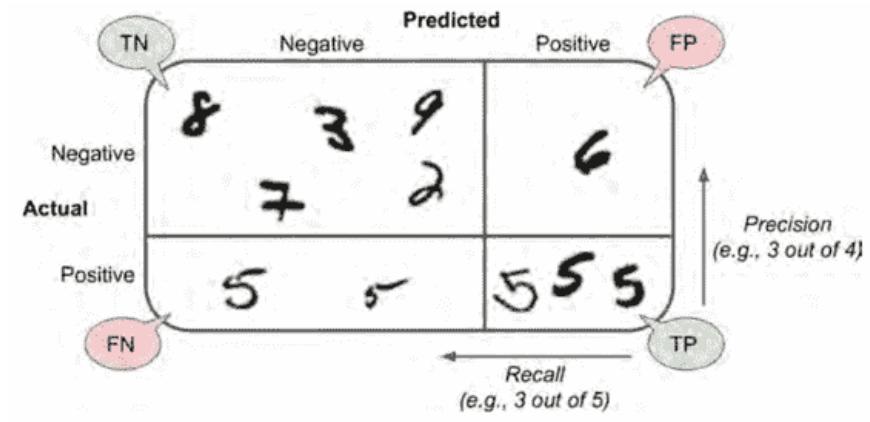
想要一个完美的准确率，一个平凡的方法是构造一个单一正例的预测和确保这个预测是正确的（`precision = 1/1 = 100%`）。但是这没什么用，因为分类器会忽略所有样例，除了那一个正例。所以准确率一般会伴随另一个指标一起使用，这个指标叫做召回率（recall），也叫做敏感度（sensitivity）或者真正例率（true positive rate，TPR）。这是正例被分类器正确探测出的比率。

## 公式 3-2 Recall

$$\text{recall} = \frac{TP}{TP + FN}$$

$FN$  是假反例的数目。

如果你对于混淆矩阵感到困惑，图 3-2 将对你有帮助



## 准确率与召回率

Scikit-Learn 提供了一些函数去计算分类器的指标，包括准确率和召回率。

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_pred) # == 4344 / (4344 + 1307)
0.76871350203503808
>>> recall_score(y_train_5, y_train_pred) # == 4344 / (4344 + 1077)
0.79136690647482011
```

当你去观察精度的时候，你的“数字 5 探测器”看起来还不够好。当它声明某张图片是 5 的时候，它只有 77% 的可能性是正确的。而且，它也只检测出“是 5”类图片当中的 79%。

通常结合准确率和召回率会更加方便，这个指标叫做“F1 值”，特别是当你需要一个简单的方法去比较两个分类器的优劣的时候。F1 值是准确率和召回率的调和平均。普通的平均值平等地看待所有的值，而调和平均会给小的值更大的权重。所以，要想分类器得到一个高的 F1 值，需要召回率和准确率同时高。

## 公式 3-3 F1 值

$$F1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = 2 * \frac{precision * recall}{precision + recall} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

为了计算 F1 值，简单调用 `f1_score()`

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.78468208092485547
```

F1 支持那些有着相近准确率和召回率的分类器。这不会总是你想要的。有的场景你会绝大部分地关心准确率，而另外一些场景你会更关心召回率。举例子，如果你训练一个分类器去检测视频是否适合儿童观看，你会倾向选择那种即便拒绝了很多好视频、但保证所保留的视频都是好（高准确率）的分类器，而不是那种高召回

率、但让坏视频混入的分类器（这种情况下你或许想增加人工去检测分类器选择出来的视频）。另一方面，加入你训练一个分类器去检测监控图像当中的窃贼，有着 30% 准确率、99% 召回率的分类器或许是合适的（当然，警卫会得到一些错误的报警，但是几乎所有的窃贼都会被抓到）。

不幸的是，你不能同时拥有两者。增加准确率会降低召回率，反之亦然。这叫做准确率与召回率之间的折衷。

## 准确率/召回率之间的折衷

为了弄懂这个折衷，我们看一下 `SGDClassifier` 是如何做分类决策的。对于每个样例，它根据决策函数计算分数，如果这个分数大于一个阈值，它会将样例分配给正例，否则它将分配给反例。图 3-3 显示了几个数字从左边的最低分数排到右边的最高分。假设决策阈值位于中间的箭头（介于两个 5 之间）：您将发现 4 个真正例（数字 5）和一个假正例（数字 6）在该阈值的右侧。因此，使用该阈值，准确率为 80% ( $4/5$ )。但实际有 6 个数字 5，分类器只检测 4 个，所以召回是 67% ( $4/6$ )。现在，如果你提高阈值（移动到右侧的箭头），假正例（数字 6）成为一个真反例，从而提高准确率（在这种情况下高达 100%），但一个真正例变成假反例，召回率降低到 50%。相反，降低阈值可提高召回率、降低准确率。

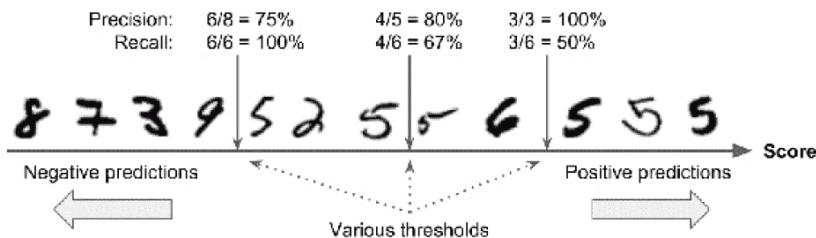


Figure 3-3. Decision threshold and precision/recall tradeoff

Scikit-Learn 不让你直接设置阈值，但是它给你提供了设置决策分数的方法，这个决策分数可以用来产生预测。它不是调用分类器的 `predict()` 方法，而是调用 `decision_function()` 方法。这个方法返回每一个样例的分数值，然后基于这个分数值，使用你想要的任何阈值做出预测。

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([-161855.74572176])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True], dtype=bool)
```

`SGDClassifier` 用了一个等于 0 的阈值，所以前面的代码返回了跟 `predict()` 方法一样的结果（都返回了 `true`）。让我们提高这个阈值：

```
>>> threshold = 200000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False], dtype=bool)
```

这证明了提高阈值会降低召回率。这个图片实际就是数字 5，当阈值等于 0 的时候，分类器可以探测到这是一个 5，当阈值提高到 20000 的时候，分类器将不能探测到这是数字 5。

那么，你应该如何使用哪个阈值呢？首先，你需要再次使用 `cross_val_predict()` 得到每一个样例的分数值，但是这一次指定返回一个决策分数，而不是预测值。

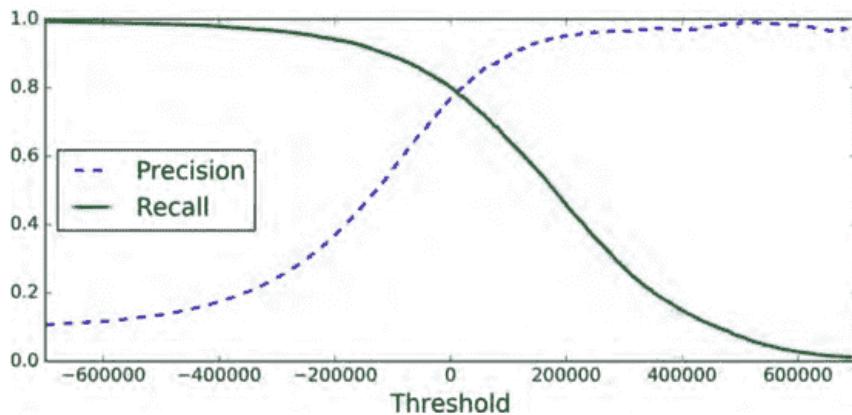
```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

现在有了这些分数值。对于任何可能的阈值，使用 `precision_recall_curve()`，你都可以计算准确率和召回率：

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

最后，你可以使用 Matplotlib 画出准确率和召回率（图 3-4），这里把准确率和召回率当作是阈值的一个函数。

```
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")
    plt.xlabel("Threshold")
    plt.legend(loc="upper left")
    plt.ylim([0, 1])
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```



你也许会好奇为什么准确率曲线比召回率曲线更加起伏不平。原因是准确率有时候会降低，尽管当你提高阈值的时候，通常来说准确率会随之提高。回头看图 3-3，留意当你从中间箭头开始然后向右移动一个数字会发生什么：准确率会由  $4/5$  (80%) 降到  $3/4$  (75%)。另一方面，当阈值提高时候，召回率只会降低。这也说明了为什么召回率的曲线更加平滑。

现在你可以选择适合你任务的最佳阈值。另一个选出好的准确率/召回率折衷的方法是直接画出准确率对召回率的曲线，如图 3-5 所示。

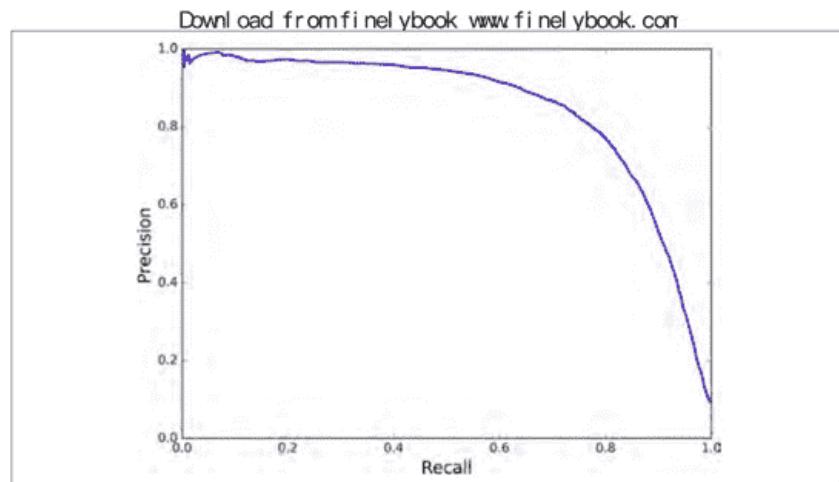


Figure 3-5. Precision versus recall

可以看到，在召回率在 80% 左右的时候，准确率急剧下降。你可能会想选择在急剧下降之前选择出一个准确率/召回率折衷点。比如说，在召回率 60% 左右的点。当然，这取决于你的项目需求。

我们假设你决定达到 90% 的准确率。你查阅第一幅图（放大一些），在 70000 附近找到一个阈值。为了作出预测（目前为止只在训练集上预测），你可以运行以下代码，而不是运行分类器的 `predict()` 方法。

```
y_train_pred_90 = (y_scores > 70000)
```

让我们检查这些预测的准确率和召回率：

```
>>> precision_score(y_train_5, y_train_pred_90)
0.8998702983138781
>>> recall_score(y_train_5, y_train_pred_90)
0.63991883416343853
```

很棒！你拥有了一个（近似）90% 准确率的分类器。它相当容易去创建一个任意准确率的分类器，只要将阈值设置得足够高。但是，一个高准确率的分类器不是非常有用，如果它的召回率太低！

如果有人说“让我们达到 99% 的准确率”，你应该问“相应的召回率是多少？”

## ROC 曲线

受试者工作特征 (ROC) 曲线是另一个二分类器常用的工具。它非常类似与准确率/召回率曲线，但不是画出准确率对召回率的曲线，ROC 曲线是真正例率 (true positive rate，另一个名字叫做召回率) 对假正例率 (false positive rate, FPR) 的曲线。FPR 是反例被错误分成正例的比率。它等于 1 减去真反例率 (true negative rate, TNR)。TNR 是反例被正确分类的比率。TNR 也叫做特异性。所以 ROC 曲线画出召回率对 (1 减特异性) 的曲线。

为了画出 ROC 曲线，你首先需要计算各种不同阈值下的 TPR、FPR，使用 `roc_curve()` 函数：

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

然后你可以使用 matplotlib，画出 FPR 对 TPR 的曲线。下面的代码生成图 3-6.

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
plot_roc_curve(fpr, tpr)
plt.show()
```

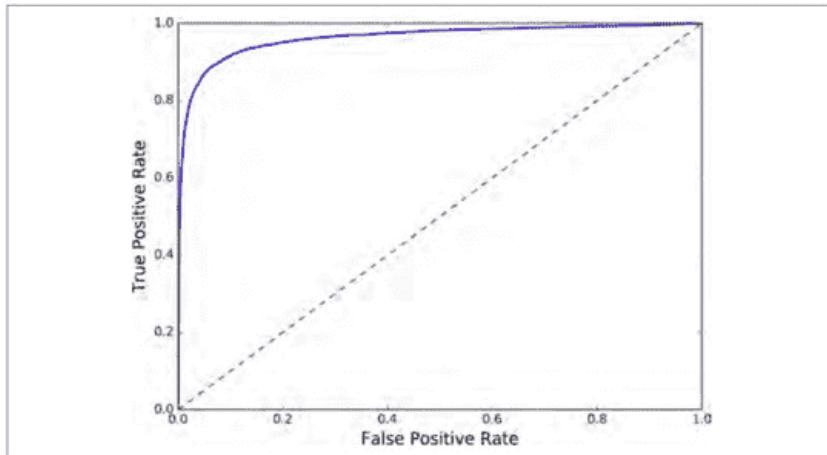


Figure 3-6. ROC curve

这里同样存在折衷的问题：召回率（TPR）越高，分类器就会产生越多的假正例（FPR）。图中的点线是一个完全随机的分类器生成的 ROC 曲线；一个好的分类器的 ROC 曲线应该尽可能远离这条线（即向左上角方向靠拢）。

一个比较分类器之间优劣的方法是：测量 ROC 曲线下方的面积（AUC）。一个完美的分类器的 ROC AUC 等于 1，而一个纯随机分类器的 ROC AUC 等于 0.5。

Scikit-Learn 提供了一个函数来计算 ROC AUC：

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.97061072797174941
```

因为 ROC 曲线跟准确率/召回率曲线（或者叫 PR）很类似，你或许会好奇如何决定使用哪一个曲线呢？一个笨拙的规则是，优先使用 PR 曲线当正例很少，或者当你关注假正例多于假反例的时候。其他情况使用 ROC 曲线。举例子，回顾前面的 ROC 曲线和 ROC AUC 数值，你或许认为这个分类器很棒。但是这几乎全是因为只有少数正例（“是 5”），而大部分是反例（“非 5”）。相反，PR 曲线清楚显示出这个分类器还有很大的改善空间（PR 曲线应该尽可能地靠近右上角）。

让我们训练一个 `RandomForestClassifier`，然后拿它的的 ROC 曲线和 ROC AUC 数值去跟 `SGDClassifier` 的比较。首先你需要得到训练集每个样例的数值。但是由于随机森林分类器的工作方式，`RandomForestClassifier` 不提供 `decision_function()` 方法。相反，它提供了 `predict_proba()` 方法。Skikit-Learn 分类器通常二者中的一个。`predict_proba()` 方法返回一个数组，数组的每一行代表一个样例，每一列代表一个类。数组当中的值的意思是：给定一个样例属于给定类的概率。比如，70% 的概率这幅图是数字 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")
```

但是要画 ROC 曲线，你需要的是样例的分数，而不是概率。一个简单的解决方法是使用正例的概率当作样例的分数。

```
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)
```

现在你即将得到 ROC 曲线。将前面一个分类器的 ROC 曲线一并画出来是很有用的，可以清楚地进行比较。见图 3-7。

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

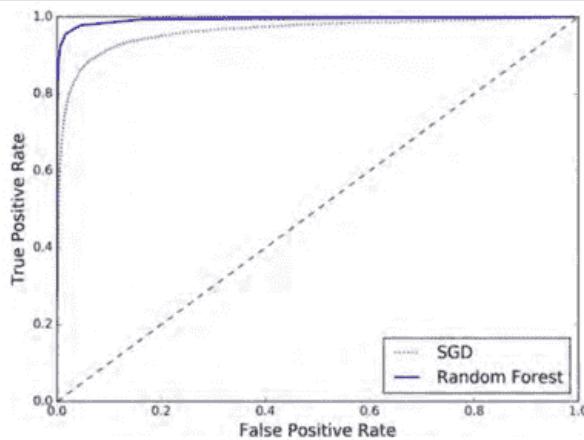


Figure 3-7. Comparing ROC curves

如你所见，`RandomForestClassifier` 的 ROC 曲线比 `SGDClassifier` 的好得多：它更靠近左上角。所以，它的 ROC AUC 也会更大。

```
>>> roc_auc_score(y_train_5, y_scores_forest)
0.99312433660038291
```

计算一下准确率和召回率：98.5% 的准确率，82.8% 的召回率。还不错。

现在你知道如何训练一个二分类器，选择合适的标准，使用交叉验证去评估你的分类器，选择满足你需要的准确率/召回率折衷方案，和比较不同模型的 ROC 曲线和 ROC AUC 数值。现在让我们检测更多的数字，而不仅仅是一个数字 5。

## 多类分类

二分类器只能区分两个类，而多类分类器（也被叫做多项式分类器）可以区分多于两个类。

一些算法（比如随机森林分类器或者朴素贝叶斯分类器）可以直接处理多类分类问题。其他一些算法（比如 SVM 分类器或者线性分类器）则是严格的二分类器。然后，有许多策略可以让你用二分类器去执行多类分类。

举例子，创建一个可以将图片分成 10 类（从 0 到 9）的系统的一个方法是：训练 10 个二分类器，每一个对应一个数字（探测器 0，探测器 1，探测器 2，以此类推）。然后当你想对某张图片进行分类的时候，让每一个分类器对这个图片进行分类，选出决策分数最高的那个分类器。这叫做“一对所有”（OvA）策略（也被叫做“一对其他”）。

另一个策略是对每一对数字都训练一个二分类器：一个分类器用来处理数字 0 和数字 1，一个用来处理数字 0 和数字 2，一个用来处理数字 1 和 2，以此类推。这叫做“一对一”（OvO）策略。如果有 N 个类。你需要训练  $N*(N-1)/2$  个分类器。对于 MNIST 问题，需要训练 45 个二分类器！当你想对一张图片进行分类，你必须将这张图片跑在全部 45 个二分类器上。然后看哪个类胜出。OvO 策略的主要优点是：每个分类器只需要在训练集的部分数据上面进行训练。这部分数据是它所需要区分的那两个类对应的数据。

一些算法（比如 SVM 分类器）在训练集的大小上很难扩展，所以对于这些算法，OvO 是比较好的，因为它可以在小的数据集上面可以更多地训练，较之于巨大的数据集而言。但是，对于大部分的二分类器来说，OvA 是更好的选择。

Scikit-Learn 可以探测出你想使用一个二分类器去完成多分类的任务，它会自动地执行 OvA（除了 SVM 分类器，它使用 OvO）。让我们试一下 `SGDClassifier`。

```
>>> sgd_clf.fit(X_train, y_train) # y_train, not y_train_5
>>> sgd_clf.predict([some_digit])
array([ 5.])
```

很容易。上面的代码在训练集上训练了一个 `SGDClassifier`。这个分类器处理原始的目标类，从 0 到 9（`y_train`），而不是仅仅探测是否为 5（`y_train_5`）。然后它做出一个判断（在这个案例下只有一个正确的数字）。在幕后，Scikit-Learn 实际上训练了 10 个二分类器，每个分类器都产生到一张图片的决策数值，选择数值最高的那个类。

为了证明这是真实的，你可以调用 `decision_function()` 方法。不是返回每个样例的一个数值，而是返回 10 个数值，一个数值对应于一个类。

```
>>> some_digit_scores = sgd_clf.decision_function([some_digit])
>>> some_digit_scores
array([-311402.62954431, -363517.28355739, -446449.5306454,
       -183226.61023518, -414337.15339485, 161855.74572176,
       -452576.39616343, -471957.14962573, -518542.33997148,
       -536774.63961222])
```

最高数值是对应于类别 5：

```
>>> np.argmax(some_digit_scores)
5
>>> sgd_clf.classes_
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> sgd_clf.classes_[5]
5.0
```

一个分类器被训练好了之后，它会保存目标类别列表到它的属性 `classes_` 中去，按照值排序。在本例子当中，在 `classes_` 数组当中的每个类的索引方便地匹配了类本身，比如，索引为 5 的类恰好是类别 5 本身。但通常不会这么幸运。

如果你想强制 Scikit-Learn 使用 OvO 策略或者 OvA 策略，你可以使用 `OneVsOneClassifier` 类或者 `OneVsRestClassifier` 类。创建一个样例，传递一个二分类器给它的构造函数。举例子，下面的代码会创建一个多类分类器，使用 OvO 策略，基于 `SGDClassifier`。

```
>>> from sklearn.multiclass import OneVsOneClassifier
>>> ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42))
>>> ovo_clf.fit(X_train, y_train)
>>> ovo_clf.predict([some_digit])
array([ 5.])
>>> len(ovo_clf.estimators_)
45
```

训练一个 `RandomForestClassifier` 同样简单：

```
>>> forest_clf.fit(X_train, y_train)
>>> forest_clf.predict([some_digit])
array([ 5.])
```

这次 Scikit-Learn 没有必要去运行 OvO 或者 OvA，因为随机森林分类器能够直接将一个样例分到多个类别。你可以调用 `predict_proba()`，得到样例对应的类别的概率值的列表：

```
>>> forest_clf.predict_proba([some_digit])
array([[ 0.1,  0. ,  0. ,  0.1,  0. ,  0.8,  0. ,  0. ,  0. ,  0. ]])
```

你可以看到这个分类器相当确信它的预测：在数组的索引 5 上的 0.8，意味着这个模型以 80% 的概率估算这张图片代表数字 5。它也认为这个图片可能是数字 0 或者数字 3，分别都是 10% 的几率。

现在当然你想评估这些分类器。像平常一样，你想使用交叉验证。让我们用 `cross_val_score()` 来评估 `SGDClassifier` 的精度。

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([ 0.84063187,  0.84899245,  0.86652998])
```

在所有测试折（test fold）上，它有 84% 的精度。如果你是一个随机的分类器，你将会得到 10% 的正确率。所以这不是一个坏的分数，但是你可以做的更好。举例子，简单将输入正则化，将会提高精度到 90% 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([ 0.91011798,  0.90874544,  0.906636 ])
```

## 误差分析

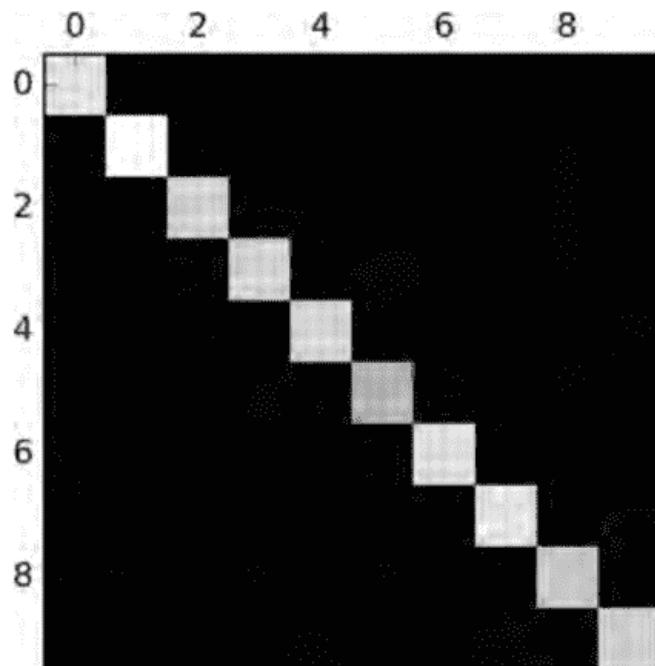
当然，如果这是一个实际的项目，你会在你的机器学习项目当中，跟随以下步骤（见附录 B）：探索准备数据的候选方案，尝试多种模型，把最好的几个模型列为入围名单，用 `GridSearchCV` 调试超参数，尽可能地自动化，像你前面的章节做的那样。在这里，我们假设你已经找到一个不错的模型，你试图找到方法去改善它。一个方式是分析模型产生的误差的类型。

首先，你可以检查混淆矩阵。你需要使用 `cross_val_predict()` 做出预测，然后调用 `confusion_matrix()` 函数，像你早前做的那样。

```
>>> 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([[5725,  3, 24,  9, 10, 49, 50, 10, 39,  4],
       [ 2, 6493, 43, 25,  7, 40,  5, 10, 109, 8],
       [ 51, 41, 5321, 104, 89, 26, 87, 60, 166, 13],
       [ 47, 46, 141, 5342, 1, 231, 40, 50, 141, 92],
       [ 19, 29, 41, 10, 5366, 9, 56, 37, 86, 189],
       [ 73, 45, 36, 193, 64, 4582, 111, 30, 193, 94],
       [ 29, 34, 44, 2, 42, 85, 5627, 10, 45, 0],
       [ 25, 24, 74, 32, 54, 12, 6, 5787, 15, 236],
       [ 52, 161, 73, 156, 10, 163, 61, 25, 5027, 123],
       [ 43, 35, 26, 92, 178, 28, 2, 223, 82, 5240]])
```

这里是一对数字。使用 Matplotlib 的 `matshow()` 函数，将混淆矩阵以图像的方式呈现，将会更加方便。

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



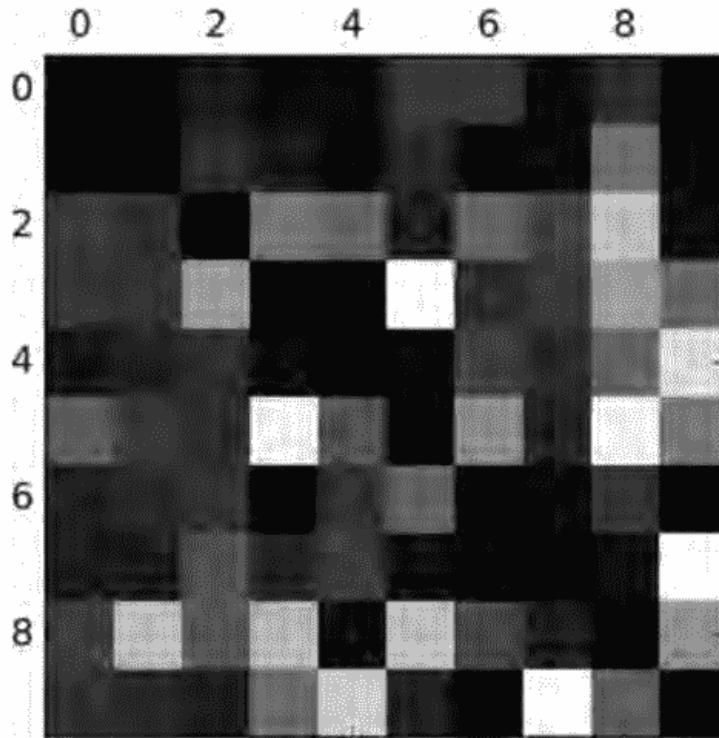
这个混淆矩阵看起来相当好，因为大多数的图片在主对角线上。在主对角线上意味着被分类正确。数字 5 对应的格子看起来比其他数字要暗淡许多。这可能是数据集中数字 5 的图片比较少，又或者是分类器对于数字 5 的表现不如其他数字那么好。你可以验证两种情况。

让我们关注仅包含误差数据的图像呈现。首先你需要将混淆矩阵的每一个值除以相应类别的图片的总数目。这样子，你可以比较错误率，而不是绝对的错误数（这对大的类别不公平）。

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

现在让我们用 0 来填充对角线。这样子就只保留了被错误分类的数据。让我们画出这个结果。

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

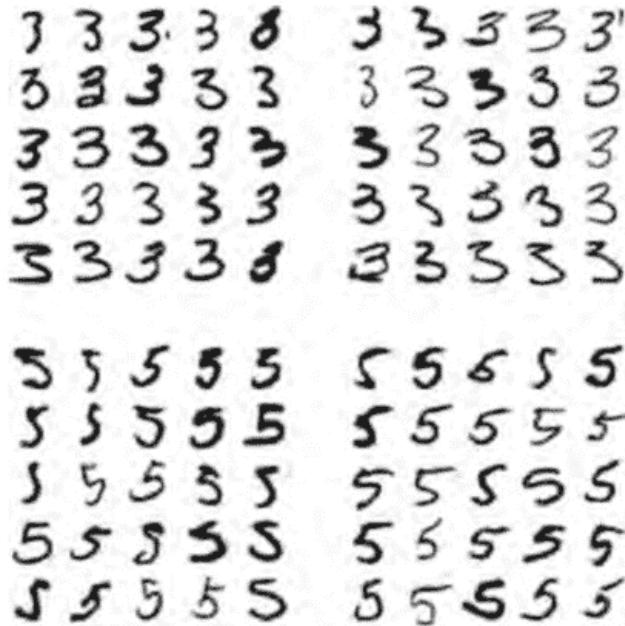


现在你可以清楚看出分类器制造出来的各类误差。记住：行代表实际类别，列代表预测的类别。第 8、9 列相当亮，这告诉你许多图片被误分成数字 8 或者数字 9。相似的，第 8、9 行也相当亮，告诉你数字 8、数字 9 经常被误以为是其他数字。相反，一些行相当黑，比如第一行：这意味着大部分的数字 1 被正确分类（一些被误分类为数字 8）。留意到误差图不是严格对称的。举例子，比起将数字 8 误分类为数字 5 的数量，有更多的数字 5 被误分类为数字 8。

分析混淆矩阵通常可以给你提供深刻的见解去改善你的分类器。回顾这幅图，看样子你应该努力改善分类器在数字 8 和数字 9 上的表现，和纠正 3/5 的混淆。举例子，你可以尝试去收集更多的数据，或者你可以构造新的、有助于分类器的特征。举例子，写一个算法去数闭合的环（比如，数字 8 有两个环，数字 6 有一个，5 没有）。又或者你可以预处理图片（比如，使用 Scikit-Learn, Pillow, OpenCV）去构造一个模式，比如闭合的环。

分析独特的误差，是获得关于你的分类器是如何工作及其为什么失败的洞见的一个好途径。但是这相对难和耗时。举例子，我们可以画出数字 3 和 5 的例子

```
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()
```



左边两个  $5 \times 5$  的块将数字识别为 3，右边的将数字识别为 5。一些被分类器错误分类的数字（比如左下角和右上角的块）是书写地相当差，甚至让人类分类都会觉得很难（比如第 8 行第 1 列的数字 5，看起来非常像数字 3）。但是，大部分被误分类的数字，在我们看来都是显而易见的错误。很难明白为什么分类器会分错。原因是我们使用的简单的 `SGDClassifier`，这是一个线性模型。它所做的全部工作就是分配一个类权重给每一个像素，然后当它看到一张新的图片，它就将加权的像素强度相加，每个类得到一个新的值。所以，因为 3 和 5 只有一小部分的像素有差异，这个模型很容易混淆它们。

3 和 5 之间的主要差异是连接顶部的线和底部的线的细线的位置。如果你画一个 3，连接处稍微向左偏移，分类器很可能将它分类成 5。反之亦然。换一个说法，这个分类器对于图片的位移和旋转相当敏感。所以，减轻 3/5 混淆的一个方法是对图片进行预处理，确保它们都很好地中心化和不过度旋转。这同样很可能帮助减轻其他类型的错误。

## 多标签分类

到目前为止，所有的样例都总是被分配到仅一个类。有些情况下，你也许想让你的分类器给一个样例输出多个类别。比如说，思考一个人脸识别器。如果对于同一张图片，它识别出几个人，它应该做什么？当然它应该给每一个它识别出的人贴上一个标签。比方说，这个分类器被训练成识别三个人脸，Alice, Bob, Charlie；然后当它被输入一张含有 Alice 和 Bob 的图片，它应该输出 `[1, 0, 1]`（意思是：Alice 是，Bob 不是，Charlie 是）。这种输出多个二值标签的分类系统被叫做多标签分类系统。

目前我们不打算深入脸部识别。我们可以先看一个简单点的例子，仅仅是为了阐明的目的。

```
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)
```

这段代码创造了一个 `y_multilabel` 数组，里面包含两个目标标签。第一个标签指出这个数字是否为大数字（7, 8 或者 9），第二个标签指出这个数字是否是奇数。接下来几行代码会创建一个 `KNeighborsClassifier` 样例（它支持多标签分类，但不是所有分类器都可以），然后我们使用多目标数组来训练它。现在你可以生成一个预测，然后它输出两个标签：

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

它工作正确。数字 5 不是大数（`False`），同时是一个奇数（`True`）。

有许多方法去评估一个多标签分类器，和选择正确的量度标准，这取决于你的项目。举个例子，一个方法是对每个个体标签去量度 F1 值（或者前面讨论过的其他任意的二分类器的量度标准），然后计算平均值。下面的代码计算全部标签的平均 F1 值：

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_train, cv=3)
>>> f1_score(y_train, y_train_knn_pred, average="macro")
0.96845540180280221
```

这里假设所有标签有着同等的重要性，但可能不是这样。特别是，如果你的 Alice 的照片比 Bob 或者 Charlie 更多的时候，也许你想让分类器在 Alice 的照片上具有更大的权重。一个简单的选项是：给每一个标签的权重等于它的支持度（比如，那个标签的样例的数目）。为了做到这点，简单地在上面代码中设置 `average="weighted"`。

## 多输出分类

我们即将讨论的最后一种分类任务被叫做“多输出-多类分类”（或者简称为多输出分类）。它是多标签分类的简单泛化，在这里每一个标签可以是多类别的（比如说，它可以有多于两个可能值）。

为了说明这点，我们建立一个系统，它可以去除图片当中的噪音。它将一张混有噪音的图片作为输入，期待它输出一张干净的数字图片，用一个像素强度的数组表示，就像 MNIST 图片那样。注意到这个分类器的输出是多标签的（一个像素一个标签）和每个标签可以有多个值（像素强度取值范围从 0 到 255）。所以它是一个多输出分类系统的例子。

分类与回归之间的界限是模糊的，比如这个例子。按理说，预测一个像素的强度更类似于一个回归任务，而不是一个分类任务。而且，多输出系统不限于分类任务。你甚至可以让你一个系统给每一个样例都输出多个标签，包括类标签和值标签。

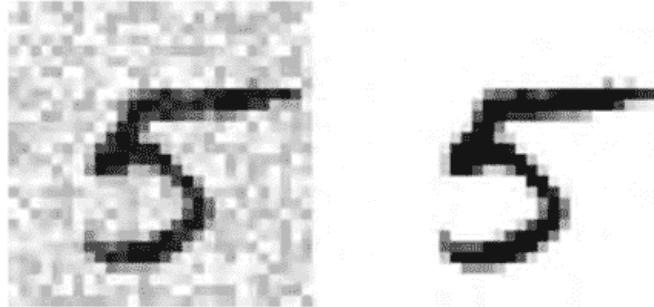
让我们从 MNIST 的图片创建训练集和测试集开始，然后给图片的像素强度添加噪声，这里是用 NumPy 的 `randint()` 函数。目标图像是原始图像。

```

noise = rnd.randint(0, 100, (len(X_train), 784))
noise = rnd.randint(0, 100, (len(X_test), 784))
X_train_mod = X_train + noise
X_test_mod = X_test + noise
y_train_mod = y_train
y_test_mod = y_test

```

让我们看一下测试集当中的一张图片（是的，我们在窥探测试集，所以你应该马上皱眉）：

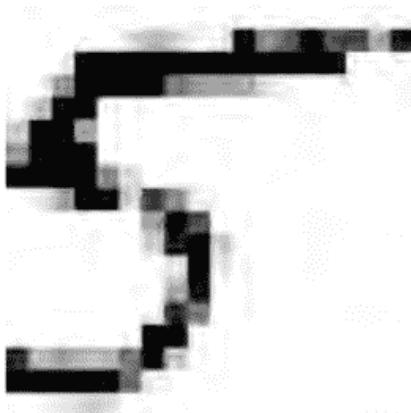


左边的加噪声的输入图片。右边是干净的目标图片。现在我们训练分类器，让它清洁这张图片：

```

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

```



看起来足够接近目标图片。现在总结我们的分类之旅。希望你现在应该知道如何选择好的量度标准，挑选出合适的准确率/召回率的折衷方案，比较分类器，更概括地说，就是为不同的任务建立起好的分类系统。

## 练习

- 尝试在 MNIST 数据集上建立一个分类器，使它在测试集上的精度超过 97%。  
提示：KNeighborsClassifier 非常适合这个任务。你只需要找出一个好的超参数值（试一下对权重和超参数 n\_neighbors 进行网格搜索）。
- 写一个函数可以是 MNIST 中的图像任意方向移动（上下左右）一个像素。然后，对训练集上的每张图片，复制四个移动后的副本（每个方向一个副本），

把它们加到训练集当中去。最后在扩展后的训练集上训练你最好的模型，并且在测试集上测量它的精度。你应该会观察到你的模型会有更好的表现。这种人工扩大训练集的方法叫做数据增强，或者训练集扩张。

3. 拿 Titanic 数据集去捣鼓一番。开始这个项目有一个很棒的平台 : Kaggle!
4. 建立一个垃圾邮件分类器（这是一个更有挑战性的练习）：
  - 下载垃圾邮件和非垃圾邮件的样例数据。地址是 [Apache SpamAssassin 的公共数据集](#)
  - 解压这些数据集，并且熟悉它的数据格式。
  - 将数据集分成训练集和测试集
  - 写一个数据准备的流水线，将每一封邮件转换为特征向量。你的流水线应该将一封邮件转换为一个稀疏向量，对于所有可能的词，这个向量标志哪个词出现了，哪个词没有出现。举例子，如果所有邮件只包含了 "Hello", "How", "are", "you" 这四个词，那么一封邮件（内容是： "Hello you Hello Hello you" ）将会被转换为向量 [1, 0, 0, 1] (意思是："Hello" 出现， "How" 不出现， "are" 不出现， "you" 出现)，或者 [3, 0, 0, 2] ，如果你想数出每个单词出现的次数。
  - 你也许想给你的流水线增加超参数，控制是否剥过邮件头、将邮件转换为小写、去除标点符号、将所有 URL 替换成 "URL" ，将所有数字替换成 "NUMBER" ，或者甚至提取词干（比如，截断词尾。有现成的 Python 库可以做到这点）。
  - 然后 尝试几个不同的分类器，看看你可否建立一个很棒的垃圾邮件分类器，同时有着高召回率和高准确率。

## 四、训练模型

译者：[@C-PIG](#)

校对者：[@PeterHo](#)、[@飞龙](#)、[@YuWang](#)、[@AlecChen](#)

在之前的描述中，我们通常把机器学习模型和训练算法当作黑箱子来处理。如果你实践过前几章的一些示例，你惊奇的发现你可以优化回归系统，改进数字图像的分类器，你甚至可以零基础搭建一个垃圾邮件的分类器，但是你却对它们内部的工作流程一无所知。事实上，许多场合你都不需要知道这些黑箱子的内部有什么，干了什么。

然而，如果你对其内部的工作流程有一定了解的话，当面对一个机器学习任务时候，这些理论可以帮助你快速的找到恰当的机器学习模型，合适的训练算法，以及一个好的假设集。同时，了解黑箱子内部的构成，有助于你更好地调试参数以及更有效的误差分析。本章讨论的大部分话题对于机器学习模型的理解，构建，以及神经网络（详细参考本书的第二部分）的训练都是非常重要的。

首先我们将以一个简单的线性回归模型为例，讨论两种不同的训练方法来得到模型的最优解：

- 直接使用封闭方程进行求根运算，得到模型在当前训练集上的最优参数（即在训练集上使损失函数达到最小值的模型参数）
- 使用迭代优化方法：梯度下降（GD），在训练集上，它可以逐渐调整模型参数以获得最小的损失函数，最终，参数会收敛到和第一种方法相同的值。同时，我们也会介绍一些梯度下降的变体形式：批量梯度下降（Batch GD）、小批量梯度下降（Mini-batch GD）、随机梯度下降（Stochastic GD），在第二部分的神经网络部分，我们会多次使用它们。

接下来，我们将研究一个更复杂的模型：多项式回归，它可以拟合非线性数据集，由于它比线性模型拥有更多的参数，于是它更容易出现模型的过拟合。因此，我们将介绍如何通过学习曲线去判断模型是否出现了过拟合，并介绍几种正则化方法以减少模型出现过拟合的风险。

最后，我们将介绍两个常用于分类的模型：Logistic 回归和 Softmax 回归

### 提示

在本章中包含许多数学公式，以及一些线性代数和微积分基本概念。为了理解这些公式，你需要知道什么是向量，什么是矩阵，以及它们直接是如何转化的，以及什么是点积，什么是矩阵的逆，什么是偏导数。如果你对这些不是很熟悉的话，你可以阅读本书提供的 Jupyter 在线笔记，它包括了线性代数和微积分的入门指导。对于那些不喜欢数学的人，你也应该快速简单的浏览这些公式。希望它足以帮助你理解大多数的概念。

## 线性回归

在第一章，我们介绍了一个简单的生活满意度回归模型：

$$\text{life\_satisfaction} = \theta_0 + \theta_1 * \text{GDP\_per\_capita}$$

这个模型仅仅是输入量 `GDP_per_capita` 的线性函数，`θ[0]` 和 `θ[1]` 是这个模型的参数，线性模型更一般化的描述指通过计算输入变量的加权和，并加上一个常数偏置项（截距项）来得到一个预测值。如公式 4-1：

公式 4-1：线性回归预测模型

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

- `y_hat` 表示预测结果
- `n` 表示特征的个数
- `x[i]` 表示第 `i` 个特征的值
- `θ[j]` 表示第 `j` 个参数（包括偏置项 `θ[0]` 和特征权重值 `θ[1], θ[2], …, θ[nj]`）

上述公式可以写成更为简洁的向量形式，如公式 4-2：

公式 4-2：线性回归预测模型（向量形式）

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

- `θ` 表示模型的参数向量包括偏置项 `θ[0]` 和特征权重值 `θ[1] 到 θ[n]`
- `θ^T` 表示向量 `θ` 的转置（行向量变为了列向量）
- `x` 为每个样本中特征值的向量形式，包括 `x[1] 到 x[n]`，而且 `x[0]` 恒为 1
- `θ^T · x` 表示 `θ^T` 和 `x` 的点积
- `h[θ]` 表示参数为 `θ` 的假设函数

怎么样去训练一个线性回归模型呢？好吧，回想一下，训练一个模型指的是设置模型的参数使得这个模型在训练集的表现较好。为此，我们首先需要找到一个衡量模型好坏的评定方法。在第二章，我们介绍到在回归模型上，最常见的评定标准是均方根误差（RMSE，详见公式 2-1）。因此，为了训练一个线性回归模型，你需要找到一个 `θ` 值，它使得均方根误差（标准误差）达到最小值。实践过程中，最小化均方误差比最小化均方根误差更加的简单，这两个过程会得到相同的 `θ`，因为函数在最小值时候的自变量，同样能使函数的方根运算得到最小值。

在训练集 `x` 上使用公式 4-3 来计算线性回归假设 `h[θ]` 的均方差（MSE）。

公式 4-3：线性回归模型的 MSE 损失函数

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

公式中符号的含义大多数都在第二章（详见“符号”）进行了说明，不同的是：为了突出模型的参数向量 `θ`，使用 `h[θ]` 来代替 `h`。以后的使用中为了公式的简洁，使用 `MSE(θ)` 来代替 `MSE(X, h[θ])`。

## 正规方程(The Normal Equation)

为了找到最小化损失函数的 `θ` 值，可以采用公式解，换句话说，就是可以通过解正规方程直接得到最后的结果。

公式 4-4：正规方程

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

- `θ_hat` 指最小化损失 `θ` 的值

- $y$  是一个向量，其包含了  $y^{(1)}$  到  $y^{(m)}$  的值

让我们生成一些近似线性的数据（如图 4-1）来测试一下这个方程。

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

图 4-1：随机线性数据集

现在让我们使用正规方程来计算  $\theta_{\text{hat}}$ ，我们将使用 Numpy 的线性代数模块 (`np.linalg`) 中的 `inv()` 函数来计算矩阵的逆，以及 `dot()` 方法来计算矩阵的乘法。

```
X_b = np.c_[np.ones((100, 1)), X]
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

我们生产数据的函数实际上是  $y = 4 + 3x[0] + \text{高斯噪声}$ 。让我们看一下最后的计算结果。

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

我们希望最后得到的参数为  $\theta[0] = 4$ ,  $\theta[1] = 3$  而不是  $\theta[0] = 3.865$ ,  $\theta[1] = 3.139$ （译者注：我认为应该  $\theta[0] = 4.2150$ ,  $\theta[1] = 2.7701$ ）。这已经足够了，由于存在噪声，参数不可能达到到原始函数的值。

现在我们能够使用  $\theta_{\text{hat}}$  来进行预测：

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new]
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[4.21509616], [9.75532293]])
```

画出这个模型的图像，如图 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：线性回归预测

使用下面的 Scikit-Learn 代码可以达到相同的效果：

```
>>> 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]])
```

## 计算复杂度

正规方程需要计算矩阵  $X^T \cdot X$  的逆，它是一个  $n * n$  的矩阵（ $n$  是特征的个数）。这样一个矩阵求逆的运算复杂度大约在  $O(n^{2.4})$  到  $O(n^3)$  之间，具体值取决于计算方式。换句话说，如果你将你的特征个数翻倍的话，其计算时间大概会变为原来的  $5.3 (2^{2.4})$  到  $8 (2^3)$  倍。

#### 提示

当特征的个数较大的时候（例如：特征数量为 100000），正规方程求解将会非常慢。

有利的一面是，这个方程在训练集上对于每一个实例来说是线性的，其复杂度为  $O(m)$ ，因此只要有能放得下它的内存空间，它就可以对大规模数据进行训练。同时，一旦你得到了线性回归模型（通过解正规方程或者其他的方法），进行预测是非常快的。因为模型中计算复杂度对于要进行预测的实例数量和特征个数都是线性的。换句话说，当实例个数变为原来的两倍多的时候（或特征个数变为原来的两倍多），预测时间也仅仅是原来的两倍多。

接下来，我们将介绍另一种方法去训练模型。这种方法适合在特征个数非常多，训练实例非常多，内存无法满足要求的时候使用。

## 梯度下降

梯度下降是一种非常通用的优化算法，它能够很好地解决一系列问题。梯度下降的整体思路是通过的迭代来逐渐调整参数使得损失函数达到最小值。

假设浓雾下，你迷失在了大山中，你只能感受到自己脚下的坡度。为了最快到达山底，一个最好的方法就是沿着坡度最陡的地方下山。这其实就是在梯度下降所做的：它计算误差函数关于参数向量  $\theta$  的局部梯度，同时它沿着梯度下降的方向进行下一次迭代。当梯度值为零的时候，就达到了误差函数最小值。

具体来说，开始时，需要选定一个随机的  $\theta$ （这个值称为随机初始值），然后逐渐去改进它，每一次变化一小步，每一步都试着降低损失函数（例如：均方差损失函数），直到算法收敛到一个最小值（如图 4-3）。

图 4-3：梯度下降

在梯度下降中一个重要的参数是步长，超参数学习率的值决定了步长的大小。如果学习率太小，必须经过多次迭代，算法才能收敛，这是非常耗时的（如图 4-4）。

图 4-4: 学习率过小

另一方面，如果学习率太大，你将跳过最低点，到达山谷的另一面，可能下一次的值比上一次还要大。这可能使的算法是发散的，函数值变得越来越大，永远不可能找到一个好的答案（如图 4-5）。

图 4-5：学习率过大

最后，并不是所有的损失函数看起来都像一个规则的碗。它们可能是洞，山脊，高原和各种不规则的地形，使它们收敛到最小值非常的困难。图 4-6 显示了梯度下降的两个主要挑战：如果随机初始值选在了图像的左侧，则它将收敛到局部最小值，这个值要比全局最小值要大。如果它从右侧开始，那么跨越高原将需要很长时间，如果你早早地结束训练，你将永远到不了全局最小值。

图 4-6：梯度下降的陷阱

幸运的是线性回归模型的均方差损失函数是一个凸函数，这意味着如果你选择曲线上的任意两点，它们的连线段不会与曲线发生交叉（译者注：该线段不会与曲线有第三个交点）。这意味着这个损失函数没有局部最小值，仅仅只有一个全局最小值。同时它也是一个斜率不能突变的连续函数。这两个因素导致了一个好的结果：梯度下降可以无限接近全局最小值。（只要你训练时间足够长，同时学习率不是太大）。

事实上，损失函数的图像呈现碗状，但是不同特征的取值范围相差较大的时，这个碗可能是细长的。图 4-7 展示了梯度下降在不同训练集上的表现。在左图中，特征 1 和特征 2 有着相同的数值尺度。在右图中，特征 1 比特征 2 的取值要小的多，由于特征 1 较小，因此损失函数改变时， $\theta[1]$  会有较大的变化，于是这个图像会在  $\theta[1]$  轴方向变得细长。

图 4-7：有无特征缩放的梯度下降

正如你看到的，左面的梯度下降可以直接快速地到达最小值，然而在右面的梯度下降第一次前进的方向几乎和全局最小值的方向垂直，并且最后到达一个几乎平坦的山谷，在平坦的山谷走了很长时间。它最终会达到最小值，但它需要很长时间。

#### 提示

当我们使用梯度下降的时候，应该确保所有的特征有着相近的尺度范围（例如：使用 Scikit Learn 的 `StandardScaler` 类），否则它将需要很长的时间才能够收敛。

这幅图也表明了一个事实：训练模型意味着找到一组模型参数，这组参数可以在训练集上使得损失函数最小。这是对于模型参数空间的搜索，模型的参数越多，参数空间的维度越多，找到合适的参数越困难。例如在 300 维的空间找到一枚针要比在三维空间里找到一枚针复杂的多。幸运的是线性回归模型的损失函数是凸函数，这个最优参数一定在碗的底部。

## 批量梯度下降

使用梯度下降的过程中，你需要计算每一个  $\theta[j]$  下损失函数的梯度。换句话说，你需要计算当  $\theta[j]$  变化一点点时，损失函数改变了多少。这称为偏导数，它就像当你面对东方的时候问：“我脚下的坡度是多少？”然后面向北方的时候问同样的问题（如果你能想象一个超过三维的宇宙，可以对所有的方向都这样做）。公式 4-5 计算关于  $\theta[j]$  的损失函数的偏导数，记为： $\partial \text{MSE} / \partial \theta[j]$ 。

公式 4-5：损失函数的偏导数

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

为了避免单独计算每一个梯度，你也可以使用公式 4-6 来一起计算它们。梯度向量记为  $\nabla[\theta] \text{MSE}(\theta)$ ，其包含了损失函数所有的偏导数（每个模型参数只出现一次）。

公式 4-6：损失函数的梯度向量

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

提示

在这个方程中每一步计算时都包含了整个训练集  $\mathbf{x}$ ，这也是为什么这个算法称为批量梯度下降：每一次训练过程都使用所有的的训练数据。因此，在大数据集上，其会变得相当的慢（但是我们接下来将会介绍更快的梯度下降算法）。然而，梯度下降的运算规模和特征的数量成正比。训练一个数千数量特征的线性回归模型使用梯度下降要比使用正规方程快的多。

一旦求得了方向是上山的梯度向量，你就可以向着相反的方向去下山。这意味着从  $\theta$  中减去  $\nabla_{\theta} MSE(\theta)$ 。学习率  $\eta$  和梯度向量的积决定了下山时每一步的大小，如公式 4-7。

公式 4-7：梯度下降步长

$$\theta^{(next\ step)} = \theta - \eta \nabla_{\theta} MSE(\theta)$$

让我们看一下这个算法的应用：

```
eta = 0.1 # 学习率
n_iterations = 1000
m = 100

theta = np.random.randn(2, 1) # 随机初始值
for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

这不是太难，让我们看一下最后的结果  $\theta$ ：

```
>>> theta
array([[4.2109616], [2.77011339]])
```

看！正规方程的表现非常好。完美地求出了梯度下降的参数。但是当你换一个学习率会发生什么？图 4-8 展示了使用了三个不同的学习率进行梯度下降的前 10 步运算（虚线代表起始位置）。

图 4-8：不同学习率的梯度下降

在左面的那副图中，学习率是最小的，算法几乎不能求出最后的结果，而且还会花费大量的时间。在中间的这幅图中，学习率的表现看起来不错，仅仅几次迭代后，它就收敛到了最后的结果。在右面的那副图中，学习率太大了，算法是发散的，跳过了所有的训练样本，同时每一步都离正确的结果越来越远。

为了找到一个好的学习率，你可以使用网格搜索（详见第二章）。当然，你一般会限制迭代的次数，以便网格搜索可以消除模型需要很长时间才能收敛这个问题。

你可能想知道如何选取迭代的次数。如果它太小了，当算法停止的时候，你依然没有找到最优解。如果它太大了，算法会非常的耗时同时后来的迭代参数也不会发生改变。一个简单的解决方法是：设置一个非常大的迭代次数，但是当梯度向量变得

非常小的时候，结束迭代。非常小指的是：梯度向量小于一个值  $\epsilon$ （称为容差）。这时候可以认为梯度下降几乎已经达到了最小值。

收敛速率：

当损失函数是凸函数，同时它的斜率不能突变（就像均方差损失函数那样），那么它的批量梯度下降算法固定学习率之后，它的收敛速率是  $O(1/\text{iterations})$ 。换句话说，如果你将容差  $\epsilon$  缩小 10 倍后（这样可以得到一个更精确的结果），这个算法的迭代次数大约会变成原来的 10 倍。

## 随机梯度下降

批量梯度下降的最要问题是计算每一步的梯度时都需要使用整个训练集，这导致在规模较大的数据集上，其会变得非常的慢。与其完全相反的随机梯度下降，在每一步的梯度计算上只随机选取训练集中的一一个样本。很明显，由于每一次的操作都使用了非常少的数据，这样使得算法变得非常快。由于每一次迭代，只需要在内存中有一个实例，这使随机梯度算法可以在大规模训练集上使用。

另一方面，由于它的随机性，与批量梯度下降相比，其呈现出更多的不规律性：它到达最小值不是平缓的下降，损失函数会忽高忽低，只是在大体上呈下降趋势。随着时间的推移，它会非常的靠近最小值，但是它不会停止在一个值上，它会一直在这个值附近摆动（如图 4-9）。因此，当算法停止的时候，最后的参数还不错，但不是最优值。

图 4-9：随机梯度下降

当损失函数很不规则时（如图 4-6），随机梯度下降算法能够跳过局部最小值。因此，随机梯度下降在寻找全局最小值上比批量梯度下降表现要好。

虽然随机性可以很好的跳过局部最优值，但同时它却不能达到最小值。解决这个难题的一个办法是逐渐降低学习率。开始时，走的每一步较大（这有助于快速前进同时跳过局部最小值），然后变得越来越小，从而使算法到达全局最小值。这个过程被称为模拟退火，因为它类似于熔融金属慢慢冷却的冶金学退火过程。决定每次迭代的学习率的函数称为 `learning schedule`。如果学习速度降低得过快，你可能会陷入局部最小值，甚至在到达最小值的半路就停止了。如果学习速度降低得太慢，你可能在最小值的附近长时间摆动，同时如果过早停止训练，最终只会出现次优解。

下面的代码使用一个简单的 `learning schedule` 来实现随机梯度下降：

```
n_epochs = 50
t0, t1 = 5, 50 #learning_schedule 的超参数

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

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

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
```

按习惯来讲，我们进行  $m$  轮的迭代，每一轮迭代被称为一代。在整个训练集上，随机梯度下降迭代了 1000 次时，一般在第 50 次的时候就可以达到一个比较好的结果。

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

图 4-10 展示了前 10 次的训练过程（注意每一步的不规则程度）。

图 4-10：随机梯度下降的前 10 次迭代

由于每个实例的选择是随机的，有的实例可能在每一代中都被选到，这样其他的实例也可能一直不被选到。如果你想保证每一代迭代过程，算法可以遍历所有实例，一种方法是将训练集打乱重排，然后选择一个实例，之后再继续打乱重排，以此类推一直进行下去。但是这样收敛速度会非常的慢。

通过使用 Scikit-Learn 完成线性回归的随机梯度下降，你需要使用 `SGDRegressor` 类，这个类默认优化的是均方差损失函数。下面的代码迭代了 50 代，其学习率为 0.1 (`eta0=0.1`)，使用默认的 `learning schedule`（与前面的不一样），同时也没有添加任何正则项 (`penalty = None`)：

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X,y.ravel())
```

你可以再一次发现，这个结果非常的接近正规方程的解：

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.18380366]), array([2.74205299]))
```

## 小批量梯度下降

最后一个梯度下降算法，我们将介绍小批量梯度下降算法。一旦你理解了批量梯度下降和随机梯度下降，再去理解小批量梯度下降是非常简单的。在迭代的每一步，批量梯度使用整个训练集，随机梯度时候用仅仅一个实例，在小批量梯度下降中，它则使用一个随机的小型实例集。它比随机梯度的主要优点在于你可以通过矩阵运算的硬件优化得到一个较好的训练表现，尤其当你使用 GPU 进行运算的时候。

小批量梯度下降在参数空间上的表现比随机梯度下降要好的多，尤其在有大量的小型实例集时。作为结果，小批量梯度下降会比随机梯度更靠近最小值。但是，另一方面，它有可能陷在局部最小值中（在遇到局部最小值问题的情况下，和我们之前看到的线性回归不一样）。图 4-11 显示了训练期间三种梯度下降算法在参数空间中所采用的路径。他们都接近最小值，但批量梯度的路径最后停在了最小值，而随机梯度和小批量梯度最后都在最小值附近摆动。但是，不要忘记，批量梯度需要花费大量时间来完成每一步，但是，如果你使用了一个较好的 `learning schedule`，随机梯度和小批量梯度也可以得到最小值。

图 4-11：参数空间的梯度下降路径

让我比较一下目前我们已经探讨过的对线性回归的梯度下降算法。如表 4-1 所示，其中  $m$  表示训练样本的个数， $n$  表示特征的个数。

表 4-1：比较线性回归的不同梯度下降算法

**提示**

上述算法在完成训练后，得到的参数基本没什么不同，它们会得到非常相似的模型，最后会以一样的方式去进行预测。

## 多项式回归

如果你的数据实际上比简单的直线更复杂呢？令人惊讶的是，你依然可以使用线性模型来拟合非线性数据。一个简单的方法是对每个特征进行加权后作为新的特征，然后训练一个线性模型在这个扩展的特征集。这种方法称为多项式回归。

让我们看一个例子。首先，我们根据一个简单的二次方程（并加上一些噪声，如图 4-12）来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)
```

图 4-12：生产加入噪声的非线性数据

很清楚的看出，直线不能恰当的拟合这些数据。于是，我们使用 Scikit-Learning 的 `PolynomialFeatures` 类进行训练数据集的转换，让训练集中每个特征的平方（2 次多项式）作为新特征（在这种情况下，仅存在一个特征）：

```
>>> 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` 现在包含原始特征 `x` 并加上了这个特征的平方 `x^2`。现在你可以在这个扩展训练集上使用 `LinearRegression` 模型进行拟合，如图 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

还是不错的，模型预测函数 `y_hat = 0.56 x[1]^2 + 0.93x[1] + 1.78`，事实上原始函数为 `y = 0.5x[1]^2 + 1.0x[1] + 2.0` 再加上一些高斯噪声。

请注意，当存在多个特征时，多项式回归能够找出特征之间的关系（这是普通线性回归模型无法做到的）。这是因为 `LinearRegression` 会自动添加当前阶数下特征的所有组合。例如，如果有两个特征 `a, b`，使用 3 阶（`degree=3`）的 `LinearRegression` 时，不仅有 `a^2, a^3, b^2` 以及 `b^3`，同时也会有它们的其他组合项 `ab, a^2b, ab^2`。

**提示**

`PolynomialFeatures(degree=d)` 把一个包含 `n` 个特征的数组转换为一个包含  $(n+d)!/(d!n!)$  特征的数组，`n!` 表示 `n` 的阶乘，等于  $1 * 2 * 3 \dots * n$ 。小心大量特征的组合爆炸！

## 学习曲线

如果你使用一个高阶的多项式回归，你可能发现它的拟合程度要比普通的线性回归要好的多。例如，图 4-14 使用一个 300 阶的多项式模型去拟合之前的数据集，并同简单线性回归、2 阶的多项式回归进行比较。注意 300 阶的多项式模型如何摆动以尽可能接近训练实例。

图 4-14：高阶多项式回归

当然，这种高阶多项式回归模型在这个训练集上严重过拟合了，线性模型则欠拟合。在这个训练集上，二次模型有着较好的泛化能力。那是因为在生成数据时使用了二次模型，但是一般我们不知道这个数据生成函数是什么，那我们该如何决定我们模型的复杂度呢？你如何告诉我你的模型是过拟合还是欠拟合？

在第二章，你可以使用交叉验证来估计一个模型的泛化能力。如果一个模型在训练集上表现良好，通过交叉验证指标却得出其泛化能力很差，那么你的模型就是过拟合了。如果在这两方面都表现不好，那么它就是欠拟合了。这种方法可以告诉我们，你的模型是太复杂还是太简单了。

另一种方法是观察学习曲线：画出模型在训练集上的表现，同时画出以训练集规模为自变量的训练集函数。为了得到图像，需要在训练集的不同规模子集上进行多次训练。下面的代码定义了一个函数，用来画出给定训练集后的模型学习曲线：

```
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_predict, y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict, y_val))

    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

我们一起看一下简单线性回归模型的学习曲线（图 4-15）：

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```

图 4-15：学习曲线

这幅图值得我们深究。首先，我们观察训练集的表现：当训练集只有一两个样本的时候，模型能够非常好的拟合它们，这也是为什么曲线是从零开始的原因。但是当加入了一些新的样本的时候，训练集上的拟合程度变得难以接受，出现这种情况有两个原因，一是因为数据中含有噪声，另一个是数据根本不是线性的。因此随着数据规模的增大，误差也会一直增大，直到达到高原地带并趋于稳定，在之后，继续加入新的样本，模型的平均误差不会变得更好或者更差。我们继续来看模型在验证集上的表现，当以非常少的样本去训练时，模型不能恰当的泛化，也就是为什么验证误差一开始是非常大的。当训练样本变多的时候，模型学习的东西变多，验证误差开始缓慢的下降。但是一条直线不可能很好的拟合这些数据，因此最后误差会到达在一个高原地带并趋于稳定，最后和训练集的曲线非常接近。

上面的曲线表现了一个典型的欠拟合模型，两条曲线都到达高原地带并趋于稳定，并且最后两条曲线非常接近，同时误差值非常大。

#### 提示

如果你的模型在训练集上是欠拟合的，添加更多的样本是没用的。你需要使用一个更复杂的模型或者找到更好的特征。

现在让我们看一个在相同数据上 10 阶多项式模型拟合的学习曲线（图 4-16）：

```
from sklearn.pipeline import Pipeline
polynomial_regression = Pipeline([
    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
    ("sgd_reg", LinearRegression()),
])
plot_learning_curves(polynomial_regression, X, y)
```

这幅图像和之前的有一点点像，但是其有两个非常重要的不同点：

- 在训练集上，误差要比线性回归模型低的多。
- 图中的两条曲线之间有间隔，这意味着模型在训练集上的表现要比验证集上好的多，这也是模型过拟合的显著特点。当然，如果你使用了更大的训练数据，这两条曲线最后会非常的接近。

图 4-16：多项式模型的学习曲线

#### 提示

改善模型过拟合的一种方法是提供更多的训练数据，直到训练误差和验证误差相等。

#### 偏差和方差的权衡

在统计和机器学习领域有个重要的理论：一个模型的泛化误差由三个不同误差的和决定：

- 偏差：泛化误差的这部分误差是由于错误的假设决定的。例如实际是一个二次模型，你却假设了一个线性模型。一个高偏差的模型最容易出现欠拟合。
- 方差：这部分误差是由于模型对训练数据的微小变化较为敏感，一个自由度的模型更容易有高的方差（例如一个高阶多项式模型），因此会导致模型过拟合。
- 不可约误差：这部分误差是由于数据本身的噪声决定的。降低这部分误差的唯一方法就是进行数据清洗（例如：修复数据源，修复坏的传感器，识别和剔除异常值）。

## 线性模型的正则化

正如我们在第一和第二章看到的那样，降低模型的过拟合的好方法是正则化这个模型（即限制它）：模型有越少的自由度，就越难以拟合数据。例如，正则化一个多项式模型，一个简单的方法就是减少多项式的阶数。

对于一个线性模型，正则化的典型实现就是约束模型中参数的权重。接下来我们将介绍三种不同约束权重的方法：Ridge 回归，Lasso 回归和 Elastic Net。

## 岭 (Ridge) 回归

岭回归（也称为 Tikhonov 正则化）是线性回归的正则化版：在损失函数上直接加上一个正则项  $\alpha \sum \theta[i]^2, i = 1 \rightarrow n$ 。这使得学习算法不仅能够拟合数据，而且能够使模型的参数权重尽量的小。注意到这个正则项只有在训练过程中才会被加到损失函数。当得到完成训练的模型后，我们应该使用没有正则化的测量方法去评价模型的表现。

### 提示

一般情况下，训练过程使用的损失函数和测试过程使用的评价函数是不一样的。除了正则化，还有一个不同：训练时的损失函数应该在优化过程中易于求导，而在测试过程中，评价函数更应该接近最后的客观表现。一个好的例子：在分类训练中我们使用对数损失（马上我们会讨论它）作为损失函数，但是我们却使用精确率/召回率来作为它的评价函数。

超参数  $\alpha$  决定了你想正则化这个模型的强度。如果  $\alpha = 0$  那此时的岭回归便变为了线性回归。如果  $\alpha$  非常的大，所有的权重最后都接近于零，最后结果将是一条穿过数据平均值的水平直线。公式 4-8 是岭回归的损失函数：

公式 4-8：岭回归损失函数

$$J(\theta) = MSE(\theta) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

值得注意的是偏差  $\theta[0]$  是没有被正则化的（累加运算的开始是  $i=1$  而不是  $i=0$ ）。如我定义  $w$  作为特征的权重向量（ $\theta[1]$  到  $\theta[n]$ ），那么正则项可以简写成  $1/2 (||w||_2)^2$ ，其中  $||\cdot||_2$  表示权重向量的  $l_2$  范数。对于梯度下降来说仅仅在均方差梯度向量（公式 4-6）加上一项  $\alpha w$ 。

### 提示

在使用岭回归前，对数据进行放缩（可以使用 `StandardScaler`）是非常重要的，算法对于输入特征的数值尺度（`scale`）非常敏感。大多数的正则化模型都是这样的。

图 4-17 展示了在相同线性数据上使用不同  $\alpha$  值的岭回归模型最后的表现。左图中，使用简单的岭回归模型，最后得到了线性的预测。右图中的数据首先使用 10 阶的 `PolynomialFeatures` 进行扩展，然后使用 `StandardScaler` 进行缩放，最后将岭模型应用在处理过后的特征上。这就是带有岭正则项的多项式回归。注意当  $\alpha$  增大的时候，导致预测曲线变得扁平（即少了极端值，多了一般值），这样减少了模型的方差，却增加了模型的偏差。

对线性回归来说，对于岭回归，我们可以使用封闭方程去计算，也可以使用梯度下降去处理。它们的缺点和优点是一样的。公式 4-9 表示封闭方程的解（矩阵  $A$  是一个除了左上角有一个  $\theta$  的  $n * n$  的单位矩，这个  $\theta$  代表偏差项。译者注：偏差  $\theta[0]$  不被正则化）。

图 4-17：岭回归

公式 4-9：岭回归的封闭方程的解

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

下面是如何使用 Scikit-Learn 来进行封闭方程的求解（使用 Cholesky 法进行矩阵分解对公式 4-9 进行变形）：

```
>>> 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])
```

使用随机梯度法进行求解：

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([ 1.13500145])
```

`penalty` 参数指的是正则项的惩罚类型。指定 `l2` 表明你要在损失函数上添加一项：权重向量  $\ell_2$  范数平方的一半，这就是简单的岭回归。

## Lasso 回归

Lasso 回归（也称 Least Absolute Shrinkage，或者 Selection Operator Regression）是另一种正则化版的线性回归：就像岭回归那样，它也在损失函数上添加了一个正则化项，但是它使用权重向量的  $\ell_1$  范数而不是权重向量  $\ell_2$  范数平方的一半。（如公式 4-10）

公式 4-10：Lasso 回归的损失函数

$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^n |\theta_i|$$

图 4-18 展示了和图 4-17 相同的事情，仅仅是用 Lasso 模型代替了 Ridge 模型，同时调小了  $\alpha$  的值。

图 4-18: Lasso 回归

Lasso 回归的一个重要特征是它倾向于完全消除最不重要的特征的权重（即将它们设置为零）。例如，右图中的虚线所示 ( $\alpha = 10^{-7}$ )，曲线看起来像一条二次曲线，而且几乎是线性的，这是因为所有的高阶多项特征都被设置为零。换句话说，Lasso 回归自动的进行特征选择同时输出一个稀疏模型（即，具有很少的非零权重）。

你可以从图 4-19 知道为什么会出现这种情况：在左上角图中，后背景的等高线（椭圆）表示了没有正则化的均方差损失函数 ( $\alpha = 0$ )，白色的小圆圈表示在当前损失函数上批量梯度下降的路径。前背景的等高线（菱形）表示  $\ell_1$  惩罚，黄色的三角形表示了仅在这个惩罚下批量梯度下降的路径 ( $\alpha \rightarrow \infty$ )。注意路径第一次是如何到达  $\theta[1] = 0$ ，然后向下滚动直到它到达  $\theta[2] = 0$ 。在右上角图中，等高线表示的是相同损失函数再加上一个  $\alpha = 0.5$  的  $\ell_1$  惩罚。这幅图中，它的全局最小值在  $\theta[2] = 0$  这根轴上。批量梯度下降首先到达  $\theta[2] = 0$ ，然后向下滚动直到达到全局最小值。两个底部图显示了相同的情况，只是使用了  $\ell_2$  惩罚。规则化的最小值比非规范化的最小值更接近于  $\theta = 0$ ，但权重不能完全消除。

图 4-19: Ridge 回归和 Lasso 回归对比

**提示**

在 Lasso 损失函数中，批量梯度下降的路径趋向与在低谷有一个反弹。这是因为  $\theta[2] = 0$  时斜率会有一个突变。为了最后真正收敛到全局最小值，你需要逐渐的降低学习率。

Lasso 损失函数在  $\theta[i] = 0, i = 1, 2, \dots, n$  处无法进行微分运算，但是梯度下降如果你使用子梯度向量  $g$  后它可以在任何  $\theta[i] = 0$  的情况下进行计算。公式 4-11 是在 Lasso 损失函数上进行梯度下降的子梯度向量公式。

公式 4-11: Lasso 回归子梯度向量

$$g(\theta, J) = \nabla_{\theta} MSE(\theta) + \alpha \begin{pmatrix} sign(\theta_1) \\ sign(\theta_2) \\ \vdots \\ sign(\theta_n) \end{pmatrix} \text{ where } sign(\theta_i) = \begin{cases} -1, & \theta_i < 0 \\ 0, & \theta_i = 0 \\ +1, & \theta_i > 0 \end{cases}$$

下面是一个使用 Scikit-Learn 的 Lasso 类的小例子。你也可以使用 `SGDRegressor(penalty="l1")` 来代替它。

```
>>> 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])
```

**弹性网络 (ElasticNet)**

弹性网络介于 Ridge 回归和 Lasso 回归之间。它的正则项是 Ridge 回归和 Lasso 回归正则项的简单混合，同时你可以控制它们的混合率  $r$ ，当  $r = 0$  时，弹性网络就是 Ridge 回归，当  $r = 1$  时，其就是 Lasso 回归。具体表示如公式 4-12。

公式 4-12：弹性网络损失函数

$$J(\theta) = MSE(\theta) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^n \theta_i^2$$

那么我们该如何选择线性回归，岭回归，Lasso 回归，弹性网络呢？一般来说有一点正则项的表现更好，因此通常你应该避免使用简单的线性回归。岭回归是一个很好的首选项，但是如果你的特征仅有少数是真正有用的，你应该选择 Lasso 和弹性网络。就像我们讨论的那样，它能够将无用特征的权重降为零。一般来说，弹性网络的表现要比 Lasso 好，因为当特征数量比样本的数量大的时候，或者特征之间有很强的相关性时，Lasso 可能会表现的不规律。下面是一个使用 Scikit-Learn `ElasticNet` (`l1_ratio` 指的就是混合率  $r$ ) 的简单样本：

```
>>> 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])
```

**早期停止法 (Early Stopping)**

对于迭代学习算法，有一种非常特殊的正则化方法，就像梯度下降在验证错误达到最小值时立即停止训练那样。我们称为早期停止法。图 4-20 表示使用批量梯度下降来训练一个非常复杂的模型（一个高阶多项式回归模型）。随着训练的进行，算法一直学习，它在训练集上的预测误差（RMSE）自然而然的下降。然而一段时间后，验证误差停止下降，并开始上升。这意味着模型在训练集上开始出现过拟合。一旦验证错误达到最小值，便提早停止训练。这种简单有效的正则化方法被 Geoffrey Hinton 称为“完美的免费午餐”

图 4-20：早期停止法

#### 提示

随机梯度和小批量梯度下降不是平滑曲线，你可能很难知道它是否达到最小值。一种解决方案是，只有在验证误差高于最小值一段时间后（你确信该模型不会变得更好了），才停止，之后将模型参数回滚到验证误差最小值。

下面是一个早期停止法的基础应用：

```
from sklearn.base import clone
sgd_reg = SGDRegressor(n_iter=1, warm_start=True, penalty=None, learning_rate=""
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)
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val_predict, y_val)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

注意：当 `warm_start=True` 时，调用 `fit()` 方法后，训练会从停下来的地方继续，而不是从头重新开始。

## 逻辑回归

正如我们在第 1 章中讨论的那样，一些回归算法也可以用于分类（反之亦然）。`Logistic` 回归（也称为 `Logit` 回归）通常用于估计一个实例属于某个特定类别的概率（例如，这电子邮件是垃圾邮件的概率是多少？）。如果估计的概率大于 50%，那么模型预测这个实例属于当前类（称为正类，标记为“1”），反之预测它不属于当前类（即它属于负类，标记为“0”）。这样便成为了一个二元分类器。

## 概率估计

那么它是怎样工作的？就像线性回归模型一样，`Logistic` 回归模型计算输入特征的加权和（加上偏差项），但它不像线性回归模型那样直接输出结果，而是把结果输入 `logistic()` 函数进行二次加工后进行输出（详见公式 4-13）。

公式 4-13：逻辑回归模型的概率估计（向量形式）

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\theta^T \cdot \mathbf{x})$$

`Logistic` 函数（也称为 `logit`），用  $\sigma()$  表示，其是一个 `sigmoid` 函数（图像呈 S 型），它的输出是一个介于 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 Logistic 回归模型估计得到了  $x$  属于正类的概率  $p_{\text{hat}} = h[\theta](x)$ ，那它很容易得到预测结果  $y_{\text{hat}}$ （见公式 4-15）。

公式 4-15：逻辑回归预测模型

$$\hat{y} = \begin{cases} 0, & \hat{p} < 0.5 \\ 1, & \hat{p} \geq 0.5 \end{cases}$$

注意当  $t < 0$  时  $\sigma(t) < 0.5$ ，当  $t \geq 0$  时  $\sigma(t) \geq 0.5$ ，因此当  $\theta^T \cdot x$  是正数的话，逻辑回归模型输出 1，如果它是负数的话，则输出 0。

## 训练和损失函数

好，现在你知道了 Logistic 回归模型如何估计概率并进行预测。但是它是如何训练的？训练的目的是设置参数向量  $\theta$ ，使得正例 ( $y = 1$ ) 概率增大，负例 ( $y = 0$ ) 的概率减小，其通过在单个训练实例  $x$  的损失函数来实现（公式 4-16）。

公式 4-16：单个样本的损失函数

$$c(\theta) = \begin{cases} -\log(\hat{p}), & y = 1 \\ -\log(1 - \hat{p}), & y = 0 \end{cases}$$

这个损失函数是合理的，因为当  $t$  接近 0 时， $-\log(t)$  变得非常大，所以如果模型估计一个正例概率接近于 0，那么损失函数将会很大，同时如果模型估计一个负例的概率接近 1，那么损失函数同样会很大。另一方面，当  $t$  接近于 1 时， $-\log(t)$  接近 0，所以如果模型估计一个正例概率接近于 0，那么损失函数接近于 0，同时如果模型估计一个负例的概率接近 0，那么损失函数同样会接近于 0，这正是我们想的。

整个训练集的损失函数只是所有训练实例的平均值。可以用一个表达式（你可以很容易证明）来统一表示，称为对数损失，如公式 4-17 所示。

公式 4-17：逻辑回归的损失函数（对数损失）

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})]$$

但是这个损失函数对于求解最小化损失函数的  $\theta$  是没有公式解的（没有等价的正规方程）。但好消息是，这个损失函数是凸的，所以梯度下降（或任何其他优化算法）一定能够找到全局最小值（如果学习速率不是太大，并且你等待足够长的时间）。公式 4-18 给出了损失函数关于第  $j$  个模型参数  $\theta[j]$  的偏导数。

公式 4-18：逻辑回归损失函数的偏导数

$$\frac{\partial}{\partial \theta_j} J(\theta_j) = \frac{1}{m} \sum_{i=1}^m (\sigma(\theta^T \cdot \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

这个公式看起来非常像公式 4-5：首先计算每个样本的预测误差，然后误差项乘以第  $j$  项特征值，最后求出所有训练样本的平均值。一旦你有了包含所有的偏导数的梯度向量，你便可以在梯度向量上使用批量梯度下降算法。也就是说：你已经知道如何训练 Logistic 回归模型。对于随机梯度下降，你当然只需要每一次使用一个实例，对于小批量梯度下降，你将每一次使用一个小型实例集。

## 决策边界

我们使用鸢尾花数据集来分析 Logistic 回归。这是一个著名的数据集，其中包含 150 朵三种不同的鸢尾花的萼片和花瓣的长度和宽度。这三种鸢尾花为：Setosa, Versicolor, Virginica（如图 4-22）。

图 4-22：三种不同的鸢尾花

让我们尝试建立一个分类器，仅仅使用花瓣的宽度特征来识别 Virginica，首先让我们加载数据：

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target_names', 'feature_names', 'target', 'DESCR']
>>> X = iris["data"][:, 3:] # petal width
>>> y = (iris["target"] == 2).astype(np.int)
```

接下来，我们训练一个逻辑回归模型：

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)
```

我们来看看模型估计的花瓣宽度从 0 到 3 厘米的概率估计（如图 4-23）：

```
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")
```

图 4-23：概率估计和决策边界

Virginica 花的花瓣宽度（用三角形表示）在 1.4 厘米到 2.5 厘米之间，而其他种类的花（由正方形表示）通常具有较小的花瓣宽度，范围从 0.1 厘米到 1.8 厘米。注意，它们之间会有一些重叠。在大约 2 厘米以上时，分类器非常肯定这朵花是 Virginica 花（分类器此时输出一个非常高的概率值），而在 1 厘米以下时，它非常肯定这朵花不是 Virginica 花（不是 Virginica 花有非常高的概率）。在这两个极端之间，分类器是不确定的。但是，如果你使用它进行预测（使用 `predict()` 方法而不是 `predict_proba()` 方法），它将返回一个最可能的结果。因此，在 1.6 厘米左右存在一个决策边界，这时两类情况出现的概率都等于 50%：如果花瓣宽度大于 1.6 厘米，则分类器将预测该花是 Virginica，否则预测它不是（即使它有可能错了）：

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

图 4-24 表示相同的数据集，但是这次使用了两个特征进行判断：花瓣的宽度和长度。一旦训练完毕，Logistic 回归分类器就可以根据这两个特征来估计一朵花是 Virginica 的可能性。虚线表示这时两类情况出现的概率都等于 50%：这是模型的决策边界。请注意，它是一个线性边界。每条平行线都代表一个分类标准下的两两个不同类的概率，从 15%（左下角）到 90%（右上角）。越过右上角分界线的点都有超过 90% 的概率是 Virginica 花。

图 4-24：线性决策边界

就像其他线性模型，逻辑回归模型也可以  $\ell_1$  或者  $\ell_2$  惩罚使用进行正则化。Scikit-Learn 默认添加了  $\ell_2$  惩罚。

#### 注意

在 Scikit-Learn 的 LogisticRegression 模型中控制正则化强度的超参数不是  $\alpha$ （与其他线性模型一样），而是它的逆： $c$ 。 $c$  的值越大，模型正则化强度越低。

## Softmax 回归

Logistic 回归模型可以直接推广到支持多类别分类，不必组合和训练多个二分类器（如第 3 章所述），其称为 Softmax 回归或多类别 Logistic 回归。

这个想法很简单：当给定一个实例  $x$  时，Softmax 回归模型首先计算  $k$  类的分数  $s_k(x)$ ，然后将分数应用在 Softmax 函数（也称为归一化指数）上，估计出每类的概率。计算  $s_k(x)$  的公式看起来很熟悉，因为它就像线性回归预测的公式一样（见公式 4-19）。

公式 4-19： $k$  类的 Softmax 得分

$$s_k(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

注意，每个类都有自己独一无二的参数向量  $\theta[k]$ 。所有这些向量通常作为行放在参数矩阵  $\theta$  中。

一旦你计算了样本  $x$  的每一类的得分，你便可以通过 Softmax 函数（公式 4-20）估计出样本属于第  $k$  类的概率  $p_{\text{hat}}[k]$ ：通过计算  $e$  的  $s_k(x)$  次方，然后对它们进行归一化（除以所有分子的总和）。

公式 4-20：Softmax 函数

$$\hat{p}_k = \sigma(s(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

- $k$  表示有多少类
- $s(\mathbf{x})$  表示包含样本  $x$  每一类得分的向量
- $\sigma(s(\mathbf{x})[k])$  表示给定每一类分数之后，实例  $x$  属于第  $k$  类的概率

和 Logistic 回归分类器一样，Softmax 回归分类器将估计概率最高（它只是得分最高的类）的那类作为预测结果，如公式 4-21 所示。

公式 4-21: Softmax 回归模型分类器预测结果

$$\hat{y} = \operatorname{argmax} \sigma(s(\mathbf{x}))_k = \operatorname{argmax} s_k(\mathbf{x}) = \operatorname{argmax} (\theta_k^T \cdot \mathbf{x})$$

- `argmax` 运算返回一个函数取到最大值的变量值。在这个等式，它返回使  $\sigma(s(\mathbf{x})[k])$  最大时的  $k$  的值

注意

Softmax 回归分类器一次只能预测一个类（即它是多类的，但不是多输出的），因此它只能用于判断互斥的类别，如不同类型的植物。你不能用它来识别一张照片中的多个人。

现在我们知道这个模型如何估计概率并进行预测，接下来将介绍如何训练。我们的目标是建立一个模型在目标类别上有着较高的概率（因此其他类别的概率较低），最小化公式 4-22 可以达到这个目标，其表示了当前模型的损失函数，称为交叉熵，当模型对目标类得出了一个较低的概率，其会惩罚这个模型。交叉熵通常用于衡量待测类别与目标类别的匹配程度（我们将在后面的章节中多次使用它）

公式 4-22：交叉熵

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log \left( \hat{p}_k^{(i)} \right)$$

- 如果对于第  $i$  个实例的目标类是  $k$ ，那么  $y[k]^{(i)} = 1$ ，反之  $y[k]^{(i)} = 0$ 。

可以看出，当只有两个类 ( $K = 2$ ) 时，此损失函数等同于 Logistic 回归的损失函数（对数损失；请参阅公式 4-17）。

交叉熵

交叉熵源于信息论。假设你想要高效地传输每天的天气信息。如果有八个选项（晴天，雨天等），则可以使用 3 位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，更高效的编码“晴天”的方式是：只用一位 (0)。剩下的七项使用四位（从 1 开始）。交叉熵度量每个选项实际发送的平均比特数。如果你对天气的假设是完美的，交叉熵就等于天气本身的熵（即其内部的不确定性）。但是，如果你的假设是错误的（例如，如果经常下雨）交叉熵将会更大，称为 Kullback-Leibler 散度（KL 散度）。

两个概率分布  $p$  和  $q$  之间的交叉熵定义为（分布至少是离散的）：

$$H(p, q) = - \sum_x p(x) \log q(x)$$

这个损失函数关于  $\theta[k]$  的梯度向量为公式 4-23：

公式 4-23： $k$  类交叉熵的梯度向量

$$\nabla_{\theta_k} J(\Theta) = \frac{1}{m} \sum_{i=1}^m \left( \hat{p}_k^{(i)} - y_k^{(i)} \right) \mathbf{x}^{(i)}$$

现在你可以计算每一类的梯度向量，然后使用梯度下降（或者其他优化算法）找到使得损失函数达到最小值的参数矩阵  $\theta$ 。

让我们使用 Softmax 回归对三种鸢尾花进行分类。当你使用 LogisticRegression 对模型进行训练时，Scikit Learn 默认使用的是一对多模型，但是你可以设置 multi\_class 参数为“multinomial”来把它改变为 Softmax 回归。你还必须指定一个支持 Softmax 回归的求解器，例如“lbfgs”求解器（有关更多详细信息，请参阅 Scikit-Learn 的文档）。其默认使用  $\ell_2$  正则化，你可以使用超参数 c 控制它。

```
X = iris[["data"][:, (2, 3)] # petal length, petal width
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial", solver="lbfgs", C=1)
softmax_reg.fit(X, y)
```

所以下次你发现一个花瓣长为 5 厘米，宽为 2 厘米的鸢尾花时，你可以问你的模型你它是哪一类鸢尾花，它会回答 94.2% 是 Virginica 花（第二类），或者 5.8% 是其他鸢尾花。

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[ 6.33134078e-07, 5.75276067e-02, 9.42471760e-01]])是
```

图 4-25: Softmax 回归的决策边界

图 4-25 用不同背景色表示了结果的决策边界。注意，任何两个类之间的决策边界是线性的。该图的曲线表示 Versicolor 类的概率（例如，用 0.450 标记的曲线表示 45% 的概率边界）。注意模型也可以预测一个概率低于 50% 的类。例如，在所有决策边界相遇的地方，所有类的估计概率相等，分别为 33%。

## 练习

1. 如果你有一个数百万特征的训练集，你应该选择哪种线性回归训练算法？
2. 假设你训练集中特征的数值尺度 (scale) 有着非常大的差异，哪种算法会受到影响？有多大的影响？对于这些影响你可以做什么？
3. 训练 Logistic 回归模型时，梯度下降是否会陷入局部最低点？
4. 在有足够的训练时间下，是否所有的梯度下降都会得到相同的模型参数？
5. 假设你使用批量梯度下降法，画出每一代的验证误差。当你发现验证误差一直增大，接下来会发生什么？你怎么解决这个问题？
6. 当验证误差升高时，立即停止小批量梯度下降是否是一个好主意？
7. 哪个梯度下降算法（在我们讨论的那些算法中）可以最快到达解的附近？哪个的确实会收敛？怎么使其他算法也收敛？
8. 假设你使用多项式回归，画出学习曲线，在图上发现学习误差和验证误差之间有着很大的间隙。这表示发生了什么？有哪三种方法可以解决这个问题？
9. 假设你使用岭回归，并发现训练误差和验证误差都很高，并且几乎相等。你的模型表现是高偏差还是高方差？这时你应该增大正则化参数  $\alpha$ ，还是降低它？
10. 你为什么要这样做：
  - 使用岭回归代替线性回归？
  - Lasso 回归代替岭回归？
  - 弹性网络代替 Lasso 回归？
11. 假设你想判断一副图片是室内还是室外，白天还是晚上。你应该选择二个逻辑回归分类器，还是一个 Softmax 分类器？

12. 在 Softmax 回归上应用批量梯度下降的早期停止法（不使用 Scikit-Learn）。

附录 A 提供了这些练习的答案。

## 五、支持向量机

译者：[@QiaoXie](#)

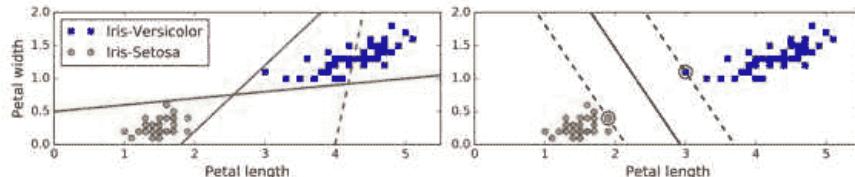
校对者：[@飞龙](#)、[@PeterHo](#)、[@yanmengk](#)、[@YuWang](#)

支持向量机（SVM）是个非常强大并且有多种功能的机器学习模型，能够做线性或者非线性的分类，回归，甚至异常值检测。机器学习领域中最为流行的模型之一，是任何学习机器学习的人必备的工具。SVM 特别适合应用于复杂但中小规模数据集的分类问题。

本章节将阐述支持向量机的核心概念，怎么使用这个强大的模型，以及它是如何工作的。

### 线性支持向量机分类

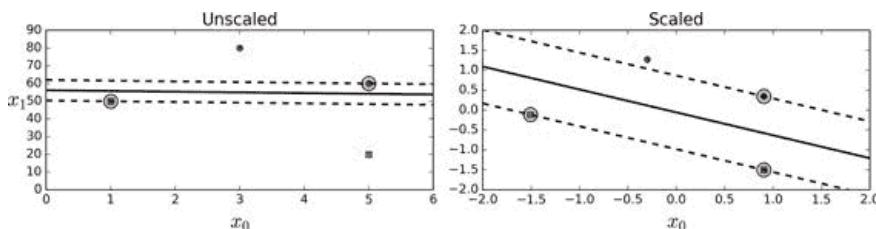
SVM 的基本思想能够用一些图片来解释得很好，图 5-1 展示了我们在第 4 章结尾处介绍的鸢尾花数据集的一部分。这两个种类能够被非常清晰，非常容易的用一条直线分开（即线性可分的）。左边的图显示了三种可能的线性分类器的判定边界。其中用虚线表示的线性模型判定边界很差，甚至不能正确地划分类别。另外两个线性模型在这个数据集表现的很好，但是它们的判定边界很靠近样本点，在新的数据上可能不会表现的很好。相比之下，右边图中 SVM 分类器的判定边界实线，不仅分开了两种类别，而且还尽可能地远离了最靠近的训练数据点。你可以认为 SVM 分类器在两种类别之间保持了一条尽可能宽敞的街道（图中平行的虚线），其被称为最大间隔分类。



我们注意到添加更多的样本点在“街道”外并不会影响到判定边界，因为判定边界是由位于“街道”边缘的样本点确定的，这些样本点被称为“支持向量”（图 5-1 中被圆圈圈起来的点）

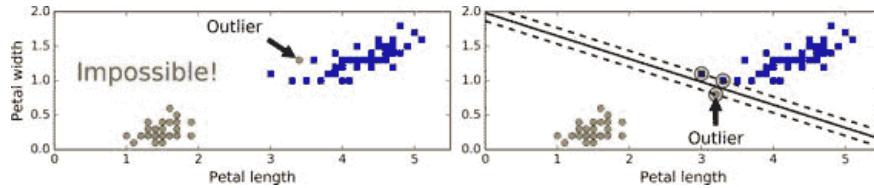
警告

SVM 对特征缩放比较敏感，可以看到图 5-2：左边的图中，垂直的比例要更大于水平的比例，所以最宽的“街道”接近水平。但对特征缩放后（例如使用 Scikit-Learn 的 StandardScaler），判定边界看起来要好得多，如右图。



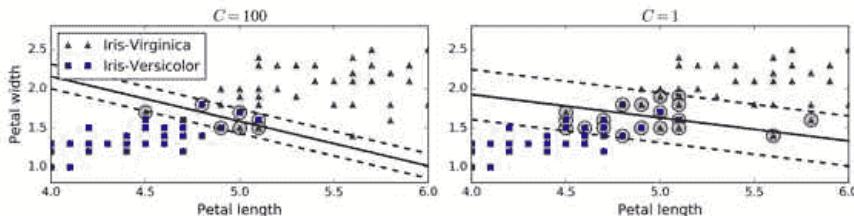
## 软间隔分类

如果我们严格地规定所有的数据都不在“街道”上，都在正确地两边，称为硬间隔分类，硬间隔分类有两个问题，第一，只对线性可分的数据起作用，第二，对异常点敏感。图 5-3 显示了只有一个异常点的鸢尾花数据集：左边的图中很难找到硬间隔，右边的图中判定边界和我们之前在图 5-1 中没有异常点的判定边界非常不一样，它很难一般化。



为了避免上述的问题，我们更倾向于使用更加软性的模型。目的在保持“街道”尽可能大和避免间隔违规（例如：数据点出现在“街道”中央或者甚至在错误的一边）之间找到一个良好的平衡。这就是软间隔分类。

在 Scikit-Learn 库的 SVM 类，你可以用 `c` 超参数（惩罚系数）来控制这种平衡：较小的 `c` 会导致更宽的“街道”，但更多的间隔违规。图 5-4 显示了在非线性可分的数据集上，两个软间隔 SVM 分类器的判定边界。左边图中，使用了较大的 `c` 值，导致更少的间隔违规，但是间隔较小。右边的图，使用了较小的 `c` 值，间隔变大了，但是许多数据点出现在了“街道”上。然而，第二个分类器似乎泛化地更好：事实上，在这个训练数据集上减少了预测错误，因为实际上大部分的间隔违规点出现在了判定边界正确的一侧。



### 提示

如果你的 SVM 模型过拟合，你可以尝试通过减小超参数 `c` 去调整。

以下的 Scikit-Learn 代码加载了内置的鸢尾花 (Iris) 数据集，缩放特征，并训练一个线性 SVM 模型（使用 `LinearSVC` 类，超参数 `C=1`，`hinge` 损失函数）来检测 Virginica 鸢尾花，生成的模型在图 5-4 的右图。

```
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)

Then, as usual, you can use the model to make predictions:
>>> svm_clf.predict([[5.5, 1.7]])
array([ 1.])
```

## 注

不同于 Logistic 回归分类器，SVM 分类器不会输出每个类别的概率。

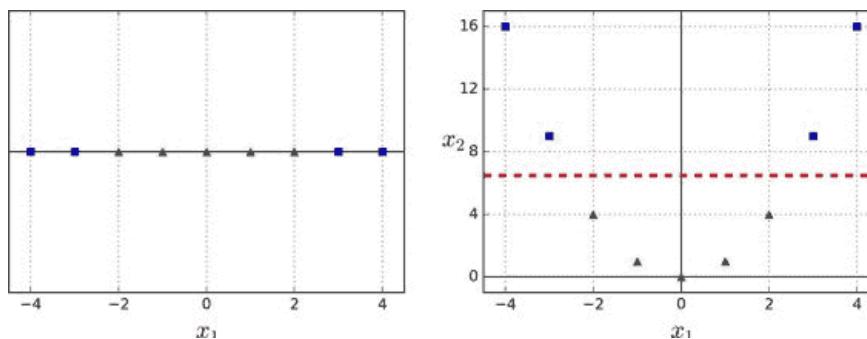
作为一种选择，你可以在 `SVC` 类，使用 `SVC(kernel="linear", C=1)`，但是它比较慢，尤其在较大的训练集上，所以一般不被推荐。另一个选择是使用 `SGDClassifier` 类，即 `SGDClassifier(loss="hinge", alpha=1/(m*C))`。它应用了随机梯度下降（SGD 见第四章）来训练一个线性 SVM 分类器。尽管它不会和 `LinearSVC` 一样快速收敛，但是对于处理那些不适合放在内存的大数据集是非常有用的，或者处理在线分类任务同样有用。

## 提示

`LinearSVC` 要使偏置项规范化，首先你应该集中训练集减去它的平均数。如果你使用了 `StandardScaler`，那么它会自动处理。此外，确保你设置 `loss` 参数为 `hinge`，因为它不是默认值。最后，为了得到更好的效果，你需要将 `dual` 参数设置为 `False`，除非特征数比样本量多（我们将在本章后面讨论二元性）

## 非线性支持向量机分类

尽管线性 SVM 分类器在许多案例上表现得出乎意料的好，但是很多数据集并不是线性可分的。一种处理非线性数据集方法是增加更多的特征，例如多项式特征（正如你在第 4 章所做的那样）；在某些情况下可以变成线性可分的数据。在图 5-5 的左图中，它只有一个特征  $x_1$  的简单数据集，正如你看到的，该数据集不是线性可分的。但是如果你增加了第二个特征  $x_2=(x_1)^2$ ，产生的 2D 数据集就能很好的线性可分。

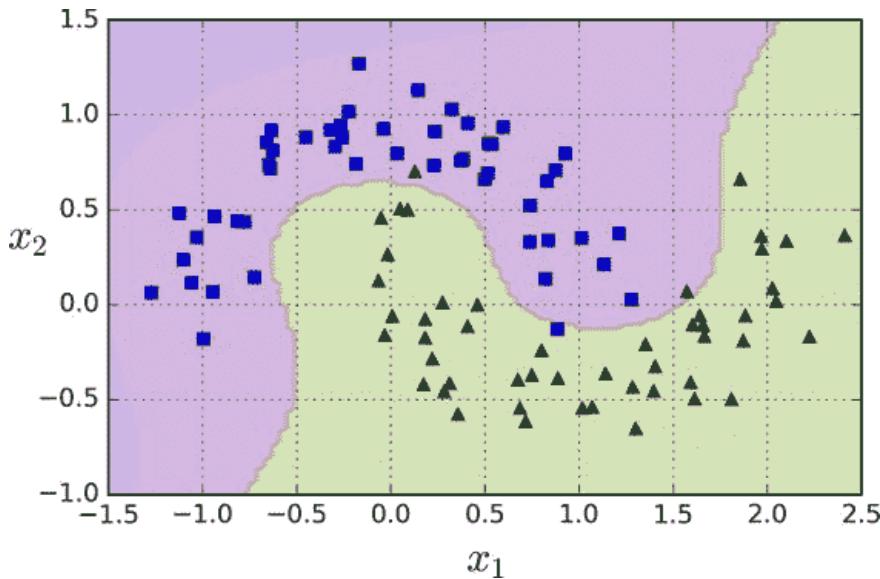


为了实施这个想法，通过 Scikit-Learn，你可以创建一个流水线（Pipeline）去包含多项式特征（`PolynomialFeatures`）变换（在 121 页的“Polynomial Regression”中讨论），然后一个 `StandardScaler` 和 `LinearSVC`。让我们在卫星数据集（moons datasets）测试一下效果。

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
])

polynomial_svm_clf.fit(X, y)
```



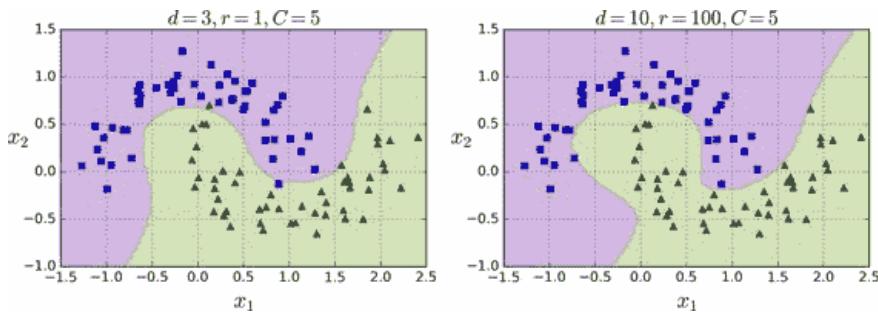
## 多项式核

添加多项式特征很容易实现，不仅仅在 SVM，在各种机器学习算法都有不错的表现，但是低次数的多项式不能处理非常复杂的数据集，而高次数的多项式却产生了大量的特征，会使模型变得慢。

幸运的是，当你使用 SVM 时，你可以运用一个被称为“核技巧”(kernel trick) 的神奇数学技巧。它可以取得就像你添加了许多多项式，甚至有高次数的多项式，一样的好的结果。所以不会大量特征导致的组合爆炸，因为你并没有增加任何特征。这个技巧可以用 SVC 类来实现。让我们在卫星数据集测试一下效果。

```
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)
```

这段代码用 3 阶的多项式核训练了一个 SVM 分类器，即图 5-7 的左图。右图是使用了 10 阶的多项式核 SVM 分类器。很明显，如果你的模型过拟合，你可以减小多项式核的阶数。相反的，如果是欠拟合，你可以尝试增大它。超参数 `coef0` 控制了高阶多项式与低阶多项式对模型的影响。



通用的方法是用网格搜索 (grid search 见第 2 章) 去找到最优超参数。首先进行非常粗略的网格搜索一般会很快，然后在找到的最佳值进行更细的网格搜索。对每个超参数的作用有一个很好的理解可以帮助你在正确的超参数空间找到合适的值。

## 增加相似特征

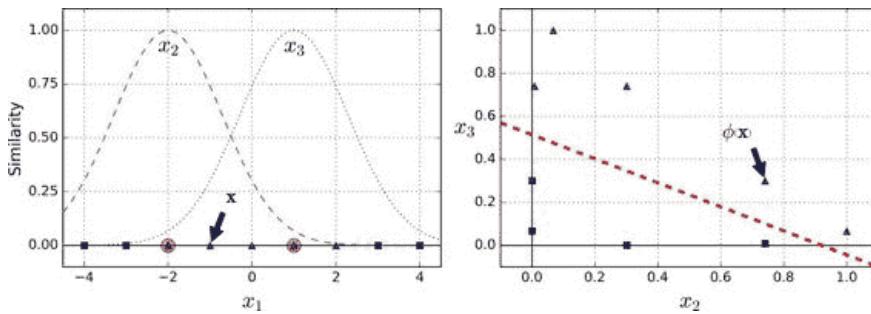
另一种解决非线性问题的方法是使用相似函数 (similarity function) 计算每个样本与特定地标 (landmark) 的相似度。例如，让我们来看看前面讨论过的一维数据集，并在  $x_1=-2$  和  $x_1=1$  之间增加两个地标 (图 5-8 左图)。接下来，我们定义一个相似函数，即高斯径向基函数 (Gaussian Radial Basis Function, RBF)，设置  $\gamma = 0.3$  (见公式 5-1)

公式 5-1 RBF

$$\phi_\gamma(x, \ell) = \exp(-\gamma \|x - \ell\|^2)$$

它是个从 0 到 1 的钟型函数，值为 0 的离地标很远，值为 1 的在地标上。现在我们准备计算新特征。例如，我们看一下样本  $x_1=-1$ ：它距离第一个地标距离是 1，距离第二个地标是 2。因此它的新特征

为  $x_2=\exp(-0.3 \times (1^2)) \approx 0.74$  和  $x_3=\exp(-0.3 \times (2^2)) \approx 0.30$ 。图 5-8 右边的图显示了特征转换后的数据集 (删除了原始特征)，正如你看到的，它现在是线性可分了。



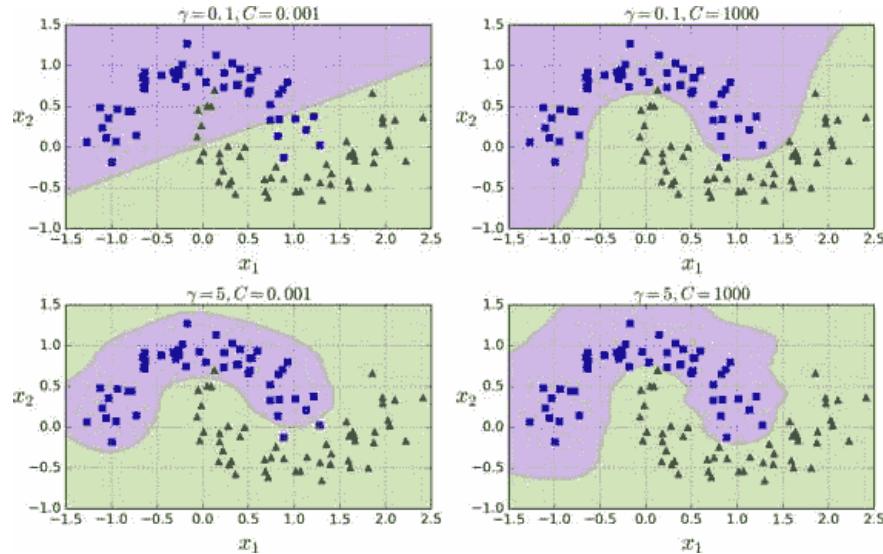
你可能想知道如何选择地标。最简单的方法是在数据集中的每一个样本的位置创建地标。这将产生更多的维度从而增加了转换后数据集是线性可分的可能性。但缺点是， $m$  个样本， $n$  个特征的训练集被转换成了  $m$  个实例， $m$  个特征的训练集 (假设你删除了原始特征)。这样一来，如果你的训练集非常大，你最终会得到同样大的特征。

## 高斯 RBF 核

就像多项式特征法一样，相似特征法对各种机器学习算法同样也有不错的表现。但是在所有额外特征上的计算成本可能很高，特别是在大规模的训练集上。然而，“核”技巧再一次显现了它在 SVM 上的神奇之处：高斯核让你可以获得同样好的结果成为可能，就像你在相似特征法添加了许多相似特征一样，但事实上，你并不需要在 RBF 添加它们。我们使用 SVC 类的高斯 RBF 核来检验一下。

```
rbf_kernel_svm_clf = Pipeline((
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
))
rbf_kernel_svm_clf.fit(X, y)
```

这个模型在图 5-9 的左下角表示。其他的图显示了用不同的超参数  $\gamma$  和  $C$  训练的模型。增大  $\gamma$  使钟型曲线更窄 (图 5-8 左图)，导致每个样本的影响范围变得很小：即判定边界最终变得不规则，在单个样本周围环绕。相反的，较小的  $\gamma$  值使钟型曲线更宽，样本有更大的影响范围，判定边界最终则更加平滑。所以  $\gamma$  是可调整的超参数：如果你的模型过拟合，你应该减小  $\gamma$  值，若欠拟合，则增大  $\gamma$  (与超参数  $C$  相似)。



还有其他的核函数，但很少使用。例如，一些核函数是专门用于特定的数据结构。在对文本文档或者 DNA 序列进行分类时，有时会使用字符串核（String kernels）（例如，使用 SSK 核（string subsequence kernel）或者基于编辑距离（Levenshtein distance）的核函数）。

#### 提示

这么多可供选择的核函数，你如何决定使用哪一个？一般来说，你应该先尝试线性核函数（记住 `LinearSVC` 比 `SVC(kernel='linear')` 要快得多），尤其是当训练集很大或者有大量的特征的情况下。如果训练集不太大，你也可以尝试高斯径向基核（Gaussian RBF Kernel），它在大多数情况下都很有效。如果你有空闲的时间和计算能力，你还可以使用交叉验证和网格搜索来试验其他的核函数，特别是有专门用于你的训练集数据结构的核函数。

## 计算复杂性

`LinearSVC` 类基于 `liblinear` 库，它实现了线性 SVM 的优化算法。它并不支持核技巧，但是它样本和特征的数量几乎是线性的：训练时间复杂度大约为  $O(m \times n)$ 。

如果你要非常高的精度，这个算法需要花费更多时间。这是由容差值超参数  $\epsilon$ （在 Scikit-learn 称为 `tol`）控制的。大多数分类任务中，使用默认容差值的效果是已经可以满足一般要求。

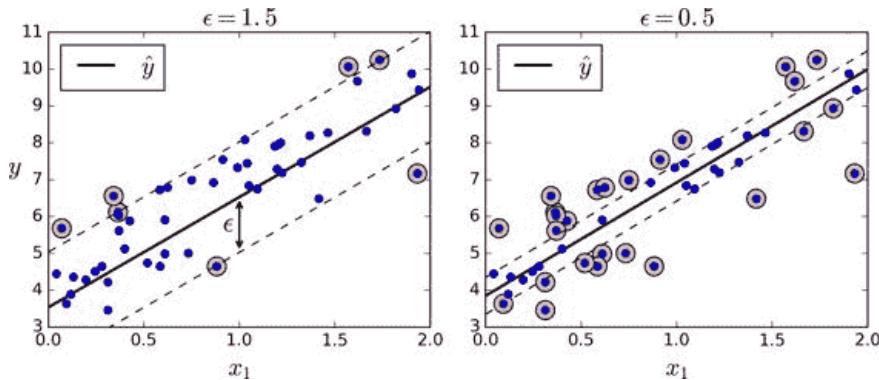
`SVC` 类基于 `libsvm` 库，它实现了支持核技巧的算法。训练时间复杂度通常介于  $O(m^2 \times n)$  和  $O(m^3 \times n)$  之间。不幸的是，这意味着当训练样本变大时，它将变得极其慢（例如，成千上万个样本）。这个算法对于复杂但小型或中等数量的数据集表现是完美的。然而，它能对特征数量很好的缩放，尤其对稀疏特征来说（sparse features）（即每个样本都有一些非零特征）。在这个情况下，算法对每个样本的非零特征的平均数量进行大概的缩放。表 5-1 对 Scikit-learn 的 SVM 分类模型进行比较。

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
LinearSVC	$O(m \times n)$	No	Yes	No
SGDClassifier	$O(m \times n)$	Yes	Yes	No
svc	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

## SVM 回归

正如我们之前提到的，SVM 算法应用广泛：不仅仅支持线性和非线性的分类任务，还支持线性和非线性的回归任务。技巧在于逆转我们的目标：限制间隔违规的情况下，不是试图在两个类别之间找到尽可能大的“街道”（即间隔）。SVM 回归任务是限制间隔违规情况下，尽量放置更多的样本在“街道”上。“街道”的宽度由超参数  $\epsilon$  控制。图 5-10 显示了在一些随机生成的线性数据上，两个线性 SVM 回归模型的训练情况。一个有较大的间隔 ( $\epsilon=1.5$ )，另一个间隔较小 ( $\epsilon=0.5$ )。

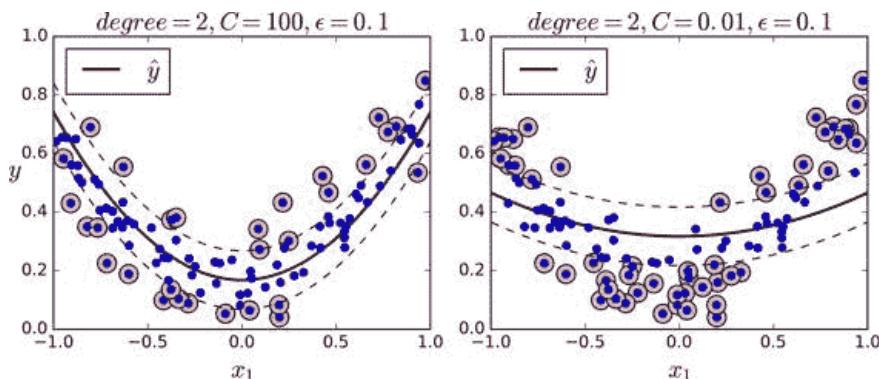


添加更多的数据样本在间隔之内并不会影响模型的预测，因此，这个模型认为是不敏感的 ( $\epsilon$ -insensitive)。

你可以使用 Scikit-Learn 的 LinearSVR 类去实现线性 SVM 回归。下面的代码产生的模型在图 5-10 左图（训练数据需要被中心化和标准化）

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

处理非线性回归任务，你可以使用核化的 SVM 模型。比如，图 5-11 显示了在随机二次方的训练集，使用二次方多项式核函数的 SVM 回归。左图是较小的正则化（即更大的  $c$  值），右图则是更大的正则化（即小的  $c$  值）



下面的代码的模型在图 5-11，其使用了 Scikit-Learn 的 SVR 类（支持核技巧）。在回归任务上，SVR 类和 SVC 类是一样的，并且 LinearSVR 是和 LinearSVC 等价。LinearSVR 类和训练集的大小成线性（就像 LinearSVC 类），当训练集变大，SVR 会变的很慢（就像 SVC 类）

```
from sklearn.svm import SVR
svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

### 注

SVM 也可以用来做异常值检测，详情见 Scikit-Learn 文档

## 背后机制

这个章节从线性 SVM 分类器开始，将解释 SVM 是如何做预测的并且算法是如何工作的。如果你是刚接触机器学习，你可以跳过这个章节，直接进入本章末尾的练习。等到你想深入了解 SVM，再回头研究这部分内容。

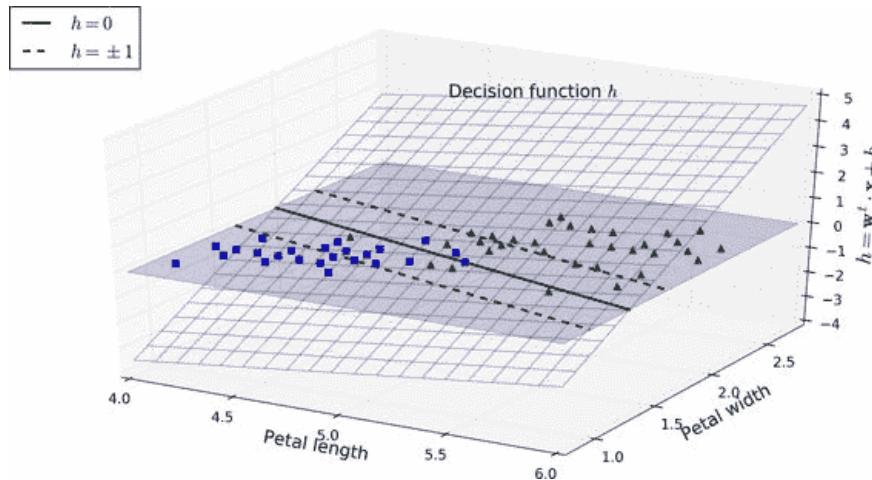
首先，关于符号的约定：在第 4 章，我们将所有模型参数放在一个向量  $\theta$  里，包括偏置项  $\theta_0$ ， $\theta_1$  到  $\theta_n$  的输入特征权重，和增加一个偏差输入  $x_0 = 1$  到所有样本。在本章中，我们将使用一个不同的符号约定，在处理 SVM 上，这更方便，也更常见：偏置项被命名为  $b$ ，特征权重向量被称为  $w$ ，在输入特征向量中不再添加偏置特征。

## 决策函数和预测

线性 SVM 分类器通过简单地计算决策函数  $w \cdot x + b = w[1]x[1] + \dots + w[n]x[n] + b$  来预测新样本的类别：如果结果是正的，预测类别  $\hat{y}$  是正类，为 1，否则他就是负类，为 0。见公式 5-2

$$\hat{y} = \begin{cases} 0 & \text{if } w^T \cdot x + b < 0, \\ 1 & \text{if } w^T \cdot x + b \geq 0 \end{cases}$$

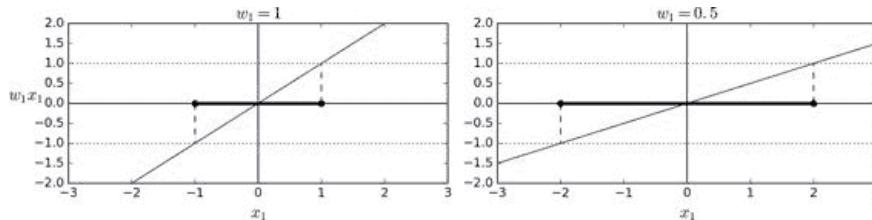
图 5-12 显示了和图 5-4 右边图模型相对应的决策函数：因为这个数据集有两个特征（花瓣的宽度和花瓣的长度），所以是个二维的平面。决策边界是决策函数等于 0 的点的集合，图中两个平面的交叉处，即一条直线（图中的实线）



虚线表示的是那些决策函数等于 1 或 -1 的点：它们平行，且到决策边界的距离相等，形成一个间隔。训练线性 SVM 分类器意味着找到  $w$  值和  $b$  值使得这一个间隔尽可能大，同时避免间隔违规（硬间隔）或限制它们（软间隔）

## 训练目标

看下决策函数的斜率：它等于权重向量的范数  $\|w\|$ 。如果我们把这个斜率除于 2，决策函数等于  $\pm 1$  的点将会离决策边界原来的两倍大。换句话，即斜率除于 2，那么间隔将增加两倍。在图 5-13 中，2D 形式比较容易可视化。权重向量  $w$  越小，间隔越大。



所以我们的目标是最小化  $\|w\|$ ，从而获得大的间隔。然而，如果我们想要避免间隔违规（硬间隔），对于正的训练样本，我们需要决策函数大于 1，对于负训练样本，小于 -1。若我们对负样本（即  $y^{(i)} = 0$ ）定义  $t^{(i)} = -1$ ，对正样本（即  $y^{(i)} = 1$ ）定义  $t^{(i)} = 1$ ，那么我们可以对所有的样本表示为  $t^{(i)}(w^T x^{(i)} + b) > 1$ 。

因此，我们可以将硬间隔线性 SVM 分类器表示为公式 5-3 中的约束优化问题

$$\begin{aligned} \underset{w, b}{\text{minimize}} \quad & \frac{1}{2} w^T \cdot w \\ \text{subject to} \quad & t^{(i)}(w^T \cdot x^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

注

$1/2 w^T w$  等于  $1/2 \|w\|^2$ ，我们最小化  $1/2 w^T w$ ，而不是最小化  $|w|$ 。这会给我们相同的结果（因为最小化  $w$  值和  $b$  值，也是最小化该值一半的平方），但是  $1/2 \|w\|^2$  有很好又简单的导数（只有  $w$ ）， $|w|$  在  $w=0$  处是不可微的。优化算法在可微函数表现得更好。

为了获得软间隔的目标，我们需要对每个样本应用一个松弛变量（slack variable） $\zeta^{(i)} > 0$ 。 $\zeta^{(i)}$  表示了第  $i$  个样本允许违规间隔的程度。我们现在有两个不一致的目标：一个是使松弛变量尽可能的小，从而减小间隔违规，另一个是使  $1/2 \mathbf{w} \cdot \mathbf{w}$  尽量小，从而增大间隔。这时  $C$  超参数发挥作用：它允许我们在两个目标之间权衡。我们得到了公式 5-4 的约束优化问题。

$$\begin{aligned} \underset{\mathbf{w}, b, \zeta}{\text{minimize}} \quad & \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)} \\ \text{subject to} \quad & t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

## 二次规划

硬间隔和软间隔都是线性约束的凸二次规划优化问题。这些问题被称之为二次规划 (QP) 问题。现在有许多解决方案可以使用各种技术来处理 QP 问题，但这超出了本书的范围。一般问题的公式在公式 5-5 给出。

$$\begin{aligned} \text{Minimize} \quad & \frac{1}{2} \mathbf{p}^T \cdot \mathbf{H} \mathbf{p} + \mathbf{f}^T \cdot \mathbf{p} \\ \text{subject to} \quad & \mathbf{A} \cdot \mathbf{p} \leq \mathbf{b} \\ \text{where} \quad & \begin{cases} \mathbf{p} \text{ is an } n_p \text{-dimensional vector } (n_p = \text{number of parameters}), \\ \mathbf{H} \text{ is an } n_p \times n_p \text{ matrix}, \\ \mathbf{f} \text{ is an } n_p \text{-dimensional vector}, \\ \mathbf{A} \text{ is an } n_c \times n_p \text{ matrix } (n_c = \text{number of constraints}), \\ \mathbf{b} \text{ is an } n_c \text{-dimensional vector.} \end{cases} \end{aligned}$$

注意到表达式  $\mathbf{A}\mathbf{p} \leq \mathbf{b}$  实际上定义了  $n[c]$  约束：

$$\mathbf{p}^T \mathbf{a}^{(i)} \leq b^{(i)}, \text{ for } i = 1, 2, \dots, n$$

$\mathbf{a}^{(i)}$  是个包含了  $\mathbf{A}$  的第  $i$  行元素的向量， $b^{(i)}$  是  $\mathbf{b}$  的第  $i$  个元素。

可以很容易地看到，如果你用以下的方式设置 QP 的参数，你将获得硬间隔线性 SVM 分类器的目标：

- $n[p] = n + 1$ ， $n$  表示特征的数量 (+1 是偏置项)
- $n[c] = m$ ， $m$  表示训练样本数量
- $H$  是  $n[p] \times n[p]$  单位矩阵，除了左上角为 0 (忽略偏置项)
- $f = \mathbf{0}$ ，一个全为 0 的  $n[p]$  维向量
- $b = \mathbf{1}$ ，一个全为 1 的  $n[c]$  维向量
- $a^{(i)} = -t^{(i)} \mathbf{x}_{dot}^{(i)}$ ， $\mathbf{x}_{dot}^{(i)}$  等于  $\mathbf{x}^{(i)}$  带一个额外的偏置特征  $x_{dot}[0] = 1$

所以训练硬间隔线性 SVM 分类器的一种方式是使用现有的 QP 解决方案，即上述的参数。由此产生的向量  $p$  将包含偏置项  $b = p[0]$  和特征权重  $w[i] = p[i]$  ( $i=1, 2, \dots, m$ )。同样的，你可以使用 QP 解决方案来解决软间隔问题（见本章最后的练习）

然而，使用核技巧我们将会看到一个不同的约束优化问题。

## 对偶问题

给出一个约束优化问题，即原始问题（primal problem），它可能表示不同但是和另一个问题紧密相连，称为对偶问题（Dual Problem）。对偶问题的解通常是对原始问题的解给出一个下界约束，但在某些条件下，它们可以获得相同解。幸运的是，SVM 问题恰好满足这些条件，所以你可以选择解决原始问题或者对偶问题，两者将会有相同解。公式 5-6 表示了线性 SVM 的对偶形式（如果你对怎么从原始问题获得对偶问题感兴趣，可以看下附录 C）

$$\underset{\alpha}{\text{minimize}} \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)}$$

subject to  $\alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m$

一旦你找到最小化公式的向量  $\alpha$ （使用 QP 解决方案），你可以通过使用公式 5-7 的方法计算  $w$  和  $b$ ，从而使原始问题最小化。

$$\mathbf{w} = \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m (1 - t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)}))$$

当训练样本的数量比特征数量小的时候，对偶问题比原始问题要快得多。更重要的是，它让核技巧成为可能，而原始问题则不然。那么这个核技巧是怎么样的呢？

## 核化支持向量机

假设你想把一个 2 次多项式变换应用到二维空间的训练集（例如卫星数据集），然后在变换后的训练集上训练一个线性 SVM 分类器。公式 5-8 显示了你想应用的 2 次多项式映射函数  $\phi$ 。

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2} x_1 x_2 \\ x_2^2 \end{pmatrix}$$

注意到转换后的向量是 3 维的而不是 2 维。如果我们应用这个 2 次多项式映射，然后计算转换后向量的点积（见公式 5-9），让我们看下两个 2 维向量  $\mathbf{a}$  和  $\mathbf{b}$  会发生什么。

$$\begin{aligned}\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) &= \begin{pmatrix} a_1^2 \\ \sqrt{2} a_1 a_2 \\ a_2^2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1^2 \\ \sqrt{2} b_1 b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2 \\ &= (a_1 b_1 + a_2 b_2)^2 = \left( \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^T \cdot \mathbf{b})^2\end{aligned}$$

转换后向量的点积等于原始向量点积的平方： $\phi(\mathbf{a})^T \phi(\mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$ 。

关键点是：如果你应用转换  $\phi$  到所有训练样本，那么对偶问题（见公式 5-6）将会包含点积  $\phi(x^{(i)})^T \phi(x^{(j)})$ 。但如果  $\phi$  像在公式 5-8 定义的 2 次多项式转换，那么你可以将这个转换后的向量点积替换成  $(x^{(i)} \cdot x^{(j)})^2$ 。所以实际上你根本不需要对训练样本进行转换：仅仅需要在公式 5-6 中，将点积替换成它点积的平方。结果将会和你经过麻烦的训练集转换并拟合出线性 SVM 算法得出的结果一样，但是这个技巧使得整个过程在计算上面更有效率。这就是核技巧的精髓。

函数  $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$  被称为二次多项式核（polynomial kernel）。在机器学习，核函数是一个能计算点积的函数，并只基于原始向量  $\mathbf{a}$  和  $\mathbf{b}$ ，不需要计算（甚至知道）转换  $\phi$ 。公式 5-10 列举了一些最常用的核函数。

Linear:	$K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b}$
Polynomial:	$K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \cdot \mathbf{b} + r)^d$
Gaussian RBF:	$K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \ \mathbf{a} - \mathbf{b}\ ^2)$
Sigmoid:	$K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^T \cdot \mathbf{b} + r)$

#### Mercer 定理

根据 Mercer 定理，如果函数  $K(\mathbf{a}, \mathbf{b})$  满足一些 Mercer 条件的数学条件（ $K$  函数在参数内必须是连续，对称，即  $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$ ，等），那么存在函数  $\phi$ ，将  $\mathbf{a}$  和  $\mathbf{b}$  映射到另一个空间（可能有更高的维度），有  $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \phi(\mathbf{b})$ 。所以你可以用  $K$  作为核函数，即使你不知道  $\phi$  是什么。使用高斯核（Gaussian RBF kernel）情况下，它实际是将每个训练样本映射到无限维空间，所以你不需要知道是怎么执行映射的也是一件好事。

注意一些常用核函数（例如 Sigmoid 核函数）并不满足所有的 Mercer 条件，然而在实践中通常表现得很好。

我们还有一个问题要解决。公式 5-7 展示了线性 SVM 分类器如何从对偶解到原始解，如果你应用了核技巧那么得到的公式会包含  $\phi(x^{(i)})$ 。事实上， $w$  必须和  $\phi(x^{(i)})$  有同样的维度，可能是巨大的维度或者无限的维度，所以你很难计算它。但怎么在不知道  $w$  的情况下做出预测？好消息是你可以将公式 5-7 的  $w$  代入到新的样本  $x^{(n)}$  的决策函数中，你会得到一个在输入向量之间只有点积的方程式。这时，核技巧将派上用场，见公式 5-11

$$\begin{aligned}
h_{\hat{\mathbf{w}}, \hat{b}}(\phi(\mathbf{x}^{(n)})) &= \mathbf{w}^T \cdot \phi(\mathbf{x}^{(n)}) + \hat{b} = \left( \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \phi(\mathbf{x}^{(i)}) \right)^T \cdot \phi(\mathbf{x}^{(n)}) + \hat{b} \\
&= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} (\phi(\mathbf{x}^{(i)})^T \cdot \phi(\mathbf{x}^{(n)})) + \hat{b} \\
&= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \hat{\alpha}^{(i)} t^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}) + \hat{b}
\end{aligned}$$

注意到支持向量才满足  $\alpha^{(i)} \neq 0$ ，做出预测只涉及计算为支持向量部分的输入样本  $\mathbf{x}^{(n)}$  的点积，而不是全部的训练样本。当然，你同样也需要使用同样的技巧来计算偏置项  $b$ ，见公式 5-12

$$\begin{aligned}
\hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m (1 - t^{(i)} \mathbf{w}^T \cdot \phi(\mathbf{x}^{(i)})) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left( 1 - t^{(i)} \left( \sum_{j=1}^m \hat{\alpha}^{(j)} t^{(j)} \phi(\mathbf{x}^{(j)}) \right)^T \cdot \phi(\mathbf{x}^{(i)}) \right) \\
&= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left( 1 - t^{(i)} \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)} > 0}}^m \hat{\alpha}^{(j)} t^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \right)
\end{aligned}$$

如果你开始感到头痛，这很正常：因为这是核技巧一个不幸的副作用

## 在线支持向量机

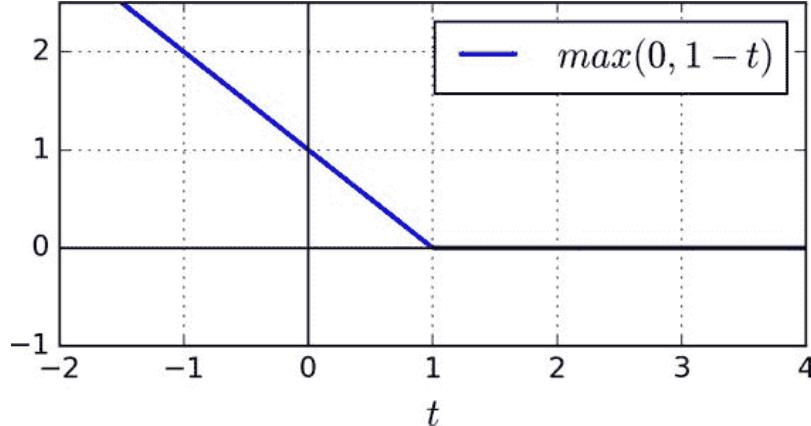
在结束这一章之前，我们快速地了解一下在线 SVM 分类器（回想一下，在线学习意味着增量地学习，不断有新实例）。对于线性 SVM 分类器，一种方式是使用梯度下降（例如使用 `SGDClassifire`）最小化代价函数，如从原始问题推导出的公式 5-13。不幸的是，它比基于 QP 方式收敛慢得多。

$$J(\mathbf{w}, b) = \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^m \max(0, 1 - t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b))$$

代价函数第一个和会使模型有一个小的权重向量  $\mathbf{w}$ ，从而获得一个更大的间隔。第二个和计算所有间隔违规的总数。如果样本位于“街道”上和正确的一边，或它与“街道”正确一边的距离成比例，则间隔违规等于 0。最小化保证了模型的间隔违规尽可能小并且少。

## Hinge 损失

函数  $\max(0, 1-t)$  被称为 Hinge 损失函数（如下）。当  $t \geq 1$  时，Hinge 值为 0。如果  $t < 1$ ，它的导数（斜率）为 -1，若  $t > 1$ ，则等于 0。在  $t=1$  处，它是不可微的，但就像套索回归（Lasso Regression）（参见 130 页套索回归）一样，你仍然可以在  $t=0$  时使用梯度下降法（即 -1 到 0 之间任何值）



我们也可以实现在线核化的 SVM。例如使用“增量和递减 SVM 学习”或者“在线和主动的快速核分类器”。但是，这些都是用 Matlab 和 C++ 实现的。对于大规模的非线性问题，你可能需要考虑使用神经网络（见第二部分）

## 练习

1. 支持向量机背后的基本思想是什么
2. 什么是支持向量
3. 当使用 SVM 时，为什么标准化输入很重要？
4. 分类一个样本时，SVM 分类器能够输出一个置信值吗？概率呢？
5. 在一个有数百万训练样本和数百特征的训练集上，你是否应该使用 SVM 原始形式或对偶形式来训练一个模型？
6. 假设你用 RBF 核来训练一个 SVM 分类器，如果对训练集欠拟合：你应该增大或者减小  $\gamma$  吗？调整参数  $C$  呢？
7. 使用现有的 QP 解决方案，你应该怎么样设置 QP 参数 ( $H$ ,  $f$ ,  $A$ , 和  $b$ ) 去解决一个软间隔线性 SVM 分类器问题？
8. 在一个线性可分的数据集训练一个 `LinearSVC`，并在同一个数据集上训练一个 `SVC` 和 `SGDCClassifier`，看它们是否产生了大致相同效果的模型。
9. 在 MNIST 数据集上训练一个 SVM 分类器。因为 SVM 分类器是二元的分类，你需要使用一对多 (one-versus-all) 来对 10 个数字进行分类。你可能需要使用小的验证集来调整超参数，以加快进程。最后你能达到多少准确度？
10. 在加利福尼亚住宅 (California housing) 数据集上训练一个 SVM 回归模型

这些练习的答案在附录 A。

## 六、决策树

译者：[@Lisanaaa](#)、[@y3534365](#)

校对者：[@飞龙](#)、[@YuWang](#)

和支持向量机一样，决策树是一种多功能机器学习算法，即可以执行分类任务也可以执行回归任务，甚至包括多输出（multioutput）任务。

它是一种功能很强大的算法，可以对很复杂的数据集进行拟合。例如，在第二章中我们对加利福尼亚住房数据集使用决策树回归模型进行训练，就很好的拟合了数据集（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），而随机森林是当今最强大的机器学习算法之一。

在本章中，我们将首先讨论如何使用决策树进行训练，可视化和预测。

然后我们会学习在 Scikit-learn 上面使用 CART 算法，并且探讨如何调整决策树让它可用于执行回归任务。

最后，我们当然也需要讨论一下决策树目前存在的一些局限性。

### 决策树的训练和可视化

为了理解决策树，我们需要先构建一个决策树并亲身体验它到底如何进行预测。

接下来的代码就是在我们熟知的鸢尾花数据集上进行一个决策树分类器的训练。

```
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)
```

你可以通过使用 `export_graphviz()` 方法，通过生成一个叫做 `iris_tree.dot` 的图形定义文件将一个训练好的决策树模型可视化。

```
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
)
```

译者注：原文中的 `image_path` 用于获得示例程序的相对路径。这里直接去掉改成 `out_file="iris_tree.dot"` 即可参见  
[https://github.com/ageron/handson-ml/blob/master/06\\_decision\\_trees.ipynb](https://github.com/ageron/handson-ml/blob/master/06_decision_trees.ipynb)

然后，我们可以利用 graphviz package [1] 中的 `dot` 命令行，将 `.dot` 文件转换成 PDF 或 PNG 等多种数据格式。例如，使用命令行将 `.dot` 文件转换成 `.png` 文件的命令如下：

[1] Graphviz 是一款开源图形可视化软件包，<http://www.graphviz.org/>。

```
$ dot -Tpng iris_tree.dot -o iris_tree.png
```

我们的第一个决策树如图 6-1。

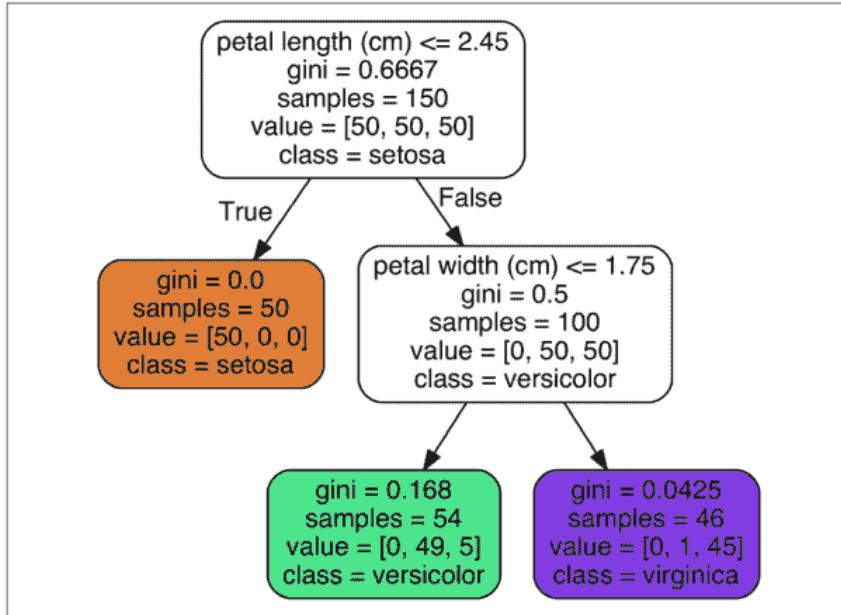


Figure 6-1. Iris Decision Tree

## 开始预测

现在让我们来看看在图 6-1 中的树是如何进行预测的。假设你找到了一朵鸢尾花并且想对它进行分类，你从根节点开始（深度为 0，顶部）：该节点询问花朵的花瓣长度是否小于 2.45 厘米。如果是，您将向下移动到根的左侧子节点（深度为 1，左侧）。在这种情况下，它是一片叶子节点（即它没有任何子节点），所以它不会问任何问题：你可以方便地查看该节点的预测类别，决策树预测你的花是 Iris-Setosa ( class = setosa )。

现在假设你找到了另一朵花，但这次的花瓣长度是大于 2.45 厘米的。你必须向下移动到根的右侧子节点（深度为 1，右侧），而这个节点不是叶节点，所以它会问另一个问题：花瓣宽度是否小于 1.75 厘米？如果是，那么你的花很可能是一个 Iris-Versicolor（深度为 2，左）。如果不是，那很可能一个 Iris-Virginica（深度为 2，右），真的是太简单了，对吧！

决策树的众多特性之一就是，它不需要太多的数据预处理，尤其是不需要进行特征的缩放或者归一化。

节点的 `samples` 属性统计出它应用于多少个训练样本实例。

例如，我们有一百个训练实例是花瓣长度大于 2.45 里面的（深度为 1，右侧），在这 100 个样例中又有 54 个花瓣宽度小于 1.75cm（深度为 2，左侧）。

节点的 `value` 属性告诉你这个节点对于每一个类别的样例有多少个。

例如：右下角的节点中包含 0 个 Iris-Setosa, 1 个 Iris-Versicolor 和 45 个 Iris-Virginica。

最后，节点的 Gini 属性用于测量它的纯度：如果一个节点包含的所有训练样例全都是同一类别的，我们就说这个节点是纯的（ $Gini=0$ ）。

例如，深度为 1 的左侧节点只包含 Iris-Setosa 训练实例，它就是一个纯节点，Gini 指数为 0。

公式 6-1 显示了训练算法如何计算第  $i$  个节点的 gini 分数  $G[i]$ 。例如，深度为 2 的左侧节点基尼指数为： $1 - (0 / 54)^2 - (49 / 54)^2 = 0.68$ 。另外一个纯度指数也将在后文很快提到。

### Equation 6-1. Gini impurity

$$G_i = 1 - \sum_{k=1}^n P_{i,k}^2$$

- $p[i,k]$  是第  $i$  个节点中训练实例为的  $k$  类实例的比例

Scikit-Learn 用的是 CART 算法，CART 算法仅产生二叉树：每一个非叶节点总是只有两个子节点（只有是或否两个结果）。然而，像 ID3 这样的算法可以产生超过两个子节点的决策树模型。

图 6-2 显示了决策树的决策边界。粗的垂直线代表根节点（深度为 0）的决定边界：花瓣长度为 2.45 厘米。由于左侧区域是纯的（只有 Iris-Setosa），所以不能再进一步分裂。然而，右边的区域是不纯的，所以深度为 1 的右边节点在花瓣宽度为 1.75 厘米处分裂（用虚线表示）。又由于 `max_depth` 设置为 2，决策树在那里停了下来。但是，如果将 `max_depth` 设置为 3，两个深度为 2 的节点，每个都将会添加另一个决策边界（用虚线表示）。

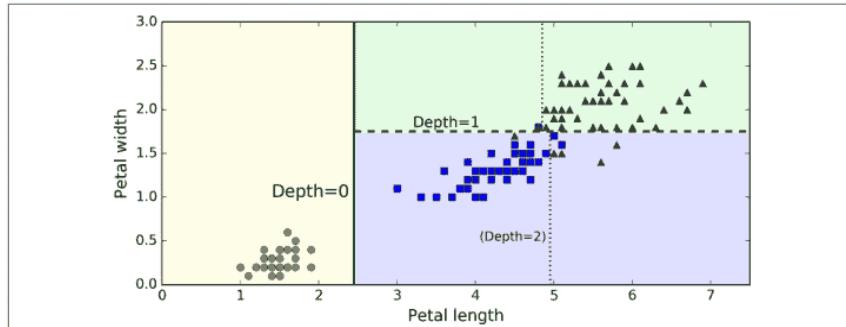


Figure 6-2. Decision Tree decision boundaries

#### 模型小知识：白盒与黑盒

正如我们看到的一样，决策树非常直观，他们的决定很容易被解释。这种模型通常被称为白盒模型。相反，随机森林或神经网络通常被认为是黑盒模型。他们能做出很好的预测，并且您可以轻松检查它们做出这些预测过程中计算的执行过程。然而，人们通常很难用简单的术语来解释为什么模型会做出这样的预测。例如，如果一个神经网络说一个特定的人出现在图片上，我们很难知道究竟是什么导致了这一个预测的出现：

模型是否认出了那个人的眼睛？她的嘴？她的鼻子？她的鞋？或者是否坐在沙发上？相反，决策树提供良好的、简单的分类规则，甚至可以根据需要手动操作（例如鸢尾花分类）。

## 估计分类概率

决策树还可以估计某个实例属于特定类  $k$  的概率：首先遍历树来查找此实例的叶节点，然后它返回此节点中类  $k$  的训练实例的比例。

例如，假设你发现了一个花瓣长 5 厘米，宽 1.5 厘米的花朵。相应的叶节点是深度为 2 的左节点，因此决策树应该输出以下概率：*Iris-Setosa* 为 0%（0/54），*Iris-Versicolor* 为 90.7%（49/54），*Iris-Virginica* 为 9.3%（5/54）。当然，如果你要求它预测具体的类，它应该输出 *Iris-Versicolor*（类别 1），因为它具有最高的概率。我们测试一下：

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[0., 0.90740741, 0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

完美！请注意，估计概率在任何地方都是相同的，除了图 6-2 中右下角的矩形部分，例如花瓣长 6 厘米和宽 1.5 厘米（尽管在这种情况下它看起来很可能是 *Iris-Virginica*）。

## CART 训练算法

*Scikit-Learn* 用分裂回归树（Classification And Regression Tree，简称 CART）算法训练决策树（也叫“增长树”）。这种算法思想真的非常简单：

首先使用单个特征  $k$  和阈值  $t[k]$ （例如，“花瓣长度  $\leq 2.45\text{cm}$ ”）将训练集分成两个子集。它如何选择  $k$  和  $t[k]$  呢？它寻找到能够产生最纯粹的子集一对  $(k, t[k])$ ，然后通过子集大小加权计算。

算法会尝试最小化成本函数。方法如公式 6-2

*Equation 6-2. CART cost function for classification*

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

where  $\begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset,} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset.} \end{cases}$

当它成功的将训练集分成两部分之后，它将会继续使用相同的递归式逻辑继续的分割子集，然后是子集的子集。当达到预定的最大深度之后将会停止分裂（由 `max_depth` 超参数决定），或者是它找不到可以继续降低不纯度的分裂方法的时候。几个其他超参数（之后介绍）控制了其他的停止生长条件（`min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf`, `max_leaf_`）。

正如您所看到的，CART 算法是一种贪婪算法：它贪婪地搜索最高级别的最佳分割方式，然后在每个深度重复该过程。它不检查分割是否能够在几个级别的全部分割可能中找到最佳方法。贪婪算法通常会产生一个相当好的解决方法，但它不保证这是全局中的最佳解决方案。

不幸的是，找到最优树是一个 NP 完全问题（自行百度）：它需要  $O(\exp^m)$  时间，即使对于相当小的训练集也会使问题变得棘手。这就是为什么我们必须设置一个“合理的”（而不是最佳的）解决方案。

## 计算复杂度

在建立好决策树模型后，做出预测需要遍历决策树，从根节点一直到叶节点。决策树通常近似左右平衡，因此遍历决策树需要经历大致  $O(\log_2(m))$  [2] 个节点。由于每个节点只需要检查一个特征的值，因此总体预测复杂度仅为  $O(\log_2(m))$ ，与特征的数量无关。所以即使在处理大型训练集时，预测速度也非常快。

[2]  $\log_2$  是二进制对数，它等于  $\log_2(m) = \log(m) / \log(2)$ 。

然而，训练算法的时候（训练和预测不同）需要比较所有特征（如果设置了 `max_features` 会更少一些）

在每个节点的所有样本上。就有了  $O(n \times m \log(m))$  的训练复杂度。对于小型训练集（少于几千例），Scikit-Learn 可以通过预先设置数据（`presort = True`）来加速训练，但是这对于较大训练集来说会显着减慢训练速度。

## 基尼不纯度或是信息熵

通常，算法使用 Gini 不纯度来进行检测，但是你也可以通过将标准超参数设置为 "entropy" 来使用熵不纯度进行检测。这里熵的概念是源于热力学中分子混乱程度的概念，当分子井然有序的时候，熵值接近于 0。

熵这个概念后来逐渐被扩展到了各个领域，其中包括香农的信息理论，这个理论被用于测算一段信息中的平均信息密度 [3]。当所有信息相同的时候熵被定义为零。

在机器学习中，熵经常被用作不纯度的衡量方式，当一个集合内只包含一类实例时，我们称为数据集的熵为 0。

[3] 熵的减少通常称为信息增益。

公式 6-3 显示了第  $i$  个节点的熵的定义，例如，在图 6-1 中，深度为 2 左节点的熵为  $-49/54 \log(49/54) - 5/54 \log(5/54) = 0.31$ 。

$$\text{Equation 6 - 3. Entropy}$$

$$H_i = - \sum_{k=1}^n P_{i,k} \log(p_{i,k})$$

那么我们到底应该使用 Gini 指数还是熵呢？事实上大部分情况都没多大的差别：他们会生成类似的决策树。

基尼指数计算稍微快一点，所以这是一个很好的默认值。但是，也有的时候它们会产生不同的树，基尼指数会趋于在树的分支中将最多的类隔离出来，而熵指数趋向于产生略微平衡一些的决策树模型。

## 正则化超参数

决策树几乎不对训练数据做任何假设（于此相反的是线性回归等模型，这类模型通常会假设数据是符合线性关系的）。

如果不添加约束，树结构模型通常将根据训练数据调整自己，使自身能够很好的拟合数据，而这种情况下大多数会导致模型过拟合。

这一类的模型通常会被称为非参数模型，这不是因为它没有任何参数（通常也有很多），而是在训练之前没有确定参数的具体数量，所以模型结构可以根据数据的特性自由生长。

于此相反的是，像线性回归这样的参数模型有事先设定好的参数数量，所以自由度是受限的，这就减少了过拟合的风险（但是增加了欠拟合的风险）。

`DecisionTreeClassifier` 类还有一些其他的参数用于限制树模型的形状：

```
min_samples_split (节点在被分裂之前必须具有的最小样本数) , min_samples_leaf (叶节点必须具有的最小样本数) , min_weight_fraction_leaf (和 min_samples_leaf 相同，但表示为加权总数的一小部分实例) , max_leaf_nodes (叶节点的最大数量) 和 max_features (在每个节点被评估是否分裂的时候，具有的最大特征数量)。增加 min_* hyperparameters 或者减少 max_* hyperparameters 会使模型正则化。
```

一些其他算法的工作原理是在没有任何约束条件下训练决策树模型，让模型自由生长，然后再对不需要的节点进行剪枝。

当一个节点的全部子节点都是叶节点时，如果它对纯度的提升不具有统计学意义，我们就认为这个分支是不必要的。

标准的假设检验，例如卡方检测，通常会被用于评估一个概率值 -- 即改进是否纯粹是偶然性的结果（也叫原假设）

如果 p 值比给定的阈值更高（通常设定为 5%，也就是 95% 置信度，通过超参数设置），那么节点就被认为是非必要的，它的子节点会被删除。

这种剪枝方式将会一直进行，直到所有的非必要节点都被删光。

图 6-3 显示了对 `moons` 数据集（在第 5 章介绍过）进行训练生成的两个决策树模型，左侧的图形对应的决策树使用默认超参数生成（没有限制生长条件），右边的决策树模型设置为 `min_samples_leaf=4`。很明显，左边的模型过拟合了，而右边的模型泛用性更好。

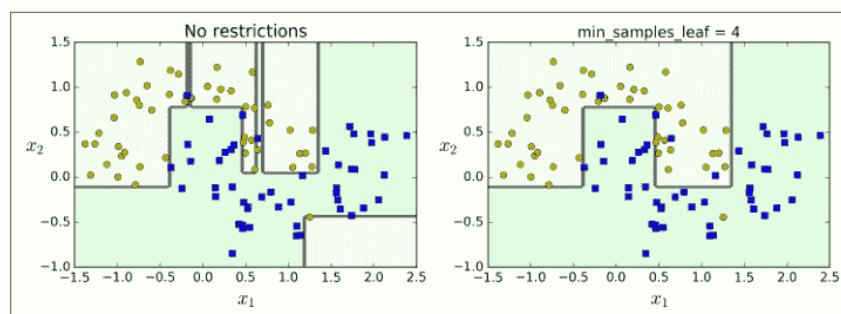


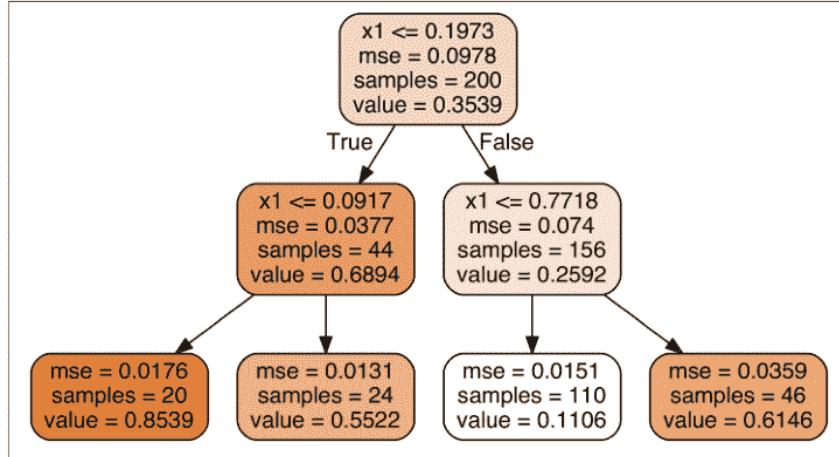
Figure 6-3. Regularization using `min_samples_leaf`

## 回归

决策树也能够执行回归任务，让我们使用 Scikit-Learn 的 `DecisionTreeRegressor` 类构建一个回归树，让我们用 `max_depth = 2` 在具有噪声的二次项数据集上进行训练。

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

结果如图 6-4 所示



这棵树看起来非常类似于你之前建立的分类树，它的主要区别在于，它不是预测每个节点中的样本所属的分类，而是预测一个具体的数值。例如，假设您想对  $x[1] = 0.6$  的新实例进行预测。从根开始遍历树，最终到达预测值等于 0.1106 的叶节点。该预测仅仅是与该叶节点相关的 110 个训练实例的平均目标值。而这个预测结果在对应的 110 个实例上的均方误差（MSE）等于 0.0151。

在图 6-5 的左侧显示的是模型的预测结果，如果你将 `max_depth=3` 设置为 3，模型就会如 6-5 图右侧显示的那样。注意每个区域的预测值总是该区域中实例的平均目标值。算法以一种使大多数训练实例尽可能接近该预测值的方式分割每个区域。

译者注：图里面的红线就是训练实例的平均目标值，对应上图中的 `value`

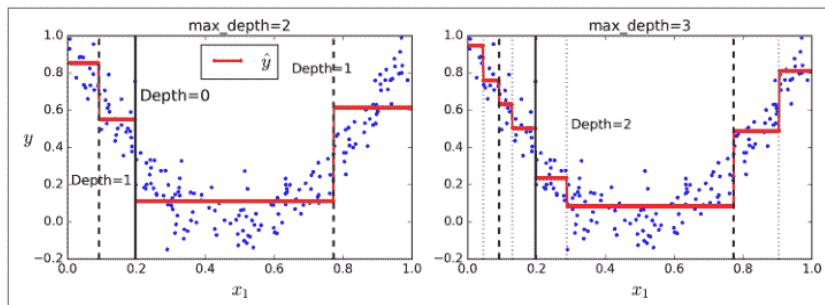


Figure 6-5. Predictions of two Decision Tree regression models

CART 算法的工作方式与之前处理分类模型基本一样，不同之处在于，现在不再以最小化不纯度的方式分割训练集，而是试图以最小化 MSE 的方式分割训练集。

公式 6-4 显示了成本函数，该算法试图最小化这个成本函数。

Equation 6-4. CART cost function for regression

$$J(k, t_k) = \frac{m_{\text{left}}}{m} \text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m} \text{MSE}_{\text{right}} \quad \text{where} \quad \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} (\hat{y}_{\text{node}} - y^{(i)})^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

和处理分类任务时一样，决策树在处理回归问题的时候也容易过拟合。如果不添加任何正则化（默认的超参数），你就会得到图 6-6 左侧的预测结果，显然，过度拟合的程度非常严重。而当我们设置了 `min_samples_leaf = 10`，相对就会产生一个更加合适的模型了，就如图 6-6 所示的那样。

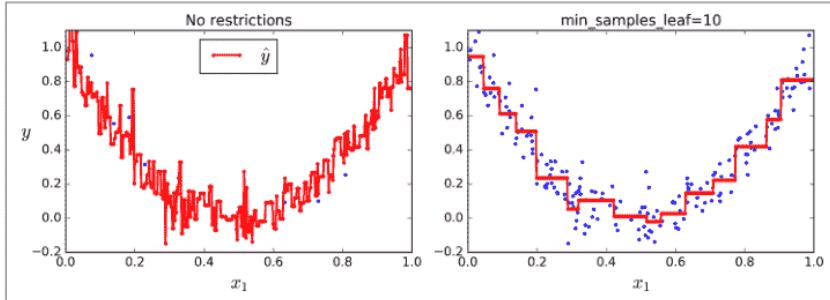


Figure 6-6. Regularizing a Decision Tree regressor

## 不稳定性

我希望你现在了解了决策树到底有哪些特点：

它很容易理解和解释，易于使用且功能丰富而强大。然而，它也有一些限制，首先，你可能已经注意到了，决策树很喜欢设定正交化的决策边界，（所有边界都是和某一个轴相垂直的），这使得它对训练数据集的旋转很敏感，例如图 6-7 显示了一个简单的线性可分数据集。在左图中，决策树可以轻易的将数据分隔开，但是在右图中，当我们把数据旋转了  $45^\circ$  之后，决策树的边界看起来变的格外复杂。尽管两个决策树都完美的拟合了训练数据，右边模型的泛化能力很可能非常差。

解决这个难题的一种方式是使用 PCA 主成分分析（第八章），这样通常能使训练结果变得更好一些。

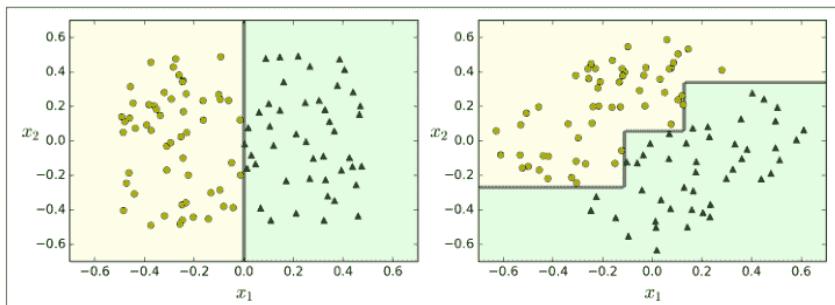


Figure 6-7. Sensitivity to training set rotation

更加通俗的讲，决策时的主要问题是它对训练数据的微小变化非常敏感，举例来说，我们仅仅从鸢尾花训练数据中将最宽的 Iris-Versicolor 拿掉（花瓣长 4.8 厘米，宽 1.8 厘米），然后重新训练决策树模型，你可能就会得到图 6-8 中的模型。正如我们看到的那样，决策树有了非常大的变化（原来的如图 6-2），事实上，由于 Scikit-Learn 的训练算法是非常随机的，即使是相同的训练数据你也可能得到差别很大的模型（除非你设置了随机数种子）。

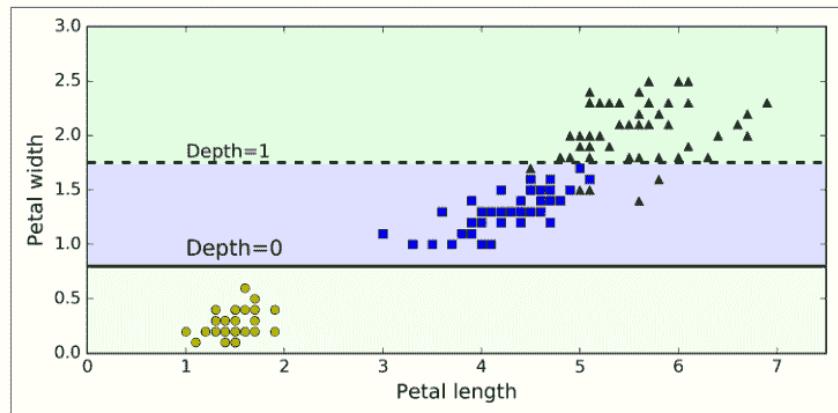


Figure 6-8. Sensitivity to training set details

我们下一章中将会看到，随机森林可以通过多棵树的平均预测值限制这种不稳定性。

## 练习

1. 在 100 万例训练集上训练（没有限制）的决策树的近似深度是多少？
2. 节点的基尼指数比起它的父节点是更高还是更低？它是通常情况下更高/更低，还是永远更高/更低？
3. 如果决策树过拟合了，减少最大深度是一个好的方法吗？
4. 如果决策树对训练集欠拟合了，尝试缩放输入特征是否是一个好主意？
5. 如果对包含 100 万个实例的数据集训练决策树模型需要一个小时，在包含 1000 万个实例的培训集上训练另一个决策树大概需要多少时间呢？
6. 如果你的训练集包含 100,000 个实例，设置 `presort=True` 会加快训练的速度吗？
7. 对 `moons` 数据集进行决策树训练并优化模型。
  - i. 通过语句 `make_moons(n_samples=10000, noise=0.4)` 生成 `moons` 数据集
  - ii. 通过 `train_test_split()` 将数据集分割为训练集和测试集。
  - iii. 进行交叉验证，并使用网格搜索法寻找最好的超参数值（使用 `GridSearchCV` 类的帮助文档）
 

提示：尝试各种各样的 `max_leaf_nodes` 值
  - iv. 使用这些超参数训练全部的训练集数据，并在测试集上测量模型的表现。你应该获得大约 85% 到 87% 的准确度。
8. 生成森林
  - i. 接着前边的练习，现在，让我们生成 1,000 个训练集的子集，每个子集包含 100 个随机选择的实例。提示：你可以使用 Scikit-Learn 的 `ShuffleSplit` 类。

- ii. 使用上面找到的最佳超参数值，在每个子集上训练一个决策树。在测试集上测试这 1000 个决策树。由于它们是在较小的集合上进行了训练，因此这些决策树可能会比第一个决策树效果更差，只能达到约 80% 的准确度。
- iii. 见证奇迹的时刻到了！对于每个测试集实例，生成 1,000 个决策树的预测结果，然后只保留出现次数最多的预测结果（您可以使用 SciPy 的 mode() 函数）。这个函数使你可以对测试集进行多数投票预测。
- iv. 在测试集上评估这些预测结果，你应该获得了一个比第一个模型高一点的准确率，（大约 0.5% 到 1.5%），恭喜，你已经弄出了一个随机森林分类器模型！

## 七、集成学习和随机森林

译者：@friedhelm739

校对者：@飞龙、@PeterHo、@yanmengk、@XinQiu、@YuWang

假设你去随机问很多人一个很复杂的问题，然后把它们的答案合并起来。通常情况下你会发现这个合并的答案比一个专家的答案要好。这就叫做群体智慧。同样的，如果你合并了一组分类器的预测（像分类或者回归），你也会得到一个比单一分类器更好的预测结果。这一组分类器就叫做集成；因此，这个技术就叫做集成学习，一个集成学习算法就叫做集成方法。

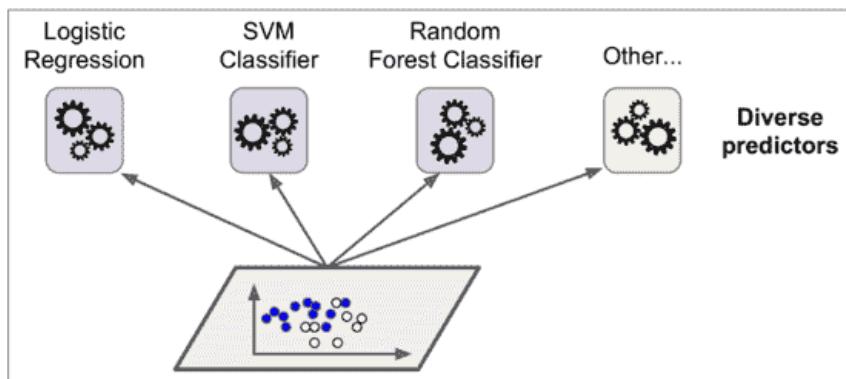
例如，你可以训练一组决策树分类器，每一个都在一个随机的训练集上。为了去做预测，你必须得到所有单棵树的预测值，然后通过投票（例如第六章的练习）来预测类别。例如一种决策树的集成就叫做随机森林，它除了简单之外也是现今存在的最强大的机器学习算法之一。

向我们在第二章讨论的一样，我们会在一个项目快结束的时候使用集成算法，一旦你建立了一些好的分类器，就把他们合并为一个更好的分类器。事实上，在机器学习竞赛中获得胜利的算法经常会包含一些集成方法。

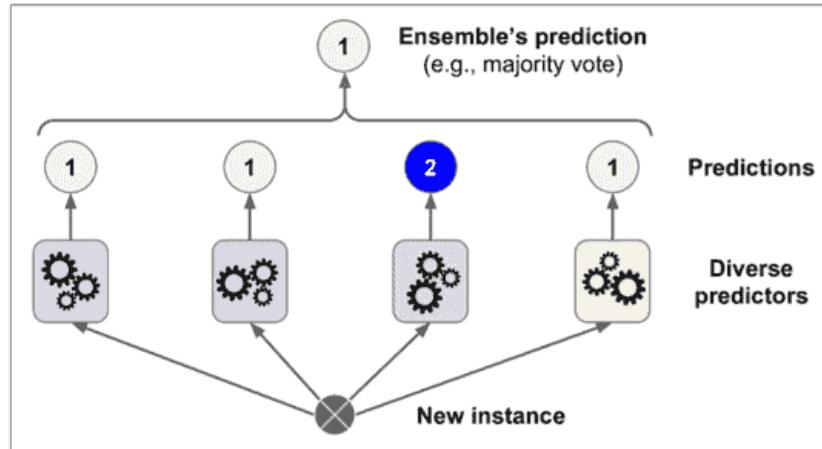
在本章中我们会讨论一下特别著名的集成方法，包括 *bagging*, *boosting*, *stacking*，和其他一些算法。我们也会讨论随机森林。

### 投票分类

假设你已经训练了一些分类器，每一个都有 80% 的准确率。你可能有了一个逻辑斯蒂回归、或一个 SVM、或一个随机森林，或者一个 KNN，或许还有更多（详见图 7-1）

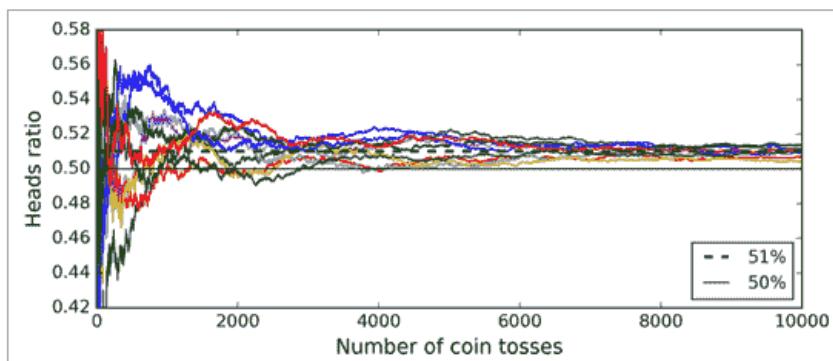


一个非常简单去创建一个更好的分类器的方法就是去整合每一个分类器的预测然后经过投票去预测分类。这种分类器就叫做硬投票分类器（详见图 7-2）。



令人惊奇的是这种投票分类器得出的结果经常会比集成中最好的一个分类器结果更好。事实上，即使每一个分类器都是一个弱学习器（意味着它们也就比瞎猜好点），集成后仍然是一个强学习器（高准确率），只要有足够数量的弱学习者，他们就足够多样化。

这怎么可能？接下来的分析将帮助你解决这个疑问。假设你有一个有偏差的硬币，他有 51% 的几率为正面，49% 的几率为背面。如果你实验 1000 次，你会得到差不多 510 次正面，490 次背面，因此大多数都是正面。如果你用数学计算，你会发现实验 1000 次后，正面概率为 51% 的人比例为 75%。你实验的次数越多，正面的比例越大（例如你试验了 10000 次，总体比例可能性就会达到 97%）。这是因为大数定律：当你一直用硬币实验时，正面的比例会越来越接近 51%。图 7-3 展示了始终有偏差的硬币实验。你可以看到当实验次数上升时，正面的概率接近于 51%。最终所有 10 种实验都会收敛到 51%，它们都大于 50%。



同样的，假设你创建了一个包含 1000 个分类器的集成模型，其中每个分类器的正确率只有 51%（仅比瞎猜好一点点）。如果你用投票去预测类别，你可能得到 75% 的准确率！然而，这仅仅在所有的分类器都独立运行的很好、不会发生有相关性的错误的情况下才会这样，然而每一个分类器都在同一个数据集上训练，导致其很可能会发生这样的错误。他们可能会犯同一种错误，所以也会有很多票投给了错误类别导致集成的准确率下降。

如果使每一个分类器都独立自主的分类，那么集成模型会工作的很好。去得到多样化的分类器的方法之一就是用完全不同的算法，这会使它们会做出不同种类的错误，这会提高集成的正确率

接下来的代码创建和训练了在 sklearn 中的投票分类器。这个分类器由三个不同的分类器组成（训练集是第五章中的 moons 数据集）：

```
>>> 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)
```

让我们看一下在测试集上的准确率：

```
>>> 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 0.872
SVC 0.888
VotingClassifier 0.896
```

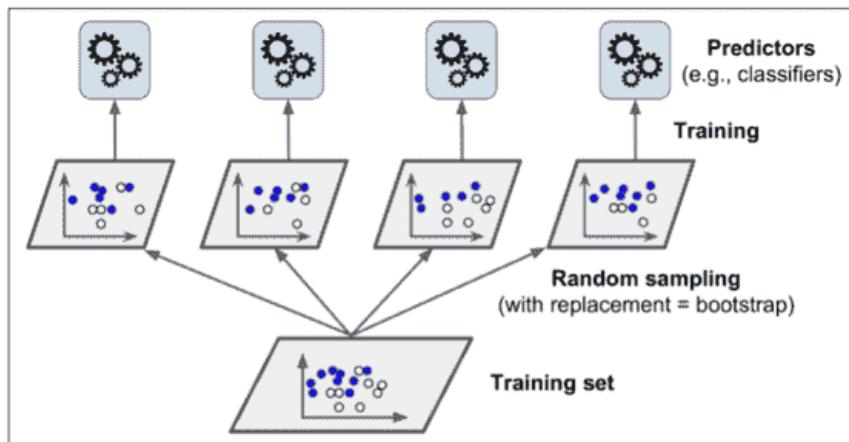
你看！投票分类器比其他单独的分类器表现的都要好。

如果所有的分类器都能够预测类别的概率（例如他们有一个 `predict_proba()` 方法），那么你就可以让 `sklearn` 以最高的类概率来预测这个类，平均在所有的分类器上。这种方式叫做软投票。他经常比硬投票表现的更好，因为它给予高自信的投票更大的权重。你可以通过把 `voting="hard"` 设置为 `voting="soft"` 来保证分类器可以预测类别概率。然而这不是 `SVC` 类的分类器默认的选项，所以你需要把它 `probability hyperparameter` 设置为 `True`（这会使 `SVC` 使用交叉验证去预测类别概率，其降低了训练速度，但会添加 `predict_proba()` 方法）。如果你修改了之前的代码去使用软投票，你会发现投票分类器正确率高达 91%

## Bagging 和 Pasting

就像之前讲到的，可以通过使用不同的训练算法去得到一些不同的分类器。另一种方法就是对每一个分类器都使用相同的训练算法，但是在不同的训练集上去训练它们。有放回采样被称为装袋（*Bagging*，是 *bootstrap aggregating* 的缩写）。无放回采样称为粘贴（*pasting*）。

换句话说，*Bagging* 和 *Pasting* 都允许在多个分类器上对训练集进行多次采样，但只有 *Bagging* 允许对同一种分类器上对训练集进行多次采样。采样和训练过程如图 7-4 所示。



当所有的分类器被训练后，集成可以通过对所有分类器结果的简单聚合来对新的实例进行预测。聚合函数通常对分类是统计模式（例如硬投票分类器）或者对回归是平均。每一个单独的分类器在如果在原始训练集上都是高偏差，但是聚合降低了偏差和方差。通常情况下，集成的结果是有一个相似的偏差，但是对比与在原始训练集上的单一分类器来讲有更小的方差。

正如你在图 7-4 上所看到的，分类器可以通过不同的 CPU 核或其他的服务器一起被训练。相似的，分类器也可以一起被制作。这就是为什么 Bagging 和 Pasting 是如此流行的原因之一：它们的可扩展性很好。

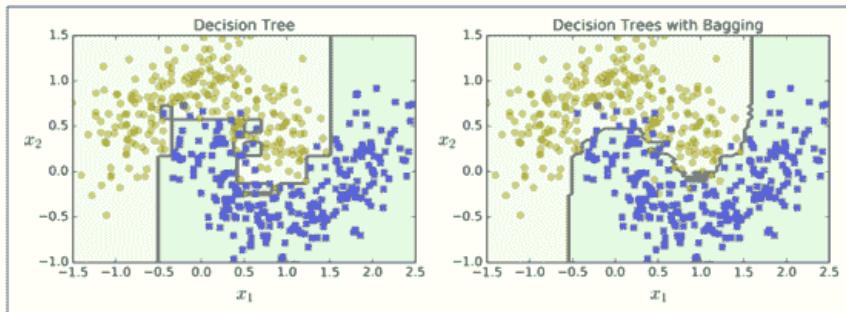
## 在 sklearn 中的 Bagging 和 Pasting

sklearn 为 Bagging 和 Pasting 提供了一个简单的 API: `BaggingClassifier` 类（或者对于回归可以是 `BaggingRegressor`）。接下来的代码训练了一个 500 个决策树分类器的集成，每一个都是在数据集上有放回采样 100 个训练实例下进行训练（这是 Bagging 的例子，如果你想尝试 Pasting，就设置 `bootstrap=False`）。`n_jobs` 参数告诉 sklearn 用于训练和预测所需要 CPU 核的数量。（-1 代表着 sklearn 会使用所有空闲核）：

```
>>> 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)
```

如果基分类器可以预测类别概率（例如它拥有 `predict_proba()` 方法），那么 `BaggingClassifier` 会自动的运行软投票，这是决策树分类器的情况。

图 7-5 对比了单一决策树的决策边界和 Bagging 集成 500 个树的决策边界，两者都在 moons 数据集上训练。正如你所看到的，集成的分类比起单一决策树的分类产生情况更好：集成有一个可比较的偏差但是有一个较小的方差（它在训练集上的错误数目大致相同，但决策边界较不规则）。



Bootstrap 在每个预测器被训练的子集中引入了更多的分集，所以 Bagging 结束时的偏差比 Pasting 更高，但这也意味着预测因子最终变得不相关，从而减少了集合的方差。总体而言，Bagging 通常会导致更好的模型，这就解释了为什么它通常是首选的。然而，如果你有空闲时间和 CPU 功率，可以使用交叉验证来评估 Bagging 和 Pasting 哪一个更好。

## Out-of-Bag 评价

对于 Bagging 来说，一些实例可能被一些分类器重复采样，但其他的有可能不会被采样。`BaggingClassifier` 默认采样。`BaggingClassifier` 默认是有放回的采样  $m$  个实例 (`bootstrap=True`)，其中  $m$  是训练集的大小，这意味着平均下来只有 63% 的训练实例被每个分类器采样，剩下的 37% 个没有被采样的训练实例就叫做 *Out-of-Bag* 实例。注意对于每一个的分类器它们的 37% 不是相同的。

因为在训练中分类器从来没有看到过 oob 实例，所以它可以在这些实例上进行评估，而不需要单独的验证集或交叉验证。你可以拿出每一个分类器的 oob 来评估集本身。

在 `sklearn` 中，你可以在训练后需要创建一个 `BaggingClassifier` 来自动评估时设置 `oob_score=True` 来自动评估。接下来的代码展示了这个操作。评估结果通过变量 `oob_score_` 来显示：

```
>>> bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,boo
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.9306666666666664
```

根据这个 oob 评估，`BaggingClassifier` 可以在测试集上达到 93.1% 的准确率，让我们修改一下：

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.9360000000000005
```

我们在测试集上得到了 93.6% 的准确率，足够接近了！

对于每个训练实例 oob 决策函数也可通过 `oob_decision_function_` 变量来展示。在这种情况下（当基决策器有 `predict_proba()` 时）决策函数会对每个训练实例返回类别概率。例如，oob 评估预测第二个训练实例有 60.6% 的概率属于正类（39.4% 属于负类）：

```
>>> bag_clf.oob_decision_function_
array([[ 0.,  1.], [ 0.60588235,  0.39411765], [ 1.,  0.],
... [ 1.,  0.], [ 0.,  1.], [ 0.48958333,  0.51041667]])
```

## 随机贴片与随机子空间

`BaggingClassifier` 也支持采样特征。它被两个超参数 `max_features` 和 `bootstrap_features` 控制。他们的工作方式和 `max_samples` 和 `bootstrap` 一样，但这是对于特征采样而不是实例采样。因此，每一个分类器都会被在随机的输入特征内进行训练。

当你在处理高维度输入下（例如图片）此方法尤其有效。对训练实例和特征的采样被叫做随机贴片。保留了所有的训练实例（例如 `bootstrap=False` 和 `max_samples=1.0`），但是对特征采样（`bootstrap_features=True` 并且/或者 `max_features` 小于 1.0）叫做随机子空间。

采样特征导致更多的预测多样性，用高偏差换低方差。

## 随机森林

正如我们所讨论的，随机森林是决策树的一种集成，通常是通过 bagging 方法（有时是 pasting 方法）进行训练，通常用 `max_samples` 设置为训练集的大小。与建立一个 `BaggingClassifier` 然后把它放入 `DecisionTreeClassifier` 相反，你可以使用更方便的也是对决策树优化够的 `RandomForestClassifier`（对于回归是 `RandomForestRegressor`）。接下来的代码训练了带有 500 个树（每个被限制为 16 叶子结点）的决策森林，使用所有空闲的 CPU 核：

```
>>>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)
```

除了一些例外，`RandomForestClassifier` 使用 `DecisionTreeClassifier` 的所有超参数（决定数怎么生长），把 `BaggingClassifier` 的超参数加起来控制集成本身。

随机森林算法在树生长时引入了额外的随机；与在节点分裂时需要找到最好分裂特征相反（详见第六章），它在一个随机的特征集中找最好的特征。它导致了树的差异性，并且再一次用高偏差换低方差，总的来说是一个更好的模型。以下是 `BaggingClassifier` 大致相当于之前的 `randomforestclassifier`：

```
>>>bag_clf = BaggingClassifier(DecisionTreeClassifier(splitter="random", max_l
```

## 极随机树

当你在随机森林上生长树时，在每个结点分裂时只考虑随机特征集上的特征（正如之前讨论过的一样）。相比于找到更好的特征我们可以通过使用对特征使用随机阈值使树更加随机（像规则决策树一样）。

这种极随机的树被简称为 *Extremely Randomized Trees*（极随机树），或者更简单的称为 *Extra-Tree*。再一次用高偏差换低方差。它还使得 *Extra-Tree* 比规则的随机森林更快地训练，因为在每个节点上找到每个特征的最佳阈值是生长树最耗时的任务之一。

你可以使用 `sklearn` 的 `ExtraTreesClassifier` 来创建一个 *Extra-Tree* 分类器。他的 API 跟 `RandomForestClassifier` 是相同的，相似的，`ExtraTreesRegressor` 跟 `RandomForestRegressor` 也是相同的 API。

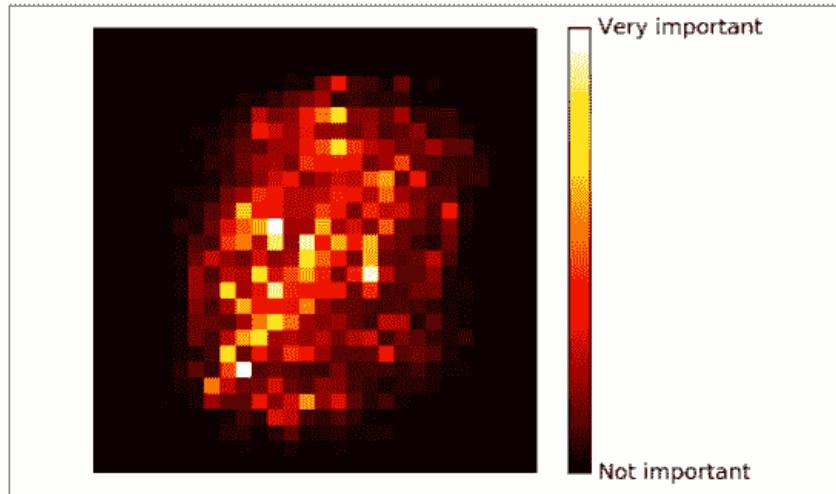
我们很难去分辨 `ExtraTreesClassifier` 和 `RandomForestClassifier` 到底哪个更好。通常情况下是通过交叉验证来比较它们（使用网格搜索调整超参数）。

## 特征重要度

最后，如果你观察一个单一决策树，重要的特征会出现在更靠近根部的位置，而不重要的特征会经常出现在靠近叶子的位置。因此我们可以通过计算一个特征在森林的全部树中出现的平均深度来预测特征的重要性。`sklearn` 在训练后会自动计算每个特征的重要性。你可以通过 `feature_importances_` 变量来查看结果。例如如下代码在鸢尾花数据集（第四章介绍）上训练了一个 `RandomForestClassifier` 模型，然后输出了每个特征的重要性。看来，最重要的特征是花瓣长度（44%）和宽度（42%），而萼片长度和宽度相对比较是不重要的（分别为 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
```

相似的，如果你在 MNIST 数据及上训练随机森林分类器（在第三章上介绍），然后画出每个像素的重要性，你可以得到图 7-6 的图片。



随机森林可以非常方便快速得了解哪些特征实际上是重要的，特别是你需要进行特征选择的时候。

## 提升

提升（Boosting，最初称为假设增强）指的是可以将几个弱学习者组合成强学习者的集成方法。对于大多数的提升方法的思想就是按顺序去训练分类器，每一个都要尝试修正前面的分类。现如今已经有很多的提升方法了，但最著名的就是 *Adaboost*（适应性提升，是 *Adaptive Boosting* 的简称）和 *Gradient Boosting*（梯度提升）。让我们先从 *Adaboost* 说起。

### Adaboost

使一个新的分类器去修正之前分类结果的方法就是对之前分类结果不对的训练实例多加关注。这导致新的预测因子越来越多地聚焦于这种情况。这是 *Adaboost* 使用的技术。

举个例子，去构建一个 Adaboost 分类器，第一个基分类器（例如一个决策树）被训练然后在训练集上做预测，在误分类训练实例上的权重就增加了。第二个分类机使用更新过的权重然后再一次训练，权重更新，以此类推（详见图 7-7）

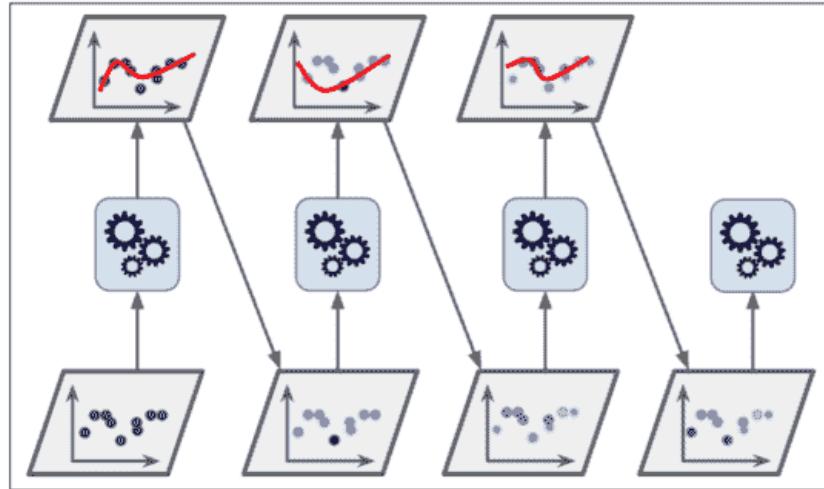
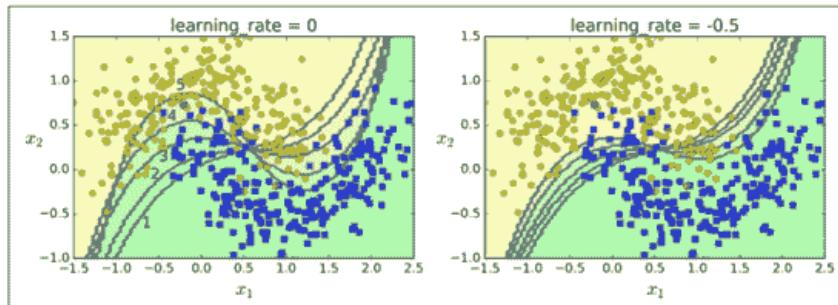


图 7-8 显示连续五次预测的 moons 数据集的决策边界（在本例中，每一个分类器都是高度正则化带有 RBF 核的 SVM）。第一个分类器误分类了很多实例，所以它们的权重被提升了。第二个分类器因此对这些误分类的实例分类效果更好，以此类推。右边的图代表了除了学习率减半外（误分类实例权重每次迭代上升一半）相同的预测序列。你可以看出，序列学习技术与梯度下降很相似，除了调整单个预测因子的参数以最小化代价函数之外，AdaBoost 增加了集合的预测器，逐渐使其更好。



一旦所有的分类器都被训练后，除了分类器根据整个训练集上的准确率被赋予的权重外，集成预测就非常像 Bagging 和 Pasting 了。

序列学习技术的一个重要的缺点就是：它不能被并行化（只能按步骤），因为每个分类器只能在之前的分类器已经被训练和评价后再进行训练。因此，它不像 Bagging 和 Pasting 一样。

让我们详细看一下 Adaboost 算法。每一个实例的权重  $w_i$  初始都被设为  $1/m$  第一个分类器被训练，然后他的权重误差率  $r_1$  在训练集上算出，详见公式 7-1。

公式 7-1：第  $j$  个分类器的权重误差率

$$r_j = \frac{\sum_{i=1}^m w^{(i)}}{\sum_{i=1}^m w^{(i)} - \hat{y}_j^{(i)} \neq y^{(i)}}$$

其中  $y_{\text{tilde}[j]}^{(i)}$  是第  $j$  个分类器对于第  $i$  实例的预测。

分类器的权重  $\alpha[j]$  随后用公式 7-2 计算出来。其中  $\eta$  是超参数学习率（默认为 1）。分类器准确率越高，它的权重就越高。如果它只是瞎猜，那么它的权重会趋近于 0。然而，如果它总是出错（比瞎猜的几率都低），它的权重会使负数。

公式 7-2：分类器权重

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

接下来实例的权重会按照公式 7-3 更新：误分类的实例权重会被提升。

公式 7-3 权重更新规则

对于  $i=1, 2, \dots, m$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

随后所有实例的权重都被归一化（例如被  $\sum w[i], i = 1 \rightarrow m$  整除）

最后，一个新的分类器通过更新过的权重训练，整个过程被重复（新的分类器权重被计算，实例的权重被更新，随后另一个分类器被训练，以此类推）。当规定的分类器数量达到或者最好的分类器被找到后算法就会停止。

为了进行预测，Adaboost 通过分类器权重  $\alpha[j]$  简单的计算了所有的分类器和权重。预测类别会是权重投票中主要的类别。（详见公式 7-4）

公式 7-4: Adaboost 分类器

$$\hat{y}(\mathbf{x}) = \operatorname{argmax}_k \sum_{j=1}^N \alpha_j \hat{y}_j^{(x)} = k$$

其中  $N$  是分类器的数量。

sklearn 通常使用 Adaboost 的多分类版本 *SAMME* (这就代表了 分段加建模使用多类指数损失函数)。如果只有两类别，那么 *SAMME* 是与 Adaboost 相同的。如果分类器可以预测类别概率 (例如如果它们有 `predict_proba()`)，如果 sklearn 可以使用 *SAMME* 叫做 *SAMME.R* 的变量 (*R* 代表“REAL”)，这种依赖于类别概率的通常比依赖于分类器的更好。

接下来的代码训练了使用 sklearn 的 `AdaBoostClassifier` 基于 200 个决策树桩 Adaboost 分类器 (正如你说期待的，对于回归也有 `AdaBoostRegressor`)。一个决策树桩是 `max_depth=1` 的决策树-换句话说，是一个单一的决策节点加上两个叶子结点。这就是 `AdaBoostClassifier` 的默认基分类器：

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> ada_clf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1), n_estimators=200, learning_rate=0.5)
>>> ada_clf.fit(X_train, y_train)
```

如果你的 Adaboost 集成过拟合了训练集，你可以尝试减少基分类器的数量或者对基分类器使用更强的正则化。

## 梯度提升

另一个非常著名的提升算法是梯度提升。与 Adaboost 一样，梯度提升也是通过向集成中逐步增加分类器运行的，每一个分类器都修正之前的分类结果。然而，它并不像 Adaboost 那样每一次迭代都更改实例的权重，这个方法是去使用新的分类器去拟合前面分类器预测的残差。

让我们通过一个使用决策树当做基分类器的简单的回归例子 (回归当然也可以使用梯度提升)。这被叫做梯度提升回归树 (GBRT, *Gradient Tree Boosting* 或者 *Gradient Boosted Regression Trees*)。首先我们用 `DecisionTreeRegressor` 去拟合训练集 (例如一个有噪二次训练集)：

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> tree_reg1 = DecisionTreeRegressor(max_depth=2)
>>> tree_reg1.fit(X, y)
```

现在在第一个分类器的残差上训练第二个分类器：

```
>>> y2 = y - tree_reg1.predict(X)
>>> tree_reg2 = DecisionTreeRegressor(max_depth=2)
>>> tree_reg2.fit(X, y2)
```

随后在第二个分类器的残差上训练第三个分类器：

```
>>> 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个回归器的集成。它可以通过集成所有树的预测来在一个新的实例上进行预测。

```
>>> y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_re
```

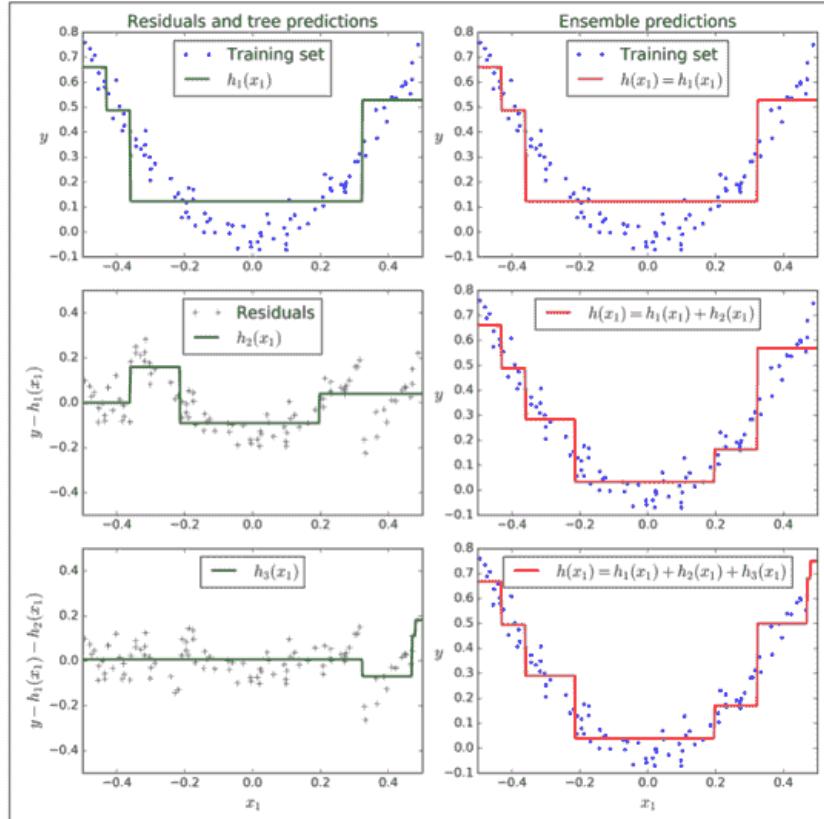
图 7-9 在左栏展示了这三个树的预测，在右栏展示了集成的预测。在第一行，集成只有一个树，所以它与第一个树的预测相似。在第二行，一个新的树在第一个树的残差上进行训练。在右边栏可以看出集成的预测等于前两个树预测的和。相同的，

在第三行另一个树在第二个数的残差上训练。你可以看到集成的预测会变的更好。

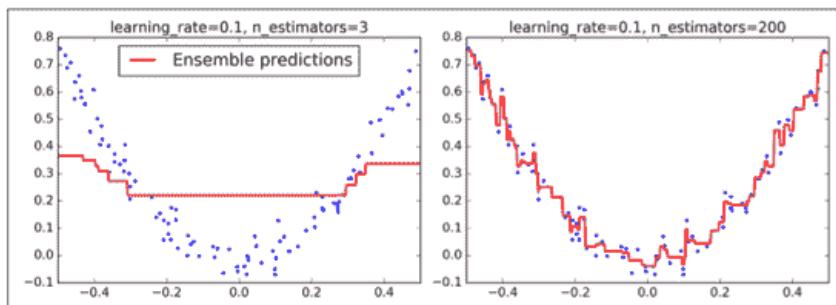
我们可以使用 sklearn 中的 GradientBoostingRegressor 来训练 GBRT 集成。

与 RandomForestClassifier 相似，它也有超参数去控制决策树的生长（例如 max\_depth，min\_samples\_leaf 等等），也有超参数去控制集成训练，例如基分类器的数量（n\_estimators）。接下来的代码创建了与之前相同的集成：

```
>>>from sklearn.ensemble import GradientBoostingRegressor
>>>gbdt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=0.1)
>>>gbdt.fit(X, y)
```



超参数 learning\_rate 确立了每个树的贡献。如果你把它设置为一个很小的树，例如 0.1，在集成中就需要更多的树去拟合训练集，但预测通常会更好。这个正则化技术叫做 *shrinkage*。图 7-10 展示了两个在低学习率上训练的 GBRT 集成：其中左面是一个没有足够树去拟合训练集的树，右面是有过多的树过拟合训练集的树。



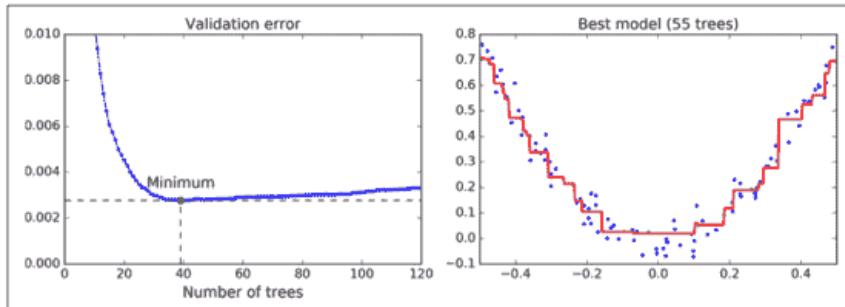
为了找到树的最优数量，你可以使用早停技术（第四章讨论）。最简单使用这个技术的方法就是使用 staged\_predict()：它在训练的每个阶段（用一棵树，两棵树等）返回一个迭代器。接下来的代码用 120 个树训练了一个 GBRT 集成，然后在

训练的每个阶段验证错误以找到树的最佳数量，最后使用 GBRT 树的最优数量训练另一个集成：

```
>>>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)
>>>gbrt_best = GradientBoostingRegressor(max_depth=2, n_estimators=bst_n_estimators)
>>>gbrt_best.fit(X_train, y_train)
```

验证错误在图 7-11 的左面展示，最优模型预测被展示在右面。



你也可以早早的停止训练来实现早停（与先在一大堆树中训练，然后再回头去找最优数目相反）。你可以通过设置 `warm_start=True` 来实现，这使得当 `fit()` 方法被调用时 `sklearn` 保留现有树，并允许增量训练。接下来的代码在当一行中的五次迭代验证错误没有改善时会停止训练：

```
>>>gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)
min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbdt.n_estimators = n_estimators
    gbdt.fit(X_train, y_train)
    y_pred = gbdt.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
```

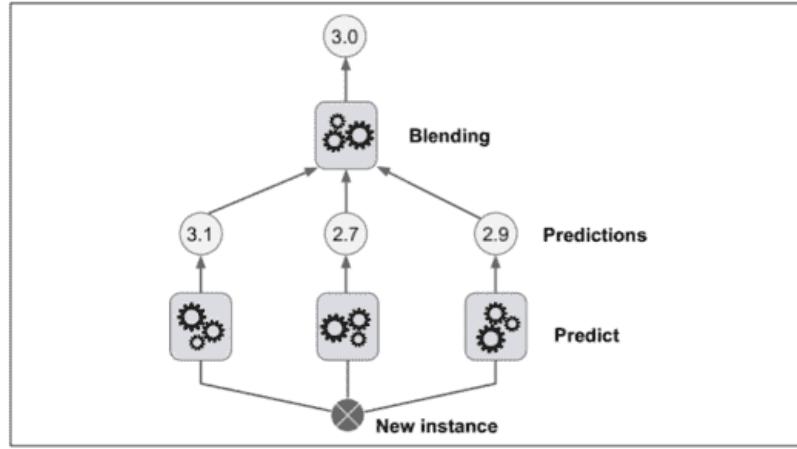
`GradientBoostingRegressor` 也支持指定用于训练每棵树的训练实例比例的超参数 `subsample`。例如如果 `subsample=0.25`，那么每个树都会在 25% 随机选择的训练实例上训练。你现在也能猜出来，这也是个高偏差换低方差的作用。它同样也加速了训练。这个技术叫做随机梯度提升。

也可能对其他损失函数使用梯度提升。这是由损失超参数控制（见 `sklearn` 文档）。

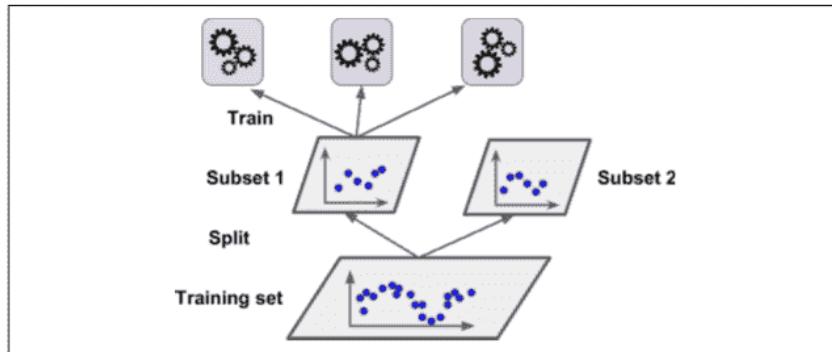
## Stacking

本章讨论的最后一个集成方法叫做 *Stacking* (*stacked generalization* 的缩写)。这个算法基于一个简单的想法：不使用琐碎的函数（如硬投票）来聚合集合中所有分类器的预测，我们为什么不训练一个模型来执行这个聚合？图 7-12 展示了这样

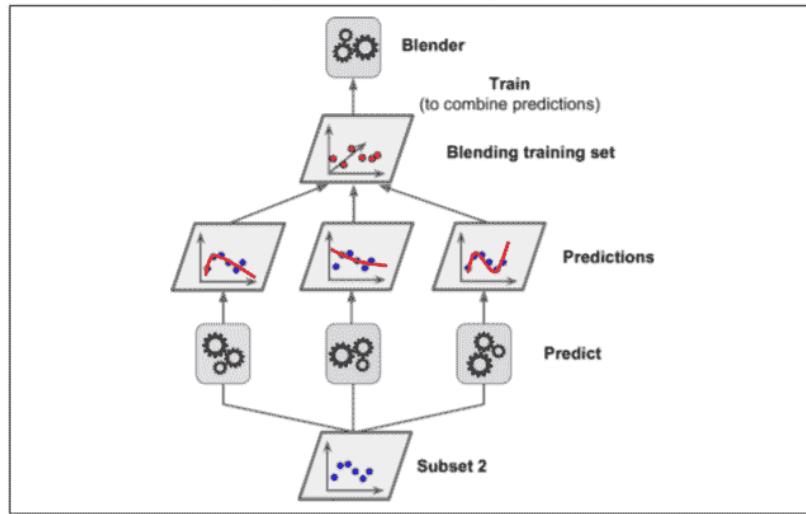
一个在新的回归实例上预测的集成。底部三个分类器每一个都有不同的值（3.1, 2.7 和 2.9），然后最后一个分类器（叫做 *blender* 或者元学习器）把这三个分类器的结果当做输入然后做出最终决策（3.0）。



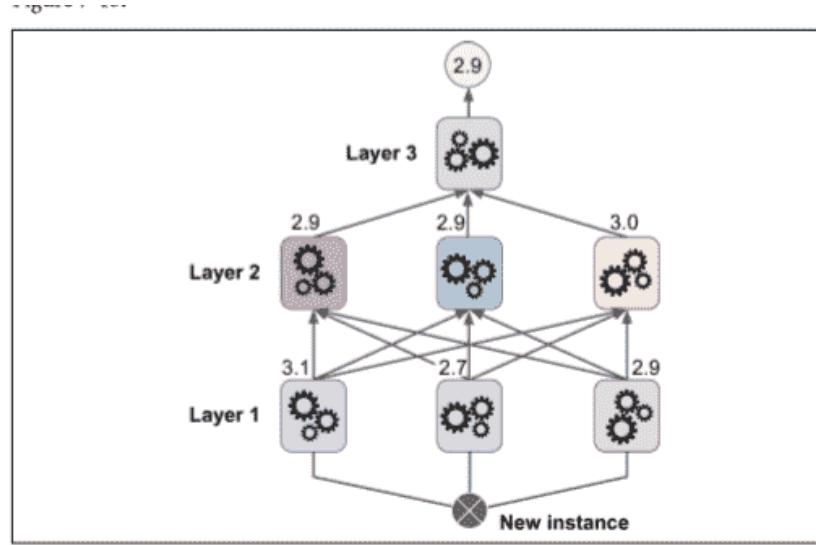
为了训练这个 *blender*，一个通用的方法是采用保持集。让我们看看它怎么工作。首先，训练集被分为两个子集，第一个子集被用作训练第一层（详见图 7-13）。



接下来，第一层的分类器被用来预测第二个子集（保持集）（详见 7-14）。这确保了预测结果很“干净”，因为这些分类器在训练的时候没有使用过这些实例。现在对在保持集中的每一个实例都有三个预测值。我们现在可以使用这些预测结果作为输入特征来创建一个新的训练集（这使得这个训练集是三维的），并且保持目标数值不变。随后 *blender* 在这个新的训练集上训练，因此，它学会了预测第一层预测的目标值。



显然我们可以用这种方法训练不同的 *blender*（例如一个线性回归，另一个是随机森林等等）：我们得到了一层 *blender*。诀窍是将训练集分成三个子集：第一个子集用来训练第一层，第二个子集用来创建训练第二层的训练集（使用第一层分类器的预测值），第三个子集被用来创建训练第三层的训练集（使用第二层分类器的预测值）。以上步骤做完了，我们可以通过逐个遍历每个层来预测一个新的实例。详见图 7-15。



然而不幸的是，sklearn 并不直接支持 stacking，但是你自己组建是很容易的（看接下来的练习）。或者你也可以使用开源的项目例如 *brew*（网址为 <https://github.com/viisar/brew>）

## 练习

1. 如果你在相同训练集上训练 5 个不同的模型，它们都有 95% 的准确率，那么你是否可以通过组合这个模型来得到更好的结果？如果可以那怎么做呢？如果不可以请给出理由。
2. 软投票和硬投票分类器之间有什么区别？
3. 是否有可能通过分配多个服务器来加速 bagging 集成系统的训练？pasting 集成，boosting 集成，随机森林，或 stacking 集成怎么样？

4. out-of-bag 评价的好处是什么？
5. 是什么使 Extra-Tree 比规则随机森林更随机呢？这个额外的随机有什么帮助呢？那这个 Extra-Tree 比规则随机森林谁更快呢？
6. 如果你的 Adaboost 模型欠拟合，那么你需要怎么调整超参数？
7. 如果你的梯度提升过拟合，那么你应该调高还是调低学习率呢？
8. 导入 MNIST 数据（第三章中介绍），把它切分进一个训练集，一个验证集，和一个测试集（例如 40000 个实例进行训练，10000 个进行验证，10000 个进行测试）。然后训练多个分类器，例如一个随机森林分类器，一个 Extra-Tree 分类器和一个 SVM。接下来，尝试将它们组合成集成，使用软或硬投票分类器来胜过验证集上的所有集合。一旦找到了，就在测试集上实验。与单个分类器相比，它的性能有多好？
9. 从练习 8 中运行个体分类器来对验证集进行预测，并创建一个新的训练集并生成预测：每个训练实例是一个向量，包含来自所有分类器的图像的预测集，目标是图像类别。祝贺你，你刚刚训练了一个 *blender*，和分类器一起组成了一个叠加组合！现在让我们来评估测试集上的集合。对于测试集中的每个图像，用所有分类器进行预测，然后将预测馈送到 *blender* 以获得集合的预测。它与你早期训练过的投票分类器相比如何？

练习的答案都在附录 A 上。

## 八、降维

译者：[@loveSnowBest](#)

校对者：[@飞龙](#)、[@PeterHo](#)、[@yanmengk](#)、[@XinQiu](#)、[@Lisanaaaa](#)

很多机器学习的问题都会涉及到有着几千甚至数百万维的特征的训练实例。这不仅让训练过程变得非常缓慢，同时还很难找到一个很好的解，我们接下来就会遇到这种情况。这种问题通常被称为维数灾难（curse of dimensionality）。

幸运的是，在现实生活中我们经常可以极大的降低特征维度，将一个十分棘手的问题转变成一个可以较为容易解决的问题。例如，对于 MNIST 图片集（第 3 章中提到）：图片四周边缘部分的像素几乎总是白的，因此你完全可以将这些像素从你的训练集中扔掉而不会丢失太多信息。图 7-6 向我们证实了这些像素的确对我们的分类任务是完全不重要的。同时，两个相邻的像素往往是高度相关的：如果你想要将他们合并成一个像素（比如取这两个像素点的平均值）你并不会丢失很多信息。

**警告：**降维肯定会丢失一些信息（这就好比将一个图片压缩成 JPEG 的格式会降低图像的质量），因此即使这种方法可以加快训练的速度，同时也会让你的系统表现的稍微差一点。降维会让你的工作流水线更复杂因而更难维护。所有你应该先尝试使用原始的数据来训练，如果训练速度太慢的话再考虑使用降维。在某些情况下，降低训练集数据的维度可能会筛选掉一些噪音和不必要的细节，这可能会让你的结果比降维之前更好（这种情况通常不会发生；它只会加快你训练的速度）。

降维除了可以加快训练速度外，在数据可视化方面（或者 DataViz）也十分有用。降低特征维度到 2（或者 3）维从而可以在图中画出一个高维度的训练集，让我们可以通过视觉直观的发现一些非常重要的信息，比如聚类。

在这一章里，我们将会讨论维数灾难问题并且了解在高维空间的数据。然后，我们将会展示两种主要的降维方法：投影（projection）和流形学习（Manifold Learning），同时我们还会介绍三种流行的降维技术：主成分分析（PCA），核主成分分析（Kernel PCA）和局部线性嵌入（LLE）。

### 维数灾难

我们已经习惯生活在一个三维的世界里，以至于当我们尝试想象更高维的空间时，我们的直觉不管用了。即使是一个基本的 4D 超正方体也很难在我们的脑中想象出来（见图 8-1），更不用说一个 200 维的椭球弯曲在一个 1000 维的空间里了。

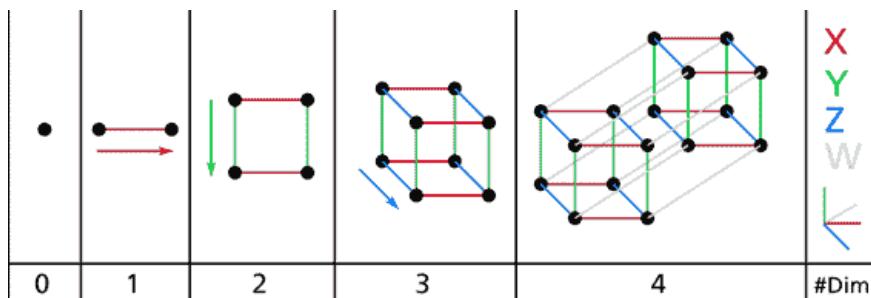


图 8-1 点，线，方形，立方体和超正方体（0D 到 4D 超正方体）

这表明很多物体在高维空间表现的十分不同。比如，如果你在一个正方形单元中随机取一个点（一个  $1 \times 1$  的正方形），那么随机选的点离所有边界大于 0.001（靠近中间位置）的概率为 0.4% ( $1 - 0.998^2$ )（换句话说，一个随机产生的点不大可能严格落在某一个维度上。但是，在一个 1,000,000 维的单位超正方体（一个  $1 \times 1 \times \dots \times 1$  的立方体，有 10,000 个 1），这种可能性超过了 99.999999%。在高维超正方体中，大多数点都分布在边界处。

还有一个更麻烦的区别：如果你在一个平方单位中随机选取两个点，那么这两个点之间的距离平均约为 0.52。如果您在单位 3D 立方体中选取两个随机点，平均距离将大致为 0.66。但是，在一个 1,000,000 维超立方体中随机抽取两点呢？那么，平均距离，信不信由你，大概为 408.25（大致  $\sqrt{1,000,000}/6$ ）！这非常违反直觉：当它们都位于同一单元超立方体内时，两点是怎么距离这么远的？这一事实意味着高维数据集有很大风险分布的非常稀疏：大多数训练实例可能彼此远离。当然，这也意味着一个新实例可能远离任何训练实例，这使得预测的可靠性远低于我们处理较低维度数据的预测，因为它们将基于更大的推测（extrapolations）。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，维数爆炸的一个解决方案是增加训练集的大小从而达到拥有足够密度的训练集。不幸的是，在实践中，达到给定密度所需的训练实例的数量随着维度的数量呈指数增长。如果只有 100 个特征（比 MNIST 问题要少得多）并且假设它们均匀分布在所有维度上，那么如果想要各个临近的训练实例之间的距离在 0.1 以内，您需要比宇宙中的原子还要多的训练实例。

## 降维的主要方法

在我们深入研究具体的降维算法之前，我们来看看降低维度的两种主要方法：投影和流形学习。

### 投影（Projection）

在大多数现实生活的问题中，训练实例并不是在所有维度上均匀分布的。许多特征几乎是常数，而其他特征则高度相关（如前面讨论的 MNIST）。结果，所有训练实例实际上位于（或接近）高维空间的低维子空间内。这听起来有些抽象，所以我们不妨来看一个例子。在图 8-2 中，您可以看到由圆圈表示的 3D 数据集。

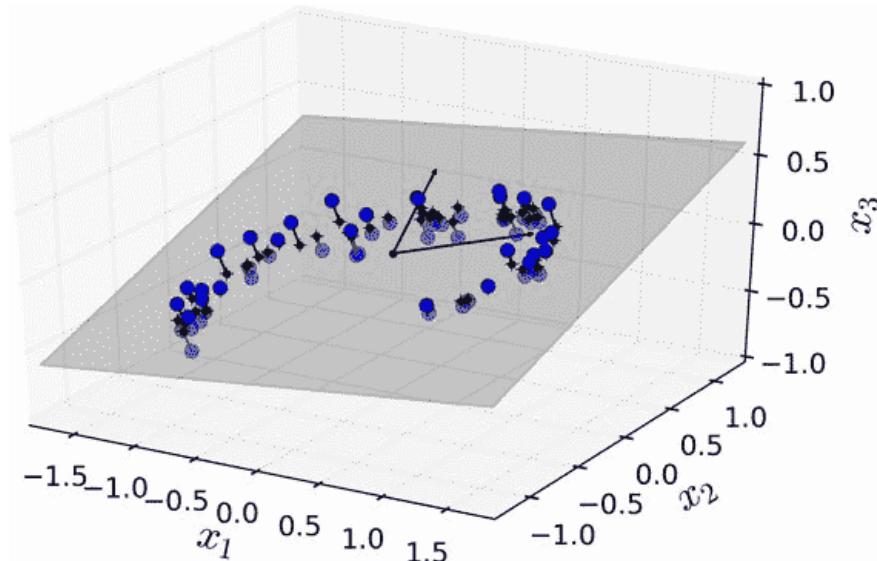


图 8-2 一个分布接近于 2D 子空间的 3D 数据集

注意到所有训练实例的分布都贴近一个平面：这是高维（3D）空间的较低维（2D）子空间。现在，如果我们将每个训练实例垂直投影到这个子空间上（就像将短线连接到平面的点所表示的那样），我们就可以得到如图 8-3 所示的新 2D 数据集。铛铛铛！我们刚刚将数据集的维度从 3D 降低到了 2D。请注意，坐标轴对应于新的特征  $z_1$  和  $z_2$ （平面上投影的坐标）。

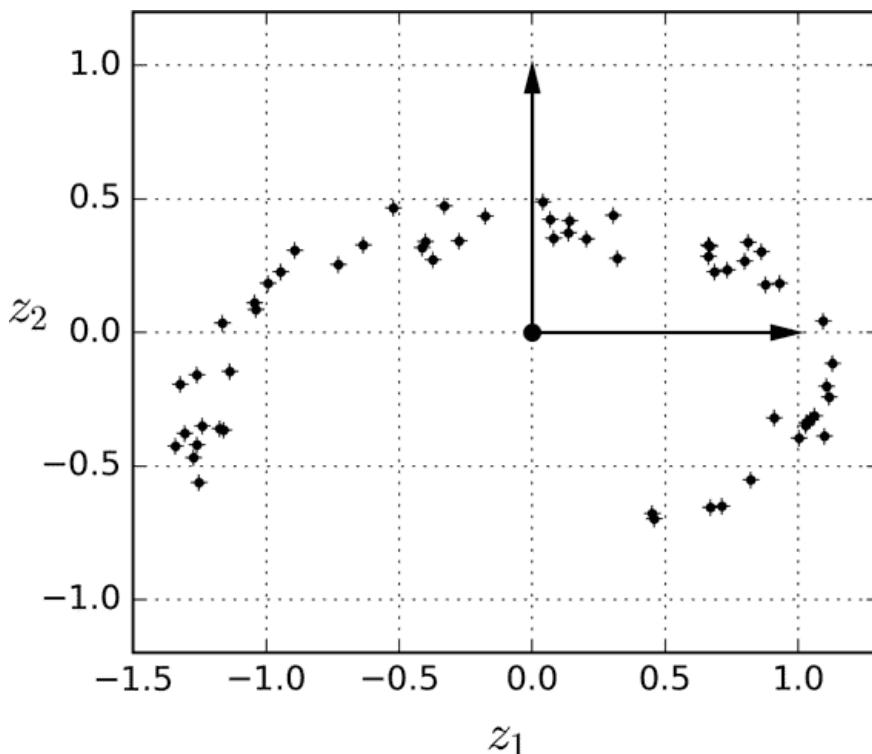


图 8-3 一个经过投影后的新的 2D 数据集

但是，投影并不总是降维的最佳方法。在很多情况下，子空间可能会扭曲和转动，比如图 8-4 所示的着名瑞士滚动玩具数据集。

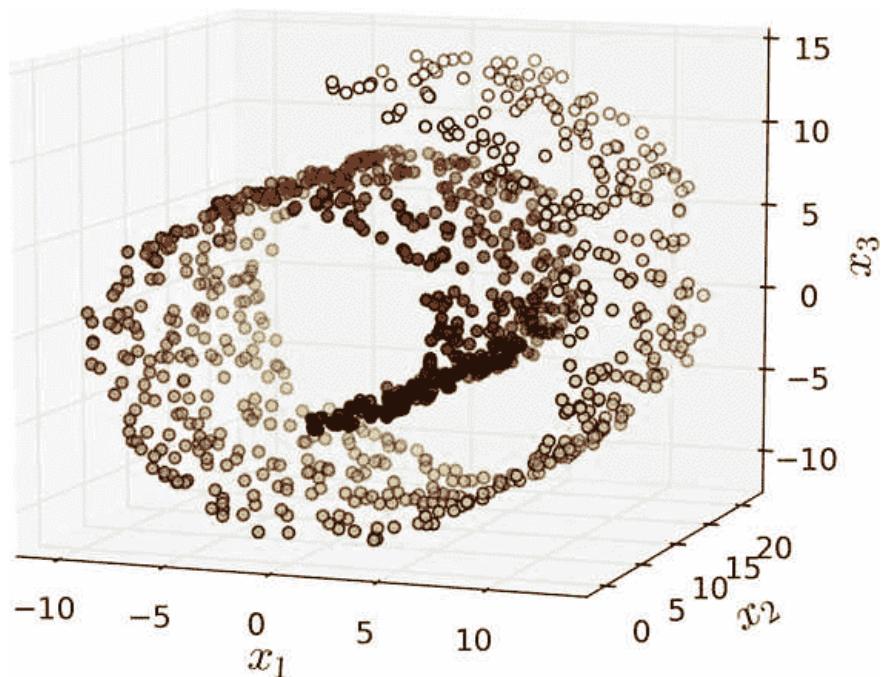


图 8-4 瑞士滚动数玩具数据集

简单地将数据集投射到一个平面上（例如，直接丢弃  $x_3$ ）会将瑞士卷的不同层叠在一起，如图 8-5 左侧所示。但是，你真正想要的是展开瑞士卷所获取到的类似图 8-5 右侧的 2D 数据集。

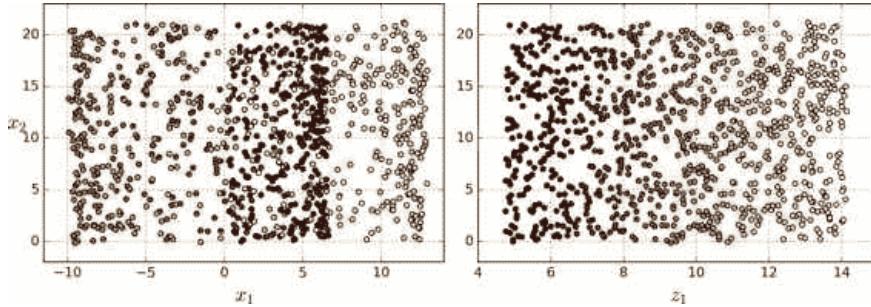


图 8-5 投射到平面的压缩（左）vs 展开瑞士卷（右）

## 流形学习

瑞士卷一个是二维流形的例子。简而言之，二维流形是一种二维形状，它可以在更高维空间中弯曲或扭曲。更一般地，一个  $d$  维流形是类似于  $d$  维超平面的  $n$  维空间（其中  $d < n$ ）的一部分。在我们瑞士卷这个例子中， $d = 2$ ， $n = 3$ ：它有些像 2D 平面，但是它实际上是在第三维中卷曲。

许多降维算法通过对训练实例所在的流形进行建模从而达到降维目的；这叫做流形学习。它依赖于流形猜想（manifold assumption），也被称为流形假设（manifold hypothesis），它认为大多数现实世界的高维数据集大都靠近一个更低维的流形。这种假设经常在实践中被证实。

让我们再回到 MNIST 数据集：所有手写数字图像都有一些相似之处。它们由连线组成，边界是白色的，大多是在图片中间的，等等。如果你随机生成图像，只有一小部分看起来像手写数字。换句话说，如果您尝试创建数字图像，那么您的自由度远低于您生成任何随便一个图像时的自由度。这些约束往往将数据集压缩到较低维流形中。

流形假设通常包含着另一个隐含的假设：你现在的手上的工作（例如分类或回归）如果在流形的较低维空间中表示，那么它们会变得更容易。例如，在图 8-6 的第一行中，瑞士卷被分为两类：在三维空间中（图左上），分类边界会相当复杂，但在二维展开的流形空间中（图右上），分类边界是一条简单的直线。

但是，这个假设并不总是成立。例如，在图 8-6 的最下面一行，决策边界位于  $x_1 = 5$ （图左下）。这个决策边界在原始三维空间（一个垂直平面）看起来非常简单，但在展开的流形中却变得更复杂了（四个独立线段的集合）（图右下）。

简而言之，如果在训练模型之前降低训练集的维数，那训练速度肯定会加快，但并不总是会得出更好的训练效果；这一切都取决于数据集。

希望你现在对于维数爆炸以及降维算法如何解决这个问题有了一定的理解，特别是对流形假设提出的内容。本章的其余部分将介绍一些最流行的降维算法。

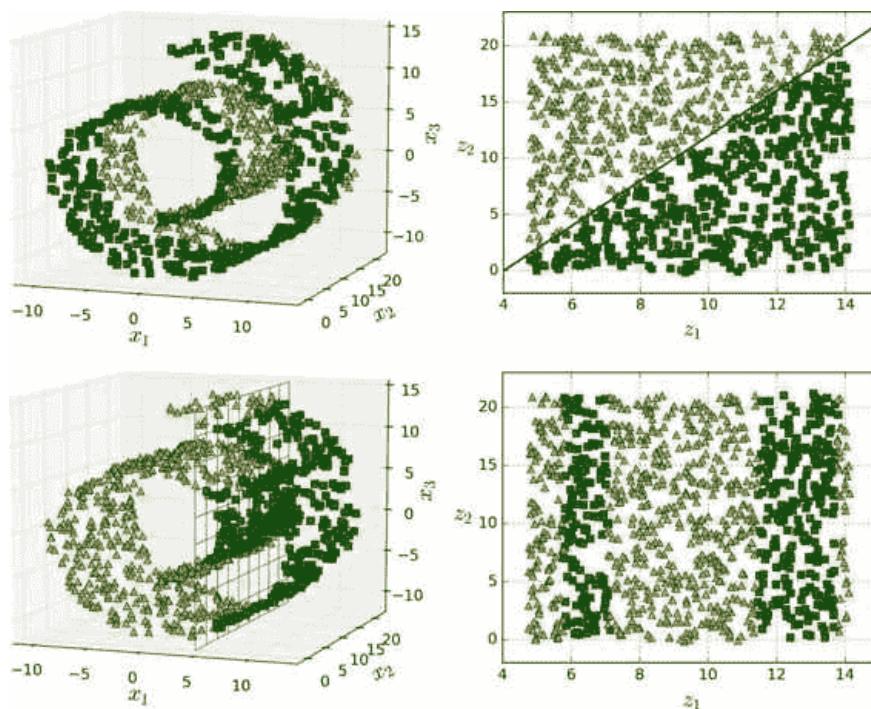


图 8-6 决策边界并不总是会在低维空间中变的简单

## 主成分分析 (PCA)

主成分分析 (Principal Component Analysis) 是目前为止最流行的降维算法。首先它找到接近数据集分布的超平面，然后将所有的数据都投影到这个超平面上。

### 保留 (最大) 方差

在将训练集投影到较低维超平面之前，您首先需要选择正确的超平面。例如图 8-7 左侧是一个简单的二维数据集，以及三个不同的轴（即一维超平面）。图右边是将数据集投影到每个轴上的结果。正如你所看到的，投影到实线上保留了最大方差，而在点线上的投影只保留了非常小的方差，投影到虚线上保留的方差则处于上述两者之间。

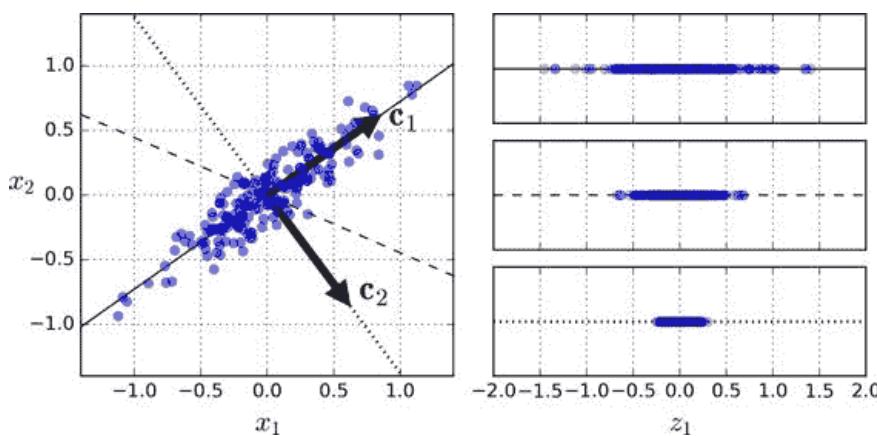


图 8-7 选择投射到哪一个子空间

选择保持最大方差的轴看起来是合理的，因为它很可能比其他投影损失更少的信息。证明这种选择的另一种方法是，选择这个轴使得将原始数据集投影到该轴上的均方距离最小。这是就 PCA 背后的思想，相当简单。

## 主成分 (Principle Components)

PCA 寻找训练集中可获得最大方差的轴。在图 8-7 中，它是一条直线。它还发现了一个与第一个轴正交的第二个轴，选择它可以获得最大的残差。在这个 2D 例子中，没有选择：就只有这条点线。但如果在一个更高维的数据集中，PCA 也可以找到与前两个轴正交的第三个轴，以及与数据集中维数相同的第四个轴，第五个轴等。定义第  $i$  个轴的单位向量被称为第  $i$  个主成分 (PC)。在图 8-7 中，第一个 PC 是  $c_1$ ，第二个 PC 是  $c_2$ 。在图 8-2 中，前两个 PC 用平面中的正交箭头表示，第三个 PC 与上述 PC 形成的平面正交（指向上或下）。

**概述：** 主成分的方向不稳定：如果您稍微打乱一下训练集并再次运行 PCA，则某些新 PC 可能会指向与原始 PC 方向相反。但是，它们通常仍位于同一轴线上。在某些情况下，一对 PC 甚至可能会旋转或交换，但它们定义的平面通常保持不变。

那么如何找到训练集的主成分呢？幸运的是，有一种称为奇异值分解 (SVD) 的标准矩阵分解技术，可以将训练集矩阵  $X$  分解为三个矩阵  $U \cdot \Sigma \cdot V^T$  的点积，其中  $V^T$  包含我们想要的所有主成分，如公式 8-1 所示。

公式 8-1 主成分矩阵

$$V^T = \begin{pmatrix} | & | & & | \\ c_1 & c_2 & \cdots & c_n \\ | & | & & | \end{pmatrix}$$

下面的 Python 代码使用了 Numpy 提供的 `svd()` 函数获得训练集的所有主成分，然后提取前两个 PC：

```
X_centered=X-X.mean(axis=0)
U,s,V=np.linalg.svd(X_centered)
c1=V.T[:,0]
c2=V.T[:,1]
```

**警告：**PCA 假定数据集以原点为中心。正如我们将看到的，Scikit-Learn 的 PCA 类负责为您的数据集中心化处理。但是，如果您自己实现 PCA（如前面的示例所示），或者如果您使用其他库，不要忘记首先要先对数据做中心化处理。

## 投影到 d 维空间

一旦确定了所有的主成分，你就可以通过将数据集投影到由前  $d$  个主成分构成的超平面上，从而将数据集的维数降至  $d$  维。选择这个超平面可以确保投影将保留尽可能多的方差。例如，在图 8-2 中，3D 数据集被投影到由前两个主成分定义的 2D 平面，保留了大部分数据集的方差。因此，2D 投影看起来非常像原始 3D 数据集。

为了将训练集投影到超平面上，可以简单地通过计算训练集矩阵  $X$  和  $W_d$  的点积， $W_d$  定义为包含前  $d$  个主成分的矩阵（即由  $V^T$  的前  $d$  列组成的矩阵），如公式 8-2 所示。

公式 8-2 将训练集投影到  $d$  维空间

$$X_{d-proj} = X \cdot W_d$$

下面的 Python 代码将训练集投影到由前两个主成分定义的超平面上：

```
W2=V.T[:, :2]
X2D=X_centered.dot(W2)
```

好了你已经知道这个东西了！你现在已经知道如何给任何一个数据集降维而又能尽可能的保留原数据集的方差了。

## 使用 Scikit-Learn

Scikit-Learn 的 PCA 类使用 SVD 分解来实现，就像我们之前做的那样。以下代码应用 PCA 将数据集的维度降至二维（请注意，它会自动处理数据的中心化）：

```
from sklearn.decomposition import PCA
pca=PCA(n_components=2)
X2D=pca.fit_transform(X)
```

将 PCA 转化器应用于数据集后，可以使用 `components_` 访问每一个主成分（注意，它返回以 PC 作为水平向量的矩阵，因此，如果我们想要获得第一个主成分则可以写成 `pca.components_.T[:, 0]`）。

## 方差解释率 (Explained Variance Ratio)

另一个非常有用的信息是每个主成分的方差解释率，可通过 `explained_variance_ratio_` 变量获得。它表示位于每个主成分轴上的数据集方差的比例。例如，让我们看一下图 8-2 中表示的三维数据集前两个分量的方差解释率：

```
>>> print(pca.explained_variance_ratio_)
array([0.84248607, 0.14631839])
```

这表明，84.2% 的数据集方差位于第一轴，14.6% 的方差位于第二轴。第三轴的那一比例不到 1.2%，因此可以认为它可能没有包含什么信息。

## 选择正确的维度

通常我们倾向于选择加起来到方差解释率能够达到足够占比（例如 95%）的维度的数量，而不是任意选择要降低到的维度数量。当然，除非您正在为数据可视化而降低维度 -- 在这种情况下，您通常希望将维度降低到 2 或 3。

下面的代码在不降维的情况下进行 PCA，然后计算出保留训练集方差 95% 所需的最小维数：

```
pca=PCA()
pac.fit(X)
cumsum=np.cumsum(pca.explained_variance_ratio_)
d=np.argmax(cumsum>=0.95)+1
```

你可以设置 `n_components = d` 并再次运行 PCA。但是，有一个更好的选择：不指定你想要保留的主成分个数，而是将 `n_components` 设置为 0.0 到 1.0 之间的浮点数，表明您希望保留的方差比率：

```
pca=PCA(n_components=0.95)
X_reduced=pca.fit_transform(X)
```

另一种选择是画出方差解释率关于维数的函数（简单地绘制 `cumsum`；参见图 8-8）。曲线中通常会有一个肘部，方差解释率停止快速增长。您可以将其视为数据集的真正的维度。在这种情况下，您可以看到将维度降低到大约 100 个维度不会失去太多的可解释方差。

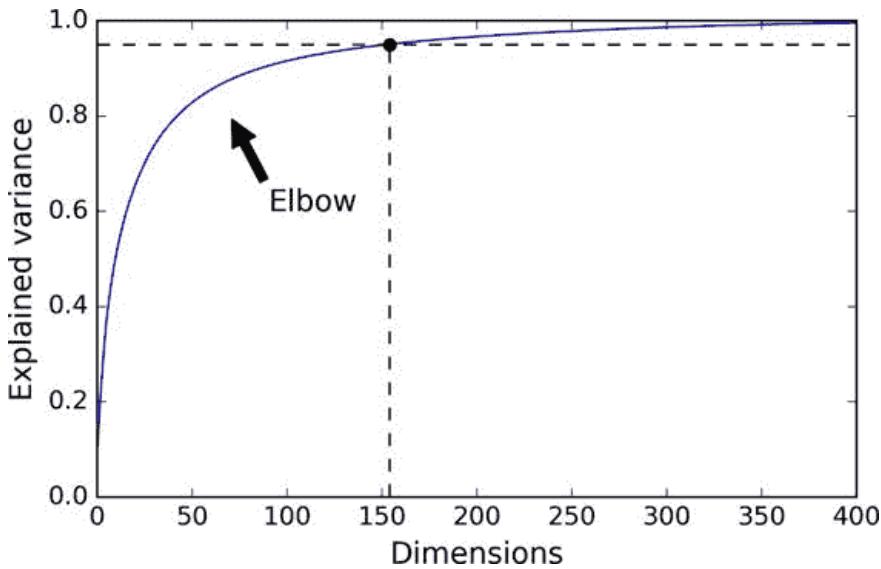


图 8-8 可解释方差关于维数的函数

## PCA 压缩

显然，在降维之后，训练集占用的空间要少得多。例如，尝试将 PCA 应用于 MNIST 数据集，同时保留 95% 的方差。你应该发现每个实例只有 150 多个特征，而不是原来的 784 个特征。因此，尽管大部分方差都保留下，但数据集现在还不到其原始大小的 20%！这是一个合理的压缩比率，您可以看到这可以如何极大地加快分类算法（如 SVM 分类器）的速度。

通过应用 PCA 投影的逆变换，也可以将缩小的数据集解压缩回 784 维。当然这并不会返回给你最原始的数据，因为投影丢失了一些信息（在 5% 的方差内），但它可能非常接近原始数据。原始数据和重构数据之间的均方距离（压缩然后解压缩）被称为重构误差（reconstruction error）。例如，下面的代码将 MNIST 数据集压缩到 154 维，然后使用 `inverse_transform()` 方法将其解压缩回 784 维。图 8-9 显示了原始训练集（左侧）的几位数字在压缩并解压缩后（右侧）的对应数字。您可以看到有轻微的图像质量降低，但数字仍然大部分完好无损。

```
pca=PCA(n_components=154)
X_mnist_reduced=pca.fit_transform(X_mnist)
X_mnist_recovered=pca.inverse_transform(X_mnist_reduced)
```

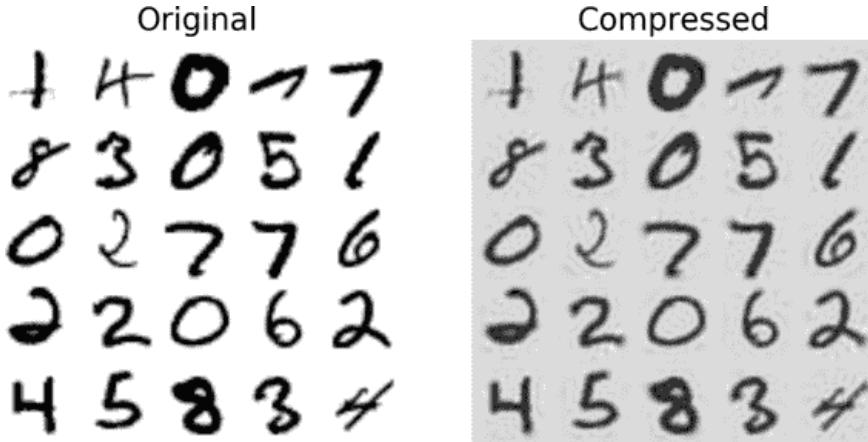


图 8-9 MNIST 保留 95 方差的压缩

逆变换的公式如公式 8-3 所示

公式 8-3 PCA 逆变换，回到原来的数据维度

$$X_{recovered} = X_{d-proj} \cdot W_d^T$$

## 增量 PCA (Incremental PCA)

先前 PCA 实现的一个问题是它需要在内存中处理整个训练集以便 SVD 算法运行。幸运的是，我们已经开发了增量 PCA (IPCA) 算法：您可以将训练集分批，并一次只对一个批量使用 IPCA 算法。这对大型训练集非常有用，并且可以在线应用 PCA（即在新实例到达时即时运行）。

下面的代码将 MNIST 数据集分成 100 个小批量（使用 NumPy 的 `array_split()` 函数），并将它们提供给 Scikit-Learn 的 `IncrementalPCA` 类，以将 MNIST 数据集的维度降低到 154 维（就像以前一样）。请注意，您必须对每个最小批次调用 `partial_fit()` 方法，而不是对整个训练集使用 `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_mnist,n_batches):
    inc_pca.partial_fit(X_batch)
X_mnist_reduced=inc_pca.transform(X_mnist)
```

或者，您可以使用 NumPy 的 `memmap` 类，它允许您操作存储在磁盘上二进制文件中的大型数组，就好像它完全在内存中；该类仅在需要时加载内存中所需的数据。由于增量 PCA 类在任何时间内仅使用数组的一小部分，因此内存使用量仍受到控制。这可以调用通常的 `fit()` 方法，如下面的代码所示：

```
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)
```

## 随机 PCA (Randomized PCA)

Scikit-Learn 提供了另一种执行 PCA 的选择，称为随机 PCA。这是一种随机算法，可以快速找到前  $d$  个主成分的近似值。它的计算复杂度是  $O(m \times d^2) + O(d^3)$ ，而不是  $O(m \times n^2) + O(n^3)$ ，所以当  $d$  远小于  $n$  时，它比之前的算法快得多。

```
rnd_pca=PCA(n_components=154,svd_solver='randomized')
X_reduced=rnd_pca.fit_transform(X_mnist)
```

## 核 PCA (Kernel PCA)

在第 5 章中，我们讨论了核技巧，一种将实例隐式映射到非常高维空间（称为特征空间）的数学技术，让支持向量机可以应用于非线性分类和回归。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，同样的技巧可以应用于 PCA，从而可以执行复杂的非线性投影来降低维度。这就是所谓的核 PCA (kPCA)。它通常能够很好地保留投影后的簇，有时甚至可以展开分布近似于扭曲流形的数据集。

例如，下面的代码使用 Scikit-Learn 的 KernelPCA 类来执行带有 RBF 核的 kPCA（有关 RBF 核和其他核的更多详细信息，请参阅第 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 展示了使用线性核（等同于简单的使用 PCA 类），RBF 核，sigmoid 核 (Logistic) 将瑞士卷降到 2 维。

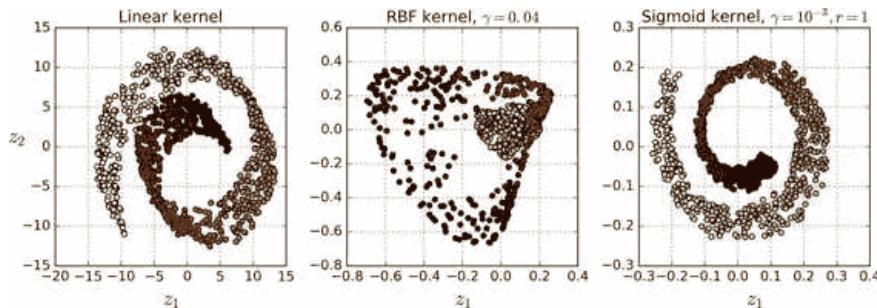


图 8-10 使用不同核的 kPCA 将瑞士卷降到 2 维

## 选择一种核并调整超参数

由于 kPCA 是无监督学习算法，因此没有明显的性能指标可以帮助您选择最佳的核方法和超参数值。但是，降维通常是监督学习任务（例如分类）的准备步骤，因此您可以简单地使用网格搜索来选择可以让该任务达到最佳表现的核方法和超参数。例如，下面的代码创建了一个两步的流水线，首先使用 kPCA 将维度降至二维，然后应用 Logistic 回归进行分类。然后它使用 GridSearchCV 为 kPCA 找到最佳的核和 gamma 值，以便在最后获得最佳的分类准确性：

```

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)

```

你可以通过调用 `best_params_` 变量来查看使模型效果最好的核和超参数：

```

>>> print(grid_search.best_params_)
{'kpca__gamma': 0.04333333333333335, 'kpca__kernel': 'rbf'}

```

另一种完全为非监督的方法，是选择产生最低重建误差的核和超参数。但是，重建并不像线性 PCA 那样容易。这里是原因：图 8-11 显示了原始瑞士卷 3D 数据集（左上角），并且使用 RBF 核应用 kPCA 后生成的二维数据集（右上角）。由于核技巧，这在数学上等同于使用特征映射  $\phi$  将训练集映射到无限维特征空间（右下），然后使用线性 PCA 将变换的训练集投影到 2D。请注意，如果我们可以在缩减空间中对给定实例实现反向线性 PCA 步骤，则重构点将位于特征空间中，而不是位于原始空间中（例如，如图中由  $x$  表示的那样）。由于特征空间是无限维的，我们不能找出重建点，因此我们无法计算真实的重建误差。幸运的是，可以在原始空间中找到一个贴近重建点的点。这被称为重建前图像（reconstruction pre-image）。一旦你有这个前图像，你就可以测量其与原始实例的平方距离。然后，您可以选择最小化重建前图像错误的核和超参数。

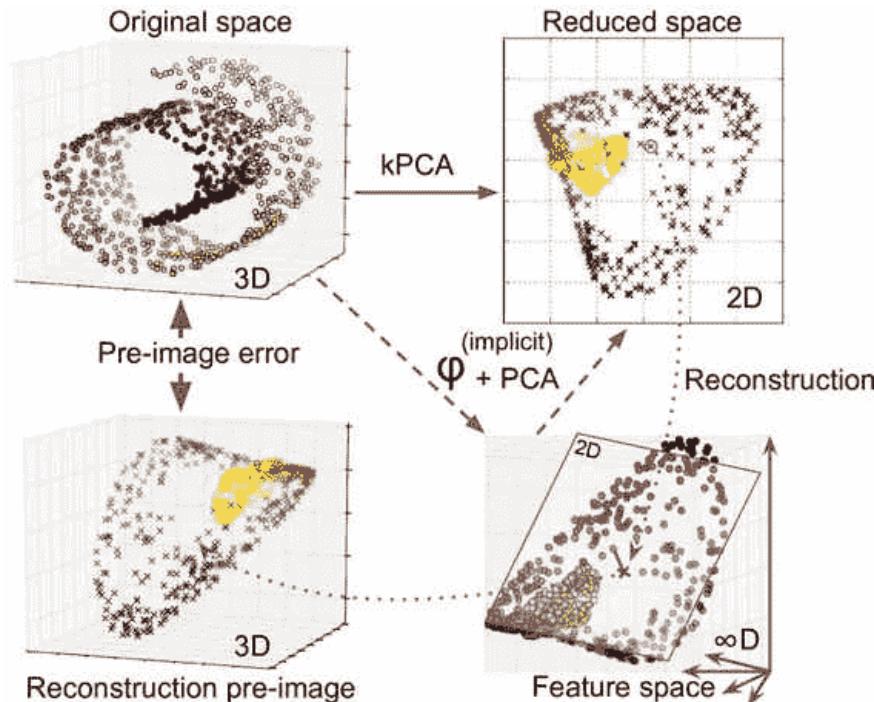


图 8-11 核 PCA 和重建前图像误差

您可能想知道如何进行这种重建。一种解决方案是训练一个监督回归模型，将预计实例作为训练集，并将原始实例作为训练目标。如果您设置了 `fit_inverse_transform = True`，Scikit-Learn 将自动执行此操作，代码如下所

示：

```
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)
```

概述：默认条件下，`fit_inverse_transform = False` 并且 `KernelPCA` 没有 `inverse_transform()` 方法。这种方法仅仅当 `fit_inverse_transform = True` 的情况下才会创建。

你可以计算重建前图像误差：

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(X, X_preimage) 32.786308795766132
```

现在你可以使用交叉验证的方格搜索来寻找可以最小化重建前图像误差的核方法和超参数。

## LLE

局部线性嵌入（Locally Linear Embedding）是另一种非常有效的非线性降维 (NLDR) 方法。这是一种流形学习技术，不依赖于像以前算法那样的投影。简而言之，LLE 首先测量每个训练实例与其最近邻 (c.n.) 之间的线性关系，然后寻找能最好地保留这些局部关系的训练集的低维表示（稍后会详细介绍）。这使得它特别擅长展开扭曲的流形，尤其是在没有太多噪音的情况下。

例如，以下代码使用 Scikit-Learn 的 `LocallyLinearEmbedding` 类来展开瑞士卷。得到的二维数据集如图 8-12 所示。正如您所看到的，瑞士卷被完全展开，实例之间的距离保存得很好。但是，距离不能在较大范围内保留的很好：展开的瑞士卷的左侧被挤压，而右侧的部分被拉长。尽管如此，LLE 在对流形建模方面做得非常好。

```
from sklearn.manifold import LocallyLinearEmbedding
lle=LocallyLinearEmbedding(n_components=2,n_neighbors=10)
X_reduced=lle.fit_transform(X)
```

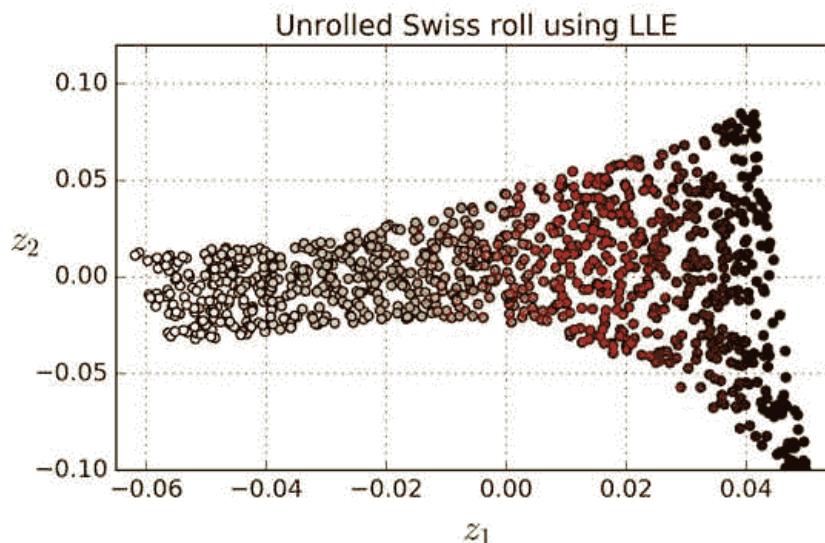


图 8-12 使用 LLE 展开瑞士卷

这是 LLE 的工作原理：首先，对于每个训练实例  $x^{(i)}$ ，该算法识别其最近的  $k$  个邻居（在前面的代码中  $k = 10$  中），然后尝试将  $x^{(i)}$  重构为这些邻居的线性函数。更具体地，找到权重  $w[i, j]$  从而使  $x^{(i)}$  和  $\sum w[i, j] x^{(j)}$ ,  $j = 1 \rightarrow m$  之间的平方距离尽可能的小，假设如果  $x^{(j)}$  不是  $x^{(i)}$  的  $k$  个最近邻时  $w[i, j] = 0$ 。因此，LLE 的第一步是方程 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。第二个约束简单地对每个训练实例  $x^{(i)}$  的权重进行归一化。

公式 8-2 LLE 第一步：对局部关系进行线性建模

$$X_{d\text{-proj}} = X \cdot W_d$$

在这步之后，权重矩阵  $W_{\text{hat}}$ （包含权重  $w_{\text{hat}}[i, j]$  对训练实例的线形关系进行编码。现在第二步是将训练实例投影到一个  $d$  维空间 ( $d < n$ ) 中去，同时尽可能的保留这些局部关系。如果  $z^{(i)}$  是  $x^{(i)}$  在这个  $d$  维空间的图像，那么我们想要  $z^{(i)}$  和  $\sum w_{\text{hat}}[i, j] z^{(j)}$ ,  $j = 1 \rightarrow m$  之间的平方距离尽可能的小。这个想法让我们提出了公式 8-5 中的非限制性优化问题。它看起来与第一步非常相似，但我们要做的不是保持实例固定并找到最佳权重，而是恰相反：保持权重不变，并在低维空间中找到实例图像的最佳位置。请注意， $Z$  是包含所有  $z^{(i)}$  的矩阵。

公式 8-3 LLE 第二步：保持关系的同时进行降维

$$X_{\text{recovered}} = X_{d\text{-proj}} \cdot W_d^T$$

Scikit-Learn 的 LLE 实现具有如下的计算复杂度：查找  $k$  个最近邻为  $O(m \log(m) n \log(k))$ ，优化权重为  $O(m n k^3)$ ，建立低维表示为  $O(d m^2)$ 。不幸的是，最后一项  $m^2$  使得这个算法在处理大数据集的时候表现较差。

## 其他降维方法

还有很多其他的降维方法，Scikit-Learn 支持其中的好几种。这里是其中最流行的：

- 多维缩放 (MDS) 在尝试保持实例之间距离的同时降低了维度（参见图 8-13）
- Isomap 通过将每个实例连接到最近的邻居来创建图形，然后在尝试保持实例之间的测地距离时降低维度。
- t-分布随机邻域嵌入 (t-Distributed Stochastic Neighbor Embedding, t-SNE) 可以用于降低维度，同时试图保持相似的实例临近并将不相似的实例分开。它主要用于可视化，尤其是用于可视化高维空间中的实例（例如，可以将 MNIST 图像降维到 2D 可视化）。
- 线性判别分析 (Linear Discriminant Analysis, LDA) 实际上是一种分类算法，但在训练过程中，它会学习类之间最有区别的轴，然后使用这些轴来定义用于投影数据的超平面。LDA 的好处是投影会尽可能地保持各个类之间距离，所以在运行另一种分类算法（如 SVM 分类器）之前，LDA 是很好的降维技术。

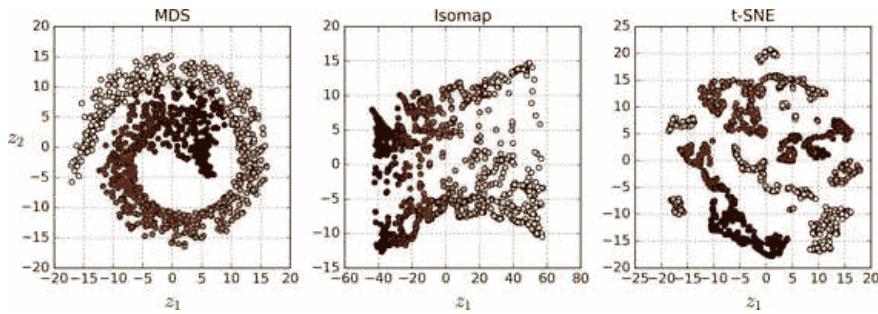


图 8-13 使用不同的技术将瑞士卷降维至 2D

## 练习

1. 减少数据集维度的主要动机是什么？主要缺点是什么？
2. 什么是维度爆炸？
3. 一旦对某数据集降维，我们可能恢复它吗？如果可以，怎样做才能恢复？如果不可以，为什么？
4. PCA 可以用于降低一个高度非线性对数据集吗？
5. 假设你对一个 1000 维的数据集应用 PCA，同时设置方差解释率为 95%，你的最终数据集将会有多少维？
6. 在什么情况下你会使用普通的 PCA，增量 PCA，随机 PCA 和核 PCA？
7. 你该如何评价你的降维算法在你数据集上的表现？
8. 将两个不同的降维算法串联使用有意义吗？
9. 加载 MNIST 数据集（在第 3 章中介绍），并将其分成一个训练集和一个测试集（将前 60,000 个实例用于训练，其余 10,000 个用于测试）。在数据集上训练一个随机森林分类器，并记录了花费多长时间，然后在测试集上评估模型。接下来，使用 PCA 降低数据集的维度，设置方差解释率为 95%。在降维后的数据集上训练一个新的随机森林分类器，并查看需要多长时间。训练速度更快？接下来评估测试集上的分类器：它与以前的分类器比较起来如何？
10. 使用 t-SNE 将 MNIST 数据集缩减到二维，并使用 Matplotlib 绘制结果图。您可以使用 10 种不同颜色的散点图来表示每个图像的目标类别。或者，您可以在每个实例的位置写入彩色数字，甚至可以绘制数字图像本身的降维版本（如果绘制所有数字，则可视化可能会过于混乱，因此您应该绘制随机样本或只在周围没有其他实例被绘制的情况下绘制）。你将会得到一个分隔良好的可视化数字集群。尝试使用其他降维算法，如 PCA，LLE 或 MDS，并比较可视化结果。

练习答案请见附录 A。

## 十、使用 Keras 搭建人工神经网络

译者：[@SeanCheney](#)

鸟类启发人类飞翔，东洋参启发了魔术贴的发明，大自然启发人类实现了无数发明创造。通过研究大脑来制造智能机器，也符合这个逻辑。人工神经网络（ANN）就是沿着这条逻辑诞生的：人工神经网络是受大脑中的生物神经元启发而来的机器学习模型。但是，虽然飞机是受鸟儿启发而来的，飞机却不用挥动翅膀。相似的，人工神经网络和生物神经元网络也是具有不同点的。一些研究者甚至认为，应该彻底摒弃这种生物学类比：例如，用“单元”取代“神经元”，以免人们将创造力局限于生物学系统的合理性上。

人工神经网络是深度学习的核心，它不仅样式多样、功能强大，还具有可伸缩性，这让人工神经网络适宜处理庞大且复杂的机器学习任务，例如对数十亿张图片分类（谷歌图片）、语音识别（苹果 Siri）、向数亿用户每天推荐视频（Youtube）、或者通过学习几百围棋世界冠军（DeepMind 的 AlphaGo）。

本章的第一部分会介绍人工神经网络，从一个简单的 ANN 架构开始，然后过渡到多层感知机（MLP），后者的应用非常广泛（后面的章节会介绍其他的架构）。第二部分会介绍如何使用流行的 Keras API 搭建神经网络，Keras API 是一个设计优美、简单易用的高级 API，可以用来搭建、训练、评估、运行神经网络。Keras 的易用性，并不妨碍它具有强大的实现能力，Keras 足以帮你搭建多种多样的神经网络。事实上，Keras 足以完成大多数的任务啦！要是你需要实现更多的功能，你可以用 Keras 的低级 API（第 12 章介绍）自己写一些组件。

### 从生物神经元到人工神经元

颇让人惊讶的地方是，其实 ANN 已经诞生相当长时间了：神经生理学家 Warren McCulloch 和数学家 Walter Pitts 在 1943 年首次提出了 ANN。在他们里程碑的论文《*A Logical Calculus of Ideas Immanent in Nervous Activity*》中，McCulloch 和 Pitts 介绍一个简单的计算模型，关于生物大脑的神经元是如何通过命题逻辑协同工作的。这是第一个 ANN 架构，后来才出现更多的 ANN 架构。

ANN 的早期成功让人们广泛相信，人类马上就能造出真正的智能机器了。1960 年代，当这个想法落空时，资助神经网络的钱锐减，ANN 进入了寒冬。1980 年代早期，诞生了新的神经网络架构和新的训练方法，连结主义（研究神经网络）复苏，但是进展很慢。到了 1990 年代，出现了一批强大的机器学习方法，比如支持向量机（见第 05 章）。这些新方法的结果更优，也比 ANN 具有更扎实的理论基础，神经网络研究又一次进入寒冬。我们正在经历的是第三次神经网络浪潮。这波浪潮会像前两次那样吗？这次与前两次有所不同，这一次会对我们的生活产生更大的影响，理由如下：

- 我们现在有更多的数据，用于训练神经网络，在大而复杂的问题上，ANN 比其它 ML 技术表现更好；
- 自从 1990 年代，计算能力突飞猛进，现在已经可以在理想的时间内训练出大规模的神经网络了。一部分原因是摩尔定律（在过去 50 年间，集成电路中的组件数每两年就翻了一倍），另外要归功于游戏产业，后者生产出了强大的

GPU 显卡。还有，云平台使得任何人都能使用这些计算能力；

- 训练算法得到了提升。虽然相比 1990 年代，算法变化不大，但这一点改进却产生了非常大的影响；
- 在实践中，人工神经网络的一些理论局限没有那么强。例如，许多人认为人工神经网络训练算法效果一般，因为它们很可能陷入局部最优，但事实证明，这在实践中是相当罕见的（或者如果它发生，它们也通常相当接近全局最优）；
- ANN 已经进入了资助和进步的良性循环。基于 ANN 的惊艳产品常常上头条，从而吸引了越来越多的关注和资金，促进越来越多的进步和更惊艳的产品。

## 生物神经元

在讨论人工神经元之前，先来看看生物神经元（见图 10-1）。这是动物大脑中一种不太常见的细胞，包括：细胞体（含有细胞核和大部分细胞组织），许多貌似树枝的树突，和一条非常长的轴突。轴突的长度可能是细胞体的几倍，也可能是一万倍。在轴突的末梢，轴突分叉成为终树突，终树突的末梢是突触，突触连接着其它神经元的树突或细胞体。

生物神经元会产生被称为“动作电位”（或称为信号）的短促电脉冲，信号沿轴突传递，使突触释放出被称为神经递质的化学信号。当神经元在几毫秒内接收了足够量的神经递质，这个神经元也会发送电脉冲（事实上，要取决于神经递质，一些神经递质会禁止发送电脉冲）。

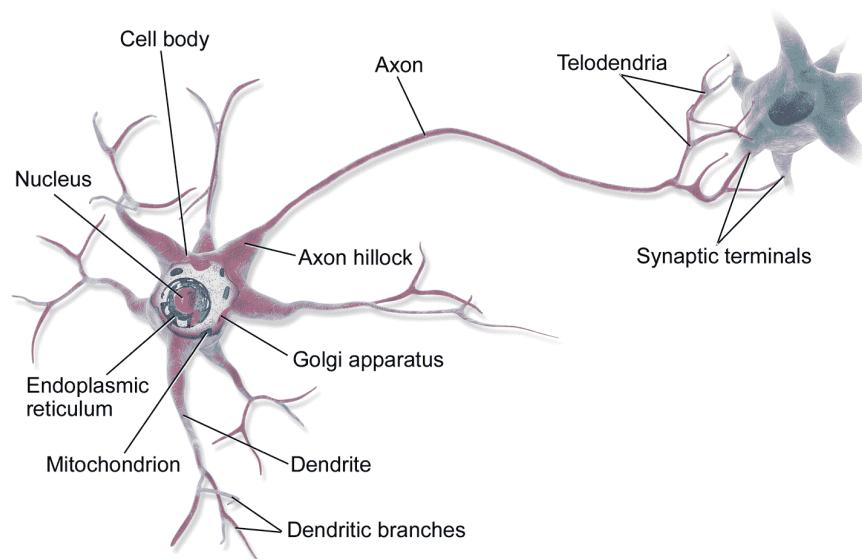


图 10-1 生物神经元

独立的生物神经元就是这样工作的，但因为神经元是处于数十亿神经元的网络中的，每个神经元都连着几千个神经元。简单神经元的网络可以完成高度复杂的计算，就好像蚂蚁齐心协力就能建成复杂的蚁冢一样。生物神经网络（BNN）如今仍是活跃的研究领域，人们通过绘制出了部分大脑的结构，发现神经元分布在连续的皮层上，尤其是在大脑皮质上（大脑外层），见图 10-2。

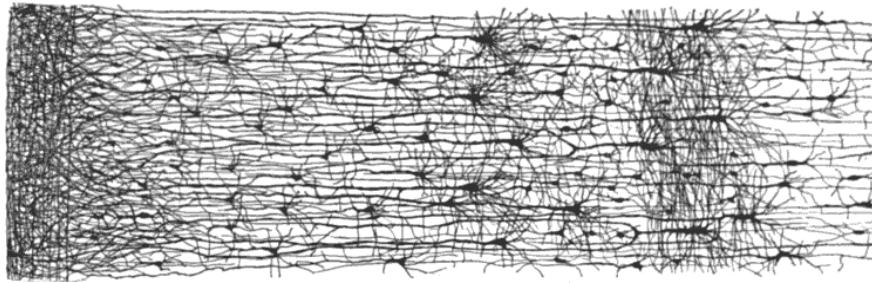


图 10-2 人类大脑皮质的多层神经元网络

## 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，它后来演化成了人工神经元：一个或多个二元（开或关）输入，一个二元输出。当达到一定的输入量时，神经元就会产生输出。在论文中，两位作者证明就算用如此简单的模型，就可以搭建一个可以完成任何逻辑命题计算的神经网络。为了展示网络是如何运行的，我们自己亲手搭建一些不同逻辑计算的 ANN（见图 10-3），假设有两个活跃的输入时，神经元就被激活。

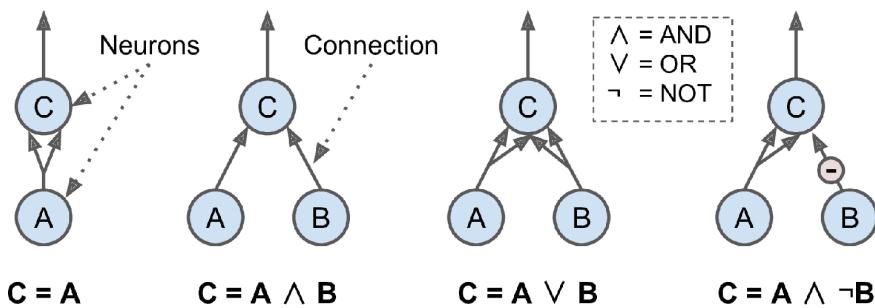


图 10-3 不同逻辑计算的 ANN

这些网络的逻辑计算如下：

- 左边第一个网络是确认函数：如果神经元 A 被激活，那么神经元 c 也被激活（因为它接收来自神经元 A 的两个输入信号），但是如果神经元 A 关闭，那么神经元 c 也关闭。
- 第二个网络执行逻辑 AND：神经元 c 只有在激活神经元 A 和 B（单个输入信号不足以激活神经元 c）时才被激活。
- 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），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（生物神经元是这样的情景），那么第四个网络计算一个稍微复杂的逻辑命题：如果神经元 B 关闭，只有当神经元 A 是激活的，神经元 c 才被激活。如果神经元 A 始终是激活的，那么你得到一个逻辑 NOT：神经元 c 在神经元 B 关闭时是激活的，反之亦然。

你可以很容易地想到，如何将这些网络组合起来计算复杂的逻辑表达式（参见本章末尾的练习）。

## 感知机

感知器是最简单的人工神经网络结构之一，由 Frank Rosenblatt 发明于 1957 年。它基于一种稍微不同的人工神经元（见图 10-4），阈值逻辑单元（TLU），或称为线性阈值单元（LTU）：输入和输出是数字（而不是二元开/关值），并且每个输入连接都一个权重。TLU 计算其输入的加权和 ( $z = w[1]x[1] + w[2]x[2] + \dots + w[n]x[n] = x^T \cdot w$ )，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  $z = x^T \cdot w$ 。

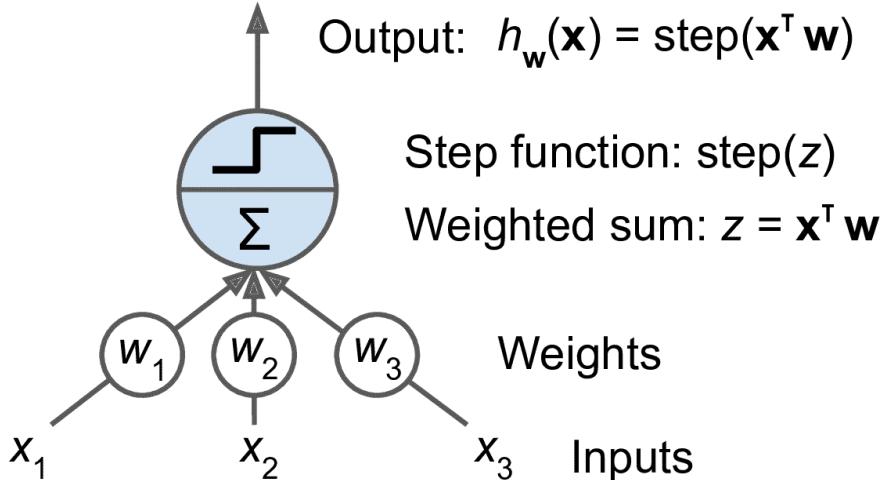


图 10-4 阈值逻辑单元：人工神经元做权重求和，然后对和做阶跃函数

感知机最常用的阶跃函数是单位阶跃函数（Heaviside step function），见公式 10-1。有时候也使用符号函数  $\text{sgn}$ 。

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

公式 10-1 感知机常用的阶跃函数，阈值为 0

单一 TLU 可用于简单的线性二元分类。它计算输入的线性组合，如果结果超过阈值，它输出正类或者输出负类（就像逻辑回归分类或线性 SVM 分类）。例如，你可以使用单一 TLU，基于花瓣长度和宽度分类鳶尾花（也可添加额外的偏置特征  $x[0] = 1$ ，就像我们在前面章节所做的那样）。训练 TLU 意味着去寻找合适的  $w[0]$ ,  $w[1]$  和  $w[2]$  值（训练算法稍后提到）。

感知器只由一层 TLU 组成，每个 TLU 连接到所有输入。当一层的神经元连接着前一层的每个神经元时，该层被称为全连接层，或紧密层。感知机的输入来自输入神经元，输入神经元只输出从输入层接收的任何输入。所有的输入神经元位于输入层。此外，通常再添加一个偏置特征 ( $x[0] = 1$ )：这种偏置特性通常用一种称为偏置神经元的特殊类型的神经元来表示，它总是输出 1。图 10-5 展示了一个具有两个输入和三个输出的感知机，它可以将实例同时分成为三个不同的二元类，这使它成为一个多输出分类器。。

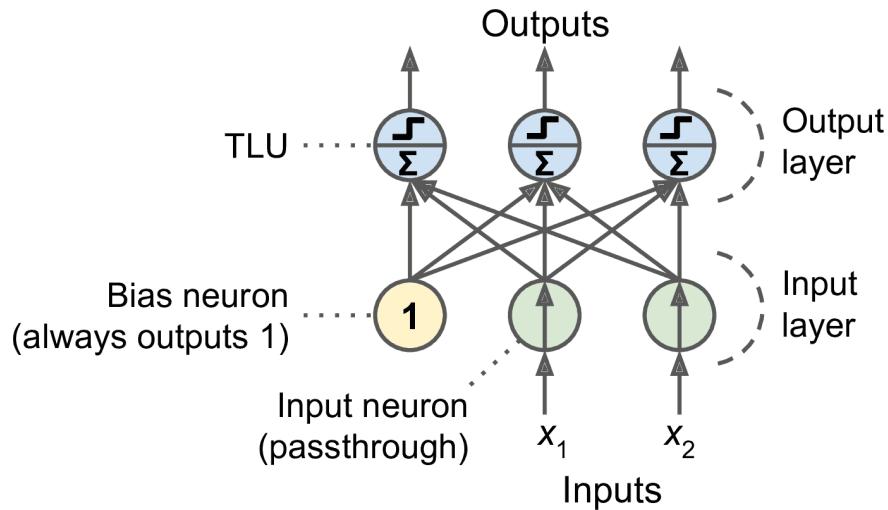


图 10-5 一个具有两个输入神经元、一个偏置神经元和三个输出神经元的感知机架构

借助线性代数，利用公式 10-2 可以方便地同时算出几个实例的一层神经网络的输出。

$$h_{\mathbf{W}, \mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

公式 10-2 计算一个全连接层的输出

在这个公式中，

- $\mathbf{x}$  表示输入特征矩阵，每行是一个实例，每列是一个特征；
- 权重矩阵  $\mathbf{w}$  包含所有的连接权重，除了偏置神经元。每有一个输入神经元权重矩阵就有一行，神经层每有一个神经元权重矩阵就有一列；
- 偏置向量  $\mathbf{b}$  含有所有偏置神经元和人工神经元的连接权重。每一个人工神经元就对应一个偏置项；
- 函数  $\phi$  被称为激活函数，当人工神经网络是 TLU 时，激活函数是阶跃函数（后面会讨论更多的激活函数）。

那么感知器是如何训练的呢？Frank Rosenblatt 提出的感知器训练算法在很大程度上受到 Hebb 规则的启发。在 1949 出版的《行为组织》一书中，Donald Hebb 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变得更强。这个想法后来被 Siegrid Löwel 总结为一经典短语：“一起燃烧的细胞，汇合在一起。”这个规则后来被称为 Hebb 规则（或 Hebbian learning）。使用这个规则的变体来训练感知器，该规则考虑了网络所犯的误差。更具体地，感知器一次被馈送一个训练实例，对于每个实例，它进行预测。对于每一个产生错误预测的输出神经元，修正输入的连接权重，以获得正确的预测。公式 10-3 展示了 Hebb 规则。

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta (y_j - \hat{y}_j) x_i$$

### 公式 10-3 感知机的学习规则（权重更新）

在这个公式中：

- 其中  $w[i, j]$  是第  $i$  个输入神经元与第  $j$  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；
- $y_{\text{hat}}[j]$  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；
- $\eta$  是学习率。

每个输出神经元的决策边界是线性的，因此感知器不能学习复杂的模式（比如 Logistic 回归分类器）。然而，如果训练实例是线性可分的，Rosenblatt 证明该算法将收敛到一个解。这被称为感知器收敛定理。

Scikit-Learn 提供了一个 `Perceptron` 类，它实现了一个单 TLU 网络。它可以实现大部分功能，例如用于鸢尾花数据集（第 4 章中介绍过）：

```
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]])
```

你可能注意到，感知器学习算法和随机梯度下降很像。事实上，sklearn 的 `Perceptron` 类相当于使用具有以下超参数的

`SGDClassifier`： `loss="perceptron"`， `learning_rate="constant"`， `eta0=1`（学习率）， `penalty=None`（无正则化）。

与逻辑回归分类器相反，感知机不输出类概率，而是基于硬阈值进行预测。这是逻辑回归优于感知机的一点。

在 1969 年题为“感知机”的专著中，Marvin Minsky 和 Seymour Papert 强调了感知器的许多严重缺陷，特别是它们不能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）。当然，其他的线性分类模型（如 Logistic 回归分类器）也都实现不了，但研究人员期望从感知器中得到更多，他们的失望是很大的，导致许多人彻底放弃了神经网络，而是转向高层次的问题，如逻辑、问题解决和搜索。

然而，事实证明，感知机的一些局限性可以通过堆叠多个感知机消除。由此产生的人工神经网络被称为多层感知机（MLP）。特别地，MLP 可以解决 XOR 问题，你可以通过计算图 10-6 右侧所示的 MLP 的输出来验证输入的每一个组合：输入  $(0, 0)$  或  $(1, 1)$  网络输出 0，输入  $(0, 1)$  或  $(1, 0)$  它输出 1。除了四个连接的权重不是 1，其它连接都是 1。

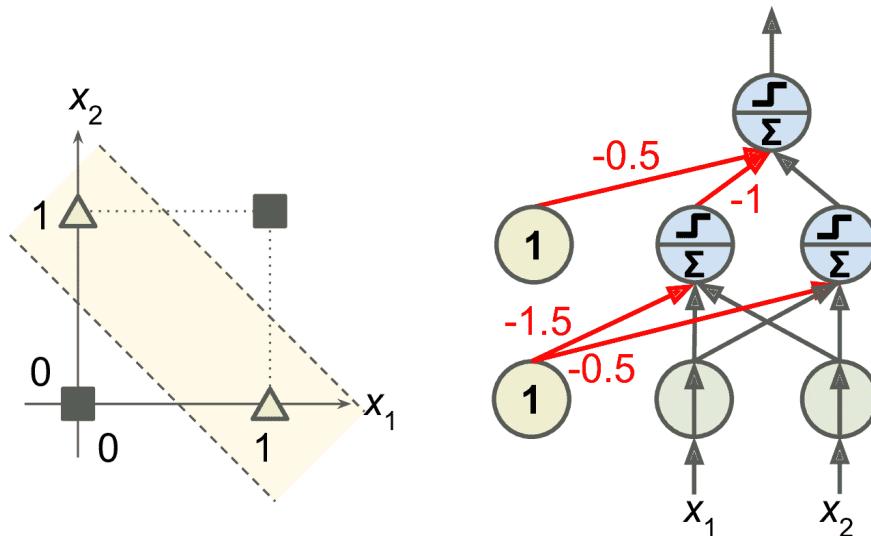


图 10-6 XOR 分类问题和 MLP

## 多层感知机与反向传播

MLP 由一个输入层、一个或多个称为隐藏层的 TLU 组成，一个 TLU 层称为输出层（见图 10-7）。靠近输入层的层，通常被称为浅层，靠近输出层的层通常被称为上层。除了输出层，每一层都有一个偏置神经元，并且全连接到下一层。

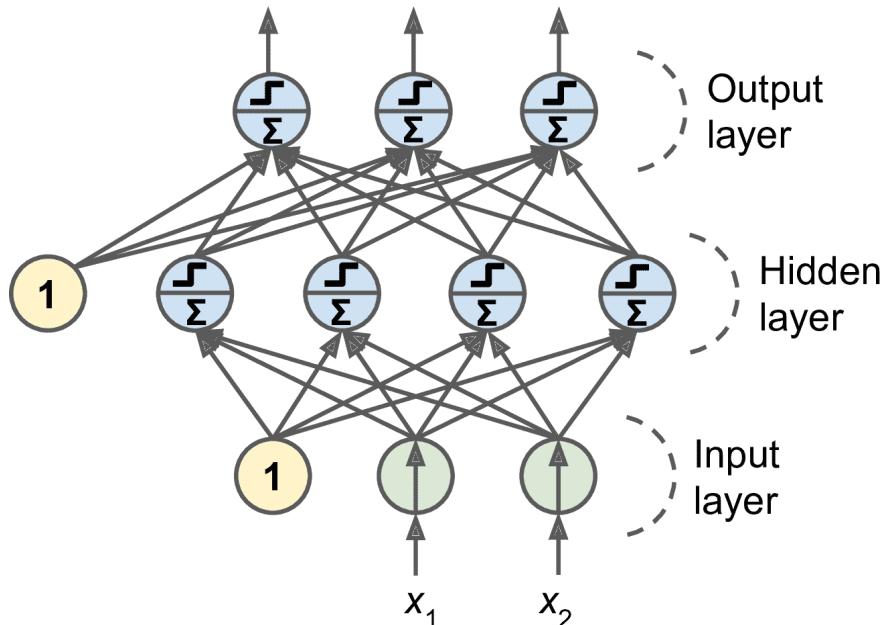


图 10-7 多层感知器

注意：信号是从输入到输出单向流动的，因此这种架构被称为前馈神经网络 (FNN)。

当人工神经网络有多个隐含层时，称为深度神经网络 (DNN)。深度学习研究的是 DNN 和深层计算模型。但是大多数人用深度学习泛化代替神经网络，即便网络很浅时。

多年来，研究人员努力寻找一种训练 MLP 的方法，但没有成功。但在 1986, David Rumelhart、Geoffrey Hinton、Ronald Williams 发表了一篇突破性的论文，提出了至今仍在使用的反向传播训练算法。总而言之，反向传播算法是使用了高效梯度计算的梯度下降算法（见第 4 章）：只需要两次网络传播（一次向前，一次向后），就可以算出网络误差的、和每个独立模型参数相关的梯度。换句话说，反向传播算法为了减小误差，可以算出每个连接权重和每个偏置项的调整量。当得到梯度之后，就做一次常规的梯度下降，不断重复这个过程，直到网络得到收敛解。

笔记：自动计算梯度被称为自动微分。有多种自动微分的方法，各有优缺点。反向传播使用的是反向模式自微分。这种方法快而准，当函数有多个变量（连接权重）和多个输出（损失函数）要微分时也能应对。附录 D 介绍了自微分。

对 BP 做详细分解：

- 每次处理一个微批次（假如每个批次包含 32 个实例），用训练集多次训练 BP，每次被称为一个周期（epoch）；
- 每个微批次先进入输入层，输入层再将其发到第一个隐藏层。计算得到该层所有神经元的（微批次的每个实例的）输出。输出接着传到下一层，直到得到输出层的输出。这个过程就是前向传播：就像做预测一样，只是保存了每个中间结果，中间结果要用于反向传播；
- 然后计算输出误差（使用损失函数比较目标值和实际输出值，然后返回误差）；
- 接着，计算每个输出连接对误差的贡献量。这是通过链式法则（就是对多个变量做微分的方法）实现的；
- 然后还是使用链式法则，计算最后一个隐藏层的每个连接对误差的贡献，这个过程不断向后传播，直到到达输入层。
- 最后，BP 算法做一次梯度下降步骤，用刚刚计算的误差梯度调整所有连接权重。

BP 算法十分重要，再归纳一下：对每个训练实例，BP 算法先做一次预测（前向传播），然后计算误差，然后反向通过每一层以测量误差贡献量（反向传播），最后调整所有连接权重以降低误差（梯度下降）。（译者注：我也总结下吧，每次训练都先是要设置周期数，每个周期其实做的就是三件事，向前传一次，向后传一次，然后调整参数，接着再进行下一周期。）

警告：随机初始化隐藏层的连接权重是很重要的。假如所有的权重和偏置都初始化为 0，则在给定一层的所有神经元都是一样的，BP 算法对这些神经元的调整也会是一样的。换句话，就算每层有几百个神经元，模型的整体表现就像每层只有一个神经元一样，模型会显得笨笨的。如果权重是随机初始化的，就可以打破对称性，训练出不同的神经元。

为了使 BP 算法正常工作，作者对 MLP 的架构做了一个关键调整：用 Logistic 函数（sigmoid）代替阶跃函数， $\sigma(z) = 1 / (1 + \exp(-z))$ 。这是必要的，因为阶跃函数只包含平坦的段，因此没有梯度（梯度下降不能在平面上移动），而 Logistic 函数处处都有一个定义良好的非零导数，允许梯度下降在每步上取得一些进展。反向传播算法也可以与其他激活函数一起使用，下面就是两个流行的激活函数：

- 双曲正切函数： $\tanh(z) = 2\sigma(2z) - 1$

类似 Logistic 函数，它是 S 形、连续可微的，但是它的输出值范围从 -1 到 1（不是 Logistic 函数的 0 到 1），这往往使每层的输出在训练开始时或多或少都变得以 0 为中心，这常常有助于加快收敛速度。

- ReLU 函数： $\text{ReLU}(z) = \max(0, z)$

ReLU 函数是连续的，但是在  $z=0$  时不可微（斜率突然改变，导致梯度下降在 0 点左右跳跃），ReLU 的变体是当  $z < 0$  时， $z=0$ 。但在实践中，ReLU 效果很好，并且具有计算快速的优点，于是成为了默认激活函数。最重要的是，它没有最大输出值，这有助于减少梯度下降期间的一些问题（第 11 章再介绍）。

这些流行的激活函数及其变体如图 10-8 所示。但是，究竟为什么需要激活函数呢？如果将几个线性变化链式组合起来，得到的还是线性变换。比如，对于

$f(x) = 2x + 3$  和  $g(x) = 5x - 1$ ，两者组合起来仍是线性变换： $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$ 。如果层之间不具有非线性，则深层网络和单层网络其实是等同的，这样就不能解决复杂问题。相反的，足够深且有非线性激活函数的 DNN，在理论上可以近似于任意连续函数。

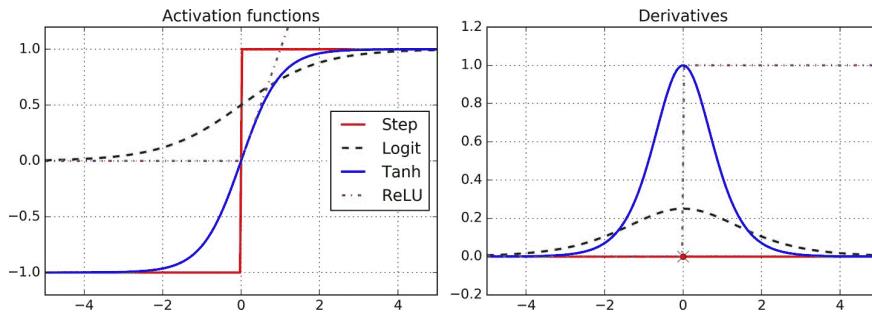


图 10-8 激活函数及其变体

知道了神经网络的起源、架构、计算方法、BP 算法，接下来看应用。

## 回归 MLP

首先，MLP 可以用来回归任务。如果想要预测一个单值（例如根据许多特征预测房价），就只需要一个输出神经元，它的输出值就是预测值。对于多变量回归（即一次预测多个值），则每一维度都要有一个神经元。例如，想要定位一张图片的中心，就要预测 2D 坐标，因此需要两个输出神经元。如果再给对象加个边框，还需要两个值：对象的宽度和高度。

通常，当用 MLP 做回归时，输出神经元不需要任何激活函数。如果要让输出是正值，则可在输出值使用 ReLU 激活函数。另外，还可以使用 softplus 激活函数，这是 ReLU 的一个平滑化变体： $\text{softplus}(z) = \log(1 + \exp(z))$ 。 $z$  是负值时，softplus 接近 0， $z$  是正值时，softplus 接近  $z$ 。最后，如果想让输出落入一定范围内，则可以使用调整过的 Logistic 或双曲正切函数：Logistic 函数用于 0 到 1，双曲正切函数用于 -1 到 1。

训练中的损失函数一般是均方误差，但如果训练集有许多异常值，则可以使用平均绝对误差。另外，也可以使用 Huber 损失函数，它是前两者的组合。

提示：当误差小于阈值 6 时（一般为 1），Huber 损失函数是二次的；误差大于阈值时，Huber 损失函数是线性的。相比均方误差，线性部分可以让 Huber 对异常值不那么敏感，二次部分可以让收敛更快，也比均绝对误差更精确。

表 10-1 总结了回归 MLP 的典型架构。

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., $28 \times 28 = 784$ for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

表 10-1 回归 MLP 的典型架构

## 分类 MLP

MLP 也可用于分类，对于二元分类问题，只需要一个使用 Logistic 激活的输出神经元：输出是一个 0 和 1 之间的值，作为正类的估计概率。

MLP 也可以处理多标签二元分类（见第 3 章）。例如，邮件分类系统可以预测一封邮件是垃圾邮件，还是正常邮件，同时预测是紧急，还是非紧急邮件。这时，就需要两个输出神经元，两个都是用 Logistic 函数：第一个输出垃圾邮件的概率，第二个输出紧急的概率。更为一般的讲，需要为每个正类配一个输出神经元。多个输出概率的和不一定非要等于 1。这样模型就可以输出各种标签的组合：非紧急非垃圾邮件、紧急非垃圾邮件、非紧急垃圾邮件、紧急垃圾邮件。

如果每个实例只能属于一个类，但可能是三个或多个类中的一个（比如对于数字图片分类，可以是类 0 到类 9），则每一类都要有一个输出神经元，整个输出层（见图 10-9）要使用 softmax 激活函数。softmax 函数可以保证，每个估计概率位于 0 和 1 之间，并且各个值相加等于 1。这被称为多类分类。

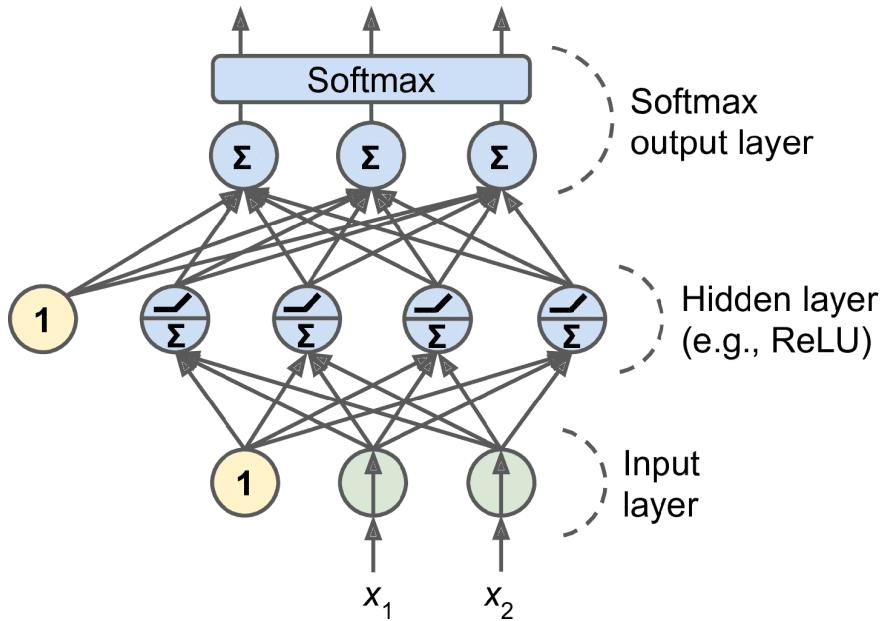


图 10-9 一个用于分类的 MLP（包括 ReLU 和 softmax）

根据损失函数，因为要预测概率分布，交叉商损失函数（也称为对数损失，见第 4 章）是不错的选择。

表 10-2 概括了分类 MLP 的典型架构。

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

表 10-2 分类 MLP 的典型架构

提示：看下面的内容前，建议看看本章末尾的习题 1。利用 TensorFlow Playground 可可视化各样的神经网络架构，可以更深入的理解 MLP 和超参数（层数、神经元数、激活函数）的作用。

## 用 Keras 实现 MLP

Keras 是一个深度学习高级 API，可以用它轻松地搭建、训练、评估和运行各种神经网络。Keras 的文档见[这里](#)。Keras 参考实现是 François Chollet 开发的，于 2015 年 3 月开源。得益于 Keras 简单易用灵活优美，迅速流行开来。为了进行神经网络计算，必须要有计算后端的支持。目前可选三个流行库：TensorFlow、CNTK 和 Theano。为避免误会，将 GitHub 上的 Keras 参考实现称为多后端 Keras。

自从 2016 年底，出现了 Keras 的其它实现。现在已经可以在 Apache MXNet、苹果 Core ML、JavaScript 或 TypeScript（浏览器）、PlaidML（各种 GPU，不限于 Nvidia）上运行 Keras。另外，TensorFlow 也捆绑了自身的 Keras 实现——`tf.keras`，它只支持 TensorFlow 作为后端，但提供了更多使用功能（见图 10-10）：例如，`tf.keras` 支持 TensorFlow 的 Data API，加载数据更轻松，预处理数据更高效。因此，本书使用的是 `tf.keras`。本章的代码不局限于 TensorFlow，只需要一些修改，比如修改引入，也可以在其它 Keras 实现上运行。

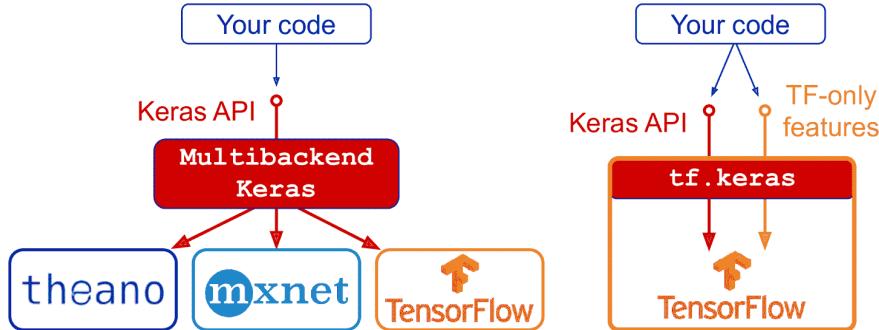


图 10-10 Keras API 的两个实现：左边是多后端 Keras，右边是 `tf.keras`

排在 Keras 和 TensorFlow 之后最流行的深度学习库，是 Facebook 的 PyTorch。PyTorch 的 API 与 Keras 很像，所以掌握了 Keras，切换到 PyTorch 也不难。得益于易用性和详实的文档（TensorFlow 1 的文档比较一般），PyTorch 在 2018 年广泛流行开来。但是，TensorFlow 2 和 PyTorch 一样简单易用，因为 TensorFlow 使用了 Keras 作为它的高级 API，并简化清理了 TensorFlow 的其它 API。TensorFlow 的文档也改观了，容易检索多了。相似的，PyTorch 的缺点（可移植性差，没有计算图分析）在 PyTorch 1.0 版本中也得到了优化。良性竞争可以使所有人获益。（作者这段讲的真好！）

## 安装 TensorFlow 2

假设已经在第 2 章中安装了 Jupyter 和 Scikit-Learn，使用 PIP 安装 TensorFlow。如果使用了 VirtualEnv，先要激活虚拟环境：

```
$ 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
```

然后安装 TensorFlow 2（如果没有使用虚拟环境，需要管理员权限，或加上选项 `--user`）：

```
$ python3 -m pip install --upgrade tensorflow
```

笔记：要使用 GPU 的话，在动笔写书的此刻，需要安装 `tensorflow-gpu`，而不是 `tensorflow`。但是 TensorFlow 团队正在开发一个既支持 CPU 也支持 GPU 的独立的库。要支持 GPU 的话，可能还要安装更多的库，参考[这里](#)。第 19 章会深入介绍 GPU。

要测试安装是否成功，可以在 Python 终端或 Jupyter 笔记本中引入 TensorFlow 和 `tf.keras`，然后打印其版本号：

```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> tf.__version__
'2.0.0'
>>> keras.__version__
'2.2.4-tf'
```

第二个版本号的末尾带有 `-tf`，表明是 `tf.keras` 实现的 Keras API，还有一些 TensorFlow 的专有功能。

## 使用顺序 API 创建图片分类器

首先加载数据集。这章用的数据集是 Fashion MNIST，它是 MNIST 一个替代品，格式与 MNIST 完全相同（70000 张灰度图，每张的像素是  $28 \times 28$ ，共有 10 类），图的内容是流行物品，而不是数字，每类中的图片更丰富，识图的挑战性比 MNIST 高得多。例如，线性模型可以在 MNIST 上达到 92% 的准确率，但在 Fashion MNIST 上只有 83% 的准确率。

### 使用 Keras 加载数据集

Keras 提供一些实用的函数用来获取和加载常见的数据集，包括 MNIST、Fashion MNIST 和第 2 章用过的加州房产数据集。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 加载 MNIST 或 Fashion MNIST 时，和 Scikit-Learn 加载数据的一个重要区别是，每张图片是  $28 \times 28$  的数组，而不是大小是 784 的 1D 数组。另外像素的强度是用整数（0 到 255）表示的，而不是浮点数（0.0 到 255.0）。看下训练集的形状和类型：

```
>>> X_train_full.shape
(60000, 28, 28)
>>> X_train_full.dtype
dtype('uint8')
```

该数据集已经分成了训练集和测试集，但没有验证集。所以要建一个验证集，另外，因为要用梯度下降训练神经网络，必须要对输入特征进行缩放。简单起见，通过除以 255.0 将强度范围变为 0-1：

```
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，表明图片是手写的数字 5。但对于 Fashion MNIST，需要分类名的列表：

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

例如，训练集的第一张图片表示外套：

```
>>> class_names[y_train[0]]
'Coat'
```

图 10-11 展示了 Fashion MNIST 数据集的一些样本。



图 10-11 Fashion MNIST 数据集的一些样本

## 用顺序 API 创建模型

搭建一个拥有两个隐含层的分类 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"))
```

逐行看下代码：

- 第一行代码创建了一个顺序模型，这是 Keras 最简单的模型，是由单层神经元顺序连起来的，被称为顺序 API；
- 接下来创建了第一层，这是一个 `Flatten` 层，它的作用是将每个输入图片转变为 1D 数组：如果输入数据是 `x`，该层则计算 `x.reshape(-1, 1)`。该层没有任何参数，只是做一些简单预处理。因为是模型的第一层，必须要指明 `input_shape`，`input_shape` 不包括批次大小，只是实例的形状。另外，第一层也可以是 `keras.layers.InputLayer`，设置 `input_shape=[28,28]`；
- 然后，添加了一个有 300 个神经元的紧密层，激活函数是 ReLU。每个紧密层只负责自身的权重矩阵，权重矩阵是神经元与输入的所有连接权重。紧密层还要负责偏置项（每个神经元都有一个偏置项）向量。当紧密层收到输入数据时，就利用公式 10-2 进行计算；
- 接着再添加第二个紧密层，激活函数仍然是 ReLU；
- 最后，加上一个拥有 10 个神经元的输出层（每有一个类就要有一个神经元），激活函数是 softmax（保证输出的概率和等于 1，因为就只有这是个类，具有排他性）。

**提示：**设置 `activation="relu"`，等同于 `activation=keras.activations.relu`。`keras.activations` 包中还有其它激活函数，完整列表见[这里](#)。

除了一层一层加层，也可以传递一个层组成的列表：

```
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 KERAS.IO 的代码实例

keras.io 上的代码也可以用于 `tf.keras`，但是需要修改引入。例如，对于下面的代码：

```
from keras.layers import Dense
output_layer = Dense(10)
```

需要改成：

```
from tensorflow.keras.layers import Dense
output_layer = Dense(10)
```

或使用完整路径：

```
from tensorflow import keras
output_layer = keras.layers.Dense(10)
```

这么写就是麻烦点，但是我在本书中是采用的这种方法，因为不仅可以容易看出使用的是哪个包，还可以避免搞混标准类和自定义类。在生产环境中，我倾向于使用前种方式。还有人喜欢这样引入，`tensorflow.keras import layers`，使用 `layers.Dense(10)`。

模型的 `summary()` 方法可以展示所有层，包括每个层的名字（名字是自动生成的，除非建层时指定名字），输出的形状（`None` 代表批次大小可以是任意值），和参数的数量。最后会输出所有参数的数量，包括可训练和不可训练参数。这章只有可训练参数（第 11 章可以看到不可训练参数的例子）：

```
>>> 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
```

紧密层通常有许多参数。比如，第一个隐含层有  $784 \times 300$  个连接权重，再加上 300 个偏置项，总共有 235500 个参数。这么多参数可以让模型具有足够的灵活度以拟合训练数据，但也意味着可能有过拟合的风险，特别是当训练数据不足时。后面再讨论这个问题。

使用属性，获取神经层很容易，可以通过索引或名称获取对应的层：

```
>>> 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
```

可以用 `get_weights()` 和 `set_weights()` 方法，获取神经层的所有参数。对于紧密层，参数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.], dtype=float32)
>>> biases.shape
(300,)
```

紧密层是随机初始化连接权重的（为了避免对称性），偏置项则是 0。如果想使用不同的初始化方法，可以在创建层时设置 `kernel_initializer`（核是连接矩阵的另一个名字）或 `bias_initializer`。第 11 章会进一步讨论初始化器，初始化器的完整列表见[这里](#)。

**笔记：**权重矩阵的形状取决于输入的数量。这就是为什么要在创建 `Sequential` 模型的第一层时指定 `input_shape`。但是，如果不指定形状也没关系：Keras 会在真正搭建模型前一直等待，直到弄清输入的形状（输入真实数据时，或调用 `build()` 方法时）。在搭建模型之前，神经层是没有权重的，也干不了什么事（比如打印模型概要或保存模型）。所以如果在创建模型时知道输入的形状，最好就设置好。

## 编译模型

创建好模型之后，必须调用 `compile()` 方法，设置损失函数和优化器。另外，还可以指定训练和评估过程中要计算的额外指标的列表：

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

**笔记：**使用 `loss="sparse_categorical_crossentropy"` 等同于 `loss=keras.losses.sparse_categorical_crossentropy`。相思的，`optimizer="sgd"` 等同于 `optimizer=keras.optimizers.SGD()`，`metrics=["accuracy"]` 等同于 `metrics=[keras.metrics.sparse_categorical_accuracy]`。后面还会使用其他的损失函数、优化器和指标，它们的完整列表见[这里](#)、[这里](#)、和[这里](#)。

解释下这段代码。首先，因为使用的是稀疏标签（每个实例只有一个目标类的索引，在这个例子中，目标类索引是 0 到 9），且就是这十个类，没有其它的，所以使用的是 `"sparse_categorical_crossentropy"` 损失函数。如果每个实例的每个类都有一个目标概率（比如独热向量，`[0., 0., 0., 1., 0., 0., 0., 0., 0.]`，来

表示类 3) , 则就要使用 "categorical\_crossentropy" 损失函数。如果是做二元分类 (有一个或多个二元标签) , 输出层就得使用 "sigmoid" 激活函数, 损失函数则变为 "binary\_crossentropy" 。

**提示** : 如果要将稀疏标签转变为独热向量标签, 可以使用函数 `keras.utils.to_categorical()` 。还可以使用函数 `np.argmax()` , `axis=1` 。

对于优化器, "sgd" 表示使用随机梯度下降训练模型。换句话说, Keras 会进行反向传播算法。第 11 章会讨论更高效的优化器 (可以提升梯度下降部分, 改善不了自动微分部分) 。

**笔记** : 使用 SGD 时, 调整学习率很重要, 必须要手动设置好, `optimizer=keras.optimizers.SGD(lr=???)`。`optimizer="sgd"` 不同, 它的学习率默认为 `lr=0.01` 。

最后, 因为是个分类器, 最好在训练和评估时测量 "accuracy" 。

## 训练和评估模型

可以训练模型了。只需调用 `fit()` 方法 :

```
>>> 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: 0.836
Epoch 2/30
55000/55000 [=====] - 2s 45us/sample - loss: 0.4840 - accuracy: 0.8327
                                         - val_loss: 0.4456 - val_accuracy: 0.848
[...]
Epoch 30/30
55000/55000 [=====] - 3s 53us/sample - loss: 0.2252 - accuracy: 0.9192
                                         - val_loss: 0.2999 - val_accuracy: 0.892
```

这里, 向 `fit()` 方法传递了输入特征 (`X_train`) 和目标类 (`y_train`) , 还要训练的周期数 (不设置的话, 默认的周期数是 1, 肯定是不能收敛到一个好的解的) 。另外还传递了验证集 (它是可选的) 。Keras 会在每个周期结束后, 测量损失和指标, 这样就可以监测模型的表现。如果模型在训练集上的表现优于在验证集上的表现, 可能模型在训练集上就过拟合了 (或者就是存在 bug, 比如训练集和验证集的数据不匹配) 。

仅需如此, 神经网络就训练好了。训练中的每个周期, Keras 会展示到目前为止一共处理了多少个实例 (还带有进度条), 每个样本的平均训练时间, 以及在训练集和验证集上的损失和准确率 (和其它指标) 。可以看到, 损失是一直下降的, 这是一个好现象。经过 30 个周期, 验证集的准确率达到了 89.26%, 与在训练集上的准确率差不多, 所以没有过拟合。

**提示** : 除了通过参数 `validation_data` 传递验证集, 也可以通过参数 `validation_split` 从训练集分割出一部分作为验证集。比如, `validation_split=0.1` 可以让 Keras 使用训练数据 (打散前) 的末尾 10% 作为验证集。

如果训练集非常倾斜, 一些类过渡表达, 一些欠表达, 在调用 `fit()` 时最好设置 `class_weight` 参数, 可以加大欠表达类的权重, 减小过渡表达类的权重。Keras 在计算损失时, 会使用这些权重。如果每个实例都要加权重, 可以设置 `sample_weight` (这个参数优先于 `class_weight` ) 。如果一些实例的标签是通过

专家添加的，其它实例是通过众包平台添加的，最好加大前者的权重，此时给每个实例都加权重就很有必要。通过在 `validation_data` 元组中，给验证集加上样本权重作为第三项，还可以给验证集添加样本权重。

`fit()` 方法会返回 `History` 对象，包含：训练参数（`history.params`）、周期列表（`history.epoch`）以及最重要的包含训练集和验证集的每个周期后的损失和指标的字典（`history.history`）。如果用这个字典创建一个 `pandas` 的 `DataFrame`，然后使用方法 `plot()`，就可以画出学习曲线，见图 10-12：

```
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()
```

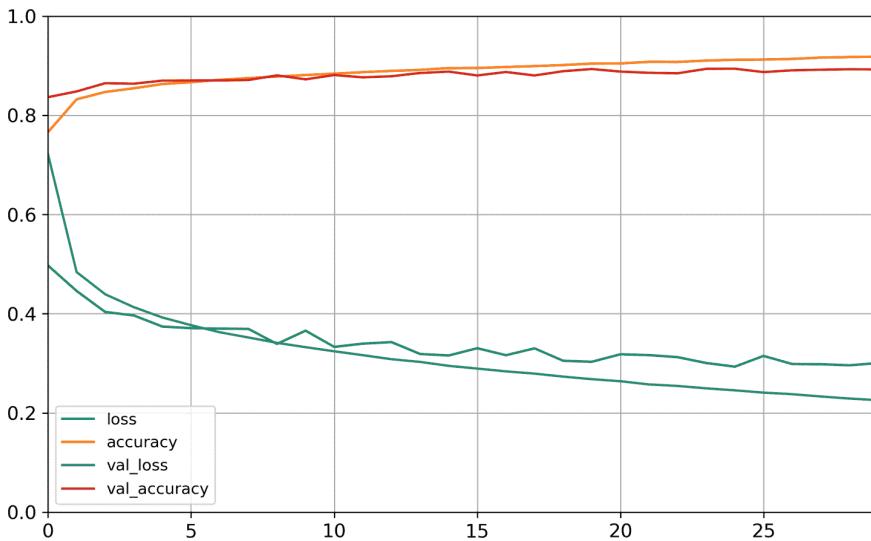


图 10-12 学习曲线：每个周期的平均训练损失和准确率，验证损失和准确率

可以看到，训练准确率和验证准确率稳步提高，训练损失和验证损失持续下降。另外，验证曲线和训练曲线靠的很近，意味着没有什么过拟合。在这个例子中，在训练一开始时，模型在验证集上的表现由于训练集。但实际情况是，验证误差是在每个周期结束后算出来的，而训练误差在每个周期期间，用流动平均误差算出来的。所以训练曲线（译者注，图中橙色的那条）实际应该向左移动半个周期。移动之后，就可以发现在训练开始时，训练和验证曲线几乎是完美重合起来的。

提示：在绘制训练曲线时，应该向左移动半个周期。

通常只要训练时间足够长，训练集的表现就能超越验证集。从图中可以看到，验证损失仍然在下降，模型收敛的还不好，所以训练应该持续下去。只需要再次调用方法 `fit()` 即可，因为 Keras 可以从断点处继续（验证准确率可以达到 89%。）

如果仍然对模型的表现不满意，就需要调节超参数了。首先是学习率。如果调节学习率没有帮助，就尝试换一个优化器（记得再调节任何超参数之后都重新调节学习率）。如果效果仍然不好，就调节模型自身的超参数，比如层数、每层的神经元数，每个隐藏层的激活函数。还可以调节其它超参数，比如批次大小（通过 `fit()` 的参数 `batch_size`，默认是 32）。本章末尾还会调节超参数。当对验证准确率达到满意之后，就可以用测试集评估泛化误差。只需使用 `evaluate()` 方法（`evaluate()` 方法包含参数 `batch_size` 和 `sample_weight`）：

```
>>> model.evaluate(X_test, y_test)
10000/10000 [=====] - 0s 29us/sample - loss: 0.3340 - accuracy: 0.8851
[0.3339798209667206, 0.8851]
```

正如第 2 章所见，测试集的表现通常比验证集上低一点，这是因为超参数根据验证集而不是测试集调节的（但是在这个例子中，我们没有调节过超参数，所以准确率下降纯粹是运气比较差而已）。一定不要在测试集上调节超参数，否则会影响泛化误差。

## 使用模型进行预测

接下来，就可以用模型的 `predict()` 方法对新实例做预测了。因为并没有新实例，所以就用测试集的前 3 个实例来演示：

```
>>> 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)
```

可以看到，模型会对每个实例的每个类（从 0 到 9）都给出一个概率。比如，对于第一张图，模型预测第 9 类（短靴）的概率是 96%，第 5 类（凉鞋）的概率是 3%，第 7 类（运动鞋）的概率是 1%，剩下的类的概率都是 0。换句话说，模型预测第一张图是鞋，最有可能是短靴，也有可能是凉鞋和运动鞋。如果只关心概率最高的类（即使概率不高），可以使用方法 `predict_classes()`：

```
>>> 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')
```

对于这 3 个实例，模型的判断都是对的（见图 10-13）：

```
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```

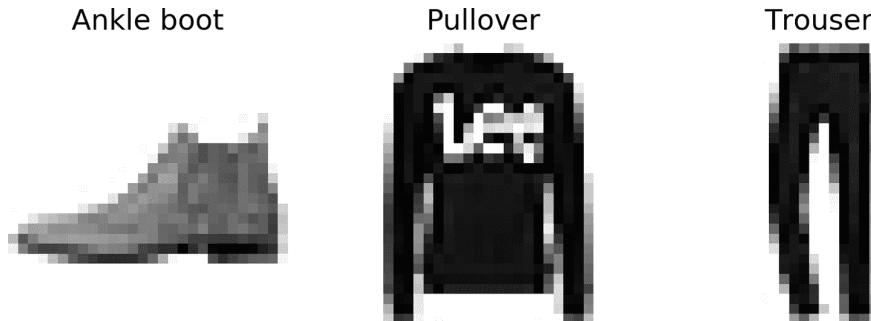


图 10-13 正确分类的 Fashion MNIST 图片

到此为止，我们学会了如何使用顺序 API 来搭建、训练、评估和使用分类 MLP？如何来做回归呢？

## 使用顺序 API 搭建回归 MLP

接下来使用回归神经网络来处理加州房价问题。简便起见，使用 Scikit-Learn 的 `fetch_california_housing()` 函数来加载数据。这个数据集比第 2 章所用的数据集简单，因为它只包括数值特征（没有 `ocean_proximity`），也不包括缺失值。加载好数据之后，将数据集分割成训练集、验证集和测试集，并做特征缩放：

```
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 搭建、训练、评估和使用回归 MLP 做预测，和前面的分类 MLP 很像。区别在于输出层只有一个神经元（因为只想预测一个值而已），也没有使用激活函数，损失函数是均方误差。因为数据集有噪音，我们就是用一个隐藏层，并且神经元也比之前少，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)
```

可以看到，使用顺序 API 是很方便的。但是，尽管 `Sequential` 十分常见，但用它搭建复杂拓扑形态或多输入多输出的神经网络还是不多。所以，Keras 还提供了函数式 API。

## 使用函数式 API 搭建复杂模型

Wide & Deep 是一个非序列化的神经网络模型。这个架构是 Heng-Tze Cheng 在 2016 年在[论文中](#)提出来的。这个模型可以将全部或部分输入与输出层连起来，见图 10-14。这样，就可以既学到深层模式（使用深度路径）和简单规则（使用短路径）。作为对比，常规 MLP 会强制所有数据流经所有层，因此数据中的简单模式在多次变换后会被扭曲。

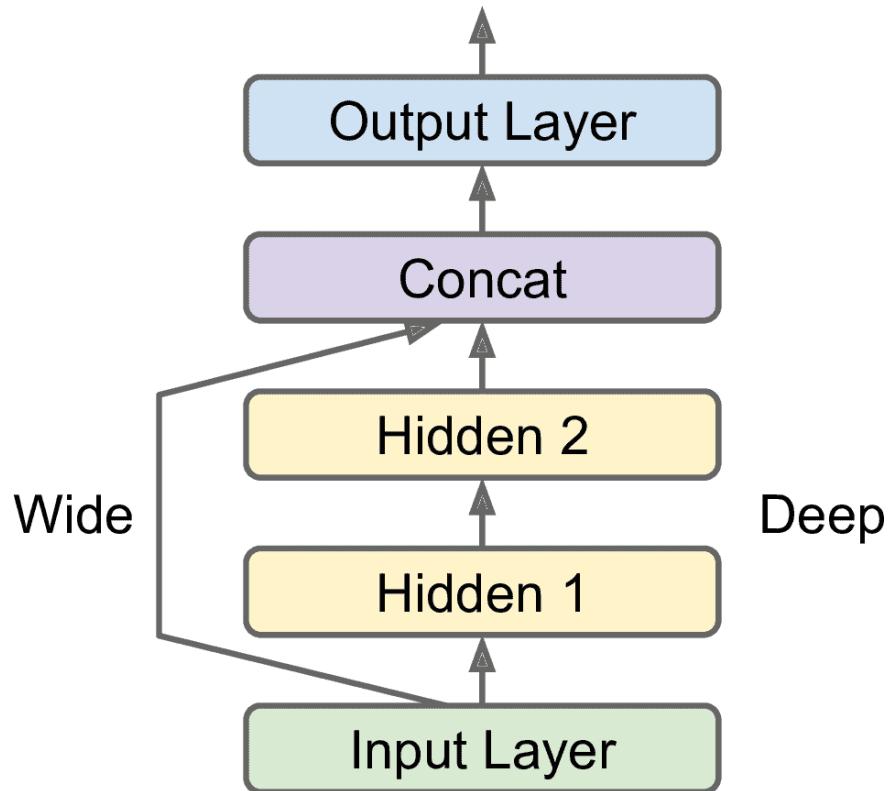


图 10-14 Wide &amp; Deep 神经网络

我们来搭建一个这样的神经网络，来解决加州房价问题：

```

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])
  
```

每行代码的作用：

- 首先创建一个 `Input` 对象。包括模型输入的形状 `shape` 和数据类型 `dtype`。模型可能会有多种输入。
- 然后，创建一个有 30 个神经元的紧密层，激活函数是 ReLU。创建好之后，将其作为函数，直接将输入传给它。这就是函数式 API 的得名原因。这里只是告诉 Keras 如何将层连起来，并没有导入实际数据。
- 然后创建第二个隐藏层，还是将其作为函数使用，输入时第一个隐藏层的输出；
- 接着，创建一个连接 `Concatenate` 层，也是作为函数使用，将输入和第二个隐藏层的输出连起来。可以使用 `keras.layers.concatenate()`。
- 然后创建输出层，只有一个神经元，没有激活函数，将连接层的输出作为输入。
- 最后，创建一个 Keras 的 `Model`，指明输入和输出。

搭建好模型之后，重复之前的步骤：编译模型、训练、评估、做预测。

但是如果你想将部分特征发送给 wide 路径，将部分特征（可以有重叠）发送给 deep 路径，该怎么做呢？答案是可以使用多输入。例如，假设向 wide 路径发送 5 个特征（特征 0 到 4），向 deep 路径发送 6 个特征（特征 2 到 7）：

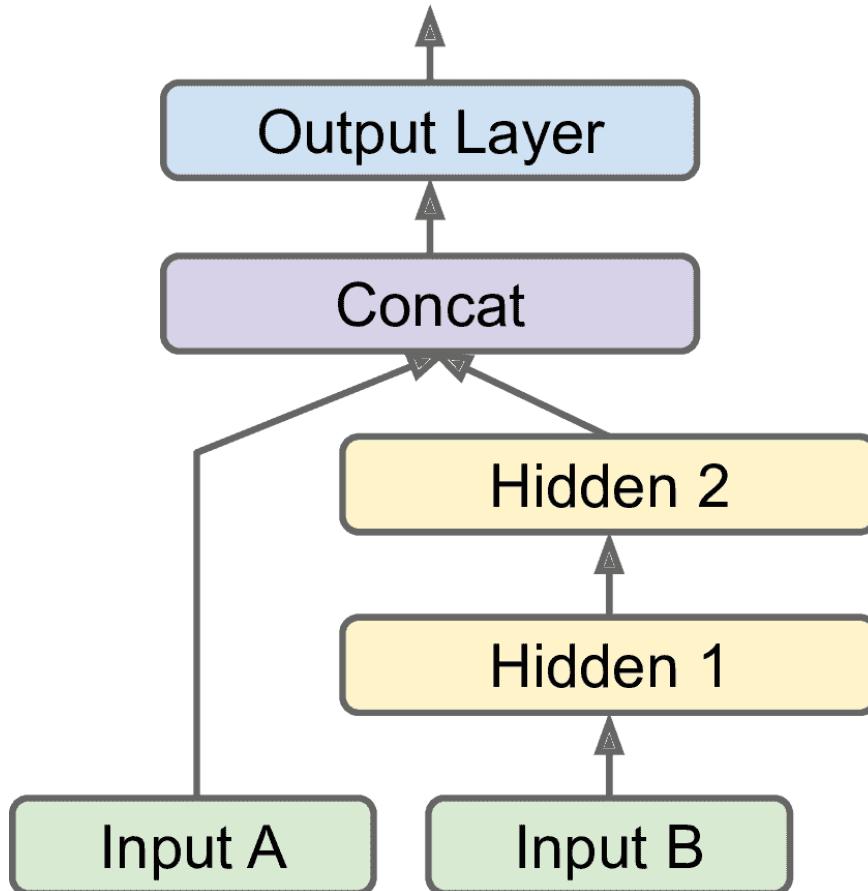


图 10-15 处理多输入

```

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])
  
```

代码非常浅显易懂。值得注意的是，在创建模型时，我们指明了 `inputs=[input_A, input_B]`。然后就可以像通常那样编译模型了，但当调用 `fit()` 时，不是传入矩阵 `X_train`，而是传入一对矩阵 (`X_train_A, X_train_B`)：每个输入一个矩阵。同理调用 `evaluate()` 或 `predict()` 时，`X_valid`、`X_test`、`X_new` 也要变化：

```

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))
  
```

有以下要使用多输入的场景：

- 任务要求。例如，你想定位和分类图片中的主要物体。这既是一个回归任务（找到目标中心的坐标、宽度和高度）和分类任务。
- 相似的，对于相同的数据，你可能有多个独立的任务。当然可以每个任务训练一个神经网络，但在多数情况下，同时对所有任务训练一个神经网络，每个任务一个输出，后者的效果更好。这是因为神经网络可以在不同任务间学习有用的数据特征。例如，在人脸的多任务分类时，你可以用一个输出做人物表情的分类（微笑惊讶等等），用另一个输出判断是否戴着眼镜。
- 另一种情况是作为一种正则的方法（即，一种降低过拟合和提高泛化能力的训练约束）。例如，你想在神经网络中加入一些辅助输出（见图 10-16），好让神经网络的一部分依靠自身就能学到一些东西。

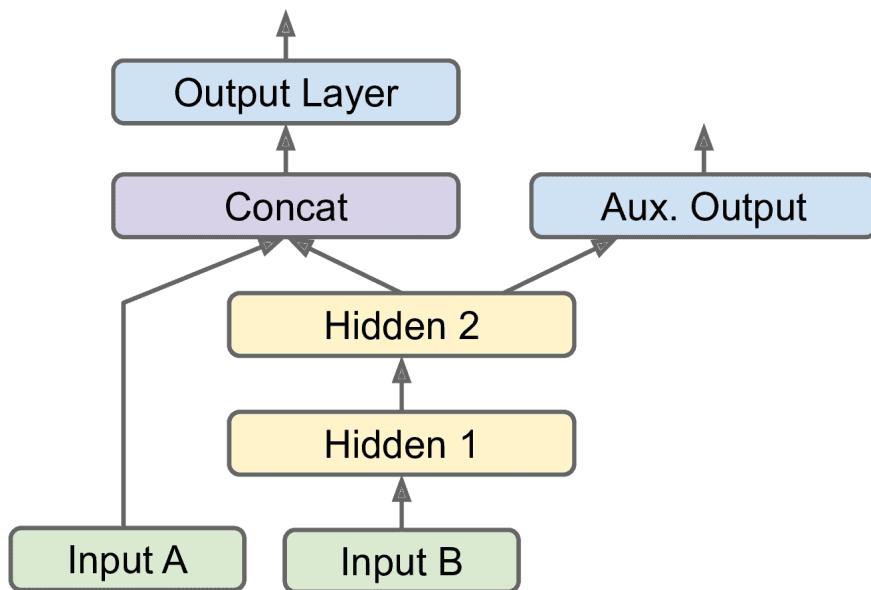


图 10-16 处理多输入，加入辅助输出作为正则

添加额外的输出很容易：只需要将输出和相关的层连起来、将输出写入输出列表就行。例如，下面的代码搭建的就是图 10-16 的架构：

```
[...] # output 层前面都一样
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])
```

每个输出都要有自己的损失函数。因此在编译模型时，需要传入损失列表（如果只传入一个损失，Keras 会认为所有输出是同一个损失函数）。Keras 默认计算所有损失，将其求和得到最终损失用于训练。主输出比辅助输出更值得关心，所以要提高它的权重，如下所示：

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1], optimizer="sgd")
```

此时若要训练模型，必须给每个输出贴上标签。在这个例子中，主输出和辅输出预测的是同一件事，因此标签相同。传入数据必须是 `(y_train, y_train)` (`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 会返回总损失和各个损失值：

```
total_loss, main_loss, aux_loss = model.evaluate(
    [X_test_A, X_test_B], [y_test, y_test])
```

相似的，方法 `predict()` 会返回每个输出的预测值：

```
y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```

可以看到，用函数式 API 可以轻易搭建任意架构。接下来再看最后一种搭建 Keras 模型的方法。

## 使用子类化 API 搭建动态模型

顺序 API 和函数式 API 都是声明式的：只有声明创建每个层以及层的连接方式，才能给模型加载数据以进行训练和推断。这种方式有其优点：模型可以方便的进行保存、克隆和分享；模型架构得以展示，便于分析；框架可以推断数据形状和类型，便于及时发现错误（加载数据之前就能发现错误）。调试也很容易，因为模型是层的静态图。但是缺点也很明显：模型是静态的。一些模型包含循环、可变数据形状、条件分支，和其它的动态特点。对于这些情况，或者你只是喜欢命令式编程，不妨使用子类化 API。

对 `Model` 类划分子类，在构造器中创建需要的层，调用 `call()` 进行计算。例如，创建一个下面的 `WideAndDeepModel` 类的实例，就可以创建与前面函数式 API 例子的同样模型，同样可以进行编译、评估、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()
```

这个例子和函数式 API 很像，除了不用创建输入；只需要在 `call()` 使用参数 `input`，另外的不同是将层的创建和使用分割了。最大的差别是，在 `call()` 方法中，你可以做任意想做的事：`for` 循环、`if` 语句、低级的 TensorFlow 操作，可以尽情发挥想象（见第 12 章）！子类化 API 可以让研究者试验各种新创意。

然而代价也是有的：模型架构隐藏在 `call()` 方法中，所以 Keras 不能对其检查；不能保存或克隆；当调用 `summary()` 时，得到的只是层的列表，没有层的连接信息。另外，Keras 不能提前检查数据类型和形状，所以很容易犯错。所以除非真的需要灵活性，还是使用顺序 API 或函数式 API 吧。

**提示：**可以像常规层一样使用 Keras 模型，组合模型搭建任意复杂的架构。

学会了搭建和训练神经网络，接下来看看如何保存。

## 保存和恢复模型

使用顺序 API 或函数式 API 时，保存训练好的 Keras 模型和训练一样简单：

```
model = keras.layers.Sequential([...]) # or keras.Model([...])
model.compile([...])
model.fit([...])
model.save("my_keras_model.h5")
```

Keras 使用 HDF5 格式保存模型架构（包括每层的超参数）和每层的所有参数值（连接权重和偏置项）。还保存了优化器（包括超参数和状态）。

通常用脚本训练和保存模型，一个或更多的脚本（或 web 服务）来加载模型和做预测。加载模型很简单：

```
model = keras.models.load_model("my_keras_model.h5")
```

**警告**：这种加载模型的方法只对顺序 API 或函数式 API 有用，不适用于子类化 API。对于后者，可以用 `save_weights()` 和 `load_weights()` 保存参数，其它的就得手动保存恢复了。

但如果训练要持续数个小时呢？在大数据集上训练，训练时间长很普遍。此时，不仅要在训练结束时保存模型检查点，在一定时间间隔内也要保存，以免电脑宕机造成损失。但是如何告诉 `fit()` 保存检查点呢？使用回调。

## 使用回调

`fit()` 方法接受参数 `callbacks`，可以让用户指明一个 Keras 列表，让 Keras 在训练开始和结束、每个周期开始和结束、甚至是每个批次的前后调用。例如，`ModelCheckpoint` 可以在每个时间间隔保存检查点，默认是每个周期结束之后：

```
[...] # 搭建编译模型
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

另外，如果训练时使用了验证集，可以在创建检查点时设定 `save_best_only=True`，只有当模型在验证集上取得最优值时才保存模型。这么做可以不必担心训练时间过长和训练集过拟合：只需加载训练好的模型，就能保证是在验证集上表现最好的模型。下面的代码演示了早停（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
```

另一种实现早停的方法是使用 `EarlyStopping` 回调。当检测到经过几个周期（周期数由参数 `patience` 确定），验证集表现没有提升时，就会中断训练，还能自动滚回到最优模型。可以将保存检查点（避免宕机）和早停（避免浪费时间和资源）结合起来：

```
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])
```

周期数可以设的很大，因为准确率没有提升时，训练就会自动停止。此时，就没有必要恢复最优模型，因为 `EarlyStopping` 调回一直在跟踪最优权重，训练结束时能自动恢复。

**提示：**包 `keras.callbacks` 中还有其它可用的调回。

如果还想有其它操控，还可以编写自定义的调回。下面的例子展示了一个可以展示验证集损失和训练集损失比例的自定义（检测过拟合）调回：

```
class PrintValTrainRatioCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print("\nval/train: {:.2f}".format(logs["val_loss"] / logs["loss"]))
```

类似的，还可以实

现 `on_train_begin()`、`on_train_end()`、`on_epoch_begin()`、`on_epoch_end()`、`on_batch_begin()` 和 `on_batch_end()`。如果需要的话，在评估和预测时也可以使用调回（例如为了调试）。对于评估，可以实

现 `on_test_begin()`、`on_test_end()`、`on_test_batch_begin()` 或 `on_test_batch_end()`（通过 `evaluate()` 调用）；对于预测，可以实

现 `on_predict_begin()`、`on_predict_end()`、`on_predict_batch_begin()` 或 `on_predict_batch_end()`（通过 `predict()` 调用）。

下面来看一个使用 `tf.keras` 的必备工具：`TensorBoard`。

## 使用 `TensorBoard` 进行可视化

`TensorBoard` 是一个强大的交互可视化工具，使用它可以查看训练过程中的学习曲线、比较每次运行的学习曲线、可视化计算图、分析训练数据、查看模型生成的图片、可视化投射到 3D 的多维数据，等等。`TensorBoard` 是 TensorFlow 自带的。

要使用 `TensorBoard`，必须修改程序，将要可视化的数据输出为二进制的日志文件 `event files`。每份二进制数据称为摘要 `summary`，`TensorBoard` 服务器会监测日志文件目录，自动加载更新并可视化：这样就能看到实时数据（稍有延迟），比如训练时的学习曲线。通常，将 `TensorBoard` 服务器指向根日志目录，程序的日志写入到它的子目录，这样一个 `TensorBoard` 服务就能可视化并比较多次运行的数据，而不会将其搞混。

我们先定义 `TensorBoard` 的根日志目录，还有一些根据当前日期生成子目录的小函数。你可能还想在目录名中加上其它信息，比如超参数的值，方便知道查询的内容：

```
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'
```

Keras 提供了一个 `TensorBoard()` 调回：

```
[...] # 搭建编译模型
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])
```

简直不能再简单了。如果运行这段代码，`TensorBoard()` 调回会负责创建日志目录（包括父级目录），在训练过程中会创建事件文件并写入概要。再次运行程序（可能修改了一些超参数）之后，得到的目录结构可能如下：

```
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
    └── validation
        └── events.out.tfevents.1559891733.mycomputer.local.38511.696430.v2
    run_2019_06_07-15_15_49
    [...]
```

每次运行都会创建一个目录，每个目录都有一个包含训练日志和验证日志的子目录。两者都包括事件文件，训练日志还包括分析追踪信息：它可以让 `TensorBoard` 展示所有设备上的模型的各个部分的训练时长，有助于定位性能瓶颈。

然后就可以启动 `TensorBoard` 服务了。一种方式是通过运行命令行。如果是在虚拟环境中安装的 `TensorFlow`，需要激活虚拟环境。接着，在根目录（也可以是其它路径，但一定要指向日志目录）运行下面的命令：

```
$ tensorboard --logdir=./my_logs --port=6006
TensorBoard 2.0.0 at http://mycomputer.local:6006/ (Press CTRL+C to quit)
```

如果终端没有找到 `tensorboard` 命令，必须更新环境变量 `PATH`（或者，可以使用 `python3 -m tensorboard.main`）。服务启动后，打开浏览器访问 `http://localhost:6006`。

或者，通过运行下面的命令，可以在 `Jupyter` 里面直接使用 `TensorBoard`。第一行代码加载了 `TensorBoard` 扩展，第二行在端口 6006 启动了一个 `TensorBoard` 服务，并连接：

```
%load_ext tensorboard
%tensorboard --logdir=./my_logs --port=6006
```

无论是使用哪种方式，都得使用 `TensorBoard` 的浏览器界面。点击栏 `SCALARS` 可以查看学习曲线（见图 10-17）。左下角选择想要可视化的路径（比如第一次和第二次运行的训练日志），再点击 `epoch_loss`。可以看到，在两次训练过程中，训练损失都是下降的，但第二次下降的更快。事实上，第二次的学习率是 0.05 (`optimizer=keras.optimizers.SGD(lr=0.05)`) 而不是 0.001。

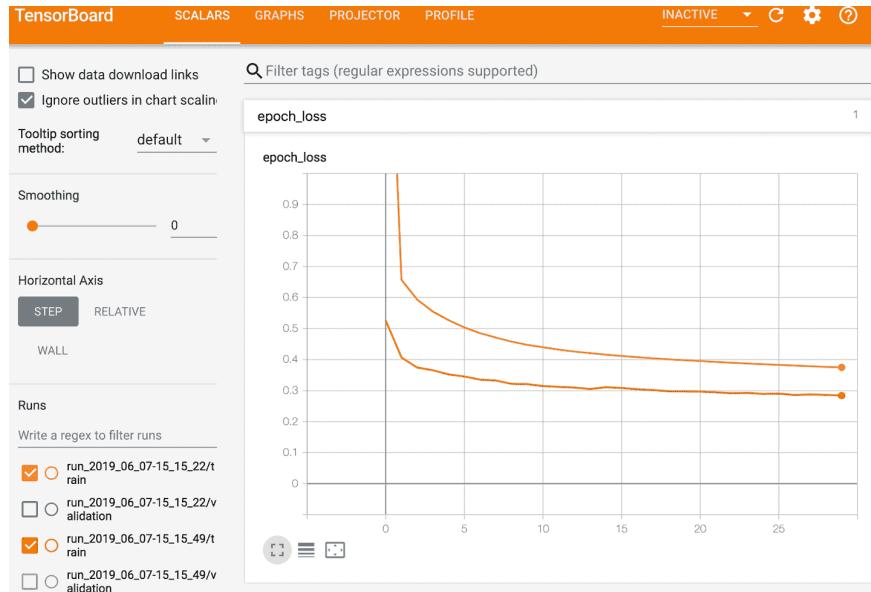


图 10-17 使用 TensorBoard 可视化学习曲线

还可以对全图、权重（投射到 3D）或其它信息做可视化。`TensorBoard()` 调用还有选项可以记录其它数据的日志，比如嵌入（见第 13 章）。另外，`TensorBoard` 在 `tf.summary` 包中还提供了低级 API。下面的代码使用方法 `create_file_writer()` 创建了 `SummaryWriter`，`TensorBoard` 使用 `SummaryWriter` 作为记录标量、柱状图、图片、音频和文本的上下文，所有这些都是可以可视化的！

```
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 32x32 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)
```

总结一下目前所学：神经网络的起源、MLP 是什么、如何用 MLP 做分类和回归、如何使用顺序 API 搭建 MLP、如何使用函数式 API 或子类化 API 搭建更复杂的模型架构、保存和恢复模型、如何使用回调创建检查点、早停，等等。最后，学了使用 TensorBoard 做可视化。这些知识已经足够解决许多问题了。但是，你可能还有疑问，如何选择隐藏层的层数、神经元的数量，以及其他超参数，下面就来讨论这些问题。

## 微调神经网络的超参数

神经网络的灵活性同时也是它的缺点：要微调的超参数太多了。不仅架构可能不同，就算对于一个简单的 MLP，就可以调节层数、每层的神经元数、每层使用什么激活函数、初始化的权重，等等。怎么才能知道哪个超参数的组合才是最佳的呢？

一种方法是直接试验超参数的组合，看哪一个在验证集（或使用 K 折交叉验证）的表现最好。例如，可以使用 `GridSearchCV` 或 `RandomizedSearchCV` 探索超参数空间，就像第 2 章中那样。要这么做的话，必须将 Keras 模型包装进模仿 Scikit-Learn 回归器的对象中。第一步是给定一组超参数，创建一个搭建和编译 Keras 模型的函数：

```
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
```

这个函数创建了一个单回归（只有一个输出神经元）顺序模型，数据形状、隐藏层的层数和神经元数是给定的，使用指定学习率的 SGD 优化器编译。最好尽量给大多数超参数都设置合理的默认值，就像 Scikit-Learn 那样。

然后使用函数 `build_model()` 创建一个 `KerasRegressor`：

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

`KerasRegressor` 是通过 `build_model()` 将 Keras 模型包装起来的。因为在创建时没有指定任何超参数，使用的是 `build_model()` 的默认参数。现在就可以像常规的 Scikit-Learn 回归器一样来使用它了：使用 `fit()` 方法训练，使用 `score()` 方法评估，使用 `predict()` 方法预测，见下面代码：

```
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)
```

任何传给 `fit()` 的参数都会传给底层的 Keras 模型。另外，分数的意义和 MSE 是相反的（即，分数越高越好）。因为超参数太多，最好使用随机搜索而不是网格搜索（见第 2 章的解释）。下面来探索下隐藏层的层数、神经元数和学习率：

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_dists = {
    "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_dists, 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)])
```

所做的和第 2 章差不多，除了这里试讲参数传给 `fit()`，`fit()` 再传给底层的 Keras。注意，`RandomizedSearchCV` 使用的是 K 折交叉验证，没有用 `X_valid` 和 `y_valid`（只有早停时才使用）。

取决于硬件、数据集大小、模型复杂度、`n_iter` 和 `cv`，求解过程可能会持续几个小时。计算完毕后，就能得到最佳参数、最佳得分和训练好的 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
```

现在就可以保存模型、在测试集上评估，如果对效果满意，就可以部署了。使用随机搜索并不难，适用于许多相对简单的问题。但是当训练较慢时（大数据集的复杂问题），这个方法就只能探索超参数空间的一小部分而已。通过手动调节可以缓解一下：首先使用大范围的超参数值先做一次随机搜索，然后根据第一次的结果再做一次小范围的计算，以此类推。这样就能缩放到最优超参数的范围了。但是，这么做很耗时。

幸好，有比随机搜索更好的探索超参数空间的方法。核心思想很简单：当某块空间的区域表现好时，就多探索这块区域。这些方法可以代替用户做“放大”工作，可以在更短的时间得到更好的结果。下面是一些可以用来优化超参数的 Python 库：

[Hyperopt](#) 一个可以优化各种复杂搜索空间（包括真实值，比如学习率和离散值，比如层数）的库。

[Hyperas](#), [kopt](#) 或 [Talos](#) 用来优化 Keras 模型超参数的库（前两个是基于 Hyperopt 的）。

[Keras Tuner](#) Google 开发的简单易用的 Keras 超参数优化库，还有可视化和分析功能。

[Scikit-Optimize \( skopt \)](#) 一个通用的优化库。类 `BayesSearchCV` 使用类似于 `GridSearchCV` 的接口做贝叶斯优化。

[Spearmint](#) 一个贝叶斯优化库。

[Hyperband](#) 一个快速超参数调节库，基于 Lisha Li 的论文 [《Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization》](#)。

[Sklearn-Deap](#) 一个基于进化算法的超参数优化库，接口类似 `GridSearchCV`。

另外，许多公司也提供超参数优化服务。第 19 章会讨论 Google Cloud AI 平台的 [超参数调节服务](#)。其它公司有 [Arimo](#)、[SigOpt](#)，和 CallDesk 的 [Oscar](#).

超参数调节仍然是活跃的研究领域，其中进化算法表现很突出。例如，在 2017 年的论文 [《Population Based Training of Neural Networks》](#) 中，Deepmind 的作者用统一优化了一组模型及其超参数。Google 也使用了一种进化算法，不仅用来搜索超参数，还可以搜索最佳的神经网络架构；[Google 的 AutoML 套件已经可以在云服务上使用了](#)。也许手动搭建神经网络的日子就要结束了？看看 [Google 的这篇文章](#)。事实上，用进化算法训练独立的神经网络很成功，已经取代梯度下降了。例如，Uber 在 2017 年介绍了名为 Deep Neuroevolution 的技术，见[这里](#)。

尽管有这些工具和服务，知道每个超参数该取什么值仍然是帮助的，可以快速创建原型和收缩搜索范围。后面的文字介绍了选择 MLP 隐藏层数和神经元数的原则，以及如何选择主要的超参数值。

## 隐藏层数

对于许多问题，开始时只用一个隐藏层就能得到不错的结果。只要有足够多的神经元，只有一个隐藏层的 MLP 就可以对复杂函数建模。但是对于复杂问题，深层网络比浅层网络有更高的参数效率：深层网络可以用指数级别更少的神经元对复杂函数建模，因此对于同样的训练数据量性能更好。

要明白为什么，假设别人让你用绘图软件画一片森林，但你不能复制和粘贴。这样的话，就得花很长时间，你需要手动来画每一棵树，一个树枝然后一个树枝，一片叶子然后一片叶子。如果可以鲜花一片叶子，然后将叶子复制粘贴到整个树枝上，再将树枝复制粘贴到整棵树上，然后再复制树，就可以画出一片森林了，所用的时间可以大大缩短。真实世界的数据通常都是有层次化结构的，深层神经网络正式利用了这一点：浅隐藏层对低级结构（比如各种形状的线段和方向），中隐藏层结合这些低级结构对中级结构（方，圆）建模，深隐藏层和输出层结合中级结构对高级结构（比如，脸）建模。

层级化的结构不仅帮助深度神经网络收敛更快，也提高了对新数据集的泛化能力。例如，如果已经训练好了一个图片人脸识别的模型，现在想训练一个识别发型的神经网络，你就可以复用第一个网络的浅层。不用随机初始化前几层的权重和偏置项，而是初始化为第一个网络浅层的权重和偏置项。这样，网络就不用从多数图片的低级结构开始学起；只要学高级结构（发型）就行了。这就称为迁移学习。

概括来讲，对于许多问题，神经网络只有一或两层就够了。例如，只用一个隐藏层和几百个神经元，就能在 MNIST 上轻松达到 97% 的准确率；同样的神经元数，两个隐藏层，训练时间几乎相同，就能达到 98% 的准确率。对于更复杂的问题，可以增加隐藏层的数量，直到在训练集上过拟合为止。非常复杂的任务，比如大图片分类或语音识别，神经网络通常需要几十层（甚至上百，但不是全连接的，见第 14 章），需要的训练数据量很大。对于这样的网络，很少是从零训练的：常见的使用预训练好的、表现出众的任务相近的网络，训练可以快得多，需要的数据也可以不那么多（见第 11 章的讨论）。

## 每个隐藏层的神经元数

输入层和输出层的神经元数是由任务确定的输入和输出类型决定的。例如，MNIST 任务需要  $28 \times 28 = 784$  个输入神经元和 10 个输出神经元。

对于隐藏层，惯用的方法是模拟金字塔的形状，神经元数逐层递减——底层思想是，许多低级特征可以聚合成少得多的高级特征。MNIST 的典型神经网络可能需要 3 个隐藏层，第一层有 300 个神经元，第二层有 200 个神经元，第三层有 100 个神经元。然而，这种方法已经被抛弃了，因为所有隐藏层使用同样多的神经元不仅表现更好，要调节的超参数也只变成了一个，而不是每层都有一个。或者，取决于数据集的情况，有时可以让第一个隐藏层比其它层更大。

和层数相同，可以逐步提高神经元的数量，知道发生过拟合为止。但在实际中，通常的简便而高效的方法是使用层数和神经元数都超量的模型，然后使用早停和其它正则技术防止过拟合。一位 Google 的科学家 Vincent Vanhoucke，称这种方法为“弹力裤”：不浪费时间选择尺寸完美匹配的裤子，而是选择一条大的弹力裤，它能自动收缩到合适的尺寸。通过这种方法，可以避免影响模型的瓶颈层。另一方面，如果某层的神经元太少，就没有足够强的表征能力，保存所有的输入信息（比如，只有两个神经元的层只能输出 2D 数据，如果用它处理 3D 数据，就会丢失信息）。无论模型网络的其它部分如何强大，丢失的信息也找不回来了。

提示：通常，增加层数比增加每层的神经元的收益更高。

## 学习率，批次大小和其它超参数

隐藏层的层数和神经元数不是 MLP 唯二要调节的参数。下面是一些其它的超参数和调节策略：

**学习率：** 学习率可能是最重要的超参数。通常，最佳学习率是最大学习率（最大学习率是超过一定值，训练算法发生分叉的学习率，见第 4 章）的大概一半。找到最佳学习率的方式之一是从一个极小值开始（比如  $10^{-5}$ ）训练模型几百次，直到学习率达到一个比较大的值（比如 10）。这是通过在每次迭代，将学习率乘以一个常数实现的（例如  $\exp(\log(10^6)/500)$ ，通过 500 次迭代，从  $10^{-5}$  到 10）。如果将损失作为学习率的函数画出来（学习率使用 log），能看到损失一开始是下降的。过了一段时间，学习率会变得非常高，损失就会升高：最佳学习率要比损失开始升高的点低一点（通常比拐点低 10 倍）。然后就可以重新初始化模型，用这个学习率开始训练了。第 11 章会介绍更多的学习率优化方法。

**优化器：** 选择一个更好的优化器（并调节超参数）而不是传统的小批量梯度下降优化器同样重要。第 11 章会介绍更先进的优化器。

**批次大小：** 批次大小对模型的表现和训练时间非常重要。使用大批量的好处是硬件（比如 GPU）可以快速处理（见第 19 章），每秒可以处理更多实例。因此，许多人建议批次大小开到 GPU 内存的最大值。但也有缺点：在实际中，大批量，会导致训练不稳定，特别是在训练开始时，并且不如小批次模型的泛化能力好。2018 年四月，Yann LeCun 甚至发了一条推特：“朋友之间不会让对方的批次大小超过 32”，引用的是 Dominic Masters 和 Carlo Luschi 的论文《Revisiting Small Batch Training for Deep Neural Networks》，在这篇论文中，作者的结论是小批次（2 到 32）更可取，因为小批次可以在更短的训练时间得到更好的模型。但是，有的论文的结论截然相反：2017 年，两篇论文《Train longer, generalize better: closing the generalization gap in large batch training of neural networks》和《Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour》建议，通过多种方法，比如给学习率热身（即学习率一开始很小，然后逐渐提高，见第 11 章），就能使用大批量（最大 8192）。这样，训练时间就能非常短，也没有泛化鸿沟。因此，一种策略是通过学习率热身使用大批量，如果训练不稳定或效果不好，就换成小批次。

**激活函数：** 本章一开始讨论过如何选择激活函数：通常来讲，ReLU 适用于所有隐藏层。对于输出层，就要取决于任务。

**迭代次数：** 对于大多数情况，用不着调节训练的迭代次数：使用早停就成了。

**提示：** 最佳学习率还取决于其它超参数，特别是批次大小，所以如果调节了任意超参数，最好也更新学习率。

想看更多关于调节超参数的实践，可以参考 Leslie Smith 的论文《A disciplined approach to neural network hyper-parameters: Part 1 -- learning rate, batch size, momentum, and weight decay》。

这章总结了对人工神经网络，以及 Keras 是实现。接下来的章节，我们会讨论训练深层网络的方法。还会使用 TensorFlow 的低级 API 实现自定义模型，和使用 Data API 高效加载和预处理数据。还会探讨其它流行的神经网络：用于图像处理的卷积神经网络，用于序列化数据的循环神经网络，用于表征学习的自编码器，用于建模和生成数据的对抗生成网络。

## 练习

---

1. [TensorFlow Playground](#) 是 TensorFlow 团队推出的一个便利的神经网络模拟器。只需点击几下，就能训练出二元分类器，通过调整架构和超参数，可以从直观上理解神经网络是如何工作的，以及超参数的作用。如下所示：

- a. 神经网络学到的模式。点击左上的运行按钮，训练默认的神经网络。注意是如何找到分类任务的最优解的。第一个隐藏层学到了简单模式，第二个隐藏层将简单模式结合为更复杂的模式。通常，层数越多，得到的模式越复杂。
- b. 激活函数。用 ReLU 激活函数代替 tanh，再训练一次网络。注意，找到解变得更快了，且是线性的，这归功于 ReLU 函数的形状。
- c. 局部最小值的风险。将网络只设定为只有一个隐藏层，且只有 3 个神经元。进行多次训练（重置网络权重，点击 Reset 按钮）。可以看到训练时间变化很大，甚至有时卡在了局部最小值。
- d. 神经网络太小的状况。去除一个神经元，只剩下两个。可以看到，即使尝试多次，神经网络现也不能找到最优解。模型的参数太少，对训练集数据欠拟合。
- e. 神经网络足够大的状况。将神经元数设为 8，再多次训练神经网络。可以看到过程很快且不会卡住。这是一个重要的发现：大神经网络几乎从不会卡在局部最小值，即使卡住了，局部最小值通常也是全局最小值。但是仍然可能在平台期卡住相当长时间。
- f. 梯度消失的风险。选择 spiral 数据集（右下角位于 DATA 下面的数据集），模型架构变为四个隐藏层，每层八个神经元。可以看到，训练耗时变长，且经常在平台期卡住很长时间。另外，最高层（右边）的神经元比最底层变得快。这个问题被称为“梯度消失”，可以通过更优的权重初始化、更好的优化器（比如 AdaGrad 或 Adam）或批次正态化（见第 11 章）解决。
- g. 再尝试尝试其它参数。
  1. 用原始神经元（像图 10-3 中的神经元）画 ANN，可以计算  $A \oplus B$  （ $\oplus$  表示 XOR 操作）。提示： $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$
  2. 为什么逻辑回归比经典感知机（即使使用感知机训练算法训练的单层的阈值逻辑单元）更好？如何调节感知机，使其等同于逻辑回归分类器？
  3. 为什么逻辑激活函数对训练 MLP 的前几层很重要？
  4. 说出三种流行的激活函数，并画出来。
  5. 假设一个 MLP 的输入层有 10 个神经元，接下来是有 50 个人工神经元的的隐藏层，最后是一个有 3 个人工神经元的输出层。所有的神经元使用 ReLU 激活函数。回答以下问题：
  6. 输入矩阵  $x$  的形状是什么？
  7. 隐藏层的权重向量  $w[h]$  和偏置项  $b[h]$  的形状是什么？
  8. 输出层的权重向量  $w[o]$  和偏置项  $b[o]$  的形状是什么？
  9. 输出矩阵  $y$  的形状是什么？
  10. 写出用  $x, w[h], b[h], w[o], b[o]$  计算矩阵  $y$  的等式。

11. 如果要将邮件分为垃圾邮件和正常邮件，输出层需要几个神经元？输出层应该使用什么激活函数？如果任务换成 MNIST，输出层需要多少神经元，激活函数是什么？再换成第 2 章中的房价预测，输出层又该怎么变？
12. 反向传播是什么及其原理？反向传播和逆向 autodiff 有什么不同？
13. 列出所有简单 MLP 中需要调节的超参数？如果 MLP 过拟合训练数据，如何调节超参数？
14. 在 MNIST 数据上训练一个深度 MLP。

使用 `keras.datasets.mnist.load_data()` 加载数据，看看能否使准确率超过 98%，利用本章介绍的方法（逐步指级提高学习率，画误差曲线，找到误差升高的点）搜索最佳学习率。保存检查点，使用早停，用 TensorBoard 画学习曲线的图。

参考答案见附录 A。

## 十一、训练深度神经网络

译者：[@SeanCheney](#)

第 10 章介绍了人工神经网络，并训练了第一个深度神经网络。但它非常浅，只有两个隐藏层。如果你需要解决非常复杂的问题，例如检测高分辨率图像中的数百种类型的对象，该怎么办？你可能需要训练更深的 DNN，也许有 10 层或更多，每层包含数百个神经元，通过数十万个连接相连。这可不像公园散步那么简单，可能碰到下面这些问题：

- 你将面临棘手的梯度消失问题（或相关的梯度爆炸问题）：在反向传播过程中，梯度变得越来越小或越来越大。二者都会使较浅层难以训练；
- 要训练一个庞大的神经网络，但是数据量不足，或者标注成本很高；
- 训练可能非常慢；
- 具有数百万参数的模型将会有严重的过拟合训练集的风险，特别是在训练实例不多或存在噪音时。

在本章中，我们将依次讨论这些问题，并给出解决问题的方法。我们将从梯度消失/爆炸问题开始，并探讨解决这个问题的一些最流行的解决方案。接下来会介绍迁移学习和无监督预训练，这可以在即使标注数据不多的情况下，也能应对复杂问题。然后我们将看看各种优化器，可以加速大型模型的训练。最后，我们将浏览一些流行的大型神经网络正则化方法。

使用这些工具，你将能够训练非常深的网络：欢迎来到深度学习的世界！

### 梯度消失/爆炸问题

正如我们在第 10 章中所讨论的那样，反向传播算法的工作原理是从输出层到输入层，传播误差的梯度。一旦该算法已经计算了网络中每个参数的损失函数的梯度，它就通过梯度下降使用这些梯度来更新每个参数。

不幸的是，随着算法进展到较低层，梯度往往变得越来越小。结果，梯度下降更新使得低层连接权重实际上保持不变，并且训练永远不会收敛到最优解。这被称为梯度消失问题。在某些情况下，可能会发生相反的情况：梯度可能变得越来越大，许多层得到了非常大的权重更新，算法发散。这是梯度爆炸的问题，在循环神经网络中最为常见（见第 145 章）。更一般地说，深度神经网络面临梯度不稳定；不同的层可能有非常不同的学习率。

虽然很早就观察到这种现象了（这是造成深度神经网络在 2000 年早期被抛弃的原因之一），但直到 2010 年左右，人们才略微清楚了导致梯度消失/爆炸的原因。

Xavier Glorot 和 Yoshua Bengio 发表的题为《[Understanding the Difficulty of Training Deep Feedforward Neural Networks](#)》的论文发现了一些疑点，包括流行的 sigmoid 激活函数和当时最受欢迎的权重初始化方法的组合，即随机初始化时使用平均值为 0，标准差为 1 的正态分布。简而言之，他们表明，用这个激活函数和这个初始化方案，每层输出的方差远大于其输入的方差。随着网络前向传播，每层的方差持续增加，直到激活函数在顶层饱和。`logistic` 函数的平均值为 0.5 而不是 0（双曲正切函数的平均值为 0，表现略好于深层网络中的 `logistic` 函数），使得情况更坏。

看一下 logistic 激活函数（参见图 11-1），可以看到当输入变大（负或正）时，函数饱和在 0 或 1，导数非常接近 0。因此，当反向传播开始时，它几乎没有梯度通过网络传播回来，而且由于反向传播通过顶层向下传递，所以存在的小梯度不断地被稀释，因此较低层得到的改善很小。

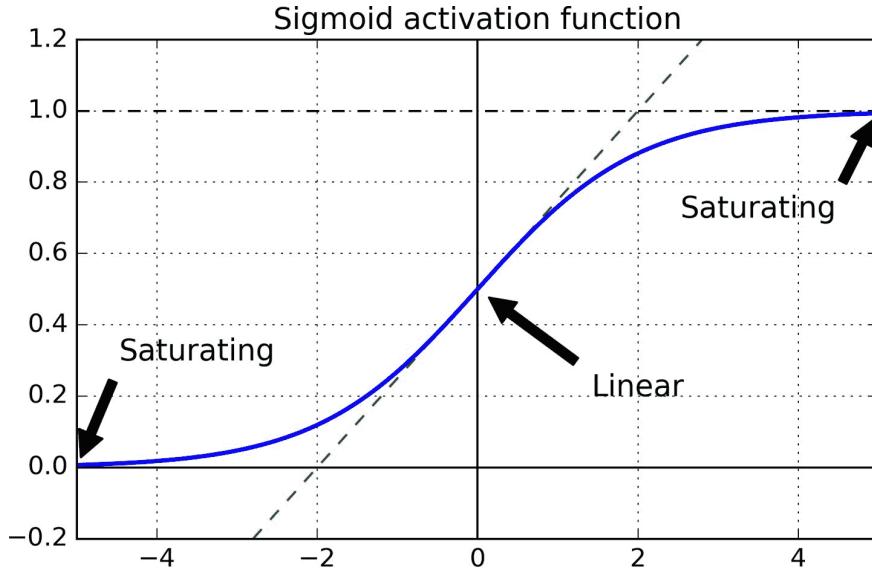


图 11-1 逻辑激活函数饱和

## Glorot 和 He 初始化

Glorot 和 Bengio 在他们的论文中提出了一种显著缓解这个问题的方法。我们需要信号在两个方向上正确地流动：在进行预测时是前向的，在反向传播梯度时是逆向的。我们不希望信号消失，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，并且反向传播时，流经一层的前后，梯度的方差也要相同（如果对数学细节感兴趣的话，请查看论文）。实际上不可能保证两者都是一样的，除非这个层具有相同数量的输入和神经元（这两个数被称为该层的扇入 fan-in 和扇出 fan-out），但是他们提出了一个很好的折衷办法，在实践中证明这个折中办法非常好：随机初始化连接权重必须如公式 11-1 这样，其中  $\text{fan}[\text{avg}] = (\text{fan}[\text{in}] + \text{fan}[\text{out}]) / 2$ 。这种初始化策略通常被称为 Xavier 初始化或 Glorot 初始化。

$$\begin{aligned} &\text{Normal distribution with mean 0 and variance } \sigma^2 = \frac{1}{\text{fan}_{\text{avg}}} \\ &\text{Or a uniform distribution between } -r \text{ and } +r, \text{ with } r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}} \end{aligned}$$

公式 11-1 Xavier 初始化（使用逻辑激活函数）

如果将公式 11-1 中的  $\text{fan}[\text{avg}]$  替换为  $\text{fan}[\text{in}]$ ，就得到了 Yann LeCun 在 1990 年代提出的初始化策略，他称其为 LeCun 初始化。Genevieve Orr 和 Klaus-Robert Müller 在 1998 年出版的书《Neural Networks: Tricks of the Trade (Springer)》中推荐了 LeCun 初始化。当  $\text{fan}[\text{in}] = \text{fan}[\text{out}]$  时，LeCun 初始化等同于 Glorot 初始化。研究者们经历了十多年才意识到初始化策略的重要性。使用 Glorot 初始化可以大大加快训练，这是促成深度学习成功的技术之一。

一些论文针对不同的激活函数提供了类似的策略。这些策略的区别在于方差大小和使用 `fan[avg]` 或 `fan[out]`，如表 11-1 所示。ReLU 激活函数（及其变体，包括简称 ELU 激活）的初始化策略有时称为 He 初始化。本章后面会介绍 SELU 激活函数，它应该与 LeCun 初始化（最好是正态分布）一起使用。

Initialization	Activation functions	$\sigma^2$ (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{avg}$
He	ReLU and variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

表 11-1 每种激活函数的初始化参数

默认情况下，Keras 使用均匀分布的 Glorot 初始化函数。创建层时，可以通过设置 `kernel_initializer="he_uniform"` 或 `kernel_initializer="he_normal"` 变更为 He 初始化，如下所示：

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

如果想让均匀分布的 He 初始化是基于 `fan[avg]` 而不是 `fan[in]`，可以使用 `VarianceScaling` 初始化器：

```
he_avg_init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg',
                                                 distribution='uniform')
keras.layers.Dense(10, activation="sigmoid", kernel_initializer=he_avg_init)
```

## 非饱和激活函数

Glorot 和 Bengio 在 2010 年的论文中的一个见解是，消失/爆炸的梯度问题部分是由于激活函数的选择不好造成的。在那之前，大多数人都认为，如果大自然选择在生物神经元中使用 sigmoid 激活函数，它们必定是一个很好的选择。但事实证明，其他激活函数在深度神经网络中表现得更好，特别是 ReLU 激活函数，主要是因为它对正值不会饱和（也因为它的计算速度很快）。

但是，ReLU 激活功能并不完美。它有一个被称为“ReLU 死区”的问题：在训练过程中，一些神经元会“死亡”，即它们停止输出 0 以外的任何东西。在某些情况下，你可能会发现你网络的一半神经元已经死亡，特别是使用大学习率时。在训练期间，如果神经元的权重得到更新，使得神经元输入的加权和为负，则它将开始输出 0。当这种情况发生时，由于当输入为负时，ReLU 函数的梯度为 0，神经元就只能输出 0 了。

为了解决这个问题，你可能需要使用 ReLU 函数的一个变体，比如 leaky ReLU。这个函数定义为  $\text{LeakyReLU}[\alpha](z) = \max(\alpha z, z)$ （见图 11-2）。超参数  $\alpha$  定义了函数“泄露”的程度：它是  $z < 0$  时函数的斜率，通常设置为 0.01。这个小斜率保证 leaky ReLU 永不死亡；他们可能会长期昏迷，但他们有机会最终醒来。[2015 年的一篇论文](#)比较了几种 ReLU 激活功能的变体，其中一个结论是 leaky Relu 总是优于严格的 ReLU 激活函数。事实上，设定  $\alpha = 0.2$ （大的泄露）似乎比  $\alpha = 0.01$ （小的泄露）有更好的性能。这篇论文还评估了随机化 leaky ReLU (RReLU)，其中  $\alpha$  在训练期间在给定范围内随机，并在测试期间固定为平

均值。它表现相当好，似乎是一个正则项（减少训练集的过拟合风险）。最后，文章还评估了参数化的 leaky ReLU (PReLU)，其中  $\alpha$  被授权在训练期间参与学习（而不是作为超参数， $\alpha$  变成可以像任何其他参数一样被反向传播修改的参数）。据报道，PReLU 在大型图像数据集上的表现强于 ReLU，但是对于较小的数据集，其具有过度拟合训练集的风险。

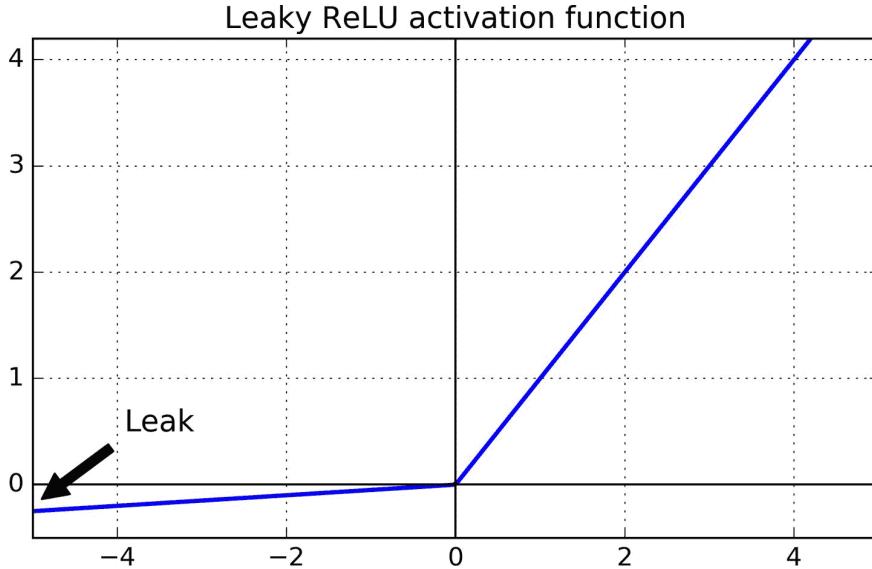


图 11-2 Leaky ReLU：很像 ReLU，但在负区间有小斜率

最后，Djork-Arné Clevert 等人在 [2015 年的一篇论文](#) 中提出了一种称为指数线性单元 (exponential linear unit, ELU) 的新激活函数，在他们的实验中，ELU 的表现优于所有 ReLU 变体：训练时间减少，神经网络在测试集上表现的更好。如图 11-3 所示，公式 11-2 给出了它的定义。

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

公式 11-2 ELU 激活函数

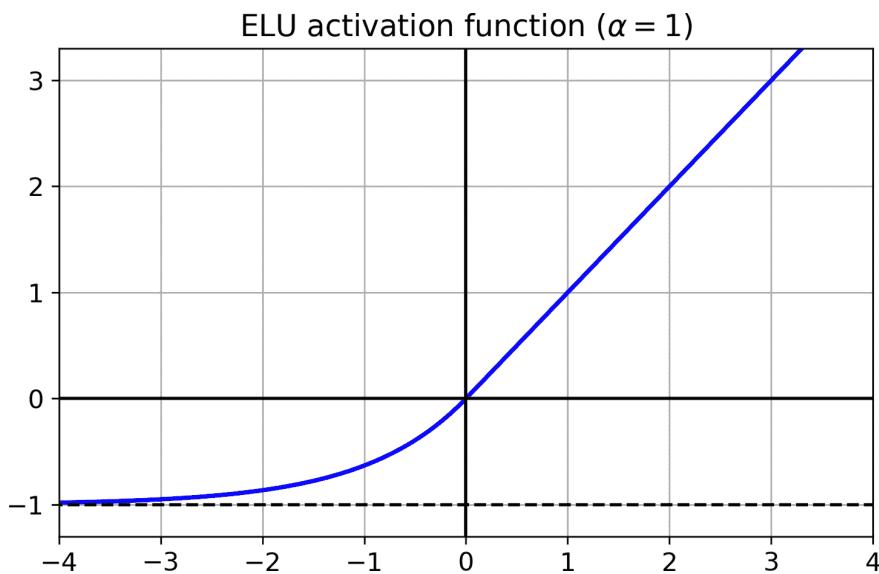


图 11-3 ELU 激活函数

ELU 看起来很像 ReLU 函数，但有一些区别，主要区别在于：

- 它在  $z < 0$  时取负值，这使得该单元的平均输出接近于 0。这有助于减轻梯度消失问题。超参数  $\alpha$  定义为当  $z$  是一个大的负数时，ELU 函数接近的值。它通常设置为 1，但是如果你愿意，你可以像调整其他超参数一样调整它。
- 它对  $z < 0$  有一个非零的梯度，避免了神经元死亡的问题。
- 如果  $\alpha$  等于 1，则函数在任何地方都是平滑的，包括  $z = 0$  附近，这有助于加速梯度下降，因为它不会在  $z = 0$  附近回弹。

ELU 激活函数的主要缺点是计算速度慢于 ReLU 及其变体（由于使用指数函数），但是在训练过程中，这是通过更快的收敛速度来补偿的。然而，在测试时间，ELU 网络将比 ReLU 网络慢。

[2017 年的一篇文章](#)中，Günter Klambauer 等人介绍了一种 Scaled ELU (SELU) 激活函数：正如它的名字所示，它是 ELU 的伸缩变体。作者证明，只要神经网络中都是紧密层，并且所有隐藏层都是用的 SELU 激活函数，则这个网络是自归一的：训练过程中，每层输出的平均值是 0，标准差是 1，这样就解决了梯度消失爆炸问题。对于全紧密层的网络（尤其是很深的），SELU 的效果常常优于其他激活函数。但是自归一是需要条件的（数学论证见论文）：

- 输入特征必须是标准的（平均值是 0，标准差是 1）；
- 每个隐藏层的权重必须是 LeCun 正态初始化的。在 Keras 中，要设置 `kernel_initializer="lecun_normal"`；
- 网络架构必须是顺序的。但是，如果要在非顺序网络（比如 RNN）或有跳连接的网络（跳过层的连接，比如 Wide&Deep）中使用 SELU，就不能保证是自归一的，所以 SELU 就不会比其它激活函数更优；
- 这篇论文只是说如果所有层都是紧密层才保证自归一，但有些研究者发现 SELU 激活函数也可以提高卷积神经网络的性能。

提示：那么深层神经网络的隐藏层应该使用哪个激活函数呢？虽然可能会有所不同，一般来说 SELU > ELU > leaky ReLU (及其变体) > ReLU > tanh > sigmoid。如果网络架构不能保证自归一，则 ELU 可能比 SELU 的性能更好（因为 SELU 在  $z=0$  时不是平滑的）。如果关心运行延迟，则 leaky ReLU 更好。如果你不想多调整另一个超参数，你可以使用前面提到的默认的  $\alpha$  值 (leaky ReLU 为 0.3)。如果有充足的时间和计算能力，可以使用交叉验证来评估其他激活函数，如果神经网络过拟合，则使用 RReLU；如果您拥有庞大的训练数据集，则为 PReLU。但是，因为 ReLU 是目前应用最广的激活函数，许多库和硬件加速器都使用了针对 ReLU 的优化，如果速度是首要的，ReLU 可能仍然是首选。

要使用 leaky ReLU，需要创建一个 `LeakyReLU` 层，并将它加到需要追加的层后面：

```
model = keras.models.Sequential([
    [...]
    keras.layers.Dense(10, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(alpha=0.2),
    [...]
])
```

对于 PReLU，用 `PReLU()` 替换 `LeakyReLU(alpha=0.2)`。目前还没有 RReLU 的 Keras 官方实现，但很容易自己实现（方法见第 12 章的练习）。

对于 SELU，当创建层时设置 `activation="selu"`，`kernel_initializer="lecun_normal"`：

```
layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")
```

## 批归一化 (Batch Normalization)

尽管使用 He 初始化和 ELU (或任何 ReLU 变体) 可以显著减少训练开始阶段的梯度消失/爆炸问题，但不能保证在训练期间问题不会再次出现。

在 2015 年的一篇论文中，Sergey Ioffe 和 Christian Szegedy 提出了一种称为批归一化 (Batch Normalization, BN) 的方法来解决梯度消失/爆炸问题。该方法包括在每层的激活函数之前或之后在模型中添加操作。操作就是将输入平均值变为 0，方差变为 1，然后用两个新参数，一个做缩放，一个做偏移。换句话说，这个操作可以让模型学习到每层输入值的最佳缩放值和平均值。大多数情况下，如果模型的第一层使用了 BN 层，则不用标准化训练集（比如使用 StandardScaler）；BN 层做了标准化工作（虽然是近似的，每次每次只处理一个批次，但能做缩放和平移）。

为了对输入进行零居中（平均值是 0）和归一化，算法需要估计输入的均值和标准差。它通过评估当前小批量输入的均值和标准差（因此命名为“批归一化”）来实现。整个操作在公式 11-3 中。

$$\begin{aligned}
 1. \quad \mu_B &= \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)} \\
 2. \quad \sigma_B^2 &= \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2 \\
 3. \quad \mathbf{x}^{(i)} &= \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\
 4. \quad \mathbf{z}^{(i)} &= \gamma \mathbf{x}^{(i)} + \beta
 \end{aligned}$$

公式 11-3 批归一化算法

其中，

- $\mu_B$  是整个小批量  $B$  的均值向量
- $\sigma_B^2$  是输入标准差向量，也是根据整个小批量估算的。

- $m[B]$  是小批量中的实例数量。
- $x_{\text{hat}}^j$  是以为零中心和标准化的实例  $i$  的输入向量。
- $\gamma$  是层的缩放参数的向量（每个输入一个缩放参数）。
- $\circ$  表示元素级别的相乘（每个输入乘以对应的缩放参数）
- $\beta$  是层的偏移参数（偏移量）向量（每个输入一个偏移参数）
- $\epsilon$  是一个很小的数字，以避免被零除（通常为  $10^{-5}$ ）。这被称为平滑项（拉布拉斯平滑，Laplace Smoothing）。
- $z^i$  是 BN 操作的输出：它是输入的缩放和移位版本。

在训练时，BN 将输入标准化，然后做了缩放和平移。测试时又如何呢？因为需要对实例而不是批次实例做预测，所以就不能计算每个输入的平均和标准差。另外，即使有批量实例，批量也可能太小，或者实例并不是独立同分布的，所以在批量上计算是不可靠的。一种解决方法是等到训练结束，用模型再运行一次训练集，算出每个 BN 层的平均值和标准差。然后就可以用这些数据做预测，而不是批输入的平均值和标准差。但是，大部分批归一化实现是通过层输入的平均值和标准差的移动平均值来计算的。这也是 Keras 在 `BatchNormalization` 中使用的方法。总的来说，每个批归一化的层都通过指数移动平均学习了四个参数： $\gamma$ （输出缩放向量）， $\beta$ （输出偏移向量）， $\mu$ （最终输入平均值向量）和  $\sigma$ （最终输入标准差向量）。 $\mu$  和  $\sigma$  都是在训练过程中计算的，但只在训练后使用（用于替换公式 11-3 中批输入平均和标准差）。

Ioffe 和 Szegedy 证明，批归一化大大改善了他们试验的所有深度神经网络，极大提高了 ImageNet 分类的效果（ImageNet 是一个图片分类数据集，用于评估计算机视觉系统）。梯度消失问题大大减少了，他们可以使用饱和激活函数，如  $\tanh$  甚至逻辑激活函数。网络对权重初始化也不那么敏感。他们能够使用更大的学习率，显著加快了学习过程。具体地，他们指出，“应用于最先进的图像分类模型，批标准减少了 14 倍的训练步骤实现了相同的精度，以显著的优势击败了原始模型。 [...] 使用批量标准化的网络集合，我们改进了 ImageNet 分类上的最佳公布结果：达到 4.9% 的前 5 个验证错误（和 4.8% 的测试错误），超出了人类评估者的准确性。批量标准化也像一个正则化项一样，减少了对其他正则化技术的需求（如本章稍后描述的丢弃）。

然而，批量标准化的确会增加模型的复杂性（尽管它不需要对输入数据进行标准化，因为第一个隐藏层会照顾到这一点，只要它是批量标准化的）。此外，还存在运行时间的损失：由于每层所需的额外计算，神经网络的预测速度较慢。但是，可以在训练之后，处理在 BN 层的前一层，就可以加快速度。方法是更新前一层的权重和偏置项，使其直接输出合适的缩放值和偏移值。例如，如果前一层计算的是  $xw + b$ ，BN 层计算的是  $\gamma \circ (xw + b - \mu) / \sigma + \beta$ （忽略了分母中的平滑项  $\epsilon$ ）。如果定义  $w' = \gamma \circ w / \sigma$  和  $b' = \gamma \circ (b - \mu) / \sigma + \beta$ ，公式就能简化为  $xw' + b'$ 。因此如果替换前一层的权重和偏置项（ $w$  和  $b$ ）为  $w'$  和  $b'$ ，就可以不用 BN 层了（TFLite 的优化器就干了这件事，见第 19 章）。

注意：你可能会发现，训练相当缓慢，这是因为每个周期都因为使用 BN 而延长了时间。但是有了 BN，收敛的速度更快，需要的周期数更少。综合来看，需要的总时长变短了。

## 使用 Keras 实现批归一化

和 Keras 大部分功能一样，实现批归一化既简单又直观。只要每个隐藏层的激活函数前面或后面添加一个 `BatchNormalization` 层就行，也可以将 BN 层作为模型的第一层。例如，这个模型在每个隐藏层的后面使用了 BN，第一层也用了 BN（在打平输入之后）：

```
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")
])
```

这样就成了！在这个只有两个隐藏层的例子中，BN 的作用不会那么大，但对于更深的网络，作用就特别大。

打印一下模型的摘要：

```
>>> model.summary()
Model: "sequential_3"
-----  

Layer (type)          Output Shape         Param #
-----  

flatten_3 (Flatten)   (None, 784)           0  

batch_normalization_v2 (BatchNormalization) (None, 784)       3136  

dense_50 (Dense)      (None, 300)           235500  

batch_normalization_v2_1 (BatchNormalization) (None, 300)       1200  

dense_51 (Dense)      (None, 100)           30100  

batch_normalization_v2_2 (BatchNormalization) (None, 100)       400  

dense_52 (Dense)      (None, 10)            1010  

-----  

Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

可以看到每个 BN 层添加了四个参数： $\gamma$ 、 $\beta$ 、 $\mu$  和  $\sigma$ （例如，第一个 BN 层添加了 3136 个参数，即  $4 \times 784$ ）。后两个参数  $\mu$  和  $\sigma$  是移动平均，不受反向传播影响，Keras 称其“不可训练”（如果将 BN 的总参数  $3,136 + 1,200 + 400$  除以 2，得到 2368，就是模型中总的不可训练的参数量）。

看下第一个 BN 层的参数。两个参数是可训练的（通过反向传播），两个不可训练：

```
>>> [(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)]
```

当在 Keras 中创建一个 BN 层时，训练过程中，还会创建两个 Keras 在迭代时的操作。该操作会更新移动平均值。因为后端使用的是 TensorFlow，这些操作就是 TensorFlow 操作（第 12 章会讨论 TF 操作）：

```
>>> model.layers[1].updates
[<tf.Operation 'cond_2/Identity' type=Identity>,
 <tf.Operation 'cond_3/Identity' type=Identity>]
```

BN 的论文作者建议在激活函数之前使用 BN 层，而不是像前面的例子添加到后面。到底是前面还是后面好存在争议，取决于具体的任务——你最好在数据集上试验一下哪种选择好。要在激活函数前添加 BN 层，必须将激活函数从隐藏层拿出来，单独做成一层。另外，因为 BN 层对每个输入有一个偏移参数，可以将前一层的偏置项去掉（设置 `use_bias=False`）：

```
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` 类可供调节的参数不多。默认值通常就可以，但有时需要调节 `momentum`，这个超参数是 `BatchNormalization` 在更新指数移动平均时使用的。给定一个新值 `v`（即，一个当前批次的输入平均或标准差新向量），BN 层使用下面的等式更新平均 `v_hat`：

$$\hat{v} \leftarrow \hat{v} \times \text{momentum} + v \times (1 - \text{momentum})$$

`momentum` 的最优值通常接近于 1：比如，0.9、0.99、0.999（大数据的 9 更多，小数据集的 9 少）。

另一个重要的超参数是 `axis`：它确定了在哪个轴上归一。默认是 `-1`，即归一化最后一个轴（使用其它轴的平均值和标准差）。当输入是 2D 时（即批的形状是 `[batch size, features]`），也就是说每个输入特征都会根据批次全部实例的平均值和标准差做归一。例如，前面例子的第一个 BN 层会分别对 784 个输入特征的每个特征做归一化（还有缩放和偏移）；因此，BN 层会计算 28 个平均值和 28 个标准差（每列 1 个值，根据每行的所有实例计算），用同样的平均值和标准差归一化给定列的所有像素。还会有 28 个缩放值和 28 个偏移值。如果仍想对 784 个像素独立处理，要设置 `axis=[1, 2]`。

在训练和训练之后，BN 层不会做同样的计算：BN 会使用训练中的批次数据和训练后的最终数据（即移动平均值的最终值）。看看源码中是如何实现的：

```
class BatchNormalization(keras.layers.Layer):
    [...]
    def call(self, inputs, training=None):
        [...]
```

`call()` 方法具体实现了方法，它有一个参数 `training`，默认是 `None`，但 `fit()` 方法在训练中将其设为 1。如果你需要写一个自定义层，要求自定义层在训练和测试中的功能不同，就可以在 `call()` 方法中添加一个参数 `training`，用这个参数决定该计算什么（第 12 张会讨论自定义层）。

`BatchNormalization` 已经成为了深度神经网络中最常使用的层，以至于计算图中经常省略，默认嘉定在每个层后面加一个 BN 层。但是 [Hongyi Zhang 的一篇文章](#) 可能改变了这种做法：通过使用一个新的 `fixed-update (fixup)` 权重初始化方法，作者没有使用 BN，训练了一个非常深的神经网络（多达 10000 层），在复杂图片分类任务上表现惊艳。但这个结论很新，最好还是再等一等，现在还是使用批归一化。

## 梯度裁剪

减少梯度爆炸问题的一种常用技术是在反向传播过程中剪切梯度，使它们不超过某个阈值，这种方法称为梯度裁剪。梯度裁剪在循环神经网络中用的很多，因为循环神经网络中用 BN 很麻烦，参见第 15 章。对于其它类型的网络，BN 就足够了。

在 Keras 中，梯度裁剪只需在创建优化器时设置 `clipvalue` 或 `clipnorm` 参数，如下：

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

优化器会将梯度向量中的每个值裁剪到 -1.0 和 1.0 之间。这意味着损失（对每个可训练参数）的所有偏导数会被裁剪到 -1.0 和 1.0 之间。阈值是一个可以调节的超参数，可能影响到梯度向量的方向。例如，如果原始梯度向量是  $[0.9, 100.0]$ ，它大体指向第二个轴；但在裁剪之后变为  $[0.9, 1.0]$ ，方向就大体指向对角线了。在实际中，梯度裁剪的效果不错。如果想确保梯度裁剪不改变梯度向量的方向，就需要设置 `clipnorm` 靠范数裁剪，这样如果梯度的 L2 范数超过了阈值，就能对整个梯度裁剪。例如，如果设置 `clipnorm = 1.0`，向量  $[0.9, 100.0]$  就会被裁剪为  $[0.00899964, 0.9999595]$ ，方向没变，但第一个量几乎被抹去了。如果再训练过程中发现了梯度爆炸（可以用 TensorBoard 跟踪梯度），最好的方法是既用值也用范数裁剪，设置不同的阈值，看看哪个在验证集上表现最好。

## 复用预训练层

从零开始训练一个非常大的 DNN 通常不是一个好主意，相反，您应该总是尝试找到一个现有的神经网络来完成与您正在尝试解决的任务类似的任务（第 14 章会介绍如何找），然后复用这个网络的较低层：这就是所谓的迁移学习。这样不仅能大大加快训练速度，还将需要更少的训练数据。

例如，假设你有一个经过训练的 DNN，能将图片分为 100 个不同的类别，包括动物，植物，车辆和日常物品。现在想要训练一个 DNN 来对特定类型的车辆进行分类。这些任务非常相似，甚至部分重叠，因此应该尝试重新使用第一个网络的一部分（请参见图 11-4）。

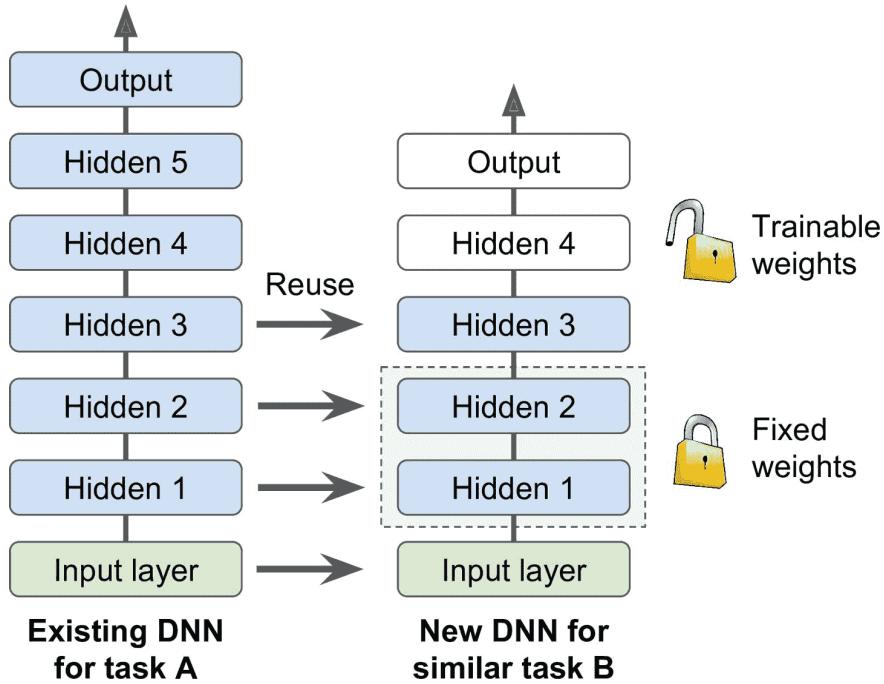


图 11-4 复用预训练层

**笔记：**如果新任务的输入图像与原始任务中使用的输入图像的大小不一致，则必须添加预处理步骤以将其大小调整为原始模型的预期大小。更一般地说，如果输入具有类似的低级层次的特征，则迁移学习将很好地工作。

原始模型的输出层通常要替换掉，因为对于新任务可能一点用也没有，输出的数量可能就不对。相似的，原始模型的上层也不如浅层管用，因为高阶特征可能相差很大。需要确定好到底用几层。

**提示：**任务越相似，可复用的层越多。对于非常相似的任务，可以尝试保留所有的吟唱层，替换输出层。

先将所有复用的层冻结（即，使其权重不可训练，梯度下降不能修改权重），然后训练模型，看其表现如何。然后将复用的最上一或两层解冻，让反向传播可以调节它们，再查看性能有无提升。训练数据越多，可以解冻的层越多。解冻时减小学习率也有帮助，可以避免破坏微调而得的权重。

如果效果不好，或者训练数据不多，可以尝试去除顶层，将其余的层都解冻。不断尝试，直到找到合适的层，如果训练数据很多，可以尝试替换顶层，或者加入更多的隐藏层。

## 用 Keras 进行迁移学习

看一个例子。假设 Fashion MNIST 只有八个类，不包括拖鞋和 T 恤。一些人在这个数据集上搭建并训练了一个 Keras 模型，且效果不错（准确率大于 90%），将其称为模型 A。现在想处理另一个问题：有拖鞋和 T 恤的图片，要训练一个二分类器 (`positive=shirt, negative=sandal`)。数据集不大，只有 200 张打了标签的图片。当训练架构与模型 A 相同的新模型时（称其为模型 B），表现非常好（准确率 97.2%）。但因为这是一个非常简单的任务（只有两类），所以准确率应该还可以更高。因为和任务 A 很像，所以可以尝试一下迁移学习。

首先，加载模型 A，创建一个新模型，除了输出层不要，保留所有的层：

```
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"))
```

`model_A` 和 `model_B_on_A` 公用了一些层。当你训练 `model_B_on_A` 时，也会影响 `model_A`。如果想避免，需要在复用前克隆 `model_A`。要这么做，可以使 `clone.model()`，然后复制权重（`clone.model()` 不能克隆权重）：

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

现在就可以训练 `model_B_on_A` 了，但是因为新输出层是随机初始化的，误差较大，较大的误差梯度可能会破坏复用的权重。为了避免，一种方法是在前几次周期中，冻结复用的层，让新层有时间学到合理的权重。要实现的话，将每层的 `trainable` 属性设为 `False`，然后编译模型：

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                      metrics=["accuracy"],
```

笔记：冻结或解冻模型之后，都需要编译。

训练几个周期之后，就可以解冻复用层（需要再次编译模型），然后接着训练以微调模型。解冻之后，最好降低学习率，目的还是避免破坏复用层的权重：

```
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))
```

最终结果，新模型的测试准确率达到了 99.25%。迁移学习将误差率从 2.8% 降低到了 0.7%，减小了 4 倍！

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.06887910133600235, 0.9925]
```

你相信这个结果吗？不要相信：因为作者作弊了！作者尝试了许多方案，才找到一组配置提升了效果。如果你尝试改变类或随机种子，就能发现效果下降。作者这里做的是“拷问数据，直到数据招供”。当某篇论文的结果太好了，你应该怀疑下：也许新方法实际没什么效果（甚至降低了表现），只是作者尝试了许多变量，只报告了最好的结果（可能只是运气），踩的坑都没说。大部分时候，这不是恶意，但确实是科学中许多结果无法复现的原因。作者为什么要作弊呢？因为迁移学习对小网络帮助不大，小型网络只能学到几个模式，紧密网络学到的具体模式，可能在其他任务中用处不大。迁移学习在深度卷积网络中表现最好，CNN 学到的特征更通用（特别是浅层）。第 14 章会用刚讨论的，回顾迁移学习（下次保证不作弊）。

## 无监督预训练

假设你想要解决一个复杂的任务，但没有多少的打了标签的训练数据，也找不到一个类似的任务训练模型。不要失去希望！首先，应该尝试收集更多的有标签的训练数据，但是如果做不到，仍然可以进行无监督的训练（见图 11-5）。通常，获得无标签的训练数据成本低，但打标签成本很高。如果收集了大量无标签数据，可以尝试训练一个无监督模型，比如自编码器或生成式对抗网络（见第 17 章）。然后可以复用自编码器或 GAN 的浅层，加上输出层，使用监督学习微调网络（使用标签数据）。

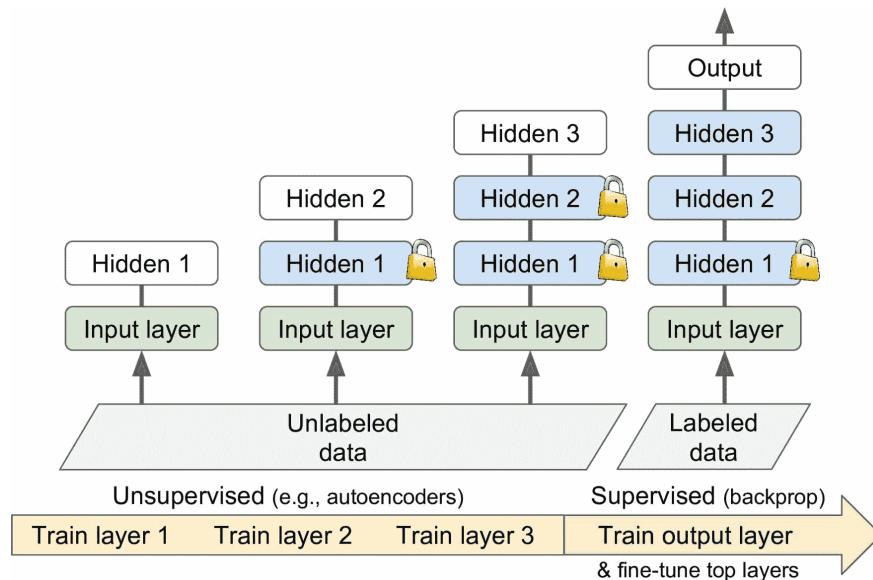


图 11-5 无监督的预训练

这是 Geoffrey Hinton 和他的团队在 2006 年使用的技术，导致了神经网络的复兴和深度学习的成功。直到 2010 年，无监督预训练（通常使用受限玻尔兹曼机 RBM）是深度网络的标准，只有在梯度消失问题得到缓解之后，监督训练 DNN 才更为普遍。然而，当你有一个复杂的任务需要解决时，没有类似的模型可以重复使用，而且标记的训练数据很少，但是大量的未标记的训练数据时，无监督训练（现在通常使用自动编码器、GAN 而不是 RBM）仍然是一个很好的选择。在深度学习的早期，训练深度模型很困难，人们使用了一种逐层预训练的方法（见图 11-5）。先训练一个单层无监督模型，通常是 RBM，然后冻结该层，加另一个层，再训练模型（只训练新层），然后冻住新层，再加一层，再次训练模型。现在变得简单了，直接跳到图 11-5 中的步骤 3，训练完整的无监督模型，使用的是自编码器或 GAN。

## 在辅助任务上预训练

如果没有多少标签训练数据，最后的选择是在辅助任务上训练第一个神经网络，在辅助任务上可以轻松获取或生成标签的训练数据，然后重新使用该网络的较低层来完成实际任务。第一个神经网络的较低层将学习可能被第二个神经网络重复使用的特征检测器。

例如，如果你想建立一个识别面孔的系统，你可能只有几个人的照片 - 显然不足以训练一个好的分类器。收集每个人的数百张照片将是不实际的。但是，您可以在互联网上收集大量随机人员的照片，并训练第一个神经网络来检测两张不同的照片是否属于同一个人。这样的网络将学习面部优秀的特征检测器，所以重复使用它的较低层将允许你使用很少的训练数据来训练一个好的面部分类器。

对于自然语言处理（NLP），可以下载大量文本，然后自动生成标签数据。例如，可以随机遮挡一些词，然后训练一个模型预测缺失词。如果能在这个任务上训练一个表现不错的模型，则该模型已经在语言层面学到不少了，就可以复用它到实际任务中，再用标签数据微调（第 15 章会讨论更多预训练任务）。

**笔记：**自监督学习是当你从数据自动生成标签，然后在标签数据上使用监督学习训练模型。因为这种方法无需人工标注，最好将其分类为无监督学习。

## 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，我们已经看到了四种加速训练的方法（并且达到更好性能的方法）：对连接权重应用良好的初始化策略，使用良好的激活函数，使用批归一化以及重用预训练网络的部分（使用辅助任务或无监督学习）。另一个速度提升的方法是使用更快的优化器，而不是常规的梯度下降优化器。在本节中，我们将介绍最流行的算法：动量优化，Nesterov 加速梯度，AdaGrad，RMSProp，最后是 Adam 和 Nadam 优化。

剧透：本节的结论是，几乎总是应该使用 `Adam_optimizer`，所以如果不关心它是如何工作的，只需使用 `AdamOptimizer` 替换 `GradientDescentOptimizer`，然后跳到下一节！只需要这么小的改动，训练通常会快几倍。但是，Adam 优化确实有三个可以调整的超参数（加上学习率）。默认值通常工作的不错，但如果您需要调整它们，知道他们怎么实现的可能会有帮助。Adam 优化结合了来自其他优化算法的几个想法，所以先看看这些算法是有用的。

## 动量优化

想象一下，一个保龄球在一个光滑的表面上平缓的斜坡上滚动：它会缓慢地开始，但是它会很快地达到最终的速度（如果有一些摩擦或空气阻力的话）。这是 Boris Polyak 在 1964 年提出的动量优化背后的一个非常简单的想法。相比之下，普通的梯度下降只需要沿着斜坡进行小的有规律的下降步骤，所以需要更多的时间才能到达底部。

回想一下，梯度下降只是通过直接减去损失函数  $J(\theta)$  相对于权重  $\theta$  的梯度 ( $\nabla_{\theta} J(\theta)$ )，乘以学习率  $\eta$  来更新权重  $\theta$ 。等式是： $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta)$ 。它不关心早期的梯度是什么。如果局部梯度很小，则会非常缓慢。

动量优化很关心以前的梯度：在每次迭代时，它将动量向量  $m$ （乘以学习率  $\eta$ ）与局部梯度相加，并且通过简单地减去该动量向量来更新权重（参见公式 11-4）。换句话说，梯度用作加速度，不用作速度。为了模拟某种摩擦机制，避免动量过大，该算法引入了一个新的超参数  $\beta$ ，简称为动量，它必须设置在 0（高摩擦）和 1（无摩擦）之间。典型的动量值是 0.9。

1.  $m \leftarrow \beta m + \eta \nabla_{\theta} J(\theta)$
2.  $\theta \leftarrow \theta - m$

公式 11-4 动量算法

可以很容易验证，如果梯度保持不变，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  $\eta$  乘以  $1/(1-\beta)$ 。例如，如果  $\beta = 0.9$ ，则最终速度等于学习率的梯度乘以 10 倍，因此动量优化比梯度下降快 10 倍！这使动量优化比梯度下降快得多。特别是，我们在第四章中看到，当输入量具有非常不同的尺度时，损失函数看起来像一个细长的碗（见图 4-7）。梯度下降速度很快，但要花很长的时间才能到达底部。相反，动量优化会越来越快地滚下山谷底部，直到到达底部（最佳）。在不使用批归一化的深度神经网络中，较高层往往能得到具有不同的尺度的输入，所以使用动量优化会有很大的帮助。它也可以帮助滚过局部最优点。

笔记：由于动量的原因，优化器可能会超调一些，然后再回来，再次超调，并在稳定在最小值之前多次振荡。这就是为什么在系统中有一点摩擦的原因之一：它消除了这些振荡，从而加速了收敛。

在 Keras 中实现动量优化很简单：只需使用 SGD 优化器，设置 `momentum` 超参数，然后就可以躺下赚钱了！

```
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。然而，0.9 的动量值通常在实践中运行良好，几乎总是比梯度下降快。

## Nesterov 加速梯度

Yurii Nesterov 在 1983 年提出的动量优化的一个小变体几乎总是比普通的动量优化更快。Nesterov 动量优化或 Nesterov 加速梯度（Nesterov Accelerated Gradient, NAG）的思想是测量损失函数的梯度不是在局部位置，而是在动量方向稍微靠前（见公式 11-5）。与普通的动量优化的唯一区别在于梯度是在  $\theta + \beta m$  而不是在  $\theta$  处测量的。

1.  $m \leftarrow \beta m + \eta \nabla_{\theta} J(\theta + \beta m)$
2.  $\theta \leftarrow \theta - m$

公式 11-5 Nesterov 加速梯度算法

这个小小的调整是可行的，因为一般来说，动量向量将指向正确的方向（即朝向最优方向），所以使用在该方向上测得的梯度稍微更精确，而不是使用原始位置的梯度，如图 11-6 所示（其中  $v_1$  代表在起点  $\theta$  处测量的损失函数的梯度， $v_2$  代表位于  $\theta + \beta m$  的点处的梯度）。

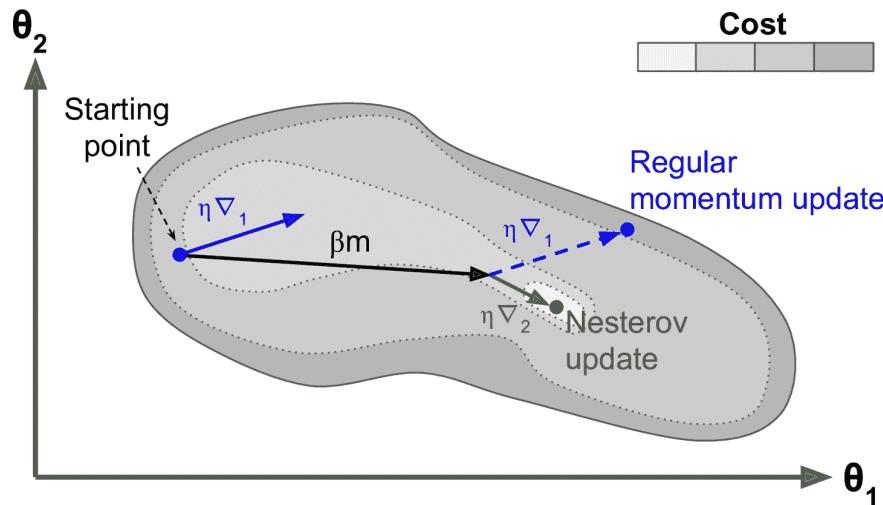


图 11-6 常规 vs Nesterov 动量优化

可以看到，Nesterov 更新稍微靠近最佳值。过了一段时间，这些小的改进加起来，NAG 最终比常规的动量优化快得多。此外，当动量推动权重横跨山谷时， $v_1$  继续推进越过山谷，而  $v_2$  推回山谷的底部。这有助于减少振荡，从而更快地收敛。

与常规的动量优化相比，NAG 几乎总能加速训练。要使用它，只需在创建 SGD 时设置 `nesterov=True`：

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

## AdaGrad

再次考虑细长碗的问题：梯度下降从最陡峭的斜坡快速下降，然后缓慢地下到谷底。如果算法能够早期检测到这个问题并且纠正它的方向来指向全局最优点，那将是非常好的。AdaGrad 算法通过沿着最陡的维度缩小梯度向量来实现这一点（见公式 11-6）：

1.  $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2.  $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$

公式 11-6 AdaGrad 算法

第一步将梯度的平方累加到向量  $s$  中 ( $\otimes$  符号表示元素级别相乘)。这个向量化形式相当于向量  $s$  的每个元素  $s[i]$  计算  $s[i] \leftarrow s[i] + (\partial J(\theta) / \partial \theta[i])^2$ 。换一种说法，每个  $s[i]$  累加损失函数对参数  $\theta[i]$  的偏导数的平方。如果损失函数沿着第  $i$  维陡峭，则在每次迭代时， $s[i]$  将变得越来越大。

第二步几乎与梯度下降相同，但有一个很大的不同：梯度向量按比例  $(s+\epsilon)^{0.5}$  缩小 ( $\oslash$  符号表示元素分割， $\epsilon$  是避免被零除的平滑项，通常设置为  $10^{-10}$ )。这个向量化的形式相当于所有  $\theta[i]$  同时计算

$$\theta_i \leftarrow \theta_i - \eta \partial / \partial \theta_i J(\theta) / \sqrt{s_i + \epsilon}$$

简而言之，这种算法会降低学习速度，但对于陡峭的维度，其速度要快于具有温和的斜率的维度。这被称为自适应学习率。它有助于将更新的结果更直接地指向全局最优（见图 11-7）。另一个好处是它不需要那么多的去调整学习率超参数  $\eta$ 。

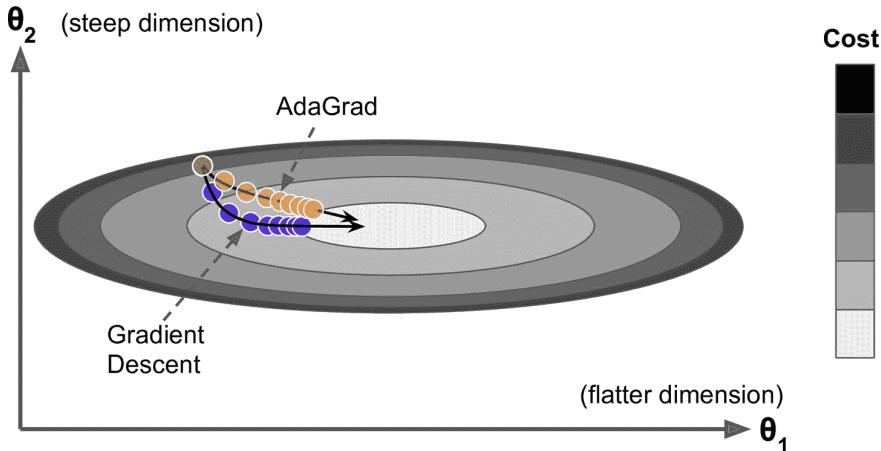


图 11-7 AdaGrad vs 梯度下降

对于简单的二次问题，AdaGrad 经常表现良好，但不幸的是，在训练神经网络时，它经常停止得太早。学习率被缩减得太多，以至于在达到全局最优之前，算法完全停止。所以，即使 Keras 有一个 Adagrad 优化器，你也不应该用它来训练深度神经网络（虽然对线性回归这样简单的任务可能是有效的）。但是，理解 AdaGrad 对掌握其它自适应学习率还是很有帮助的。

## RMSProp

前面看到，AdaGrad 的风险是降速太快，可能无法收敛到全局最优。RMSProp 算法通过仅累积最近迭代（而不是从训练开始以来的所有梯度）的梯度来修正这个问题。它通过在第一步中使用指数衰减来实现（见公式 11-7）。

1.  $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2.  $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$

公式 11-7 RMSProp 算法

它的衰变率  $\beta$  通常设定为 0.9。是的，它又是一个新的超参数，但是这个默认值通常运行良好，所以你可能根本不需要调整它。

正如所料，Keras 拥有一个 RMSProp 优化器：

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

除了非常简单的问题，这个优化器几乎总是比 AdaGrad 执行得更好。它通常也比动量优化和 Nesterov 加速梯度表现更好。事实上，这是许多研究人员首选的优化算法，直到 Adam 优化出现。

## Adam 和 Nadam 优化

Adam，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，它追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（见方程式 11-8）。

1.  $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta)$
2.  $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
3.  $\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^T}$
4.  $\widehat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^T}$
5.  $\theta \leftarrow \theta + \eta \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \epsilon}$

#### 公式 11-8 Adam 算法

$T$  代表迭代次数（从 1 开始）。

如果你只看步骤 1, 2 和 5，你会注意到 Adam 与动量优化和 RMSProp 的相似性。唯一的区别是第 1 步计算指数衰减的平均值，而不是指数衰减的和，但除了一个常数因子（衰减平均值只是衰减和的  $1 - \beta_1$  倍）之外，它们实际上是等效的。步骤 3 和步骤 4 是一个技术细节：由于  $m$  和  $s$  初始化为 0，所以在训练开始时它们会偏向 0，所以这两步将在训练开始时帮助提高  $m$  和  $s$ 。

动量衰减超参数  $\beta_1$  通常初始化为 0.9，而缩放衰减超参数  $\beta_2$  通常初始化为 0.999。如前所述，平滑项  $\epsilon$  通常被初始化为一个很小的数，例如  $10^{-7}$ 。这些是 TensorFlow 的 Adam 类的默认值（更具体地， $\epsilon$  默认为 None，Keras 将使用 `keras.backend.epsilon()`，默认为  $10^{-7}$ ，可以通过 `keras.backend.set_epsilon()` 更改），所以你可以简单地使用：

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

实际上，由于 Adam 是一种自适应学习率算法（如 AdaGrad 和 RMSProp），所以对学习率超参数  $\eta$  的调整较少。您经常可以使用默认值  $\eta=0.001$ ，使 Adam 相对于梯度下降更容易使用。

提示：如果读者对这些不同的技术感到头晕脑胀，不用担心，本章末尾会提供一些指导。

最后，Adam 还有两种变体值得一看：

#### AdaMax

公式 11-8 的第 2 步中，Adam 积累了  $s$  的梯度平方（越近，权重越高）。第 5 步中，如果忽略了  $\epsilon$ 、第 3 步和第 4 步（只是技术细节而已），Adam 是通过  $s$  的平方根更新参数。总之，Adam 通过时间损耗梯度的  $L_2$  范数更新参数（ $L_2$  范数是平方和的平方根）。AdaMax（也是在 Adam 的同一篇论文中介绍的）用  $L_\infty$  范数（max 的另一种说法）代替了  $L_2$  范数。更具体的，是在第 2 步中做了替换，舍弃了第 4 步，第 5 步中用  $s$ （即时间损耗的最大值）更新梯度。在实践中，这样可以使 AdaMax 比 Adam 更稳定，但也要取决于数据集，总体上，Adam 表现更好。因此，AdaMax 只是 Adam 碰到问题时的另一种选择。

### Nadam

Nadam 优化是 Adam 优化加上了 Nesterov 技巧，所以通常比 Adam 收敛的快一点。在[论文](#)中，作者 Timothy Dozat 在不同任务上试验了不同的优化器，发现 Nadam 通常比 Adam 效果好，但有时不如 RMSProp。

警告：自适应优化方法（包括 RMSProp, Adam, Nadam）总体不错，收敛更快。但是 Ashia C. Wilson 在 2017 年的一篇[论文](#)中说，这些自适应优化方法在有些数据集上泛化很差。所以当你对模型失望时，可以尝试下普通的 Nesterov 加速梯度：你的数据集可能只是对自适应梯度敏感。另外要调研最新的研究进展，因为这个领域进展很快。

目前所有讨论的优化方法都是基于一阶偏导（雅可比矩阵）的。文献中还介绍了基于二阶导数（黑森矩阵，黑森矩阵是雅可比矩阵的偏导）的算法。但是，后者很难应用于深度神经网络，因为每个输出有  $n^2$  个黑森矩阵（ $n$  是参数个数），每个输出只有  $n$  个雅可比矩阵。因为 DNN 通常有数万个参数，二阶优化器通常超出了内存，就算内存能装下，计算黑森矩阵也非常慢。

训练稀疏模型 所有刚刚提出的优化算法都会产生紧密模型，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，或者如果你需要它占用较少的内存，你可能更喜欢用一个稀疏模型来代替。实现这一点的一个微不足道的方法是像平常一样训练模型，然后丢掉微小的权重（将它们设置为 0）。但这通常不会生成一个稀疏的模型，而且可能使模型性能下降。更好的选择是在训练过程中应用强  $\ell_1$  正则化，因为它会推动优化器尽可能多地消除权重（如第 4 章关于 Lasso 回归的讨论）。如果这些技术可能仍然不成，就查看 [TensorFlow Model Optimization Toolkit \(TF-MOT\)](#)，它提供了一些剪枝 API，可以在训练中根据量级迭代去除权重。

表 11-2 比较了讨论过的优化器（\* 是差， \*\* 是平均， \*\*\* 是好）。

Class	Convergence speed	Convergence quality
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	***
Adagrad	***	* (stops too early)
RMSprop	***	** or ***
Adam	***	** or ***
Nadam	***	** or ***
AdaMax	***	** or ***

表 11-2 优化器比较

## 学习率调整

找到一个好的学习速率非常重要。如果设置太高，训练时可能离散。如果设置得太低，训练最终会收敛到最佳状态，但会花费很长时间。如果将其设置得稍高，开始的进度会非常快，但最终会在最优解周围跳动，永远不会停下来。如果计算资源有

限，可能需要打断训练，在最优收敛之前拿到一个次优解（见图 11-8）。

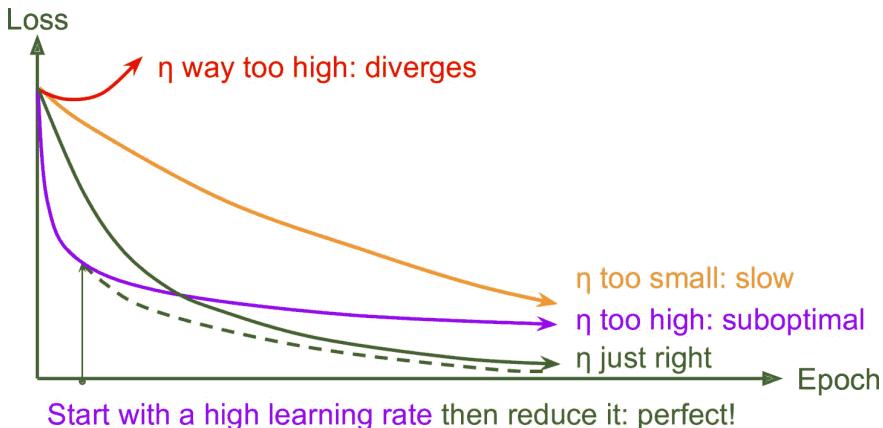


图 11-8 不同学习速率的学习曲线

正如第 10 章讨论过的，可以通过几百次迭代找到一个好的学习率，学习率一开始设的很小，然后指数级提高，查看学习曲线，找到那条要开始抬高的曲线，要找的学习率比这条曲线稍低。

但除了固定学习率，还有更好的方法：如果你从一个高的学习率开始，然后一旦它停止快速的进步就减少它，你可以比最佳的恒定学习率更快地达到一个好的解决方案。有许多不同的策略，以减少训练期间的学习率。这些策略被称为学习率调整（我们在第 4 章中简要介绍了这个概念），其中最常见的是：

**幂调度：**设学习率为迭代次数  $t$  的函数： $\eta(t) = \eta[0] (1 + t/s)^{-c}$ 。初始学习率  $\eta[0]$ ，幂  $c$ （通常被设置为 1），步数  $s$  是超参数。学习率在每步都会下降， $s$  步后，下降到  $\eta[0]/2$ 。再经过  $s$  步，下降到  $\eta[0]/3$ ，然后是  $\eta[0]/4$ 、 $\eta[0]/5$ ，以此类推。可以看到，策略是一开始很快，然后越来越慢。幂调度需要调节  $\eta[0]$  和  $s$ （也可能有  $c$ ）。

**指数调度：**将学习率设置为迭代次数  $t$  的函数： $\eta(t) = \eta[0] 0.1^{(t/s)}$ 。学习率每步都会下降 10 倍。幂调度的下降是越来越慢，指数调度保持 10 倍不变。

**预定的分段恒定学习率：**先在几个周期内使用固定的学习率（比如 5 个周期内学习率设置为  $\eta[0] = 0.1$ ），然后在另一个周期内设更小的学习率（比如 50 个周期  $\eta[0] = 0.001$ ），以此类推。虽然这个解决方案可以很好地工作，但是通常需要弄清楚正确的学习速度顺序以及使用时长。

**性能调度：**每  $N$  步测量验证误差（就像提前停止一样），当误差下降时，将学习率降低  $\lambda$  倍。

**1 循环调度：**与其它方法相反，1 循环调度（Leslie Smith 在 2018 年提出）一开始在前半个周期将学习率  $\eta[0]$  线性增加到  $\eta[1]$ ，然后在后半个周期内再线性下降到  $\eta[0]$ ，最后几个周期学习率下降几个数量级（仍然是线性的）。用前面的方法找到最优学习率的方法确定  $\eta[1]$ ， $\eta[0]$  是  $\eta[1]$  的十分之一。当使用动量时，先用一个高动量（比如 0.95），然后在训练上半段下降（比如线性下降到 0.85），然后在训练后半部分上升到最高值（0.95），最后几个周期也用最高值完成。Smith 做了许多试验，证明这个方法可以显著加速并能提高性能。例如，在 CIFAR10 图片数据集上，这个方法在 100 个周期就达到了 91.9% 的验证准确率，而标准方法经过 800 个周期才打到 90.3%（模型架构不变）。

Andrew Senior 等人在 2013 年的论文比较了使用动量优化训练深度神经网络进行语音识别时一些最流行的学习率调整的性能。作者得出结论：在这种情况下，性能调度和指数调度都表现良好，但他们更喜欢指数调度，因为它实现起来比较简单，容易调整，收敛速度略快于最佳解决方案。作者还之处，1 周期表现更好。

使用 Keras 实现学习率幂调整非常简单，只要在优化器中设定 decay 超参数：

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

decay 是 s (更新学习率的步骤数)，Keras 假定 c 等于 1。

指数调度和分段恒定学习率也很简单。首先定义一个函数接受当前周期，然后返回学习率。例如，如下实现指数调度：

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

如果不想硬实现  $\eta[1]$  和 s，可以实现一个函数返回配置函数：

```
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)
```

然后，创建一个 `LearningRateScheduler` 调回，给它一个调度函数，然后将调回传递给 `fit()`：

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

`LearningRateScheduler` 会在每个周期开始时更新优化器的 `learning_rate` 属性。每个周期更新一次学习率就够了，但如果想更新更频繁，例如每步都更新，可以通过写调回实现（看前面指数调回的例子）。如果每个周期有许多步，每步都更新学习率是非常合理的。或者，可以使用 `keras.optimizers.schedules` 方法。

调度函数可以将当前学习率作为第二个参数。例如，下面的调度函数将之前的学习率乘以  $0.1^{(1/20)}$ ，同样实现了指数下降：

```
def exponential_decay_fn(epoch, lr):
    return lr * 0.1**(1 / 20)
```

该实现依靠优化器的初始学习率（与前面的实现相反），所以一定要设置对。

当保存模型时，优化器和学习率也能保存。这意味着，只要有这个新的调度函数，就能加载模型接着训练。如果调度函数使用了周期，会稍微麻烦点：周期不会保存，每次调用 `fit()` 方法时，周期都会重置为 0。如果加载模型接着训练，可能会导致学习率很大，会破坏模型的权重。一种应对方法是手动设置 `fit()` 方法的参数 `initial_epoch`，是周期从正确的值开始。

对于分段恒定学习率调度，可以使用如下的调度函数，然后创建一个 `LearningRateScheduler` 调回，传递给 `fit()` 方法：

```
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，使用 `ReduceLROnPlateau` 调回。例如，如果将下面的调回去传递给 `fit()`，只要验证损失在连续 5 个周期内没有改进，就会将学习率乘以 0.5：

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
```

最后，`tf.keras` 还提供了一种实现学习率调度的方法：使用 `keras.optimizers.schedules` 中一种可用的调度定义学习率。这样可以在每步更新学习率。例如，还可以如下实现前面的函数 `exponential_decay_fn()`：

```
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)
```

这样又好看又简单，另外当保存模型时，学习率和调度（包括状态）也能保存。但是这个方法不属于 Keras API，是 `tf.keras` 专有的。

对于 1 循环调度，实现也不困难：只需创建一个在每个迭代修改学习率的自定义调回（通过更改 `self.model.optimizer.lr` 更新学习率）。代码见 Jupyter 笔记本的例子。

总结一下，指数调度、性能调度和 1 循环调度可以极大加快收敛，不妨一试！

## 通过正则化避免过拟合

有四个参数，我可以拟合一个大象，五个我可以让他摆动他的象鼻。—— John von Neumann,cited by Enrico Fermi in Nature 427

有数千个参数，甚至可以拟合整个动物园。深度神经网络通常具有数以万计的参数，有时甚至是数百万。有了这么多的参数，网络拥有难以置信的自由度，可以适应各种复杂的数据集。但是这个很大的灵活性也意味着它很容易过拟合训练集。所以需要正则。第 10 章用过了最好的正则方法之一：早停。另外，虽然批归一化是用来解决梯度不稳定的，但也可以作为正则器。这一节会介绍其它一些最流行的神经网络正则化技术： $\ell_1$  和  $\ell_2$  正则、丢弃和最大范数正则。

### $\ell_1$ 和 $\ell_2$ 正则

就像第 4 章中对简单线性模型所做的那样，可以使用  $\ell_2$  正则约束一个神经网络的连接权重，或  $\ell_1$  正则得到稀疏模型（许多权重为 0）。下面是对 Keras 的连接权重设置  $\ell_2$  正则，正则因子是 0.01：

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

$\ell_2$  函数返回的正则器会在训练中的每步被调用，以计算正则损失。正则损失随后被添加到最终损失。如果要使用  $\ell_1$  正则，可以使用 `keras.regularizers.l1()`；如果想使用  $\ell_1$  和  $\ell_2$  正则，可以使用 `keras.regularizers.l1_l2()`（要设置两个正则因子）。

因为想对模型中的所有层使用相同的正则器，还要使用相同的激活函数和相同的初始化策略。参数重复使代码很难看。为了好看，可以用循环重构代码。另一种方法是使用 Python 的函数 `functools.partial()`，它可以为任意可调回对象创建封装类，并有默认参数值：

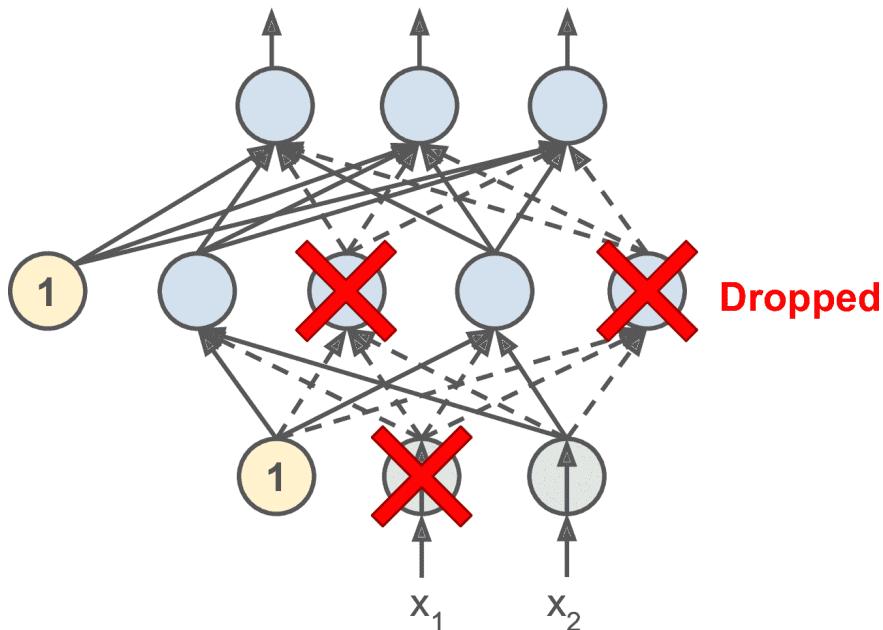
```
from functools import partial
RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

## 丢弃

丢弃是深度神经网络最流行的正则化方法之一。它由 Geoffrey Hinton 于 2012 年提出，并在 Nitish Srivastava 等人的 2014 年论文中进一步详细描述，并且已被证明是非常成功的：即使是最先进的神经网络，仅仅通过增加丢弃就可以提高 1-2% 的准确度。这听起来可能不是很多，但是当一个模型已经具有 95% 的准确率时，获得 2% 的准确度提升意味着将误差率降低近 40%（从 5% 误差降至大约 3%）。

这是一个相当简单的算法：在每个训练步骤中，每个神经元（包括输入神经元，但不包括输出神经元）都有一个暂时“丢弃”的概率  $p$ ，这意味着在这个训练步骤中它将被完全忽略，在下一步可能会激活（见图 11-9）。超参数  $p$  称为丢弃率，通常设为 10% 到 50% 之间；循环神经网络之间接近 20-30%，在卷积网络中接近 40-50%。训练后，神经元不会再丢失。这就是全部（除了我们将要讨论的技术细节）。



### 图 11-9 丢弃正则化

这个具有破坏性的方法竟然行得通，这是相当令人惊讶的。如果一个公司的员工每天早上被告知要掷硬币来决定是否上班，公司的表现会不会更好呢？那么，谁知道；也许会！公司显然将被迫适应这样的组织构架；它不能依靠任何一个人操作咖啡机或执行任何其他关键任务，所以这个专业知识将不得不分散在几个人身上。员工必须学会与其他的许多同事合作，而不仅仅是其中的一小部分。该公司将变得更有弹性。如果一个人离开了，并没有什么区别。目前还不清楚这个想法是否真的可以在公司实行，但它确实对于神经网络是可行的。神经元被丢弃训练不能与其相邻的神经元共适应；他们必须尽可能让自己变得有用。他们也不能过分依赖一些输入神经元；他们必须注意他们的每个输入神经元。他们最终对输入的微小变化会不太敏感。最后，你会得到一个更稳定的网络，泛化能力更强。

了解丢弃的另一种方法是认识到每个训练步骤都会产生一个独特的神经网络。由于每个神经元可以存在或不存在，总共有  $2^N$  个可能的网络（其中 N 是可丢弃神经元的总数）。这是一个巨大的数字，实际上不可能对同一个神经网络进行两次采样。一旦你运行了 10,000 个训练步骤，你基本上已经训练了 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，但是它们都是不同的。由此产生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。

**提示：**在实际中，可以只将丢弃应用到最上面的一到三层（包括输出层）。

有一个小而重要的技术细节。假设  $p = 50\%$ ，在这种情况下，在测试期间，在训练期间神经元将被连接到两倍于（平均）的输入神经元。为了弥补这个事实，我们需要在训练之后将每个神经元的输入连接权重乘以 0.5。如果我们不这样做，每个神经元的总输入信号大概是网络训练的两倍，这不太可能表现良好。更一般地说，我们需要将每个输入连接权重乘以训练后的保持概率 ( $1-p$ )。或者，我们可以在训练过程中将每个神经元的输出除以保持概率（这些替代方案并不完全等价，但它们工作得同样好）。

要使用 Keras 实现丢弃，可以使用 `keras.layers.Dropout` 层。在训练过程中，它随机丢弃一些输入（将它们设置为 0），并用保留概率来划分剩余输入。训练结束后，这个函数什么都不做，只是将输入传给下一层。下面的代码将丢弃正则化应用于每个紧密层之前，丢弃率为 0.2：

```
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")
])
```

**警告：**因为丢弃只在训练时有用，比较训练损失和验证损失会产生误导。特别地，一个模型可能过拟合训练集，但训练和验证损失相近。因此一定要不要带丢弃评估训练损失（比如训练后）。

如果观察到模型过拟合，则可以增加丢弃率（即，减少 `keep_prob` 超参数）。相反，如果模型欠拟合训练集，则应尝试降低丢弃率（即增加 `keep_prob`）。它也可以帮助增加大层的丢弃率，并减少小层的丢弃率。另外，许多优秀的架构只在最后一个隐藏层之后使用丢弃，如果全都加上丢弃太强了，可以这么试试。

丢弃似乎减缓了收敛速度，但通常会在调参得当时使模型更好。所以，这通常值得花费额外的时间和精力。

**提示：**如果想对一个自归一化的基于 SELU 的网络使用正则，应该使用 alpha 丢弃：这是一个丢弃的变体，可以保留输入的平均值和标准差（它是在 SELU 的论文中提出的，因为常规的丢弃会破会自归一化）。

## 蒙特卡洛 (MC) 丢弃

Yarin Gal 和 Zoubin Ghahramani 在 [2016 的一篇论文](#) 中，追加了几个使用丢弃的理由：

- 首先，这篇论文对丢弃网络（每个权重层前都有一个丢弃层）和贝叶斯推断建立了理论联系，从数学角度给予了证明。
- 第二，作者介绍了一种称为 MC 丢弃的方法，它可以提升任何训练过的丢弃模型的性能，并且无需重新训练或修改，对模型存在的不确定性提供了一种更好的方法，也很容易实现。

如果这听起来像一个广告，看下面的代码。它是 MC 丢弃的完整实现，可以提升前面训练的模型，并且没有重新训练：

```
y_probas = np.stack([model(X_test_scaled, training=True)
                      for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

我们只是在训练集上做了 100 次预测，设置 `training=True` 保证丢弃是活跃的，然后放到一起。因为丢弃是开启的，所有的预测都会不同。`predict()` 返回一个矩阵，每行包含一个实例，每列是一个类。因为测试集有 10000 个实例和 10 个类，这个矩阵的形状是 `[10000, 10]`。我们一共有 100 个这样的矩阵，因此 `y_proba` 是一个形状 `[100, 10000, 10]` 的数组。当对以一个维度做平均时，得到的是 `y_proba`，形状是 `[10000, 10]` 的数组，就像和一次独立预测的一样。对开启丢弃的多次预测做平均，就得到了一个蒙特卡洛估计，会比单独一次预测的可靠性更高。例如，看下模型对训练集第一个实例的预测，关闭丢弃：

```
>>> np.round(model.predict(X_test_scaled[:1]), 2)
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.99]],
```

这个模型大概率认定这张图属于类 9（靴子）。应该相信这个结果吗？有无质疑空间呢？

再看看开启丢弃的预测：

```
>>> np.round(y_probas[:, :1], 2)
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.68],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.64],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.97],
       [...]])
```

当开启丢弃，模型就没那么确定了。虽然仍偏向类 9，但会在类 5（凉鞋）和类 7（运动鞋）犹豫。对第一维做平均，我们得到了下面的 MC 丢弃预测：

```
>>> np.round(y_proba[:, 1], 2)
array([[0., 0., 0., 0., 0.22, 0., 0.16, 0., 0.62]],
```

模型仍认为这张图属于类 9，但置信度只有 62%，这比 99% 可信度了。知道可能属于其它什么类，也有用。还可以再查看下概率估计的标准差：

```
>>> y_std = y_probas.std(axis=0)
>>> np.round(y_std[:1], 2)
array([0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01], dtype=float32)
```

显然，概率估计的方差很大：如果搭建的是一个对风险敏感的系统（比如医疗或金融），就要对这样不确定的预测保持谨慎。另外，模型的准确率从 86.8 提升到了 86.9：

```
>>> accuracy = np.sum(y_pred == y_test) / len(y_test)
>>> accuracy
0.8694
```

**笔记：**蒙特卡洛样本的数量是一个可以调节的超参数。这个数越高，预测和不准确度的估计越高。但是，如果样本数翻倍，推断时间也要翻倍。另外，样本数超过一定数量，提升就不大了。因此要取决于任务本身，在延迟和准确性上做取舍。

如果模型包含其它层行为特殊的层（比如批归一化层），则不能像刚才那样强行训练模型。相反，你需要将 `Dropout` 层替换为 `MCDropout` 类：

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

这里，使用了 `Dropout` 的子类，并覆盖了方法 `call()`，使 `training` 参数变为 `True`（见第 12 章）。相似的，可以通过 `AlphaDropout` 的子类定义一个 `MCAlphaDropout`。如果是从零搭建模型，只需使用 `MCDropout` 而不是 `Dropout`，你需要创建一个与老模型架构相同的新模型，替换 `Dropout` 层为 `MCDropout` 层，然后复制权重到新模型上。

总之，MC 丢弃是一个可以提升丢弃模型、提供更加不准确估计的神奇方法。当然，因为在训练中仍然是常规丢弃，它仍然是一个正则器。

## 最大范数正则化

另一种在神经网络中非常流行的正则化技术被称为最大范数正则化：对于每个神经元，它约束输入连接的权重 `w`，使得  $\|w\|_2 < r$ ，其中 `r` 是最大范数超参数， $\|\cdot\|_2$  是 L2 范数。

最大范数正则没有添加正则损失项到总损失函数中。相反，只是计算我们通常通过在每个训练步骤之后计算  $\|w\|_2$ ，并且如果需要的话可以如下剪切 `w`。

$$W \leftarrow W \frac{r}{\|w\|_2}$$

减少 `r` 增加了正则化的量，并有助于减少过拟合。最大范数正则化还可以帮助减轻梯度消失/爆炸问题（如果不使用批归一化）。

要在 Keras 中实现最大范数正则，需要设置每个隐藏层的 `kernel_constraint` 的 `max_norm()` 为一个合适的值，如下所示：

```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",
    kernel_constraint=keras.constraints.max_norm(1.))
```

每次训练迭代之后，模型的 `fit()` 方法会调用 `max_norm()` 返回的对象，传给它层的权重，并返回缩放过的权重，再代替层的权重。第 12 章会看到，如果需要的话可以定义自己的约束函数。你还可以通过设置参数 `bias_constraint` 约束偏置项。

`max_norm()` 函数有一个参数 `axis`，默认为 0。紧密层权重的形状通常是[输入数，神经元数]，因此设置 `axis=0`，意味最大范数约束会独立作用在每个神经元的权重向量上。如果你想对卷积层使用最大范数，一定要合理设置 `axis`（通常 `axis=[0,1,2]`）。

## 总结和实践原则

本章介绍了许多方法，读者可能纳闷到底该用哪个呢。用哪种方法要取决于任务，并没有统一的结论，表 11-3 的总结可用于大多数情况，不需要调节太多超参数。但是，也不要死守这些默认值！

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping ( $+\ell_2$ reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

表 11-3 默认 DNN 配置

如果网络只有紧密层，则可以是自归一化的，可以使用表 11-4 的配置。

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

表 11-4 自归一化网络的 DNN 配置

不要忘了归一化输入特征！还应该尝试复用部分预训练模型，如果它处理的是一个想死任务，或者如果有许多无便数据时使用无监督预训练，或者有许多相似任务的标签数据时使用辅助任务的语序年。

虽然这些指导可以应对大部分情况，但有些例外：

- 如果需要系数模型，你可以使用  $\ell_1$  正则（可以在训练后，将部分小权重设为零）。如果需要一个再稀疏点的模型，可以使用 TensorFlow Model Optimization Toolkit，它会破坏自归一化，所以要使用默认配置。
- 如果需要一个地延迟模型（预测快），层要尽量少，对前一层使用批归一化，使用更快的激活函数，比如 leaky ReLU 或 ReLU。稀疏模型也快。最后，将浮点精度从 32 位降到 16 位，甚至 8 位。还有，尝试 TF-MOT。
- 如果搭建的是风险敏感的模型，或者推断延迟不是非常重要，可以使用 MC 丢弃提升性能，得到更可靠的概率估计和不确定估计。

有了这些原则，就可以开始训练非常深的网络了。希望你现在对 Keras 有足够的自信。随着深入，可能需要写自定义的损失函数或调解训练算法。对于这样的情况，需要使用 TensorFlow 的低级 API，见下一章。

## 练习

1. 使用 He 初始化随机选择权重，是否可以将所有权重初始化为相同的值？
2. 可以将偏置初始化为 0 吗？
3. 说出 SELU 激活功能与 ReLU 相比的三个优点。
4. 在哪些情况下，您想要使用以下每个激活函数：SELU, leaky ReLU（及其变体）, ReLU, tanh, logistic 以及 softmax？
5. 如果将 `momentum` 超参数设置得太接近 1（例如，0.99999），会发生什么情况？
6. 请列举您可以生成稀疏模型的三种方法。
7. 丢弃是否会减慢训练？它是否会减慢推断（即预测新的实例）？MC 丢弃呢？
8. 在 CIFAR10 图片数据集上训练一个深度神经网络：
  - i. 建立一个 DNN，有 20 个隐藏层，每层 100 个神经元，使用 He 初始化和 ELU 激活函数。
  - ii. 使用 Nadam 优化和早停，尝试在 CIFAR10 上进行训练，可以使用 `keras.datasets.cifar10.load_data()` 加载数据。数据集包括 60000 张 32x32 的图片（50000 张训练，10000 张测试）有 10 个类，所以需要 10 个神经元的 softmax 输出层。记得每次调整架构或超参数之后，寻找合适的学习率。
  - iii. 现在尝试添加批归一化并比较学习曲线：它是否比以前收敛得更快？它是否会产生更好的模型？对训练速度有何影响？
  - iv. 尝试用 SELU 替换批归一化，做一些调整，确保网络是自归一化的（即，标准化输入特征，使用 LeCun 正态初始化，确保 DNN 只含有紧密层）。
  - v. 使用 alpha 丢弃正则化模型。然后，不训练模型，使用 MC 丢弃能否提高准确率。

vi. 用 1 循环调度重新训练模型，是否能提高训练速度和准确率。

参考答案见附录 A。

## 十二、使用 TensorFlow 自定义模型并训练

译者：[@SeanCheney](#)

目前为止，我们只是使用了 TensorFlow 的高级 API —— `tf.keras`，它的功能很强大：搭建了各种神经网络架构，包括回归、分类网络、Wide & Deep 网络、自归一化网络，使用了各种方法，包括批归一化、丢弃和学习率调度。事实上，你在实际案例中 95% 碰到的情况只需要 `tf.keras` 就足够了（和 `tf.data`，见第 13 章）。现在来深入学习 TensorFlow 的低级 Python API。当你需要实现自定义损失函数、自定义标准、层、模型、初始化器、正则器、权重约束时，就需要低级 API 了。甚至有时需要全面控制训练过程，例如使用特殊变换或对约束梯度时。这一章就会讨论这些问题，还会学习如何使用 TensorFlow 的自动图生成特征提升自定义模型和训练算法。首先，先来快速学习下 TensorFlow。

笔记：TensorFlow 2.0 (beta) 是 2019 年六月发布的，相比前代更易使用。本书第一版使用的是 TF 1，这一版使用的是 TF 2。

### TensorFlow 速览

TensorFlow 是一个强大的数值计算库，特别适合做和微调大规模机器学习（但也可以用来做其它的重型计算）。TensorFlow 是谷歌大脑团队开发的，支持了谷歌的许多大规模服务，包括谷歌云对话、谷歌图片和谷歌搜索。TensorFlow 是 2015 年 11 月开源的，（按文章引用、公司采用、GitHub 星数）是目前最流行的深度学习库。无数的项目是用 TensorFlow 来做各种机器学习任务，包括图片分类、自然语言处理、推荐系统和时间序列预测。TensorFlow 提供的功能如下：

- TensorFlow 的核心与 NumPy 很像，但 TensorFlow 支持 GPU；
- TensorFlow 支持（多设备和服务器）分布式计算；
- TensorFlow 使用了即时 JIT 编译器对计算速度和内存使用优化。编译器的工作是从 Python 函数提取出计算图，然后对计算图优化（比如剪切无用的节点），最后高效运行（比如自动并行运行独立任务）；
- 计算图可以导出为迁移形式，因此可以在一个环境中训练一个 TensorFlow 模型（比如使用 Python 或 Linux），然后在另一个环境中运行（比如在安卓设备上用 Java 运行）；
- TensorFlow 实现了自动微分，并提供了一些高效的优化器，比如 RMSProp 和 NAdam，因此可以容易的最小化各种损失函数。

基于上面这些特点，TensorFlow 还提供了许多其他功能：最重要的是 `tf.keras`，还有数据加载和预处理操作（`tf.data`，`tf.io` 等等），图片处理操作（`tf.image`），信号处理操作（`tf.signal`），等等（图 12-1 总结了 TensorFlow 的 Python API）

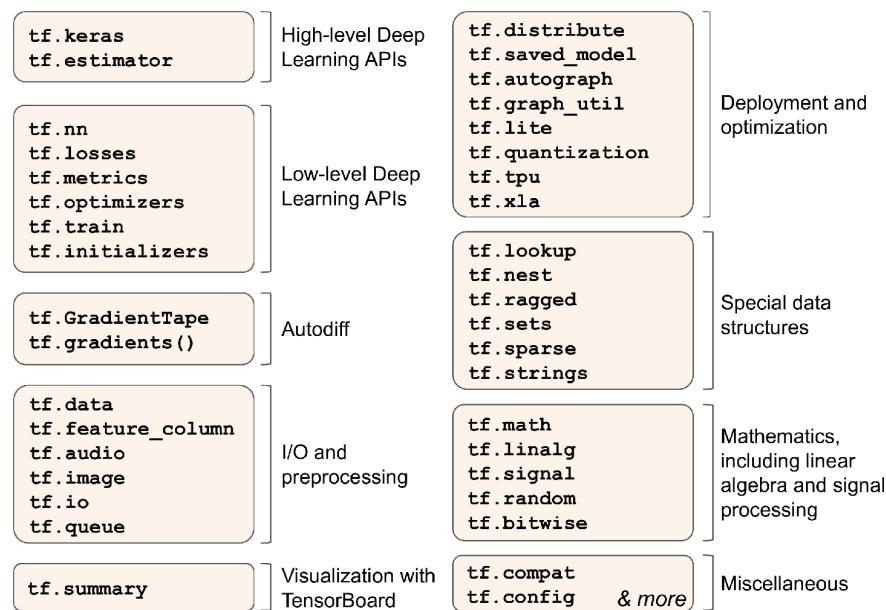


图 12-1 TensorFlow 的 Python API

提示：这一章会介绍 TensorFlow API 的多个包和函数，但来不及介绍全部，所以读者最好自己花点时间好好看看 API。TensorFlow 的 API 十分丰富，且文档详实。

TensorFlow 的低级操作都是用高效的 C++ 实现的。许多操作有多个实现，称为 核：每个核对应一个具体的设备型号，比如 CPU、GPU，甚至 TPU（张量处理单元）。GPU 通过将任务分成小块，在多个 GPU 线程中并行运行，可以极大提高提高计算的速度。TPU 更快：TPU 是自定义的 ASIC 芯片，专门用来做深度学习运算的（第 19 章会讨论适合使用 GPU 和 TPU）。

TensorFlow 的架构见图 12-2。大多数时候你的代码使用高级 API 就够了（特别是 `tf.keras` 和 `tf.data`），但如果需要更大的灵活性，就需要使用低级 Python API，来直接处理张量。TensorFlow 也支持其它语言的 API。任何情况下，甚至是跨设备和机器的情况下，TensorFlow 的执行引擎都会负责高效运行。

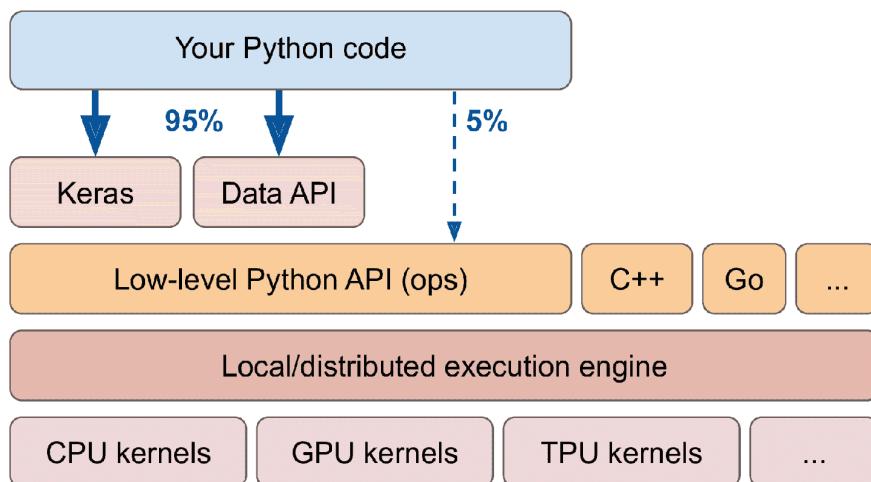


图 12-2 TensorFlow 的架构

TensorFlow 不仅可以运行在 Windows、Linux 和 macOS 上，也可以运行在移动设备上（使用 TensorFlow Lite），包括 iOS 和安卓（见第 19 章）。如果不使用 Python API，还可以使用 C++、Java、Go 和 Swift 的 API。甚至还有 JavaScript 的实现 TensorFlow.js，它可以直接在浏览器中运行。

TensorFlow 不只有这些库。TensorFlow 处于一套可扩展的生态系统库的核心位置。首先，TensorBoard 可以用来可视化。其次，TensorFlow Extended (TFX)，是谷歌推出的用来生产化的库，包括：数据确认、预处理、模型分析和服务（使用 TF Serving，见第 19 章）。谷歌的 TensorFlow Hub 上可以方便下载和复用预训练好的神经网络。你还可以从 TensorFlow 的 [model garden](#) 获取许多神经网络架构，其中一些是预训练好的。[TensorFlow Resources](#) 和[这个页面](#)上有更多的资源。你可以在 GitHub 上找到数百个 TensorFlow 项目，无论干什么都可以方便地找到现成的代码。

**提示：**越来越多的 ML 论文都附带了实现过程，一些甚至带有预训练模型。可以在[这里](#)找到。

最后，TensorFlow 有一支热忱满满的开发者团队，也有庞大的社区。要是想问技术问题，可以去[这里](#)，问题上打上 tensorflow 和 python 标签。还可以在 [GitHub](#) 上提 bug 和新功能。一般的讨论可以去[谷歌群组](#)。

下面开始写代码！

## 像 NumPy 一样使用 TensorFlow

TensorFlow 的 API 是围绕张量 (tensor) 展开的，从一个操作流动 (flow) 到另一个操作，所以名字叫做 TensorFlow。张量通常是一个多维数组（就像 NumPy 的 ndarray），但也可以是标量（即简单值，比如 42）。张量对于自定义的损失函数、标准、层等等非常重要，接下来学习如何创建和操作张量。

## 张量和运算

使用 `tf.constant()` 创建张量。例如，下面的张量表示的是两行三列的浮点数矩阵：

```
>>> 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) # 标量
<tf.Tensor: id=1, shape=(), dtype=int32, numpy=42>
```

就像 `ndarray` 一样，`tf.Tensor` 也有形状和数据类型 (`dtype`)：

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32
```

索引和 NumPy 中很像：

```
>>> 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，所有张量运算都可以执行：

```
>>> 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)>
```

可以看到，`t + 10` 等同于调用 `tf.add(t, 10)`，`-` 和 `*` 也支持。`@` 运算符是在 Python3.5 中出现的，用于矩阵乘法，等同于调用函数 `tf.matmul()`。

可以在 `tf` 中找到所有基本的数学运算

(`tf.add()`、`tf.multiply()`、`tf.square()`、`tf.exp()`、`tf.sqrt()`)，以及 NumPy 中的大部分运算（比如 `tf.reshape()`、`tf.squeeze()`、`tf.tile()`）。一些 `tf` 中的函数与 NumPy 中不同，例如，`tf.reduce_mean()`、`tf.reduce_sum()`、`tf.reduce_max()`、`tf.math.log()` 等同于 `np.mean()`、`np.sum()`、`np.max()` 和 `np.log()`。当函数名不同时，通常都是有原因的。例如，TensorFlow 中必须使用 `tf.transpose(t)`，不能像 NumPy 中那样使用 `t.T`。原因是函数 `tf.transpose(t)` 所做的和 NumPy 的属性 `t` 并不完全相同：在 TensorFlow 中，是使用转置数据的复制来生成张量的，而在 NumPy 中，`t.T` 是数据的转置视图。相似的，`tf.reduce_sum()` 操作之所以这么命名，是因为它的 GPU 核（即 GPU 实现）所采用的归约算法不能保证元素相加的顺序，因为 32 位的浮点数精度有限，每次调用的结果可能会有细微的不同。`tf.reduce_mean()` 也是这样（`tf.reduce_max()` 结果是确定的）。

**笔记：**许多函数和类都有假名。比如，`tf.add()` 和 `tf.math.add()` 是相同的。这可以让 TensorFlow 对于最常用的操作有简洁的名字，同时包可以有序安置。

Keras 的低级 API Keras API 有自己的低级 API，位于 `keras.backend`，包括：函数 `square()`、`exp()`、`sqrt()`。在 `tf.keras` 中，这些函数通常通常只是调用对应的 TensorFlow 操作。如果你想写一些可以迁移到其它 Keras 实现上，就应该使用这些 Keras 函数。但是这些函数不多，所以这本书里就直接使用 TensorFlow 的运算了。下面是一个简单的使用了 `keras.backend` 的例子，简记为 `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)>
```

## 张量和 NumPy

张量和 NumPy 融合地非常好：使用 NumPy 数组可以创建张量，张量也可以创建 NumPy 数组。可以在 NumPy 数组上运行 TensorFlow 运算，也可以在张量上运行 NumPy 运算：

```
>>> a = np.array([2., 4., 5.])
>>> tf.constant(a)
<tf.Tensor: id=111, shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>
>>> t.numpy() # 或 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)
```

**警告**：NumPy 默认使用 64 位精度，TensorFlow 默认用 32 位精度。这是因为 32 位精度通常对于神经网络就足够了，另外运行地更快，使用的内存更少。因此当你用 NumPy 数组创建张量时，一定要设置 `dtype=tf.float32`。

## 类型转换

类型转换对性能的影响非常大，并且如果类型转换是自动完成的，不容易被注意到。为了避免这样，TensorFlow 不会自动做任何类型转换：只是如果用不兼容的类型执行了张量运算，TensorFlow 就会报异常。例如，不能用浮点型张量与整数型张量相加，也不能将 32 位张量与 64 位张量相加：

```
>>> 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[...]
```

这点可能一开始有点恼人，但是有其存在的理由。如果真的需要转换类型，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>
```

## 变量

到目前为止看到的 `tf.Tensor` 值都是不能修改的。意味着不能使用常规张量实现神经网络的权重，因为权重必须要能被反向传播调整。另外，其它的参数也需要随着时间调整（比如，动量优化器要跟踪过去的梯度）。此时需要的是 `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` 和 `tf.Tensor` 很像：可以运行同样的运算，可以配合 NumPy 使用，也要注意类型。可以使用 `assign()` 方法对其就地修改

（或 `assign_add()`、`assign_sub()`）。使用切片的 `assign()` 方法可以修改独立的切片（直接赋值行不通），或使用 `scatter_update()`、`scatter_nd_update()` 方法：

```

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.]]

```

笔记：在实践中，很少需要手动创建变量，因为 Keras 有 `add_weight()` 方法可以自动来做。另外，模型参数通常会直接通过优化器更新，因此很少需要手动更新。

## 其它数据结构

TensorFlow 还支持其它几种数据结构，如下（可以参考笔记本的 `Tensors and Operations` 部分，或附录的 F）：

稀疏张量 (`tf.SparseTensor`) 高效表示含有许多 0 的张量。`tf.sparse` 包含有对稀疏张量的运算。

张量数组 (`tf.TensorArray`) 是张量的列表。有默认固定大小，但也可以做成动态的。列表中的张量必须形状相同，数据类型也相同。

嵌套张量 (`tf.RaggedTensor`) 张量列表的静态列表，张量的形状和数据结构相同。`tf.ragged` 包里有嵌套张量的运算。

字符串张量 类型是 `tf.string` 的常规张量，是字节串而不是 Unicode 字符串，因此如果你用 Unicode 字符串（比如，Python3 字符串 `café`）创建了一个字符串张量，就会自动被转换为 UTF-8（`b"caf\xc3\xa9"`）。另外，也可以用 `tf.int32` 类型的张量表示 Unicode 字符串，其中每项表示一个 Unicode 码（比如，`[99, 97, 102, 233]`）。`tf.strings` 包里有字节串和 Unicode 字符串的运算，以及二者转换的运算。要注意 `tf.string` 是原子性的，也就是说它的长度不出现在张量的形状中，一旦将其转换成了 Unicode 张量（即，含有 Unicode 码的 `tf.int32` 张量），长度才出现在形状中。

集合 表示为常规张量（或稀疏张量）。例如 `tf.constant([[1, 2], [3, 4]])` 表示两个集合{1, 2}和{3, 4}。通常，用张量的最后一个轴的向量表示集合。集合运算可以用 `tf.sets` 包。

队列 用来在多个步骤之间保存张量。TensorFlow 提供了多种队列。先进先出 (FIFO) 队列 `FIFOQueue`，优先级队列 `PriorityQueue`，随机队列 `RandomShuffleQueue`，通过填充的不同形状的批次队列 `PaddingFIFOQueue`。这些队列都在 `tf.queue` 包中。

有了张量、运算、变量和各种数据结构，就可以开始自定义模型和训练算法啦！

## 自定义模型和训练算法

先从简单又常见的任务开始，创建一个自定义的损失函数。

### 自定义损失函数

假如你想训练一个回归模型，但训练集有噪音。你当然可以通过清除或修正异常值来清理数据集，但是这样还不够：数据集还是有噪音。此时，该用什么损失函数呢？均方差可能对大误差惩罚过重，导致模型不准确。均绝对值误差不会对异常值

惩罚过重，但训练可能要比较长的时间才能收敛，训练模型也可能不准确。此时使用 Huber 损失（第 10 章介绍过）就比 MSE 好多了。目前官方 Keras API 中没有 Huber 损失，但 `tf.keras` 有（使用类 `keras.losses.Huber` 的实例）。就算 `tf.keras` 没有，实现也不难！只需创建一个函数，参数是标签和预测值，使用 TensorFlow 运算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)
```

**警告**：要提高性能，应该像这个例子使用向量。另外，如果想利用 TensorFlow 的图特性，则只能使用 TensorFlow 运算。

最好返回一个包含实例的张量，其中每个实例都有一个损失，而不是返回平均损失。这么做的话，Keras 可以在需要时，使用类权重或样本权重（见第 10 章）。

现在，编译 Keras 模型时，就可以使用 Huber 损失来训练了：

```
model.compile(loss=huber_fn, optimizer="nadam")
model.fit(X_train, y_train, [...])
```

仅此而已！对于训练中的每个批次，Keras 会调用函数 `huber_fn()` 计算损失，用损失来做梯度下降。另外，Keras 会从一开始跟踪总损失，并展示平均损失。

在保存这个模型时，这个自定义损失会发生什么呢？

## 保存并加载包含自定义组件的模型

因为 Keras 可以保存函数名，保存含有自定义损失函数的模型也不成问题。当加载模型时，你需要提供一个字典，这个字典可以将函数名和真正的函数映射起来。一般说来，当加载一个含有自定义对象的模型时，你需要将名字映射到对象上：

```
model = keras.models.load_model("my_model_with_a_custom_loss.h5",
                                custom_objects={"huber_fn": huber_fn})
```

对于刚刚的代码，在 -1 和 1 之间的误差被认为是“小”误差。如果要改变阈值呢？一个解决方法是创建一个函数，它可以产生一个可配置的损失函数：

```
def create_hubert(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_hubert(2.0), optimizer="nadam")
```

但在保存模型时，`threshold` 不能被保存。这意味着在加载模型时（注意，给 Keras 的函数名是 `Huber_fn`，不是创造这个函数的函数名），必须要指定 `threshold` 的值：

```
model = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
                                custom_objects={"huber_fn": create_hubert(2.0)})
```

要解决这个问题，可以创建一个 `keras.losses.Loss` 类的子类，然后实现 `get_config()` 方法：

```
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}
```

**警告**：Keras API 目前只使用子类来定义层、模型、回调和正则器。如果使用子类创建其它组件（比如损失、指标、初始化器或约束），它们不能迁移到其它 Keras 实现上。可能 Keras API 经过更新，就会支持所有组件了。

逐行看下这段代码：

- 构造器接收 `**kwargs`，并将其传递给父构造器，父构造器负责处理超参数：  
损失的 `name`，要使用的、用于将单个实例的损失汇总的 `reduction` 算法。默认情况下是 `"sum_over_batch_size"`，意思是损失是各个实例的损失之和，如果有样本权重，则做权重加权，再除以批次大小（不是除以权重之和，所以不是加权平均）。其它可能的值是 `"sum"` 和 `None`。
- `call()` 方法接受标签和预测值，计算所有实例的损失，并返回。
- `get_config()` 方法返回一个字典，将每个超参数映射到值上。它首先调用父类的 `get_config()` 方法，然后将新的超参数加入字典（`{**x}` 语法是 Python 3.5 引入的）。

当编译模型时，可以使用这个类的实例：

```
model.compile(loss=HuberLoss(2.), optimizer="nadam")
```

保存模型时，阈值会一起保存；加载模型时，只需将类名映射到具体的类上：

```
model = keras.models.load_model("my_model_with_a_custom_loss_class.h5",
                                custom_objects={"HuberLoss": HuberLoss})
```

保存模型时，Keras 调用损失实例的 `get_config()` 方法，将配置以 JSON 的形式保存在 HDF5 中。当加载模型时，会调用 `HuberLoss` 类的 `from_config()` 方法：这个方法是父类 `Loss` 实现的，创建一个类 `Loss` 的实例，将 `**config` 传递给构造器。

## 自定义激活函数、初始化器、正则器和约束

Keras 的大多数功能，比如损失、正则器、约束、初始化器、指标、激活函数、层，甚至是完整的模型，都可以用相似的方法做自定义。大多数时候，需要写一个简单的函数，带有合适的输入和输出。下面的例子是自定义激活函数（等价于 `keras.activations.softplus()` 或 `tf.nn.softplus()`），自定义 Glorot 初始化器（等价于 `keras.initializers.glorot_normal()`），自定义 `l1` 正则化器（等价于 `keras.regularizers.l1(0.01)`），可以保证权重都是正值的自定义约束（等价于 `equivalent to keras.constraints.nonneg()` 或 `tf.nn.relu()`）：

```

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)

```

可以看到，参数取决于自定义函数的类型。这些自定义函数可以如常使用，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)

```

激活函数会应用到这个 `Dense` 层的输出上，结果会传递到下一层。层的权重会使用初始化器的返回值。在每个训练步骤，权重会传递给正则化函数以计算正则损失，这个损失会与主损失相加，得到训练的最终损失。最后，会在每个训练步骤结束后调用约束函数，经过约束的权重会替换层的权重。

如果函数有需要连同模型一起保存的超参数，需要对相应的类做子类，比如 `keras.regularizers.Regularizer`, `keras.constraints.Constraint`, `keras.initializers.Initializer`，或 `keras.layers.Layer` (任意层，包括激活函数)。就像前面的自定义损失一样，下面是一个简单的 `l1` 正则类，可以保存它的超参数 `factor` (这次不必调用其父构造器或 `get_config()` 方法，因为它们不是父类定义的)：

```

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}

```

注意，你必须要实现损失、层（包括激活函数）和模型的 `call()` 方法，或正则化器、初始化器和约束的 `_call_()` 方法。对于指标，处理方法有所不同。

## 自定义指标

损失和指标的概念是不一样的：梯度下降使用损失（比如交叉熵损失）来训练模型，因此损失必须是可微分的（至少是在评估点可微分），梯度不能在所有地方都是 0。另外，就算损失比较难解释也没有关系。相反的，指标（比如准确率）是用来评估模型的：指标的解释性一定要好，可以是不可微分的，或者可以在任何地方的梯度都是 0。

但是，在多数情况下，定义一个自定义指标函数和定义一个自定义损失函数是完全一样的。事实上，刚才创建的 `Huber` 损失函数也可以用来当指标（持久化也是同样的，只需要保存函数名 `Huber_fn` 就成）：

```

model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])

```

对于训练中的每个批次，Keras 能计算该指标，并跟踪自周期开始的指标平均值。大多数时候，这样没有问题。但会有例外！比如，考虑一个二元分类器的准确性。第 3 章介绍过，准确率是真正值除以正预测数（包括真正值和假正值）。假设模型

在第一个批次做了 5 个正预测，其中 4 个是正确的，准确率就是 80%。再假设模型在第二个批次做了 3 次正预测，但没有一个预测对，则准确率是 0%。如果对这两个准确率做平均，则平均值是 40%。但它不是模型在两个批次上的准确率！事实上，真正值总共有 4 个，正预测有 8 个，整体的准确率是 50%。我们需要的是一个能跟踪真正值和正预测数的对象，用该对象计算准确率。这就是类 `keras.metrics.Precision` 所做的：

```
>>> 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>
```

在这个例子中，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传给它，然后传第二个批次的数据（这里也可以传样本权重）。数据和前面的真正值和正预测一样。第一个批次之后，正确率是 80%；第二个批次之后，正确率是 50%（这是完整过程的准确率，不是第二个批次的准确率）。这叫做流式指标（或者静态指标），因为他是一个批次接一个批次，逐次更新的。

任何时候，可以调用 `result()` 方法获取指标的当前值。还可以通过 `variables` 属性，查看指标的变量（跟踪正预测和负预测的数量），还可以用 `reset_states()` 方法重置变量：

```
>>> p.result()
<tf.Tensor: id=581794, shape=(), dtype=float32, numpy=0.5>
>>> p.variables
[<tf.Variable 'true_positives:0' [...] numpy=array([4.], dtype=float32)>,
 <tf.Variable 'false_positives:0' [...] numpy=array([4.], dtype=float32)>]
>>> p.reset_states() # both variables get reset to 0.0
```

如果想创建一个这样的流式指标，可以创建一个 `keras.metrics.Metric` 类的子类。下面的例子跟踪了完整的 Huber 损失，以及实例的数量。当查询结果时，就能返回比例值，该值就是平均 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}
```

逐行看下代码：

- 构造器使用 `add_weight()` 方法来创建用来跟踪多个批次的变量 —— 在这个例子中，就是 Huber 损失的和（`total`）和实例的数量（`count`）。如果愿意的话，可以手动创建变量。Keras 会跟中任何被设为属性的 `tf.Variable`（更一般的讲，任何“可追踪对象”，比如层和模型）。
- 当将这个类的实例当做函数使用时会调用 `update_state()` 方法（正如 `Precision` 对象）。它能用每个批次的标签和预测值（还有样本权重，但这个例子忽略了样本权重）来更新变量。

- `result()` 方法计算并返回最终值，在这个例子中，是返回所有实例的平均 Huber 损失。当你将指标用作函数时，`update_state()` 方法先被调用，然后调用 `result()` 方法，最后返回输出。
- 还实现了 `get_config()` 方法，用以确保 `threshold` 和模型一起存储。
- `reset_states()` 方法默认将所有值重置为 0.0（也可以改为其它值）。

笔记：Keras 能无缝处理变量持久化。

当用简单函数定义指标时，Keras 会在每个批次自动调用它，还能跟踪平均值，就和刚才的手工处理一模一样。因此，`HuberMetric` 类的唯一好处是 `threshold` 可以进行保存。当然，一些指标，比如准确率，不能简单的平均化；对于这些例子，只能实现一个流式指标。

创建好了流式指标，再创建自定义层就很简单了。

## 自定义层

有时候你可能想搭建一个架构，但 TensorFlow 没有提供默认实现。这种情况下，就需要创建自定义层。否则只能搭建出的架构会是简单重复的，包含相同且重复的层块，每个层块实际上就是一个层而已。比如，如果模型的层顺序是 A、B、C、A、B、C、A、B、C，则完全可以创建一个包含 A、B、C 的自定义层 D，模型就可以简化为 D、D、D。

如何创建自定义层呢？首先，一些层没有权重，比如 `keras.layers.Flatten` 或 `keras.layers.ReLU`。如果想创建一个没有任何权重的自定义层，最简单的方法是协议个函数，将其包装进 `keras.layers.Lambda` 层。比如，下面的层会对输入做指数运算：

```
exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
```

这个自定义层可以像任何其它层一样使用顺序 API、函数式 API 或子类化 API。你还可以将其用作激活函数（或者使

用 `activation=tf.exp`，`activation=keras.activations.exponential`，或者 `activation="exponential"`）。当预测值的数量级不同时，指数层有时用在回归模型的输出层。

你可能猜到了，要创建自定义状态层（即，有权重的层），需要创建 `keras.layers.Layer` 类的子类。例如，下面的类实现了一个紧密层的简化版本：

```

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)}

```

逐行看下代码：

- 构造器将所有超参数作为参数（这个例子中，是 `units` 和 `activation`），更重要的，它还接收一个 `**kwargs` 参数。接着初始化了父类，传给父类 `kwargs`：它负责标准参数，比如 `input_shape`、`trainable` 和 `name`。然后将超参数存为属性，使用 `keras.activations.get()` 函数（这个函数接收函数、标准字符串，比如 `"relu"`、`"selu"`、或 `"None"`），将 `activation` 参数转换为合适的激活函数。
- `build()` 方法通过对每个权重调用 `add_weight()` 方法，创建层的变量。层第一次被使用时，调用 `build()` 方法。此时，Keras 能知道该层输入的形状，并传入 `build()` 方法，这对创建权重是必要的。例如，需要知道前一层的神经元数量，来创建连接权重矩阵（即，`"kernel"`）：对应的是输入的最后一维的大小。在 `build()` 方法最后（也只是在最后），必须调用父类的 `build()` 方法：这步告诉 Keras 这个层建好了（或者设定 `self.built=True`）。
- `call()` 方法执行预想操作。在这个例子中，计算了输入 `x` 和层的核的矩阵乘法，加上了偏置向量，对结果使用了激活函数，得到了该层的输出。
- `compute_output_shape()` 方法只是返回了该层输出的形状。在这个例子中，输出和输入的形状相同，除了最后一维被替换成了层的神经元数。  
在 `tf.keras` 中，形状是 `tf.TensorShape` 类的实例，可以用 `as_list()` 转换为 Python 列表。
- `get_config()` 方法和前面的自定义类很像。注意是通过调用 `keras.activations.serialize()`，保存了激活函数的完整配置。

现在，就可以像其它层一样，使用 `MyDense` 层了！

笔记：一般情况下，可以忽略 `compute_output_shape()` 方法，因为 `tf.keras` 能自动推断输出的形状，除非层是动态的（后面会看到动态层）。在其它 Keras 实现中，要么需要 `compute_output_shape()` 方法，要么默认输出形状和输入形状相同。

要创建一个有多个输入（比如 `Concatenate`）的层，`call()` 方法的参数应该是包含所有输入的元组。相似的，`compute_output_shape()` 方法的参数应该是一个包含每个输入的批次形状的元组。要创建一个有多输出的层，`call()` 方法要返回输出

的列表，`compute_output_shape()` 方法要返回批次输出形状的列表（每个输出一个形状）。例如，下面的层有两个输入和三个输出：

```
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] # 可能需要处理广播规则
```

这个层现在就可以像其它层一样使用了，但只能使用函数式和子类化 API，顺序 API 不成（只能使用单输入和单输出的层）。

如果你的层需要在训练和测试时有不同的行为（比如，如果使用 `Dropout` 或 `BatchNormalization` 层），那么必须给 `call()` 方法加上 `training` 参数，用这个参数确定该做什么。比如，创建一个在训练中（为了正则）添加高斯造影的层，但不改动训练（`Keras` 有一个层做了同样的事，`keras.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
```

上面这些就能让你创建自定义层了！接下来看看如何创建自定义模型。

## 自定义模型

第 10 章在讨论子类化 API 时，接触过创建自定义模型的类。说白了：创建 `keras.Model` 类的子类，创建层和变量，用 `call()` 方法完成模型想做的任何事。假设你想搭建一个图 12-3 中的模型。

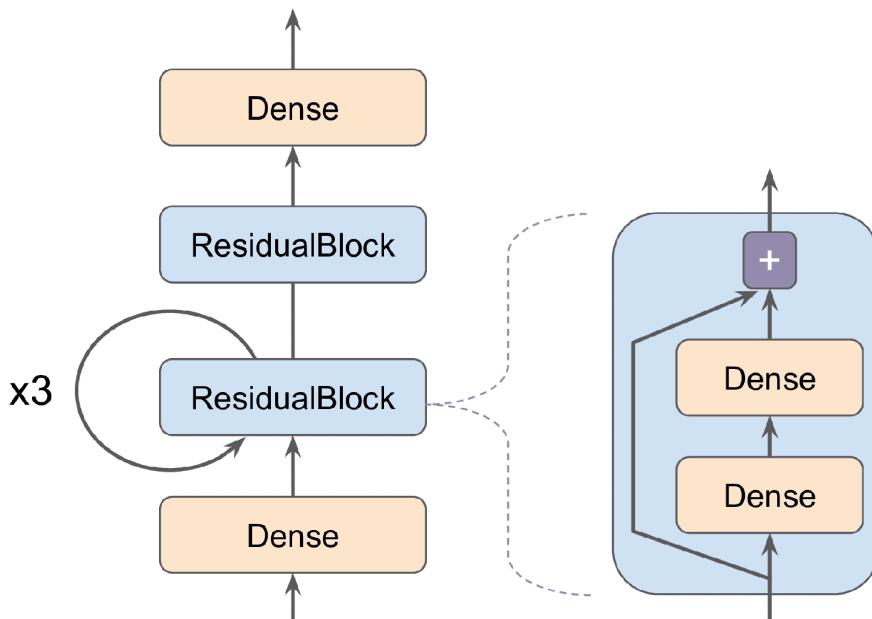


图 12-3 自定义模型案例：包含残差块层，残块层含有跳连接

输入先进入一个紧密层，然后进入包含两个紧密层和一个添加操作的残差块（第 14 章会看见，残差块将输入和输出相加），经过 3 次同样的残差块，再通过第二个残差块，最终结果通过一个紧密输出层。这个模型没什么意义，只是一个搭建任意结构（包含循环和跳连接）模型的例子。要实现这个模型，最好先创建 ResidualBlock 层，因为这个层要用好几次：

```
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
```

这个层稍微有点特殊，因为它包含了其它层。用 Keras 来实现：自动检测 hidden 属性包含可追踪对象（即，层），内含层的变量可以自动添加到整层的变量列表中。类的其它部分很好懂。接下来，使用子类化 API 定义模型：

```
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)
```

在构造器中创建层，在 call() 方法中使用。这个模型可以像其它模型那样来使用（编译、拟合、评估、预测）。如果你还想使用 save() 方法保存模型，使用 keras.models.load\_model() 方法加载模型，则必须在 ResidualBlock 类和 ResidualRegressor 类中实现 get\_config() 方法。另外，可以用 save\_weights() 方法和 load\_weights() 方法保存和加载权重。

Model 类是 Layer 类的子类，因此模型可以像层一样定义和使用。但是模型还有一些其它的功能，包括 compile()、fit()、evaluate() 和 predict()（还有一些变量），还有 get\_layers() 方法（它能通过名字或序号返回模型的任意层）、save() 方法（支持 keras.models.load\_model() 和 keras.models.clone\_model()）。

提示：如果模型提供的功能比层多，为什么不讲每一个层定义为模型呢？技术上当然可以这么做，但对内部组件和模型（即，层或可重复使用的层块）加以区别，可以更加清晰。前者应该是 Layer 类的子类，后者应该是 Model 类的子类。

掌握了上面的方法，你就可以使用顺序 API、函数式 API、子类化 API 搭建几乎任何文章上的模型了。为什么是“几乎”？因为还有些内容需要掌握：首先，如何基于模型内部定义损失或指标，第二，如何搭建自定义训练循环。

## 基于模型内部的损失和指标

前面的自定义损失和指标都是基于标签和预测（或者还有样本权重）。有时，你可能想基于模型的其它部分定义损失，比如隐藏层的权重或激活函数。这么做，可以是出于正则的目的，或监督模型的内部。

要基于模型内部自定义损失，需要先做基于这些组件的计算，然后将结果传递给 `add_loss()` 方法。例如，自定义一个包含五个隐藏层加一个输出层的回归 MLP 模型。这个自定义模型基于上层的隐藏层，还有一个辅助的输出。和辅助输出关联的损失，被称为重建损失（见第 17 章）：它是重建和输入的均方差。通过将重建误差添加到主损失上，可以鼓励模型通过隐藏层保留尽量多的信息，即便是那些对回归任务没有直接帮助的信息。在实际中，重建损失有助于提高泛化能力（它是一个正则损失）。下面是含有自定义重建损失的自定义模型：

```
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)
```

逐行看下代码：

- 构造器搭建了一个有五个紧密层和一个紧密输出层的 DNN。
- `build()` 方法创建了另一个紧密层，可以重建模型的输入。必须要在这里创建 `build()` 方法的原因，是单元的数量必须等于输入数，而输入数在调用 `build()` 方法之前是不知道的。
- `call()` 方法处理所有五个隐藏层的输入，然后将结果传给重建层，重建层产生重建。
- `call()` 方法然后计算重建损失（重建和输入的均方差），然后使用 `add_loss()` 方法，将其加到模型的损失列表上。注意，这里对重建损失乘以了 0.05（这是个可调节的超参数），做了缩小，以确保重建损失不主导主损失。
- 最后，`call()` 方法将隐藏层的输出传递给输出层，然后返回输出。

相似的，可以加上一个基于模型内部的自定义指标。例如，可以在构造器中创建一个 `keras.metrics.Mean` 对象，然后在 `call()` 方法中调用它，传递给它 `recon_loss`，最后通过 `add_metric()` 方法，将其添加到模型上。使用这种方式，在训练模型时，Keras 能展示每个周期的平均损失（损失是主损失加上 0.05 乘以重建损失），和平均重建误差。两者都会在训练过程中下降：

```
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，前面所讨论的内容已经足够搭建你想要的模型了，就算是包含复杂架构、损失和指标也行。但是，在某些极端情况，你还需要自定义训练循环。介绍之前，先来看看 TensorFlow 如何自动计算梯度。

## 使用自动微分计算梯度

要搞懂如何使用自动微分自动计算梯度，来看一个例子：

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2
```

如果你会微积分，就能算出这个函数对  $w_1$  的偏导是  $6 * w_1 + 2 * w_2$ ，还能算出它对  $w_2$  的偏导是  $2 * w_1$ 。例如，在点  $(w_1, w_2) = (5, 3)$ ，这两个偏导数分别是 36 和 10，在这个点的梯度向量就是  $(36, 10)$ 。但对于神经网络来说，函数会复杂得多，可能会有上万个参数，用手算偏导几乎是不可能的任务。一个解决方法是计算每个偏导的大概值，通过调节参数，查看输出的变化：

```
>>> 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
10.00000003174137
```

这种方法很容易实现，但只是大概。重要的是，需要对每个参数至少要调用一次  $f()$ （不是至少两次，因为可以只计算一次  $f(w_1, w_2)$ ）。这样，对于大神经网络，就不怎么可控。所以，应该使用自动微分。TensorFlow 的实现很简单：

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

先定义了两个变量  $w_1$  和  $w_2$ ，然后创建了一个 `tf.GradientTape` 上下文，它能自动记录变脸的每个操作，最后使用它算出结果  $z$  关于两个变量  $[w_1, w_2]$  的梯度。TensorFlow 计算的梯度如下：

```
>>> gradients
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: id=828229, shape=(), dtype=float32, numpy=10.0>]
```

很好！不仅结果是正确的（准确度只受浮点误差限制），`gradient()` 方法只逆向算了一次，无论有多少个变量，效率很高。

**提示：**为了节省内存，只将严格的最小值放在 `tf.GradientTape()` 中。另外，通过在 `tf.GradientTape()` 中创建一个 `tape.stop_recording()` 来暂停记录。

当调用记录器的 `gradient()` 方法时，记录器会自动清零，所以调用两次 `gradient()` 就会报错：

```

with tf.GradientTape() as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
dz_dw2 = tape.gradient(z, w2) # 运行时错误

```

如果需要调用 `gradient()` 一次以上，比续将记录器持久化，并在每次用完之后删除，释放资源：

```

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，记录器只会跟踪包含变量的操作，所以如果是计算 `z` 的梯度，`z` 和变量没关系，结果就会是 `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，你也可以强制记录器监视任何你想监视的张量，将它们当做变量来计算梯度：

```

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，这么做会有帮助，比如当输入的波动很小，而激活函数结果波动很大时，要实现一个正则损失，就可以这么做：损失会基于激活函数结果，激活函数结果会基于输入。因为输入不是变量，就需要记录器监视输入。

大多数时候，梯度记录器被用来计算单一值（通常是损失）的梯度。这就是自动微分发挥长度的地方了。因为自动微分只需要一次向前传播一次向后传播，就能计算所有梯度。如果你想计算一个向量的梯度，比如一个包含多个损失的向量，TensorFlow 就会计算向量和的梯度。因此，如果你需要计算单个梯度的话（比如每个损失相对于模型参数的梯度），你必须调用记录器的 `jacobiian()` 方法：它能做反向模式的自动微分，一次计算完向量中的所有损失（默认是并行的）。甚至还可以计算二级偏导，但在实际中用的不多（见笔记本中的“自动微分计算梯度部分”）。

某些情况下，你可能想让梯度在部分神经网络停止传播。要这么做的话，必须使用 `tf.stop_gradient()` 函数。它能在前向传播中（比如 `tf.identity()`）返回输入，并能阻止梯度反向传播（就像常量一样）：

```

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，在计算梯度时可能还会碰到数值问题。例如，如果对于很大的输入，计算 `my_softplus()` 函数的梯度，结果会是 `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这个函数的梯度，会有些数值方面的难点：因为浮点数的精度误差，自动微分最后会变成无穷除以无穷（结果是 `NaN`）。幸好，`softplus` 函数的导数是  $1 / (1 + 1 / \exp(x))$ ，它是数值稳定的。接着，让 TensorFlow 使用这个稳定的函数，通过装饰器 `@tf.custom_gradient` 计算 `my_softplus()` 的梯度，既返回正常输出，也返回计算导数的函数（注意：它会接收的输入是反向传播的梯度；根据链式规则，应该乘以函数的梯度）：

```
@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
```

计算好了 `my_better_softplus()` 的梯度，就算对于特别大的输入值，也能得到正确的结果（但是，因为指数运算，主输出还是会爆炸；绕过的方法是，当输出很大时，使用 `tf.where()` 返回输入）。

祝贺你！现在你就可以计算任何函数的梯度（只要函数在计算点可微就行），甚至可以阻止反向传播，还能写自己的梯度函数！TensorFlow 的灵活性还能让你编写自定义的训练循环。

## 自定义训练循环

在某些特殊情况下，`fit()` 方法可能不够灵活。例如，第 10 章讨论过的 Wide & Deep 论文使用了两个优化器：一个用于宽路线，一个用于深路线。因为 `fit()` 方法智能使用一个优化器（编译时设置的优化器），要实现这篇论文就需要写自定义循环。

你可能还想写自定义的训练循环，只是想让训练过程更加可控（也许你对 `fit()` 方法的细节并不确定）。但是，自定义训练循环会让代码变长、更容易出错、也难以维护。

**提示：**除非真的需要自定义，最好还是使用 `fit()` 方法，而不是自定义训练循环，特别是当你是在一个团队之中时。

首先，搭建一个简单的模型。不用编译，因为是要手动处理训练循环：

```
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)
])
```

接着，创建一个小函数，它能从训练集随机采样一个批次的实例（第 13 章会讨论更便捷的 Data API）：

```
def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]
```

再定义一个可以展示训练状态的函数，包括步骤数、总步骤数、平均损失  
(用 `Mean` 指标计算) ，和其它指标：

```
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)
```

这段代码不难，除非你对 Python 字符串的 `{:.4f}` 不熟：它的作用是保留四位小数。使用 `\r` (回车) 和 `end=""` 连用，保证状态条总是打印在一条线上。笔记本中，`print_status_bar()` 函数包括进度条，也可以使用 `tqdm` 库。

有了这些准备，就可以开干了！首先，我们定义超参数、选择优化器、损失函数和指标（这个例子中是 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()]
```

可以搭建自定义循环了：

```
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()
```

逐行看下代码：

- 创建了两个嵌套循环：一个是给周期的，一个是给周期里面的批次的。
- 然后从训练集随机批次采样。
- 在 `tf.GradientTape()` 内部，对一个批次做了预测（将模型用作函数），计算其损失：损失等于主损失加上其它损失（在这个模型中，每层有一个正则损失）。因为 `mean_squared_error()` 函数给每个实例返回一个损失，使用 `tf.reduce_mean()` 计算平均值（如果愿意的话，每个实例可以用不同的权重）。正则损失已经转变为单个的标量，所以只需求和就成（使用 `tf.add_n()`，它能将相同形状和数据类型的张量求和）。
- 接着，让记录器计算损失相对于每个可训练变量的梯度（不是所有的变量！），然后用优化器对梯度做梯度下降。

- 然后，更新（当前周期）平均损失和平均指标，显示状态条。
- 在每个周期结束后，再次展示状态条，使其完整，然后换行，重置平均损失和平均指标。

如果设定优化器的 `clipnorm` 或 `clipvalue` 超参数，就可以自动重置。如果你想对梯度做任何其它变换，在调用 `apply_gradients()` 方法之前，做变换就行。

如果你对模型添加了权重约束（例如，添加层时设置 `kernel_constraint` 或 `bias_constraint`），你需要在 `apply_gradients()` 之后，更新训练循环，以应用这些约束：

```
for variable in model.variables:
    if variable.constraint is not None:
        variable.assign(variable.constraint(variable))
```

最重要的，这个训练循环没有处理训练和测试过程中，行为不一样的层（例如，`BatchNormalization` 或 `Dropout`）。要处理的话，需要调用模型，令 `training=True`，并传播到需要这么设置的每一层。

可以看到，有这么多步骤都要做对才成，很容易出错。但另一方面，训练的控制权完全在你手里。

现在你知道如何自定义模型中的任何部分了，也知道如何训练算法了，接下来看看如何使用 TensorFlow 的自动图生成特征：它能显著提高自定义代码的速度，并且还是可迁移的（见第 19 章）。

## TensorFlow 的函数和图

在 TensorFlow 1 中，图是绕不过去的（同时图也很复杂），因为图是 TensorFlow 的 API 的核心。在 TensorFlow 2 中，图还在，但不是核心了，使用也简单多了。为了演示其易用性，从一个三次方函数开始：

```
def cube(x):
    return x ** 3
```

可以用一个值调用这个函数，整数、浮点数都成，或者用张量来调用：

```
>>> cube(2)
8
>>> cube(tf.constant(2.0))
<tf.Tensor: id=18634148, shape=(), dtype=float32, numpy=8.0>
```

现在，使用 `tf.function()` 将这个 Python 函数变为 TensorFlow 函数：

```
>>> tf_cube = tf.function(cube)
>>> tf_cube
<tensorflow.python.eager.def_function.Function at 0x1546fc080>
```

可以像原生 Python 函数一样使用这个 TF 函数，可以返回同样的结果（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>
```

`tf.function()` 在底层分析了 `cube()` 函数的计算，然后生成了一个等价的计算图！可以看到，过程十分简单（下面会讲解过程）。另外，也可以使用 `tf.function` 作为装饰器，更常见一些：

```
@tf.function
def tf_cube(x):
    return x ** 3
```

原生的 Python 函数通过 TF 函数的 `python_function` 属性仍然可用：

```
>>> tf_cube.python_function(2)
8
```

TensorFlow 优化了计算图，删掉了没用的节点，简化了表达式（比如，`1 + 2` 会替换为 `3`），等等。当优化好的计算图准备好之后，TF 函数可以在图中，按合适的顺序高效执行运算（该并行的时候就并行）。作为结果，TF 函数比普通的 Python 函数快的做，特别是在做复杂计算时。大多数时候，根本没必要知道底层到底发生了什么，如果需要对 Python 函数加速，将其转换为 TF 函数就行。

另外，当你写的自定义损失函数、自定义指标、自定义层或任何其它自定义函数，并在 Keras 模型中使用的，Keras 都自动将其转换成了 TF 函数，不用使用 `tf.function()`。

提示：创建自定义层或模型时，设置 `dynamic=True`，可以让 Keras 不转化你的 Python 函数。另外，当调用模型的 `compile()` 方法时，可以设置 `run_eagerly=True`。

默认时，TF 函数对每个独立输入的形状和数据类型的集合，生成了一个新的计算图，并缓存以备后续使用。例如，如果你调用 `tf_cube(tf.constant(10))`，就会生成一个 `int32` 张量、形状是 `[]` 的计算图。如果你调用 `tf_cube(tf.constant(20))`，会使用相同的计算图。但如果调用 `tf_cube(tf.constant([10, 20]))`，就会生成一个 `int32`、形状是 `[2]` 的新计算图。这就是 TF 如何处理多态的（即变化的参数类型和形状）。但是，这仅适用于张量参数：如果你将 Python 数值传给 TF，就会为每个独立值创建一个计算图：比如，调用 `tf_cube(10)` 和 `tf_cube(20)` 会产生两个计算图。

警告：如果用多个不同的 Python 数值调用 TF 函数，就会产生多个计算图，这样会减慢程序，使用很多的内存（必须删掉 TF 函数才能释放）。Python 的值应该复赋值给尽量重复的参数，比如超参数，每层有多少个神经元。这可以让 TensorFlow 更好的优化模型中的变量。

## 自动图和跟踪

TensorFlow 是如何生成计算图的呢？它先分析了 Python 函数源码，得出所有的数据流控制语句，比如 `for` 循环，`while` 循环，`if` 条件，还有 `break`、`continue`、`return`。这个第一步被称为自动图（AutoGraph）。TensorFlow 之所以要分析源码，试分析 Python 没有提供任何其它的方式来获取控制流语句：Python 提供了 `__add__()` 和 `__mul__()` 这样的魔术方法，但没有 `__while__()` 或 `__if__()` 这样的魔术方法。分析完源码之后，自动图中的所有控制流语句都被替换成相应的 TensorFlow 方法，比如 `tf.while_loop()`（`while` 循环）和 `tf.cond()`（`if` 判断）。例如，见图 12-4，自动图分析了 Python 函数 `sum_squares()` 的源码，然后变为函

数 `tf_sum_squares()`。在这个函数中，`for` 循环被替换成 `loop_body()`（包括原生的 `for` 循环）。然后是函数 `for_stmt()`，调用这个函数会形成运算 `tf.while_loop()`。

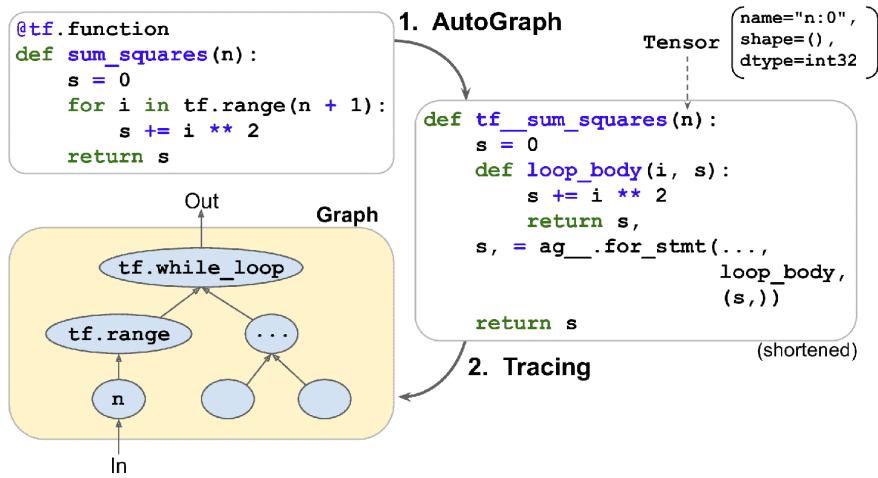


图 12-4 TensorFlow 是如何使用自动图和跟踪生成计算图的？

然后，TensorFlow 调用这个“升级”方法，但没有向其传递参数，而是传递一个符号张量（symbolic tensor）——一个没有任何真实值的张量，只有名字、数据类型和形状。例如，如果调用 `sum_squares(tf.constant(10))`，然后会调用 `tf_sum_squares()`，其符号张量的类型是 `int32`，形状是 `[]`。函数会以图模式运行，意味着每个 TensorFlow 运算会在图中添加一个表示自身的节点，然后输出 `tensor(s)`（与常规模式相对，这被称为动态图执行，或动态模式）。在图模式中，TF 运算不做任何计算。如果你懂 TensorFlow 1，这应该很熟悉，因为图模式是默认模式。在图 12-4 中，可以看到 `tf_sum_squares()` 函数被调用，参数是符号张量，最后的图是跟踪中生成的。节点表示运算，箭头表示张量（生成的函数和图都简化了）。

提示：想看生成出来的函数源码的话，可以调用 `tf.autograph.to_code(sum_squares.python_function)`。源码不美观，但可以用来调试。

## TF 函数规则

大多数时候，将 Python 函数转换为 TF 函数是琐碎的：要用 `@tf.function` 装饰，或让 Keras 来负责。但是，也有一些规则：

- 如果调用任何外部库，包括 NumPy，甚至是标准库，调用只会在跟踪中运行，不会是图的一部分。事实上，TensorFlow 图只能包括 TensorFlow 的构件（张量、运算、变量、数据集，等等）。因此，要确保使用的 `tf.reduce_sum()` 而不是 `np.sum()`，使用的是 `tf.sort()` 而不是内置的 `sorted()`，等等。还要注意：
- 如果定义了一个 TF 函数 `f(x)`，它只返回 `np.random.rand()`，当函数被追踪时，生成的是个随机数，因此 `f(tf.constant(2.))` 和 `f(tf.constant(3.))` 会返回同样的随机数，但 `f(tf.constant([2., 3.]))` 会返回不同的数。如果将 `np.random.rand()` 替换为 `tf.random.uniform([])`，每次调用都会返回新的随机数，因为运算是图的一部分。

- 如果你的非 TensorFlow 代码有副作用（比如日志，或更新 Python 计数器），则 TF 函数被调用时，副作用不一定发生，因为只有函数被追踪时才有效。
- 你可以在 `tf.py_function()` 运算中包装任意的 Python 代码，但这么做的话会使性能下降，因为 TensorFlow 不能做任何图优化。还会破坏移植性，因为图只能在有 Python 的平台上跑起来（且安装上正确的库）。
- 你可以调用其它 Python 函数或 TF 函数，但是它们要遵守相同的规则，因为 TensorFlow 会在计算图中记录它们的运算。注意，其它函数不需要用 `@tf.function` 装饰。
- 如果函数创建了一个 TensorFlow 变量（或任意其它静态 TensorFlow 对象，比如数据集或队列），它必须在第一次被调用时创建 TF 函数，否则会导致异常。通常，最好在 TF 函数的外部创建变量（比如在自定义层的 `build()` 方法中）。如果你想将一个新值赋值给变量，要确保调用它的 `assign()` 方法，而不是使用 `=`。
- Python 的源码可以被 TensorFlow 使用。如果源码用不了（比如，如果是在 Python shell 中定义函数，源码就访问不了，或者部署的是编译文件 `*.pyc`），图的生成就会失败或者缺失功能。
- TensorFlow 只能捕获迭代张量或数据集的 `for` 循环。因此要确保使用 `for i in tf.range(x)`，而不是 `for i in range(x)`，否则循环不能在图中捕获，而是在会在追踪中运行。（如果 `for` 循环使用创建计算图的，这可能是你想要的，比如创建神经网络中的每一层）。
- 出于性能原因，最好使用向量化的实现方式，而不是使用循环。

总结一下，这一章一开始介绍了 TensorFlow，然后是 TensorFlow 的低级 API，包括张量、运算、变量和特殊的数据结构。然后使用这些工具自定义了 `tf.keras` 中的几乎每个组件。最后，学习了 TF 函数如何提升性能，计算图是如何通过自动图和追踪生成的，在写 TF 函数时要遵守什么规则。（附录 G 介绍了生成图的内部黑箱）

下一章会学习如何使用 TensorFlow 高效加载和预处理数据。

## 练习

---

- 如何用一句话描述 TensorFlow？它的主要特点是什么？能列举出其它流行的深度学习库吗？
- TensorFlow 是 NumPy 的简单替换吗？二者有什么区别？
- `tf.range(10)` 和 `tf.constant(np.arange(10))` 能拿到相同的结果吗？
- 列举出除了常规张量之外，TensorFlow 的其它六种数据结构？
- 可以通过函数或创建 `keras.losses.Loss` 的子类来自定义损失函数。两种方法各在什么时候使用？
- 相似的，自定义指标可以通过定义函数或创建 `keras.metrics.Metric` 的子类。两种方法各在什么时候使用？
- 什么时候应该创建自定义层，而不是自定义模型？

8. 什么时候需要创建自定义的训练循环？
9. 自定义 Keras 组件可以包含任意 Python 代码吗，或者 Python 代码需要转换为 TF 函数吗？
10. 如果想让一个函数可以转换为 TF 函数，要遵守什么规则？
11. 什么时候需要创建一个动态 Keras 模型？怎么做？为什么不让所有模型都是动态的？
12. 实现一个具有层归一化的自定义层（第 15 章会用到）：
  - a. `build()` 方法要定义两个可训练权重  $\alpha$  和  $\beta$ ，形状都是 `input_shape[-1:]`，数据类型是 `tf.float32`。 $\alpha$  用 1 初始化， $\beta$  用 0 初始化。
  - b. `call()` 方法要计算每个实例的特征的平均值  $\mu$  和标准差  $\sigma$ 。你可以使用 `tf.nn.moments(inputs, axes=-1, keepdims=True)`，它可以返回平均值  $\mu$  和方差  $\sigma^2$ （计算其平方根得到标准差）。函数返回  $\alpha \cdot (X - \mu) / (\sigma + \epsilon) + \beta$ ，其中  $\diamond$  表示元素级别惩罚， $\epsilon$  是平滑项（避免发生除以 0，而是除以 0.001）。
  - c. 确保自定义层的输出和 `keras.layers.LayerNormalization` 层的输出一致（或非常接近）。
1. 训练一个自定义训练循环，来处理 Fashion MNIST 数据集。
  - a. 展示周期、迭代，每个周期的平均训练损失、平均准确度（每次迭代会更新），还有每个周期结束后的验证集损失和准确度。
  - b. 深层和浅层使用不同的优化器，不同的学习率。

参考答案见附录 A。

## 十三、使用 TensorFlow 加载和预处理数据

译者：[@SeanCheney](#)

目前为止，我们只是使用了存放在内存中的数据集，但深度学习系统经常需要在大数据集上训练，而内存放不下大数据集。其它的深度学习库通过对大数据集做预处理，绕过了内存限制，但 TensorFlow 通过 Data API，使一切都容易了：只需要创建一个数据集对象，告诉它去哪里拿数据，以及如何做转换就行。TensorFlow 负责所有的实现细节，比如多线程、队列、批次和预提取。另外，Data API 和 `tf.keras` 可以无缝配合！

Data API 还可以从现成的文件（比如 CSV 文件）、固定大小的二进制文件、使用 TensorFlow 的 TFRecord 格式的文件（支持大小可变的记录）读取数据。

TFRecord 是一个灵活高效的二进制格式，基于 Protocol Buffers（一个开源二进制格式）。Data API 还支持从 SQL 数据库读取数据。另外，许多开源插件也可以用来从各种数据源读取数据，包括谷歌的 BigQuery。

高效读取大数据集不是唯一的难点：数据还需要进行预处理，通常是归一化。另外，数据集中并不是只有数值字段：可能还有文本特征、类型特征，等等。这些特征需要编码，比如使用独热编码或嵌入（后面会看到，嵌入嵌入是用来标识类型或标记的紧密向量）。预处理的一种方式是写自己的自定义预处理层，另一种是使用 Keras 的标准预处理层。

本章中，我们会介绍 Data API, TFRecord 格式，以及如何创建自定义预处理层，和使用 Keras 的预处理层。还会快速学习 TensorFlow 生态的一些项目：

- **TF Transform (`tf.Transform`)**：可以用来编写单独的预处理函数，它可以在真正训练前，运行在完整训练集的批模式中，然后输出到 TF 函数，插入到训练好的模型中。只要模型在生产环境中部署好了，就能随时预处理新的实例。
- **TF Datasets (TFDS)**：提供了下载许多常见数据集的函数，包括 ImageNet，和数据集对象（可用 Data API 操作）。

### Data API

整个 Data API 都是围绕数据集 `dataset` 的概念展开的：可以猜得到，数据集表示一连串数据项。通常你是用的数据集是从硬盘里逐次读取数据的，简单起见，我们是用 `tf.data.Dataset.from_tensor_slices()` 创建一个存储于内存中的数据集：

```
>>> X = tf.range(10) # any data tensor
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
<TensorSliceDataset shapes: (), types: tf.int32>
```

函数 `from_tensor_slices()` 取出一个张量，创建了一个 `tf.data.Dataset`，它的元素是 `X` 的全部切片，因此这个数据集包括 10 项：张量 0、1、2、...、9。在这个例子中，使用 `tf.data.Dataset.range(10)` 也能达到同样的效果。

可以像下面这样对这个数据集迭代：

```
>>> 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)
```

## 链式转换

有了数据集之后，通过调用转换方法，可以对数据集做各种转换。每个方法会返回一个新的数据集，因此可以将转换像下面这样链接起来（见图 13-1）：

```
>>> 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)
```

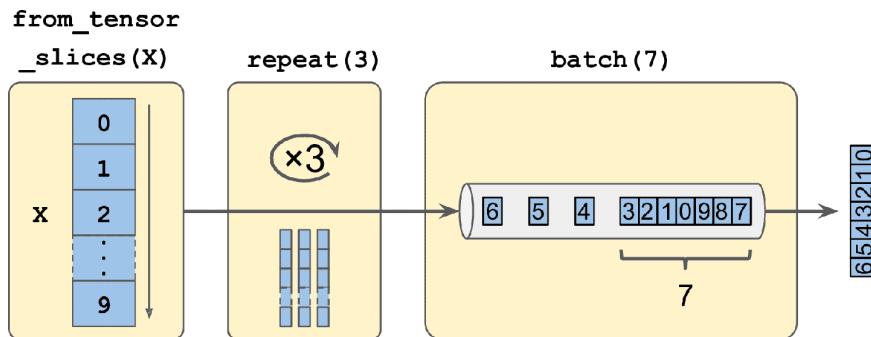


图 13-1 链接数据集转换

在这个例子中，我们先在原始数据集上调用了 `repeat()` 方法，返回了一个重复了原始数据集 3 次的新数据集。当然，这步不会复制数据集中的数据三次（如果调用这个方法时没有加参数，新数据集会一直重复源数据集，必须让迭代代码决定何时退出）。然后我们在新数据集上调用了 `batch()` 方法，这步又产生了一个新数据集。这一步会将上一个数据集的分成 7 个批次。最后，做一下迭代。可以看到，最后的批次只有两个元素，可以设置 `drop_remainder=True`，丢弃最后的两项，将数据对齐。

**警告**：数据集方法不修改数据集，只是生成新的数据集而已，所以要做新数据集的赋值（即使用 `dataset = ...`）。

还可以通过 `map()` 方法转换元素。比如，下面的代码创建了一个每个元素都翻倍的新数据集：

```
>>> dataset = dataset.map(lambda x: x * 2) # Items: [0,2,4,6,8,10,12]
```

这个函数可以用来对数据做预处理。有时可能会涉及复杂的计算，比如改变形状或旋转图片，所以通常需要多线程来加速：只需设置参数 `num_parallel_calls` 就行。注意，传递给 `map()` 方法的函数必须是可以转换为 TF 函数。

`map()` 方法是对每个元素做转换的，`apply()` 方法是对数据整体做转换的。例如，下面的代码对数据集应用了 `unbatch()` 函数（这个函数目前是试验性的，但很有可能加入到以后的版本中）。新数据集中的每个元素都是一个单整数张量，而不是批次大小为 7 的整数。

```
>>> dataset = dataset.apply(tf.data.experimental.unbatch()) # Items: 0, 2, 4, ...
```

还可以用 `filter()` 方法做过滤：

```
>>> dataset = dataset.filter(lambda x: x < 10) # Items: 0 2 4 6 8 0 2 4 6...
```

`take()` 方法可以用来查看数据：

```
>>> 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)
```

## 打散数据

当训练集中的实例是独立同分布时，梯度下降的效果最好（见第 4 章）。实现独立同分布的一个简单方法是使用 `shuffle()` 方法。它能创建一个新数据集，新数据集的前面是一个缓存，缓存中是源数据集的开头元素。然后，无论什么时候取元素，就会从缓存中随便随机取出一个元素，从源数据集中取一个新元素替换。从缓冲器取元素，直到缓存为空。必须要指定缓存的大小，最好大一点，否则随机效果不明显。不要查出内存大小，即使内存够用，缓存超过数据集也是没有意义的。可以提供一个随机种子，如果希望随机的顺序是固定的。例如，下面的代码创建并显示了一个包括 0 到 9 的数据集，重复 3 次，用大小为 5 的缓存做随机，随机种子是 42，批次大小是 7：

```
>>> 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)
```

提示：如果在随机数据集上调用 `repeat()` 方法，默认下，每次迭代的顺序都是新的。通常这样没有问题，但如果你想让每次迭代的顺序一样（比如，测试或调试），可以设置 `reshuffle_each_iteration=False`。

对于内存放不下的大数据集，这个简单的随机缓存方法就不成了，因为缓存相比于数据集就小太多了。一个解决方法是将源数据本身打乱（例如，Linux 可以用 `shuf` 命令打散文本文件）。这样肯定能提高打散的效果！即使源数据打散了，你可能还想再打散一点，否则每个周期可能还会出现同样的顺序，模型最后可能是偏的（比如，源数据顺序偶然导致的假模式）。为了将实例进一步打散，一个常用的方法是将源数据分成多个文件，训练时随机顺序读取。但是，相同文件中的实例仍然靠的太近。为了避免这点，可以同时随机读取多个文件，做交叉。在最顶层，可以用 `shuffle()` 加一个随机缓存。如果这听起来很麻烦，不用担心：Data API 都为你实现了，几行代码就行。

## 多行数据交叉

首先，假设加载了加州房价数据集，打散它（除非已经打散了），分成训练集、验证集、测试集。然后将每个数据集分成多个 csv 文件，每个如下所示（每行包含 8 个输入特征加上目标中位房价）：

```
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
[...]
```

再假设 `train_filepaths` 包括了训练文件路径的列表（还要 `valid_filepaths` 和 `test_filepaths`）：

```
>>> train_filepaths
['datasets/housing/my_train_00.csv', 'datasets/housing/my_train_01.csv', ...]
```

另外，可以使用文件模板，比

如 `train_filepaths = "datasets/housing/my_train_*.csv"`。现在，创建一个数据集，包括这些文件路径：

```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

默认，`list_files()` 函数返回一个文件路径打散的数据集。也可以设置 `shuffle=False`，文件路径就不打散了。

然后，可以调用 `leave()` 方法，一次读取 5 个文件，做交叉操作（跳过第一行表头，使用 `skip()` 方法）：

```
n_readers = 5
dataset = filepath_dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
    cycle_length=n_readers)
```

`interleave()` 方法会创建一个数据集，它从 `filepath_dataset` 读 5 条文件路径，对每条路径调用函数（例子中是用的匿名函数）来创建数据集（例子中是 `TextLineDataset`）。为了更清楚点，这一步总欧诺个由七个数据集：文件路径数据集，交叉数据集，和五个 `TextLineDatasets` 数据集。当迭代交叉数据集时，会循环 `TextLineDatasets`，每次读取一行，知道数据集为空。然后会从 `filepath_dataset` 再获取五个文件路径，做同样的交叉，直到文件路径为空。

**提示：**为了交叉得更好，最好让文件有相同的长度，否则长文件的尾部不会交叉。

默认情况下，`interleave()` 不是并行的，只是顺序从每个文件读取一行。如果想变成并行读取文件，可以设定参数 `num_parallel_calls` 为想要的线程数（`map()` 方法也有这个参数）。还可以将其设置为 `tf.data.experimental.AUTOTUNE`，让 TensorFlow 根据 CPU 自己找到合适的线程数（目前这是个试验性的功能）。看看目前数据集包含什么：

```
>>> 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'
```

忽略表头行，这是五个 csv 文件的第一行，随机选取的。看起来不错。但是也看到了，都是字节串，需要解析数据，缩放数据。

## 预处理数据

实现一个小函数来做预处理：

```
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
```

逐行看下代码：

- 首先，代码假定已经算好了训练集中每个特征的平均值和标准差。`X_mean` 和 `X_std` 是 1D 张量（或 NumPy 数组），包含八个浮点数，每个都是特征。
- `preprocess()` 函数从 csv 取一行，开始解析。使用 `tf.io.decode_csv()` 函数，接收两个参数，第一个是要解析的行，第二个是一个数组，包含 csv 文件每列的默认值。这个数组不仅告诉 TensorFlow 每列的默认值，还有总列数和数据类型。在这个例子中，是告诉 TensorFlow，所有特征列都是浮点数，缺失值默认为，但提供了一个类型是 `tf.float32` 的空数组，作为最后一列（目标）的默认值：数组告诉 TensorFlow 这一列包含浮点数，但没有默认值，所以碰到空值时会报异常。
- `decode_csv()` 函数返回一个标量张量（每列一个）的列表，但应该返回 1D 张量数组。所以在所有张量上调用了 `tf.stack()`，除了最后一个。然后对目标值做同样的操作（让其成为只包含一个值，而不是标量张量的 1D 张量数组）。
- 最后，对特征做缩放，减去平均值，除以标准差，然后返回包含缩放特征和目标值的元组。

测试这个预处理函数：

```
>>> 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)>)
```

很好，接下来将函数应用到数据集上。

## 整合

为了让代码可复用，将前面所有讨论过的东西编程一个小函数：创建并返回一个数据集，可以高效从多个 csv 文件加载加州房价数据集，做预处理、打散、选择性重复，做批次（见图 3-2）：

```
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)
```

代码条理很清晰，除了最后一行的 `prefetch(1)`，对于提升性能很关键。

## 预提取

通过调用 `prefetch(1)`，创建了一个高效的数据集，总能提前一个批次。换句话说，当训练算法在一个批次上工作时，数据集已经准备好下一个批次了（从硬盘读取数据并做预处理）。这样可以极大提升性能，解释见图 13-3。如果加载和预处理还是多线程的（通过设置 `interleave()` 和 `map()` 的 `num_parallel_calls`），可以利用多 CPU，准备批次数据可以比在 GPU 上训练还快：这样 GPU 就可以 100% 利用起来了（排除数据从 CPU 传输到 GPU 的时间），训练可以快很多。

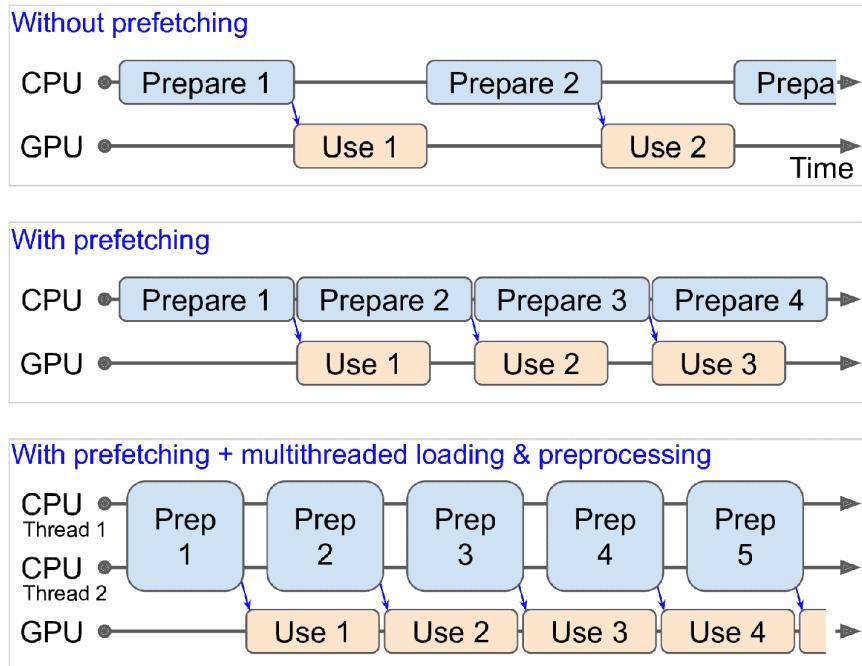


图 13-3 通过预提取，让 CPU 和 GPU 并行工作：GPU 在一个批次上工作时，CPU 准备下一个批次

**提示：**如果想买一块 GPU 显卡的话，它的处理能力和显存都是非常重要的。另一个同样重要的，是显存带宽，即每秒可以进入或流出内存的 GB 数。

如果数据集不大，内存放得下，可以使用数据集的 `cache()` 方法将数据集存入内存。通常这步是在加载和预处理数据之后，在打散、重复、分批次之前。这样做的话，每个实例只需做一次读取和处理，下一个批次仍能提前准备。

你现在知道如何搭建高效输入管道，从多个文件加载和预处理数据了。我们讨论了最常用的数据集方法，但还有一些你可能感兴趣：`concatenate()`、`zip()`、`window()`、`reduce()`、`shard()`、`flat_map()` 和 `padded_batch()`。还有两个类方法：`from_generator()` 和 `from_tensors()`，它们能从 Python 生成器或张量列表创建数据集。更多细节请查看 API 文档。`tf.data.experimental` 中还有试验性功能，其中许多功能可能会添加到未来版本中。

## tf.keras 使用数据集

现在可以使用 `csv_reader_dataset()` 函数为训练集创建数据集了。注意，不需要将数据重复，`tf.keras` 会做重复。还为验证集和测试集创建了数据集：

```
train_set = csv_reader_dataset(train_filepaths)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

现在就可以利用这些数据集来搭建和训练 Keras 模型了。我们要做的就是将训练和验证集传递给 `fit()` 方法，而不是 `X_train`、`y_train`、`X_valid`、`y_valid`：

```
model = keras.models.Sequential([...])
model.compile([...])
model.fit(train_set, epochs=10, validation_data=valid_set)
```

相似的，可以将数据集传递给 `evaluate()` 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
```

跟其它集合不同，`new_set` 通常不包含标签（如果包含标签，也会被 Keras 忽略）。注意，在所有这些情况下，还可以使用 NumPy 数组（但仍需要加载和预处理）。

如果你想创建自定义训练循环（就像 12 章那样），你可以在训练集上迭代：

```
for X_batch, y_batch in train_set:
    [...] # perform one Gradient Descent step
```

事实上，还可以创建一个 TF 函数（见第 12 章）来完成整个训练循环：

```
@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))
```

祝贺，你现在知道如何使用 Data API 创建强大的输入管道了！但是，目前为止我们使用的 CSV 文件，虽然常见又简单方便，但不够高效，不支持大或复杂的数据结构（比如图片或音频）。这就是 TFRecord 要解决的。

提示：如果你对 csv 文件感到满意（或其它任意格式），就不必使用 TFRecord。就像老话说的，只要没坏就别修！TFRecord 是为解决训练过程中加载和解析数据时碰到的瓶颈。

## TFRecord 格式

TFRecord 格式是 TensorFlow 偏爱的存储大量数据并高效读取的数据。它是非常简单的二进制格式，只包含不同大小的二进制记录的数据（每个记录包括一个长度、一个 CRC 校验和，校验和用于检查长度是否正确，真是的数据，和一个数据的 CRC 校验和，用于检查数据是否正确）。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")
```

然后可以使用 `tf.data.TFRecordDataset` 来读取一个或多个 TFRecord 文件：

```
filepaths = ["my_data.tfrecord"]
dataset = tf.data.TFRecordDataset(filepaths)
for item in dataset:
    print(item)
```

输出是：

```
tf.Tensor(b"This is the first record", shape=(), dtype=string)
tf.Tensor(b"And this is the second record", shape=(), dtype=string)
```

提示：默认情况下，`TFRecordDataset` 会逐一读取数据，但通过设定 `num_parallel_reads` 可以并行读取并交叉数据。另外，你可以使用 `list_files()` 和 `interleave()` 获得同样的结果。

## 压缩 TFRecord 文件

有的时候压缩 TFRecord 文件很有必要，特别是当需要网络传输的时候。你可以通过设定 `options` 参数，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:
    [...]
```

当读取压缩 TFRecord 文件时，需要指定压缩类型：

```
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                  compression_type="GZIP")
```

## 简要介绍协议缓存

即便每条记录可以使用任何二进制格式，TFRecord 文件通常包括序列化的协议缓存（也称为 protobuf）。这是一种可移植、可扩展的高效二进制格式，是谷歌在 2001 年开发，并在 2008 年开源的；协议缓存现在使用广泛，特别是在 gRPC，谷歌的远程调用系统中。定义语言如下：

```

syntax = "proto3";
message Person {
    string name = 1;
    int32 id = 2;
    repeated string email = 3;
}

```

定义写道，使用的是协议缓存的版本 3，指定每个 `Person` 对象可以有一个 `name`，类型是字符串，类型是 `int32` 的 `id`，0 个或多个 `email` 字段，每个都是字符串。数字 1、2、3 是字段标识符：用于每条数据的二进制表示。当你在 `.proto` 文件中有了一个定义，就可以编译了。这就需要 `protoc`，协议缓存编译器，来生成 Python（或其它语言）的访问类。注意，要使用的缓存协议的定义已经编译好了，它们的 Python 类是 TensorFlow 的一部分，所以就不必使用 `protoc` 了。你需要知道的知识如何使用 Python 的缓存协议访问类。为了讲解，看一个简单的例子，使用访问类来生成 `Person` 缓存协议：

```

>>> from person_pb2 import Person # 引入生成的访问类
>>> person = Person(name="Al", id=123, email=["a@b.com"]) # 创建一个 Person
>>> print(person) # 展示 Person
name: "Al"
id: 123
email: "a@b.com"
>>> person.name # 读取一个字段
"Al"
>>> person.name = "Alice" # 修改一个字段
>>> person.email[0] # 重复的字段可以像数组一样访问
"a@b.com"
>>> person.email.append("c@d.com") # 添加 email 地址
>>> s = person.SerializeToString() # 将对象序列化为字节串
>>> s
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person() # 创建一个新 Person
>>> person2.ParseFromString(s) # 解析字节串 (字节长度 27)
27
>>> person == person2 # 现在相等
True

```

简而言之，我们引入了 `protoc` 生成的类 `Person`，创建了一个实例，展示、读取、并写入新字段，然后使用 `SerializeToString()` 将其序列化。序列化的数据就可以保存或通过网络传输了。当读取或接收二进制数据时，可以用 `ParseFromString()` 方法来解析，就得到了序列化对象的复制。

可以将序列化的 `Person` 对象存储为 `TFRecord` 文件，然后可以加载和解析。但是 `SerializeToString()` 和 `ParseFromString()` 不是 TensorFlow 运算（这段代码中的其它代码也不是 TensorFlow 运算），因此 TensorFlow 函数中不能含有这两个方法（除非将其包装进 `tf.py_function()` 运算，但会使代码速度变慢，移植性变差）。幸好，TensorFlow 还有提供了解析运算的特殊协议缓存。

## TensorFlow 协议缓存

`TFRecord` 文件主要使用的协议缓存是 `Example`，它表示数据集中的一个实例，包括命名特征的列表，每个特征可以是字节串列表、或浮点列表、或整数列表。下面是一个协议缓存的定义：

```

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; };

```

`BytesList`、`FloatList`、`Int64List` 的定义都很清楚。注意，重复的数值字段使用了 `[packed = true]`，目的是高效编码。`Feature` 包含的是 `BytesList`、`FloatList`、`Int64List` 三者之一。`Features`（带 s）是包含特征名和对应特征值的字典。最后，一个 `Example` 值包含一个 `Features` 对象。下面是一个如何创建 `tf.train.Example` 的例子，表示的是之前同样的人，并存储为 `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"]))
        })
)

```

这段代码有点冗长和重复，但很清晰（可以很容易将其包装起来）。现在有了 `Example` 协议缓存，可以调用 `SerializeToString()` 方法将其序列化，然后将结果数据存入 `TFRecord` 文件：

```

with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    f.write(person_example.SerializeToString())

```

通常需要写不止一个 `Example`！一般来说，你需要写一个转换脚本，读取当前格式（例如 `csv`），为每个实例创建 `Example` 协议缓存，序列化并存储到若干 `TFRecord` 文件中，最好再打散。这些需要花费不少时间，如有必要再这么做（也许 `CSV` 文件就足够了）。

有了序列化好的 `Example` `TFRecord` 文件之后，就可以加载了。

## 加载和解析 `Example`

要加载序列化的 `Example` 协议缓存，需要再次使用 `tf.data.TFRecordDataset`，使用 `tf.io.parse_single_example()` 解析每个 `Example`。这是一个 `TensorFlow` 运算，所以可以包装进 `TF` 函数。它至少需要两个参数：一个包含序列化数据的字符串张量，和每个特征的描述。描述是一个字典，将每个特征名映射到 `tf.io.FixedLenFeature` 描述符，描述符指明特征的形状、类型和默认值，或（当特征列表长度可能变化时，比如 `"email"` 特征）映射到 `tf.io.VarLenFeature` 描述符，它只指向类型。

下面的代码定义了描述字典，然后迭代 `TFRecordDataset`，解析序列化的 `Example` 协议缓存：

```

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)

```

长度固定的特征会像常规张量那样解析，而长度可变的特征会作为稀疏张量解析。可以使用 `tf.sparse.to_dense()` 将稀疏张量转变为紧密张量，但只是简化了值的访问：

```

>>> 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'], [...])>

```

`BytesList` 可以包含任意二进制数据，序列化对象也成。例如，可以使 `tf.io.encode_jpeg()` 将图片编码为 JPEG 格式，然后将二进制数据放入 `BytesList`。然后，当代码读取 `TFRecord` 时，会从解析 `Example` 开始，再调用 `tf.io.decode_jpeg()` 解析数据，得到原始图片（或者可以使 `tf.io.decode_image()`，它能解析任意 BMP、GIF、JPEG、PNG 格式）。你还可以通过 `tf.io.serialize_tensor()` 序列化张量，将结果字节串放入 `BytesList` 特征，将任意张量存储在 `BytesList` 中。之后，当解析 `TFRecord` 时，可以使用 `tf.io.parse_tensor()` 解析数据。

除了使用 `tf.io.parse_single_example()` 逐一解析 `Example`，你还可以通过 `tf.io.parse_example()` 逐批次解析：

```

dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(10)
for serialized_examples in dataset:
    parsed_examples = tf.io.parse_example(serialized_examples,
                                          feature_description)

```

可以看到 `Example` 协议缓存对大多数情况就足够了。但是，如果处理的是嵌套列表，就会比较麻烦。比如，假设你想分类文本文档。每个文档可能都是句子的列表，而每个句子又是词的列表。每个文档可能还有评论列表，评论又是词的列表。可能还有上下文数据，比如文档的作者、标题和出版日期。`TensorFlow` 的 `SequenceExample` 协议缓存就是为了处理这种情况的。

## 使用 `SequenceExample` 协议缓存处理嵌套列表

下面是 `SequenceExample` 协议缓存的定义：

```

message FeatureList { repeated Feature feature = 1; };
message FeatureLists { map<string, FeatureList> feature_list = 1; };
message SequenceExample {
    Features context = 1;
    FeatureLists feature_lists = 2;
};

```

`SequenceExample` 包括一个上下文数据的 `Features` 对象，和一个包括一个或多个命名 `FeatureList` 对象（比如，一个 `FeatureList` 命名为 "content"，另一个命名为 "comments"）的 `FeatureLists` 对象。每个 `FeatureList` 包含 `Feature` 对象的列表，每个 `Feature` 对象可能是字节串、64 位整数或浮点数的列表（这个例子中，每个 `Feature` 表示的是一个句子或一条评论，格式或许是词的列表）。创

建 `SequenceExample`，将其序列化、解析，和创建、序列化、解析 `Example` 很像，但必须要使用 `tf.io.parse_single_sequence_example()` 来解析单个的 `SequenceExample` 或用 `tf.io.parse_sequence_example()` 解析一个批次。两个函数都是返回一个包含上下文特征（字典）和特征列表（也是字典）的元组。如果特征列表包含大小可变的序列（就像前面的例子），可以将其转化为嵌套张量，使用 `tf.RaggedTensor.from_sparse()`：

```
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"])
```

现在你就知道如何高效存储、加载和解析数据了，下一步是准备数据。

## 预处理输入特征

为神经网络准备数据需要将所有特征转变为数值特征，做一些归一化工作等等。特别的，如果数据包括类型特征或文本特征，也需要转变为数字。这些工作可以在准备数据文件的时候做，使用 NumPy、Pandas、Scikit-Learn 这样的工作。或者，可以在用 Data API 加载数据时，实时预处理数据（比如，使用数据集的 `map()` 方法，就像前面的例子），或者可以给模型加一个预处理层。接下来，来看最后一种方法。

例如，这个例子是使用 `Lambda` 层实现标准化层。对于每个特征，减去其平均值，再除以标准差（再加上一个平滑项，避免 0 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)),
    [...] # 其它层
])
```

并不难。但是，你也许更想要一个独立的自定义层（就像 Scikit-Learn 的 `StandardScaler`），而不是像 `means` 和 `stds` 这样的全局变量：

```
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())
```

使用这个标准化层之前，你需要使用 `adapt()` 方法将其适配到数据集样本。这么做就能使用每个特征的平均值和标准差：

```
std_layer = Standardization()
std_layer.adapt(data_sample)
```

这个样本必须足够大，可以代表数据集，但不必是完整的训练集：通常几百个随机实例就够了（但还是要取决于任务）。然后，就可以像普通层一样使用这个预处理层了：

```
model = keras.Sequential()
model.add(std_layer)
[...] # create the rest of the model
model.compile([...])
model.fit([...])
```

可能以后还会有 `keras.layers.Normalization` 层，和这个自定义 `Standardization` 层差不多：先创建层，然后对数据集做适配（向 `adapt()` 方法传递样本），最后像普通层一样使用。

接下来看看类型特征。先将其编码为独热向量。

## 使用独热向量编码类型特征

考虑下第 2 章中的加州房价数据集的 `ocean_proximity` 特征：这是一个类型特征，有五个

值：`"<1H OCEAN"`、`"INLAND"`、`"NEAR OCEAN"`、`"NEAR BAY"`、`"ISLAND"`。输入给神经网络之前，需要对其进行编码。因为类型不多，可以使用独热编码。先将每个类型映射为索引（0 到 4），使用一张查询表：

```
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)
```

逐行看下代码：

- 先定义词典：也就是所有类型的列表。
- 然后创建张量，具有索引 0 到 4。
- 接着，创建查找表的初始化器，传入类型列表和对应索引。在这个例子中，因为已经有了数据，所以直接用 `KeyValueTensorInitializer` 就成了；但如果类型是在文本中（一行一个类型），就要使用 `TextFileInitializer`。
- 最后两行创建了查找表，传入初始化器并指明未登录词（oov）桶的数量。如果查找的类型不在词典中，查找表会计算这个类型的哈希，使用哈希分配一个未知的类型给未登录词桶。索引序号接着现有序号，所以这个例子中的两个未登录词的索引是 5 和 6。

为什么使用桶呢？如果类型数足够大（例如，邮编、城市、词、产品、或用户），数据集也足够大，或者数据集持续变化，这样的话，获取类型的完整列表就不容易了。一个解决方法是根据数据样本定义（而不是整个训练集），为其它不在样本中的类型加上一些未登录词桶。训练中碰到的未知类型越多，要使用的未登录词桶就要越多。事实上，如果未登录词桶的数量不够，就会发生碰撞：不同的类型会出现在同一个桶中，所以神经网络就无法区分了。

现在用查找表将小批次的类型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)>
```

可以看到，"NEAR BAY" 映射到了索引 3，未知类型 "DESERT" 映射到了两个未登录词桶之一（索引 5），"INLAND" 映射到了索引 1 两次。然后使用 `tf.one_hot()` 来做独热编码。注意，需要告诉该函数索引的总数量，索引总数等于词典大小加上未登录词桶的数量。现在你就知道如何用 TensorFlow 将类型特征编码为独热向量了。

和之前一样，将这些操作写成一个独立的类并不难。`adapt()` 方法接收一个数据样本，提取其中的所有类型。创建一张查找表，将类型和索引映射起来。`call()` 方法会使用查找表将输入类型和索引建立映射。目前，Keras 已经有了一个名为 `keras.layers.TextVectorization` 的层，它的功能就是上面这样：`adapt()` 从样本中提取词表，`call()` 将每个类型映射到词表的索引。如果要将索引变为独热向量的话，可以将这个层添加到模型开始的地方，后面再加一个可以用 `tf.one_hot()` 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表长度加上未登录词桶的大小。当类型不多时，这么做可以，但如果词表很大，最好使用“嵌入”来做。

**提示：**一个重要的原则，如果类型数小于 10，可以使用独热编码。如果类型超过 50 个（使用哈希桶时通常如此），最好使用嵌入。类型数在 10 和 50 之间时，最好对两种方法做个试验，看哪个更合适。

## 使用嵌入编码类型特征

嵌入是一个可训练的表示类型的紧密向量。默认时，嵌入是随机初始化的，"NEAR BAY" 可能初始化为 `[0.131, 0.890]`，"NEAR OCEAN" 可能初始化为 `[0.631, 0.791]`。

这个例子中，使用的是 2D 嵌入，维度是一个可调节的超参数。因为嵌入是可以训练的，它能在训练中提高性能；当嵌入表示相似的类时，梯度下降会使相似的嵌入靠的更近，而 "INLAND" 会偏的更远（见图 13-4）。事实上，表征的越好，越利于神经网络做出准确的预测，而训练会让嵌入更好的表征类型，这被称为表征学习（第 17 章会介绍其它类型的表征学习）。

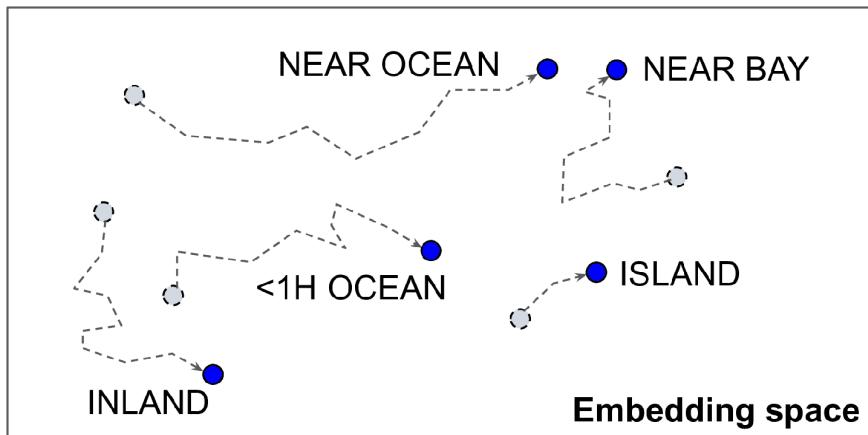


图 13-4 嵌入的表征会在训练中提高

## 词嵌入

嵌入不仅可以实现当前任务的表征，同样的嵌入也可以用于其它的任务。最常见的例子是词嵌入（即，单个词的嵌入）：对于自然语言处理任务，最好使用预训练的词嵌入，而不是使用自己训练的。

使用向量表征词可以追溯到 1960 年代，许多复杂的技术用于生成向量，包括使用神经网络。进步发生在 2013 年，Tomáš Mikolov 和谷歌其它的研究院发表了一篇论文《[Distributed Representations of Words and Phrases and their Compositionality](#)》，介绍了一种用神经网络学习词嵌入的技术，效果远超以前的技术。可以实现在大文本语料上学习嵌入：用神经网络预测给定词附近的词，得到了非常好的词嵌入。例如，同义词有非常相近的词嵌入，语义相近的词，比如法国、西班牙和意大利靠的也很近。

不止是相近：词嵌入在嵌入空间的轴上的分布也是有意义的。下面是一个著名的例子：如果计算 `King - Man + Woman`，结果与 `Queen` 非常相近（见图 13-5）。换句话说，词嵌入编码了性别。相似的，可以计算 `Madrid - Spain + France`，结果和 `Paris` 很近。

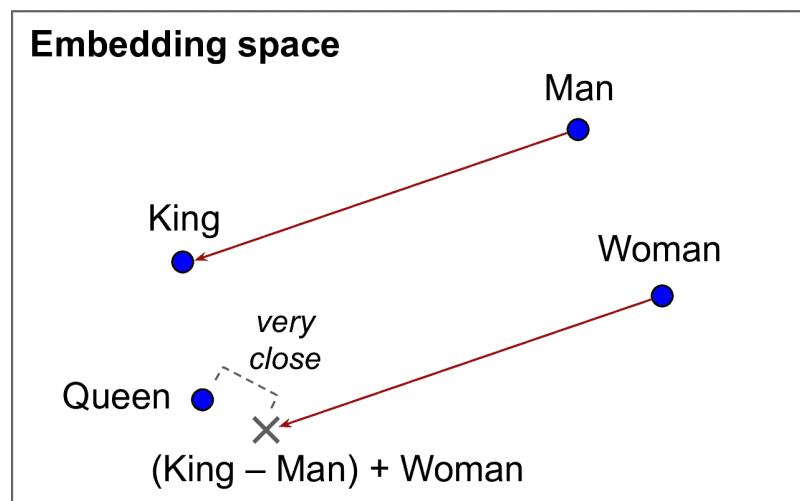


图 13-5 相似词的词嵌入也相近，一些轴编码了概念

但是，词嵌入有时偏差很大。例如，尽管词嵌入学习到了男人是国王，女人是王后，词嵌入还学到了男人是医生、女人是护士。这是非常大的性别偏差。

来看下如何手动实现嵌入。首先，需要创建一个包含每个类型嵌入（随机初始化）的嵌入矩阵。每个类型就有一行，每个未登录词桶就有一行，每个嵌入维度就有一列：

```
embedding_dim = 2
embed_init = tf.random.uniform([len(vocab) + num_oov_buckets, embedding_dim])
embedding_matrix = tf.Variable(embed_init)
```

这个例子用的是 2D 嵌入，通常的嵌入是 10 到 300 维，取决于任务和词表大小（需要调节词表大小超参数）。

嵌入矩阵是一个随机的  $6 \times 2$  矩阵，存入一个变量（因此可以在训练中被梯度下降调节）：

```
>>> 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)>
```

使用嵌入编码之前的类型特征：

```
>>> 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()` 函数根据给定的索引在嵌入矩阵中查找行。例如，查找表说 "INLAND" 类型位于索引 1，`tf.nn.embedding_lookup()` 就返回嵌入矩阵的行 1：[0.3528825, 0.46448255]。

Keras 提供了 `keras.layers.Embedding` 层来处理嵌入矩阵（默认可训练）；当这个层初始化时，会随机初始化嵌入矩阵，当被调用时，就返回索引所在的嵌入矩阵的那行：

```
>>> 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)>
```

将这些内容放到一起，创建一个 Keras 模型，可以处理类型特征（和数值特征），学习每个类型（和未登录词）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])
```

这个模型有两个输入：一个常规输入，每个实例包括 8 个数值特征，机上一个类型特征。使用 `Lambda` 层查找每个类型的索引，然后用索引查找嵌入。接着，将嵌入和常规输入连起来，作为编码输入进神经网络。此时可以加入任意种类的神经网络，但只是添加了一个紧密输出层。

当 `keras.layers.TextVectorization` 准备好之后，可以调用它的 `adapt()` 方法，从数据样本提取词表（会自动创建查找表）。然后加入到模型中，就可以执行索引查找了（替换前面代码的 `Lambda` 层）。

**笔记：**独热编码加紧密层（没有激活函数和偏差项），等价于嵌入层。但是，嵌入层用的计算更少（嵌入矩阵越大，性能差距越明显）。紧密层的权重矩阵扮演的是嵌入矩阵的角色。例如，大小为 20 的独热向量和 10 个单元的紧密层加起来，等价于 `input_dim=20`、`output_dim=10` 的嵌入层。作为结果，嵌入的维度超过后面的层的神经元数是浪费的。

再进一步看看 Keras 的预处理层。

## Keras 预处理层

Keras 团队打算提供一套标准的 Keras 预处理层，现在已经可用，[链接](#)。新的 API 可能会覆盖旧的 Feature Columns API。

我们已经讨论了其中的两个：`keras.layers.Normalization` 用来做特征标准化，`TextVectorization` 层用于将文本中的词编码为词典的索引。对于这两个层，都是用数据样本调用它的 `adapt()` 方法，然后如常使用。其它的预处理层也是这么使用的。

API 中还提供了 `keras.layers.Discretization` 层，它能将连续数据切成不同的组，将每个组赋值为独热向量。例如，可以用它将价格分成三类，低、中、高，编码为 `[1, 0, 0]`、`[0, 1, 0]`、`[0, 0, 1]`。当然，这么做会损失很多信息，但有时，相对于连续数据，这么做可以发现不那么明显的规律。

**警告：** `Discretization` 层是不可微的，只能在模型一开始使用。事实上，模型的预处理层会在训练时冻结，因此预处理层的参数不会被梯度下降影响，所以可以是不可微的。这还意味着，如果想让预处理层可训练的话，不能在自定义预处理层上直接使用嵌入层，而是应该像前文的例子那样分开来做。

还可以用类 `PreprocessingStage` 将多个预处理层链接起来。例如，下面的代码创建了一个预处理管道，先将输入归一化，然后离散（有点类似 Scikit-Learn 的管道）。当将这个管道应用到数据样本时，可以作为常规层使用（还得是在模型的前部，因为包含不可微分的预处理层）：

```
normalization = keras.layers.Normalization()
discretization = keras.layers.Discretization([...])
pipeline = keras.layers.PreprocessingStage([normalization, discretization])
pipeline.adapt(data_sample)
```

`TextVectorization` 层也有一个选项用于输出词频向量，而不是词索引。例如，如果词典包括三个词，比如 `["and", "basketball", "more"]`，则 `"more and more"` 会映射为 `[1, 0, 2]`：`"and"` 出现了一次，`"basketball"` 没有出现，`"more"` 出现了两次。这种词表征称为词袋，因为它完全失去了词的顺序。常见词，比如 `"and"`，会在文本中有更高的值，尽管没什么实际意义。因此，词频向量中应该降低常见词的影响。一个常见的方法是将词频除以出现该词的文档数的对数。这种方法称为词频-逆文档频率（TF-IDF）。例如，假设 `"and"`、`"basketball"`、`"more"` 分别出

现在了 200、10、100 个文档中：最终的向量应该是 `[1/log(200), 0/log(10), 2/log(100)]`，大约是 `[0.19, 0., 0.43]`。TextVectorization 层会有 TF-IDF 的选项。

笔记：如果标准预处理层不能满足你的任务，你还可以选择创建自定义预处理层，就像前面的 Standardization。创建一个 `keras.layers.PreprocessingLayer` 子类，`adapt()` 方法用于接收一个 `data_sample` 参数，或者再有一个 `reset_state` 参数：如果是 `True`，则 `adapt()` 方法在计算新状态之前重置现有的状态；如果是 `False`，会更新现有的状态。

可以看到，这些 Keras 预处理层可以使预处理更容易！现在，无论是自定义预处理层，还是使用 Keras 的，预处理都可以实时进行了。但在训练中，最好再提前进行预处理。下面来看看为什么，以及怎么做。

## TF Transform

预处理非常消耗算力，训练前做预处理相对于实时处理，可以极大的提高速度：数据在训练前，每个实例就处理一次，而不是在训练中每个实例在每个周期就处理一次。前面提到过，如果数据集小到可以存入内存，可以使用 `cache()` 方法。但如果太大，可以使用 Apache Beam 或 Spark。它们可以在大数据上做高效的数据预处理，还可以分布进行，使用它们就能在训练前处理所有训练数据了。

虽然训练加速了，但带来一个问题：一旦模型训练好了，假如想部署到移动 app 上，还是需要写一些预处理数据的代码。假如想部署到 TensorFlow.js，还是需要预处理代码。这是一个维护难题：无论何时想改变预处理逻辑，都需要更新 Apache Beam 的代码、移动端代码、JavaScript 代码。不仅耗时，也容易出错：不同端的可能有细微的差别。训练/实际产品表现之间的偏差会导致 bug 或使效果大打折扣。

一种解决办法是在部署到 app 或浏览器之前，给训练好的模型加上额外的预处理层，来做实时的预处理。这样好多了，只有两套代码 Apache Beam 或 Spark 代码，和预处理层代码。

如果只需定义一次预处理操作呢？这就是 TF Transform 要做的。TF Transform 是 TensorFlow Extended (TFX) 的一部分，这是一个端到端的 TensorFlow 模型生产化平台。首先，需要安装（TensorFlow 没有捆绑）。然后通过 TF Transform 函数来做缩放、分桶等操作，一次性定义预处理函数。你还可以使用任意需要的 TensorFlow 运算。如果只有两个特征，预处理函数可能如下：

```
import tensorflow_transform as tft

def preprocess(inputs): # inputs = 输入特征批次
    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
    }
```

然后，TF Transform 可以使用 Apache Beam（可以使用其 `AnalyzeAndTransformDataset` 类）在整个训练集上应用这个 `preprocess()` 函数。在使用过程中，还会计算整个训练集上的必要统计数据：这个例子中，

是 `housing_median_age` 和 `the_ocean_proximity` 的平均值和标准差。计算这些数据的组件称为分析器。

更重要的，TF Transform 还会生成一个等价的 TensorFlow 函数，可以放入部署的模型中。这个 TF 函数包括一些常量，对应于 Apache Beam 的统计值（平均值、标准差和词典）。

有了 Data API、TFRecord、Keras 预处理层和 TF Transform，可以为训练搭建高度伸缩的输入管道，可以是生产又快，迁移性又好。

但是，如果只想使用标准数据集呢？只要使用 TFDS 就成了。

## TensorFlow Datasets (TFDS) 项目

从 [TensorFlow Datasets](#) 项目，可以非常方便的下载一些常见的数据集，从小数据集，比如 MNIST 或 Fashion MNIST，到大数据集，比如 ImageNet（需要大硬盘）。包括了图片数据集、文本数据集（包括翻译数据集）、和音频视频数据集。可以访问[这里](#)，查看完整列表，每个数据集都有介绍。

TensorFlow 没有捆绑 TFDS，所以需要使用 PIP 安装库 `tensorflow-datasets`。然后调用函数 `tfds.load()`，就能下载数据集了（除非之前下载过），返回的数据是数据集的字典（通常是一个是训练集，一个是测试集）。例如，下载 MNIST：

```
import tensorflow_datasets as tfds
dataset = tfds.load(name="mnist")
mnist_train, mnist_test = dataset["train"], dataset["test"]
```

然后可以对其应用任意转换（打散、批次、预提取），然后就可以训练模型了。下面是一个简单的例子：

```
mnist_train = mnist_train.shuffle(10000).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    [...]
```

**提示：** `load()` 函数打散了每个下载的数据分片（只是对于训练集）。但还不够，最好再自己做打散。

注意，数据集中的每一项都是一个字典，包含特征和标签。但 Keras 期望每项都是一个包含两个元素（特征和标签）的元组。可以使用 `map()` 对数据集做转换，如下：

```
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)
```

更简单的方式是让 `load()` 函数来做这个工作，只要设定 `as_supervised=True`（显然这仅适用于有标签的数据集）。你还可以将数据集直接传给 `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)
```

这一章很技术，你可能觉得没有神经网络的抽象美，但事实是深度学习经常要涉及大数据集，知道如何高效加载、解析和预处理，是一个非常重要的技能。下一章会学习卷积神经网络，它是一种用于图像处理和其它应用的、非常成功的神经网络。

## 练习

1. 为什么要使用 Data API ?
  2. 将大数据分成多个文件有什么好处 ?
  3. 训练中，如何断定输入管道是瓶颈？如何处理瓶颈 ?
  4. 可以将任何二进制数据存入 TFRecord 文件吗，还是只能存序列化的协议缓存 ?
  5. 为什么要将数据转换为示例协议缓存 ? 为什么不使用自己的协议缓存 ?
  6. 使用 TFRecord 时，什么时候要压缩 ? 为什么不系统化的做 ?
  7. 数据预处理可以在写入数据文件时，或在 `tf.data` 管道中，或在预处理层中，或使用 TF Transform。这几种方法各有什么优缺点 ?
  8. 说出几种常见的编码类型特征的方法。文本如何编码 ?
9. 加载 Fashion MNIST 数据集；将其分成训练集、验证集和测试集；打散训练集；将每个数据及对应多个 TFRecord 文件。每条记录应该是有两个特征的序列化的示例协议缓存：序列化的图片（使用 `tf.io.serialize_tensor()` 序列化每张图片），和标签。然后使用 `tf.data` 为每个集合创建一个高效数据集。最后，使用 Keras 模型训练这些数据集，用预处理层标准化每个特征。让输入管道越高效越好，使用 TensorBoard 可视化地分析数据。
1. 在这道题中，你要下载一个数据集，分割它，创建一个 `tf.data.Dataset`，用于高效加载和预处理，然后搭建一个包含嵌入层的二分类模型：
    - a. 下载 [Large Movie Review Dataset](#)，它包含 50000 条 IMDB 的影评。数据分为两个目录，`train` 和 `test`，每个包含 12500 条正面评价和 12500 条负面评价。每条评论都存在独立的文本文件中。还有其他文件和文件夹（包括预处理的词袋），但这个练习中用不到。
    - b. 将测试集分给成验证集（15000）和测试集（10000）。
    - c. 使用 `tf.data`，为每个集合创建高效数据集。
    - d. 创建一个二分类模型，使用 `TextVectorization` 层来预处理每条评论。如果 `TextVectorization` 层用不了（或者你想挑战下），则创建自定义的预处理层：使用 `tf.strings` 包中的函数，比如 `lower()` 来做小写，`regex_replace()` 来替换带有空格的标点，`split()` 来分割词。用查找表输出词索引，`adapt()` 方法中要准备好。
    - e. 加入嵌入层，计算每条评论的平均嵌入，乘以词数的平方根。这个缩放过的平均嵌入可以传入剩余的模型中。
    - f. 训练模型，看看准确率能达到多少。尝试优化管道，让训练越快越好。
    - g. 使用 TFDS 加载同样的数据集：`tfds.load("imdb_reviews")`。

参考答案见附录 A。

## 十四、使用卷积神经网络实现深度计算机视觉

译者：[@SeanCheney](#)

尽管 IBM 的深蓝超级计算机在 1996 年击败了国际象棋世界冠军加里·卡斯帕罗夫，但直到最近计算机才能从图片中认出小狗，或是识别出说话时的单词。为什么这些任务对人类反而毫不费力呢？原因在于，感知过程不属于人的自我意识，而是属于专业的视觉、听觉和其它大脑感官模块。当感官信息抵达意识时，信息已经具有高级特征了：例如，当你看一张小狗的图片时，不能选择不可能，也不能回避的小狗的可爱。你解释不了你是如何识别出来的：小狗就是在图片中。因此，我们不能相信主观经验：感知并不简单，要明白其中的原理，必须探究感官模块。

卷积神经网络（CNN）起源于人们对大脑视神经的研究，自从 1980 年代，CNN 就被用于图像识别了。最近几年，得益于算力提高、训练数据大增，以及第 11 章中介绍过的训练深度网络的技巧，CNN 在一些非常复杂的视觉任务上取得了超出人类表现的进步。CNN 支撑了图片搜索、无人驾驶汽车、自动视频分类，等等。另外，CNN 也不再限于视觉，比如：语音识别和自然语言处理，但这一章只介绍视觉应用。

本章会介绍 CNN 的起源，CNN 的基本组件以及 TensorFlow 和 Keras 实现方法。然后会讨论一些优秀的 CNN 架构，和一些其它的视觉任务，比如目标识别（分类图片中的多个物体，然后画框）、语义分割（按照目标，对每个像素做分类）。

### 视神经结构

David H. Hubel 和 Torsten Wiesel 在 1958 年和 1959 年在猫的身上做了一系列研究，对视神经中枢做了研究（并在 1981 年荣获了诺贝尔生理学或医学奖）。特别的，他们指出视神经中的许多神经元都有一个局部感受野（local receptive field），也就是说，这些神经元只对有限视觉区域的刺激作反应（见图 14-1，五个神经元的局部感受野由虚线表示）。不同神经元的感受野或许是重合的，拼在一起就形成了完整的视觉区域。

另外，David H. Hubel 和 Torsten Wiesel 指出，有些神经元只对横线有反应，而其它神经元可能对其它方向的线有反应（两个神经元可能有同样的感受野，但是只能对不同防线的线有反应）。他们还注意到，一些神经元有更大的感受野，可以处理更复杂的图案，复杂图案是由低级图案构成的。这些发现启发人们，高级神经元是基于周边附近低级神经元的输出（图 14-1 中，每个神经元只是连着前一层的几个神经元）。这样的架构可以监测出视觉区域中各种复杂的图案。

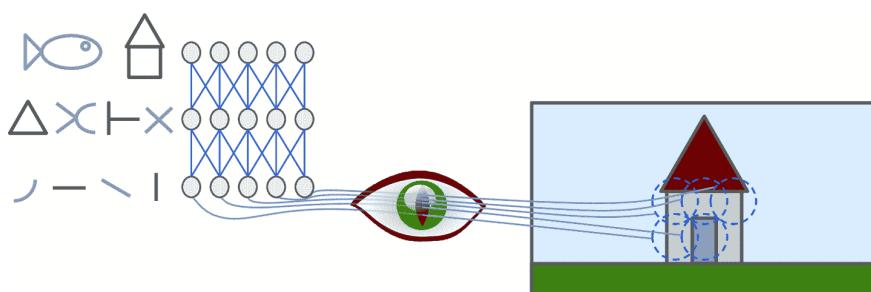


图 14-1 视神经中生物神经元可以对感受野中的图案作反应；当视神经信号上升时，神经元可以反应出更大感受野中的更为复杂的图案

对视神经的研究在 1980 年启发了[神经认知学](#)，后者逐渐演变成了今天的卷积神经网络。Yann LeCun 等人再 1998 年发表了一篇里程碑式的论文，提出了著名的 LeNet-5 架构，被银行广泛用来识别手写支票的数字。这个架构中的一些组件，我们已经学过了，比如全连接层、sigmod 激活函数，但 CNN 还引入了两个新组件：卷积层和池化层。

笔记：为什么不使用全连接层的深度神经网络来做图像识别呢？这是因为，尽管这种方案在小图片（比如 MNIST）任务上表现不错，但由于参数过多，在大图片任务上表现不佳。举个例子，一张 $100 \times 100$  像素的图片总共有 10000 个像素点，如果第一层有 1000 个神经元（如此少的神经元，已经限制信息的传输量了），那么就会有 1000 万个连接。这仅仅是第一层的情况。CNN 是通过部分连接层和权重共享解决这个问题的。

## 卷积层

卷积层是 CNN 最重要的组成部分：第一个卷积层的神经元，不是与图片中的每个像素点都连接，而是只连着局部感受野的像素（见图 14-2）。同理，第二个卷积层中的每个神经元也只是连着第一层中一个小方形内的神经元。这种架构可以让第一个隐藏层聚焦于小的低级特征，然后在下一层组成大而高级的特征，等等。这种层级式的结构在真实世界的图片很常见，这是 CNN 能在图片识别上取得如此成功的原因之一。

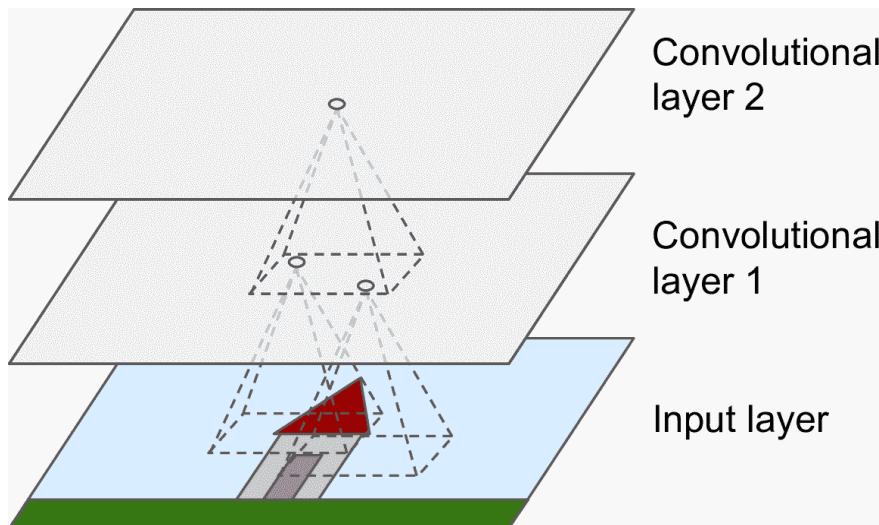


图 14-2 有方形局部感受野的 CNN 层

笔记：我们目前所学过的所有多层神经网络的层，都是由一长串神经元组成的，所以在将图片输入给神经网络之前，必须将图片打平成 1D 的。在 CNN 中，每个层都是 2D 的，更容易将神经元和输入做匹配。

位于给定层第  $i$  行、第  $j$  列的神经元，和前一层的第  $i$  行到第  $i + fh - 1$  行、第  $j$  列到第  $j + fw - 1$  列的输出相连， $fh$  和  $fw$  是感受野的高度和宽度（见图 14-3）。为了让卷积层能和前一层有相同的高度和宽度，通常给输入加上 0，见图，这被称为零填充（zero padding）。

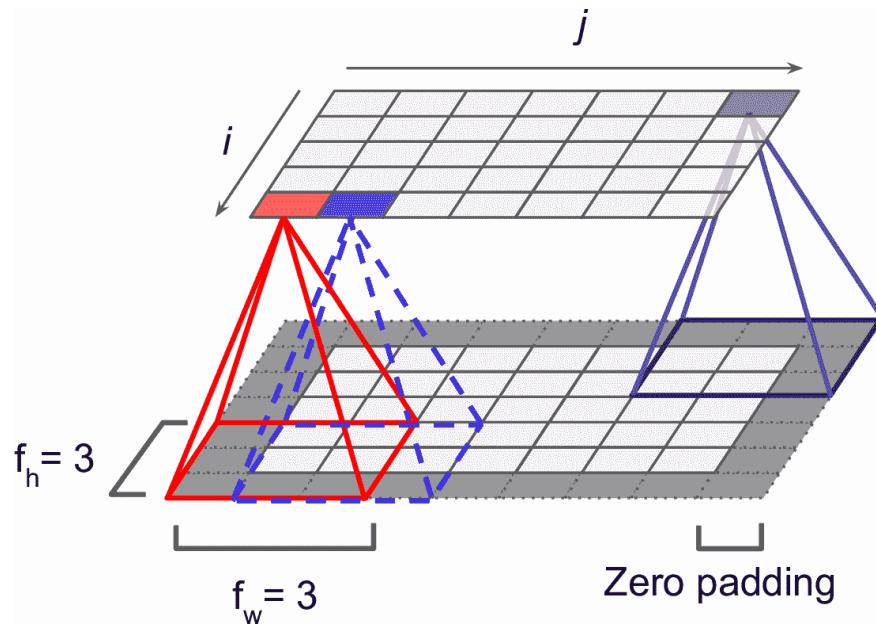


图 14-3 卷积层和零填充的连接

也可以通过间隔感受野，将大输入层和小卷积层连接起来，见图 14-4。这么做可以极大降低模型的计算复杂度。一个感受野到下一个感受野的便宜距离称为步长。在图中， $5 \times 7$  的输入层（加上零填充），连接着一个  $3 \times 4$  的层，使用  $3 \times 3$  的感受野，步长是 2（这个例子中，宽和高的步长都是 2，但也可以不同）。位于上层第  $i$  行、第  $j$  列的神经元，连接着前一层的第  $i \times s_h$  到  $i \times s_h + f_h - 1$  行、第  $j \times s_w$  到  $j \times s_w + f_w - 1$  列的神经元的输出， $s[h]$  和  $s[w]$  分别是垂直和平行步长。

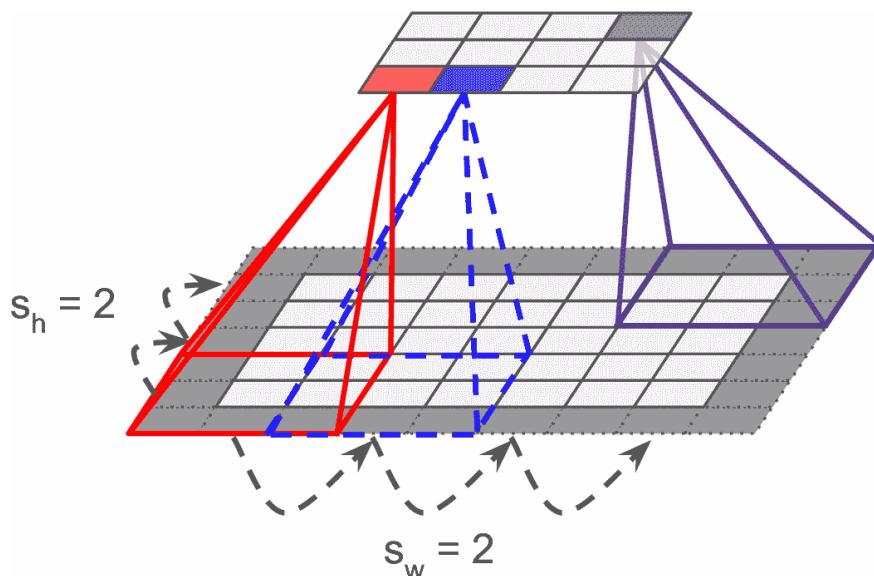


图 14-2 使用大小为 2 的步长降维

## 过滤器

神经元的权重可以表示为感受野大小的图片。例如，图 14-5 展示了两套可能的权重（称为权重，或卷积核）。第一个是黑色的方形，中央有垂直白线（ $7 \times 7$  的矩阵，除了中间的竖线都是 1，其它地方是 0）；使用这个矩阵，神经元只能注意到

中间的垂直线（因为其它地方都乘以 0 了）。第二个过滤器也是黑色的方形，但是中间是水平的白线。使用这个权重的神经元只会注意中间的白色水平线。

如果卷积层的所有神经元使用同样的垂直过滤器（和同样的偏置项），给神经网络输入图 14-5 中最底下的图片，卷积层输出的是左上的图片。可以看到，图中垂直的白线得到了加强，其余部分变模糊了。相似的，右上的图是所有神经元都是用水平线过滤器的结果，水平的白线加强了，其余模糊了。因此，一层的全部神经元都用一个过滤器，就能输出一个特征映射（feature map），特征映射可以高亮图片中最为激活过滤器的区域。当然，不用手动定义过滤器：卷积层在训练中可以自动学习对任务最有用的过滤器，上面的层则可以将简单图案组合为复杂图案。

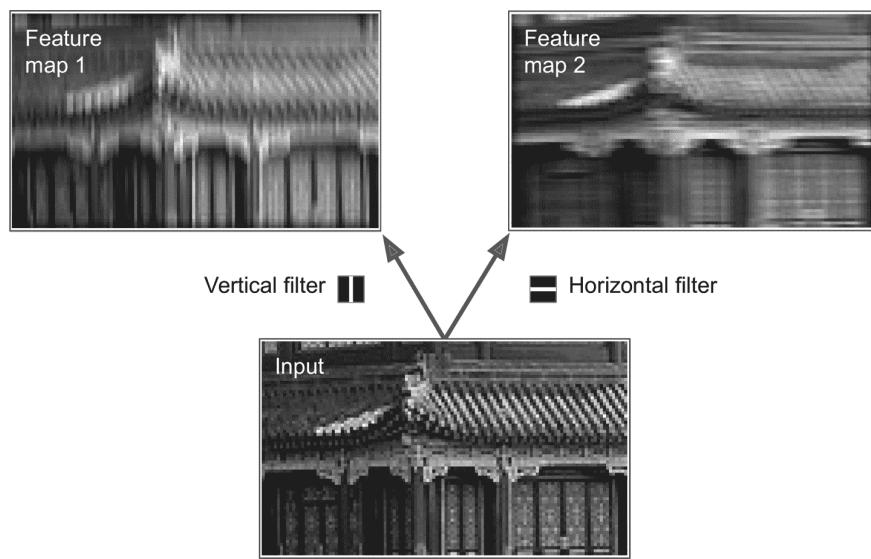


图 14-5 应用两个不同的过滤器，得到两张不同的特征映射

## 堆叠多个特征映射

简单起见，前面都是将每个卷积层的输出用 2D 层来表示的，但真实的卷积层可能有多个过滤器（过滤器数量由你确定），每个过滤器会输出一个特征映射，所以表示成 3D 更准确（见图 14-6）。每个特征映射的每个像素有一个神经元，同一特征映射中的所有神经元有同样的参数（即，同样的权重和偏置项）。不同特征映射的神经元的参数不同。神经元的感受野和之前描述的相同，但扩展到了前面所有的特征映射。总而言之，一个卷积层同时对输入数据应用多个可训练过滤器，使其可以检测出输入的任何地方的多个特征。

**笔记：**同一特征映射中的所有神经元共享一套参数，极大地减少了模型的参数数量。当 CNN 认识了一个位置的图案，就可以在任何其它位置识别出来。  
相反的，当常规 DNN 学会一个图案，只能在特定位置识别出来。

输入图像也是有多个子层构成的：每个颜色通道，一个子层。通常是三个：红，绿，蓝（RGB）。灰度图只有一个通道，但有些图可能有多个通道——例如，卫星图片可以捕捉到更多的光谱频率（比如红外线）。

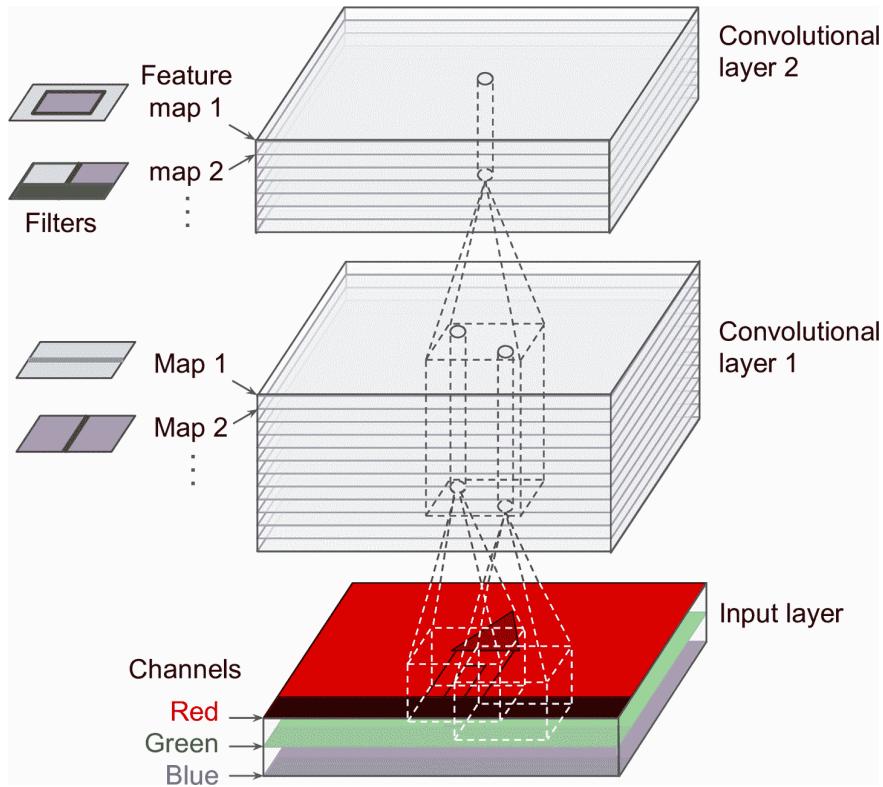


图 14-6 有多个特征映射的卷积层，有三个颜色通道的图像

特别的，位于卷积层  $l$  的特征映射  $k$  的第  $i$  行、第  $j$  列的神经元，它连接的是前一层  $l-1$  的  $i \times s_h$  到  $i \times s_h + f_h - 1$  行、 $j \times s_w$  到  $j \times s_w + f_w - 1$  列的所有特征映射。不同特征映射中，位于相同  $i$  行、 $j$  列的神经元，连接着前一层相同的神经元。

等式 14-1 用一个大等式总结了前面的知识：如何计算卷积层中给定神经元的输出。因为索引过多，这个等式不太好看，它所做的其实就是计算所有输入的加权和，再加上偏置项。

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k}$$

with  $\begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$

公式 14-1 计算卷积层中给定神经元的输出

在这个等式中：

- $z[i, j, k]$  是卷积层  $l$  中第  $i$  行、第  $j$  列、特征映射  $k$  的输出。
- $s[h]$  和  $s[w]$  是垂直和水平步长， $f[h]$  和  $f[w]$  是感受野的高和宽， $f[n']$  是前一层  $l-1$  的特征映射数。
- $x[i', j', k']$  是卷积层  $l-1$  中第  $i'$  行、第  $j'$  列、特征映射  $k'$  的输出（如果前一层是输入层，则为通道  $k'$ ）。

- $b[k]$  是特征映射  $k$  的偏置项。可以将其想象成一个旋钮，可以调节特征映射  $k$  的明亮度。
- $w[u, v, k', k]$  是层  $l$  的特征映射  $k$  的任意神经元，和位于行  $u$ 、列  $v$ （相对于神经元的感受野）、特征映射  $k'$  的输入，两者之间的连接权重。

## TensorFlow 实现

在 TensorFlow 中，每张输入图片通常都是用形状为 [高度, 宽度, 通道] 的 3D 张量表示的。一个小批次则为 4D 张量，形状是 [批次大小, 高度, 宽度, 通道]。卷积层的权重是 4D 张量，形状是  $[f[h], f[w], f[n'], f[n]]$ 。卷积层的偏置项是 1D 张量，形状是  $[f[n]]$ 。

看一个简单的例子。下面的代码使用 Scikit-Learn 的 `load_sample_image()` 加载了两张图片，一张是中国的寺庙，另一张是花，创建了两个过滤器，应用到了两张图片上，最后展示了一张特征映射：

```
from sklearn.datasets import load_sample_image

# 加载样本图片
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

# 创建两个过滤器
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1 # 垂直线
filters[3, :, :, 1] = 1 # 水平线

outputs = tf.nn.conv2d(images, filters, strides=1, padding="same")

plt.imshow(outputs[0, :, :, 1], cmap="gray") # 画出第 1 张图的第 2 个特征映射
plt.show()
```

逐行看下代码：

- 每个颜色通道的像素强度是用 0 到 255 来表示的，所以直接除以 255，将其缩放到区间 0 到 1 内。
- 然后创建了两个  $7 \times 7$  的过滤器（一个有垂直正中白线，另一个有水平正中白线）。
- 使用 `tf.nn.conv2d()` 函数，将过滤器应用到两张图片上。这个例子中使用了零填充（`padding="same"`），步长是 1。
- 最后，画出一个特征映射（相似与图 14-5 中的右上图）。

`tf.nn.conv2d()` 函数这一行，再多说说：

- `images` 是一个输入的小批次（4D 张量）。
- `filters` 是过滤器的集合（也是 4D 张量）。
- `strides` 等于 1，也可以是包含 4 个元素的 1D 数组，中间的两个元素是垂直和水平步长（`s[h]` 和 `s[w]`），第一个和最后一个元素现在必须是 1。以后可以用来指定批次步长（跳过实例）和通道步长（跳过前一层的特征映射或通道）。
- `padding` 必须是 "same" 或 "valid"：

- 如果设为 "same"，卷积层会使用零填充。输出的大小是输入神经元的数量除以步长，再取整。例如：如果输入大小是 13，步长是 5（见图 14-7），则输出大小是 3 ( $13 / 5 = 2.6$ ，再向上圆整为 3)，零填充尽量在输入上平均添加。当 `strides=1` 时，层的输出会和输入有相同的空间维度（宽和高），这就是 same 的来历。
- 如果设为 "valid"，卷积层就不使用零填充，取决于步长，可能会忽略图片的输入图片的底部或右侧的行和列，见图 14-7（简单举例，只是显示了水平维度）。这意味着每个神经元的感受野位于严格确定的图片中的位置（不会越界），这就是 valid 的来历。

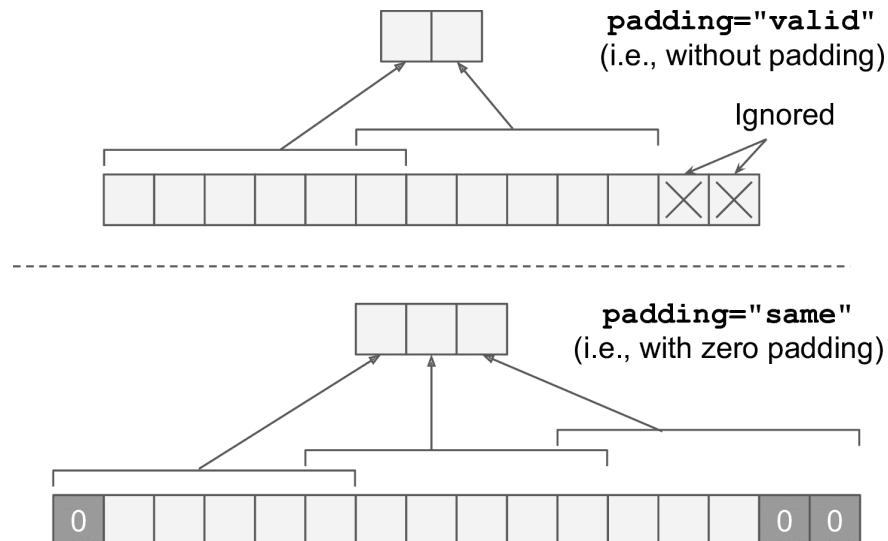


图 14-7 Padding="same" 或 "valid" (输入宽度 13，过滤器宽度 6，步长 5)

这个例子中，我们手动定义了过滤器，但在真正的 CNN 中，一般将过滤器定义为可以训练的变量，好让神经网络学习哪个过滤器的效果最好。使用 `keras.layers.Conv2D` 层：

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                           padding="same", activation="relu")
```

这段代码创建了一个有 32 个过滤器的 `Conv2D` 层，每个过滤器的形状是  $3 \times 3$ ，步长为 1（水平垂直都是 1），和 "same" 填充，输出使用 `ReLU` 激活函数。可以看到，卷积层的超参数不多：选择过滤器的数量，过滤器的高和宽，步长和填充类型。和以前一样，可以使用交叉验证来找到合适的超参数值，但很耗时间。后面会讨论常见的 CNN 架构，可以告诉你如何挑选超参数的值。

## 内存需求

CNN 的另一个问题是卷积层需要很高的内存。特别是在训练时，因为反向传播需要所有前向传播的中间值。

比如，一个有  $5 \times 5$  个过滤器的卷积层，输出 200 个特征映射，大小为  $150 \times 100$ ，步长为 1，零填充。如果如数是  $150 \times 100$  的 RGB 图片（三通道），则参数总数是  $(5 \times 5 \times 3 + 1) \times 200 = 15200$ ，加 1 是考虑偏置项。相对于全连接层，参数少很多了。但是 200 个特征映射，每个都包含  $150 \times 100$  个神经元，每个神经元都需要计算  $5 \times 5 \times 3 = 75$  个输入的权重和：总共是 2.25 亿个浮点数乘法运算。虽然比全连接层少点，但也很耗费算力。另外，如果特征映射用的

是 32 位浮点数，则卷积层输出要占用  $200 \times 150 \times 100 \times 32 = 96$  百万比特 (12MB) 的内存。这仅仅是一个实例，如果训练批次有 100 个实例，则要使用 1.2 GB 的内存。

在做推断时（即，对新实例做预测），下一层计算完，前一层占用的内存就可以释放掉内存，所以只需要两个连续层的内存就够了。但在训练时，前向传播期间的所有结果都要保存下来以为反向传播使用，所以消耗的内存是所有层的内存占用总和。

**提示：**如果因为内存不够发生训练终端，可以降低批次大小。另外，可以使步长降低纬度，或去掉几层。或者，你可以使用 16 位浮点数，而不是 32 位浮点数。或者，可以将 CNN 分布在多台设备上。

接下来，看看 CNN 的第二个组成部分：池化层。

## 池化层

明白卷积层的原理了，池化层就容易多了。池化层的目的是对输入图片做降采样（即，收缩），以降低计算负载、内存消耗和参数的数量（降低过拟合）。

和卷积层一样，池化层中的每个神经元也是之和前一层的感受野里的有限个神经元相连。和前面一样，必须定义感受野的大小、步长和填充类型。但是，池化神经元没有权重，它所要做的是使用聚合函数，比如最大或平均，对输入做聚合。图 14-8 展示了最为常用的最大池化层。在这个例子中，使用了一个  $2 \times 2$  的池化核，步长为 2，没有填充。只有感受野中的最大值才能进入下一层，其它的就丢弃了。例如，在图 14-8 左下角的感受野中，输入值是 1、5、3、2，所以只有最大值 5 进入了下一层。因为步长是 2，输出图的高度和宽度是输入图的一半（因为没有用填充，向下圆整）。

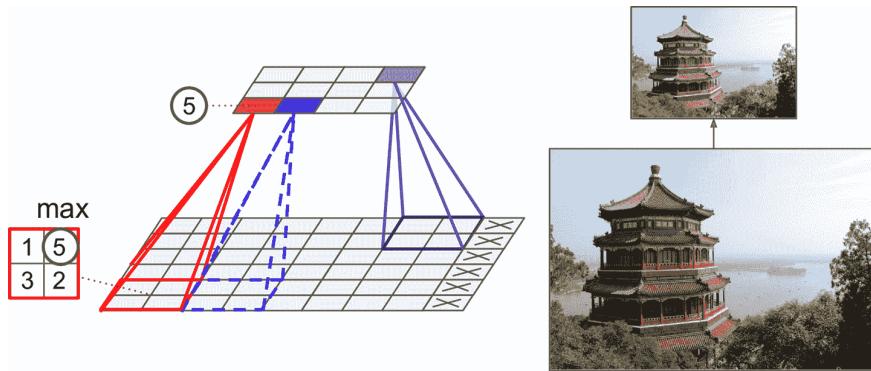


图 14-8 最大池化层 ( $2 \times 2$  的池化核，步长为 2，没有填充)

**笔记：**池化层通常独立工作在每个通道上，所以输出深度和输入深度相同。

除了可以减少计算、内存消耗、参数数量，最大池化层还可以带来对小偏移的不变性，见图 14-9。假设亮像素比暗像素的值小，用  $2 \times 2$  核、步长为 2 的最大池化层处理三张图 (A、B、C)。图 B 和 C 的图案与 A 相同，只是分别向右移动了一个和两个像素。可以看到，A、B 经过池化层处理后的结果相同，这就是所谓的平移不变性。对于图片 C，输出有所不同：向右偏移了一个像素（但仍然有 50% 没变）。在 CNN 中每隔几层就插入一个最大池化层，可以带来更大程度的平移不变性。另外，最大池化层还能带来一定程度的旋转不变性和缩放不变性。当预测不需要考虑平移、旋转和缩放时，比如分类任务，不变性可以有一定益处。

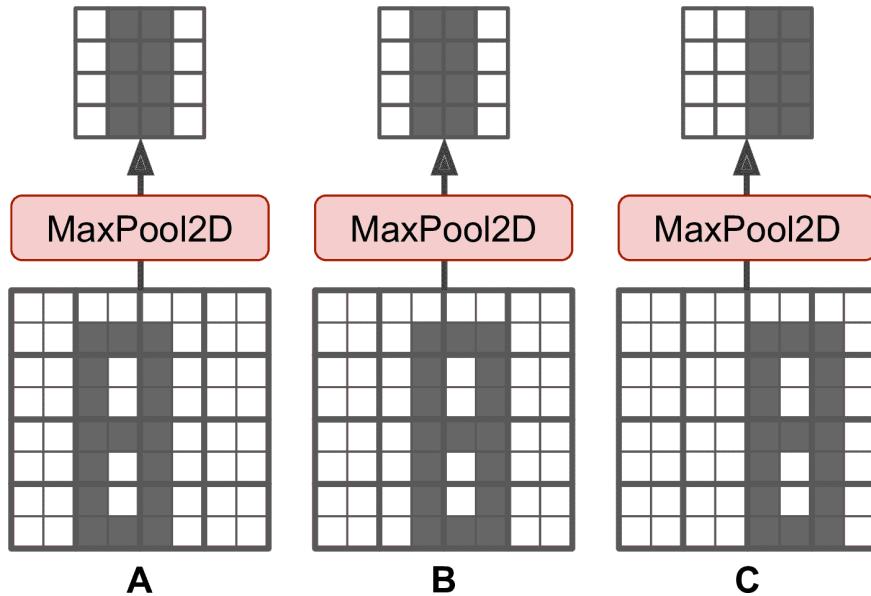


图 14-9 小平移不变性

但是，最大池化层也有缺点。首先，池化层破坏了信息：即使感受野的核是 $2 \times 2$ ，步长是 2，输出在两个方向上都损失了一半，总共损失了 75% 的信息。对于某些任务，不变性不可取。比如语义分割（将像素按照对象分类）：如果输入图片向右平移了一个像素，输出也应该向右平移一个降速。此时强调的就是等价：输入发生小变化，则输出也要有对应的小变化。

## TensorFlow 实现

用 TensorFlow 实现最大池化层很简单。下面的代码实现了最大池化层，核是 $2 \times 2$ 。步长默认等于核的大小，所以步长是 2（水平和垂直步长都是 2）。默认使用“valid”填充：

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

要创建平均池化层，则使用 `AvgPool2D`。平均池化层和最大池化层很相似，但计算的是感受野的平均值。平均池化层在过去很流行，但最近人们使用最大池化层更多，因为最大池化层的效果更好。初看很奇怪，因为计算平均值比最大值损失的信息要少。但是从反面看，最大值保留了最强特征，去除了无意义的特征，可以让下一层获得更清楚的信息。另外，最大池化层提供了更强的平移不变性，所需计算也更少。

池化层还可以沿着深度方向做计算。这可以让 CNN 学习到不同特征的不变性。比如。CNN 可以学习多个过滤器，每个过滤器检测一个相同的图案的不同旋转（比如手写字，见图 14-10），深度池化层可以使输出相同。CNN 还能学习其它的不变性：厚度、明亮度、扭曲、颜色，等等。

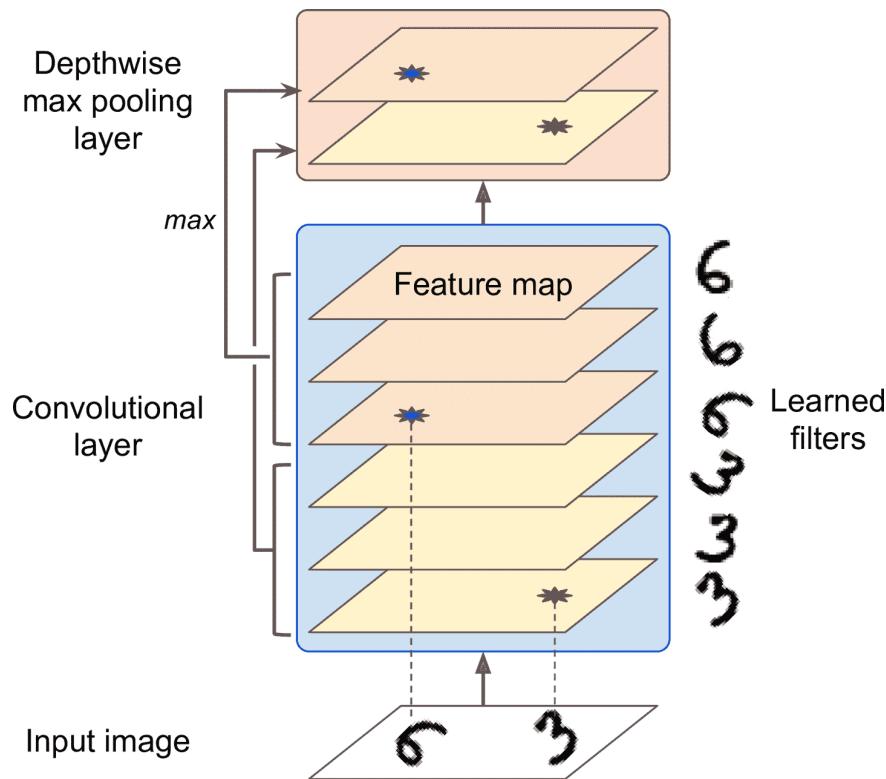


图 14-10 深度最大池化层可以让 CNN 学习到多种不变性

Keras 没有深度方向最大池化层，但 TensorFlow 的低级 API 有：使用 `tf.nn.max_pool()`，指定核的大小、步长（4 元素的元组）：元组的前三个值应该是 1，表明沿批次、高度、宽度的步长是 1；最后一个值，是深度方向的步长——比如 3（深度步长必须可以整除输入深度；如果前一个层有 20 个特征映射，步长 3 就不成）：

```
output = tf.nn.max_pool(images,
                        ksize=(1, 1, 1, 3),
                        strides=(1, 1, 1, 3),
                        padding="valid")
```

如果想将这个层添加到 Keras 模型中，可以将其包装进 `Lambda` 层（或创建一个自定义 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"))
```

最后一类常见的池化层是全局平均池化层。它的原理非常不同：它计算整个特征映射的平均值（就像是平均池化层的核的大小和输入的空间维度一样）。这意味着，全局平均池化层对于每个实例的每个特征映射，只输出一个值。虽然这么做对信息的破坏性很大，却可以用来做输出层，后面会看到例子。创建全局平均池化层的方法如下：

```
global_avg_pool = keras.layers.GlobalAvgPool2D()
```

它等同于下面的 `Lambda` 层：

```
global_avg_pool = keras.layers.Lambda(lambda X: tf.reduce_mean(X, axis=[1, 2]))
```

介绍完 CNN 的组件之后，来看看如何将它们组合起来。

## CNN 架构

CNN 的典型架构是将几个卷积层叠起来（每个卷积层后面跟着一个 ReLU 层），然后再叠一个池化层，然后再叠几个卷积层（+ReLU），接着再一个池化层，以此类推。图片在流经神经网络的过程中，变得越来越小，但得益于卷积层，却变得越来越深（特征映射变多了），见图 14-11。在 CNN 的顶部，还有一个常规的前馈神经网络，由几个全连接层（+ReLU）组成，最终层输出预测（比如，一个输出类型概率的 softmax 层）。

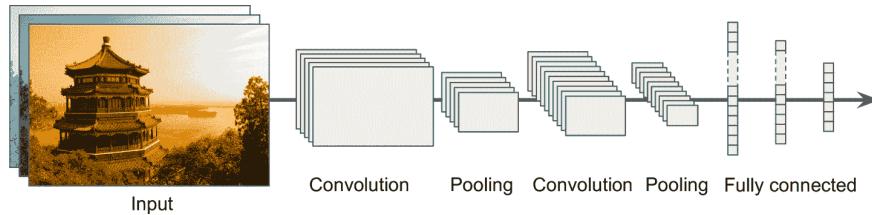


图 14-11 典型的 CNN 架构

提示：常犯的错误之一，是使用过大的卷积核。例如，要使用一个卷积层的核是  $5 \times 5$ ，再加上两个核为  $3 \times 3$  的层：这样参数不多，计算也不多，通常效果也更好。第一个卷积层是例外：可以有更大的卷积核（例如  $5 \times 5$ ），步长为 2 或更大：这样可以降低图片的空间维度，也没有损失很多信息。

下面的例子用一个简单的 CNN 来处理 Fashion MNIST 数据集（第 10 章介绍过）：

```
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")
])
```

逐行看下代码：

- 第一层使用了 64 个相当大的过滤器 ( $7 \times 7$ )，但没有用步长，因为输入图片不大。还设置了 `input_shape=[28, 28, 1]`，因为图片是  $28 \times 28$  像素的，且是单通道（即，灰度）。
- 接着，使用了一个最大池化层，核大小为 2。
- 接着，重复做两次同样的结构：两个卷积层，跟着一个最大池化层。对于大图片，这个结构可以重复更多次（重复次数是超参数）。

- 要注意，随着 CNN 向着输出层的靠近，过滤器的数量一直在提高（一开始是 64，然后是 128，然后是 256）：这是因为低级特征的数量通常不多（比如，小圆圈或水平线），但将其组合成为高级特征的方式很多。通常的做法是在每个池化层之后，将过滤器的数量翻倍：因为池化层对空间维度除以了 2，因此可以将特征映射的数量翻倍，且不用担心参数数量、内存消耗、算力的增长。
- 然后是全连接网络，由两个隐藏紧密层和一个紧密输出层组成。要注意，必须要打平输入，因为紧密层的每个实例必须是 1D 数组。还加入了两个丢弃层，丢弃率为 50%，以降低过拟合。

这个 CNN 可以在测试集上达到 92% 的准确率。虽然不是顶尖水平，但也相当好了，效果比第 10 章用的方法好得多。

过去几年，这个基础架构的变体发展迅猛，取得了惊人的进步。衡量进步的一个指标是 ILSVRC [ImageNet challenge](#) 的误差率。在六年期间，这项赛事的前五误差率从 26% 降低到了 2.3%。前五误差率的意思是，预测结果的前 5 个最高概率的图片不包含正确结果的比例。测试图片相当大（256 个像素），有 1000 个类，一些图的差别很细微（比如区分 120 种狗的品种）。学习 ImageNet 冠军代码是学习 CNN 的好方法。

我们先看看经典的 LeNet-5 架构（1998），然后看看三个 ILSVRC 竞赛的冠军：AlexNet（2012）、GoogLeNet（2014）、ResNet（2015）。

## LeNet-5

LeNet-5 也许是广为人知的 CNN 架构。前面提到过，它是由 Yann LeCun 在 1998 年创造出来的，被广泛用于手写字识别（MNIST）。它的结构如下：

层	类型	映射数量	大小	核大小	步长	激活函数
输出层	全连接层	-	10	-	-	RBF
F6	全连接层	-	84	-	-	tanh
C5	卷积层	120	1 × 1	5 × 5	1	tanh
S4	平均池化层	16	5 × 5	2 × 2	2	tanh
C3	卷积层	16	10 × 10	5 × 5	1	tanh
S2	平均池化层	6	14 × 14	2 × 2	2	tanh
C1	卷积层	6	28 × 28	5 × 5	1	tanh
输入层	输入层	1	32 × 32	-	-	-

表 14-1 LeNet-5 架构

有一些点需要注意：

- MNIST 图片是  $28 \times 28$  像素的，但在输入给神经网络之前，做了零填充，成为  $32 \times 32$  像素，并做了归一化。后面的层不用使用任何填充，这就是为什么当图片在网络中传播时，图片大小持续缩小。
- 平均池化层比一般的稍微复杂点：每个神经元计算输入的平均值，然后将记过乘以一个可学习的系数（每个映射一个系数），在加上一个可学习的偏置项（也是每个映射一个），最后使用激活函数。
- C3 层映射中的大部分神经元，只与 S2 层映射三个或四个神经元全连接（而不是 6 个）。

- 输出层有点特殊：不是计算输入和权重向量的矩阵积，而是每个神经元输出输入向量和权重向量的欧氏距离的平方。每个输出衡量图片属于每个数字类的概率程度。这里适用交叉熵损失函数，因为对错误预测惩罚更多，可以产生更大的梯度，收敛更快。

Yann LeCun 的[网站](#)展示了 LeNet-5 做数字分类的例子。

## AlexNet

[AlexNet CNN 架构](#)以极大优势，赢得了 2012 ImageNet ILSVRC 冠军：它的 Top-5 误差率达到了 17%，第二名只有 26%！它是由 Alex Krizhevsky、Ilya Sutskever 和 Geoffrey Hinton 发明的。AlexNet 和 LeNet-5 很相似，只是更大更深，是首个将卷积层堆叠起来的网络，而不是在每个卷积层上再加一个池化层。表 14-2 展示了其架构：

层	类型	映射数量	大小	核大小	步长	填充	激活函数
输出层	全连接层	-	1000	-	-	-	Softmax
F9	全连接层	-	4096	-	-	-	ReLU
F8	全连接层	-	4096	-	-	-	ReLU
C7	卷积层	256	13 × 13	3 × 3	1	same	ReLU
C6	卷积层	384	13 × 13	3 × 3	1	same	ReLU
C5	卷积层	384	13 × 13	3 × 3	1	same	ReLU
S4	最大池化层	256	13 × 13	3 × 3	2	valid	-
C3	卷积层	256	27 × 27	5 × 5	1	same	ReLU
S2	最大池化层	96	27 × 27	3 × 3	2	valid	-
C1	卷积层	96	55 × 55	11 × 11	4	valid	ReLU
输入层	输入层	3 (RGB)	227 × 227	-	-	-	-

表 14-2 AlexNet 架构

为了降低过拟合，作者使用了两种正则方法。首先，F8 和 F9 层使用了丢弃，丢弃率为 50%。其次，他们通过随机距离偏移训练图片、水平翻转、改变亮度，做了数据增强。

### 数据增强

数据增强是通过生成许多训练实例的真实变种，来人为增大训练集。因为可以降低过拟合，成为了一种正则化方法。生成出来的实例越真实越好：最理想的情况，人们无法区分增强图片是原生的还是增强过的。简单的添加白噪声没有用，增强修改要是可以学习的（白噪声不可学习）。

例如，可以轻微偏移、旋转、缩放原生图，再添加到训练集中（见图 14-12）。这么做可以使模型对位置、方向和物体在图中的大小，有更高的容忍度。如果想让模型对不同光度有容忍度，可以生成对比度不同的照片。通常，还可以水平翻转图片（文字不成、不对称物体也不成）。通过这些变换，可以极大的增大训练集。

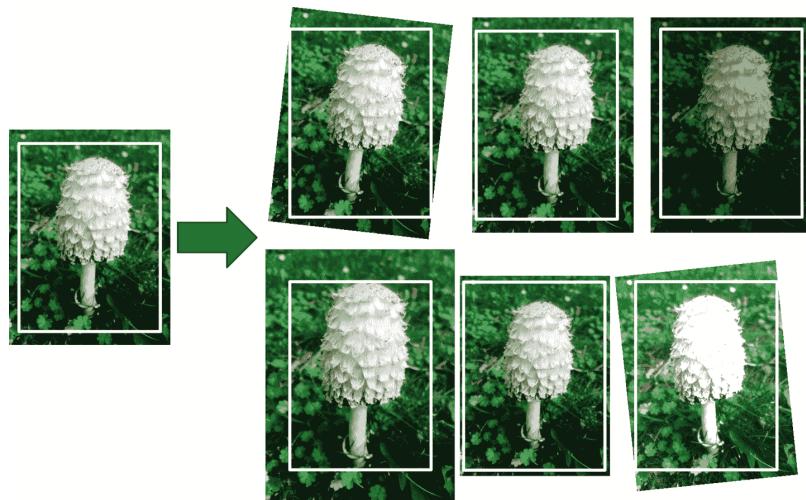


图 14-12 从原生图生成新的训练实例

AlexNet 还在 C1 和 C3 层的 ReLU 之后，使用了强大的归一化方法，称为局部响应归一化 (LRN)：激活最强的神经元抑制了相同位置的相邻特征映射的神经元（这样的竞争性激活也在生物神经元上观察到了）。这么做可以让不同的特征映射专业化，特征范围更广，提升泛化能力。等式 14-2 展示了如何使用 LRN。

$$b_i = a_i \left( k + \alpha \sum_{j=j_{\text{low}}}^{j_{\text{high}}} a_j^2 \right)^{-\beta} \quad \text{with} \quad \begin{cases} j_{\text{high}} = \min \left( i + \frac{r}{2}, f_n - 1 \right) \\ j_{\text{low}} = \max \left( 0, i - \frac{r}{2} \right) \end{cases}$$

公式 14-2 局部响应归一化 (LRN)

这个等式中：

- $b[I]$  是特征映射  $i$  的行  $u$  列  $v$  的神经元的归一化输出（注意等式中没有出现行  $u$  列  $v$ ）。
- $a[I]$  是 ReLU 之后，归一化之前的激活函数。
- $k$ 、 $\alpha$ 、 $\beta$  和  $r$  是超参。 $k$  是偏置项， $r$  是深度半径。
- $f[n]$  是特征映射的数量。

例如，如果  $r=2$ ，且神经元有强激活，能抑制其他相邻上下特征映射的神经元的激活。

在 AlexNet 中，超参数是这么设置的： $r = 2$ ， $\alpha = 0.00002$ ， $\beta = 0.75$ ， $k = 1$ 。可以通过 `tf.nn.local_response_normalization()` 函数实现，要想用在 Keras 模型中，可以包装进 Lambda 层。

AlexNet 的一个变体是 *ZF Net*，是由 Matthew Zeiler 和 Rob Fergus 发明的，赢得了 2013 年的 ILSVRC。它本质上是对 AlexNet 做了一些超参数的调节（特征映射数、核大小，步长，等等）。

## GoogLeNet

[GoogLeNet 架构](#)是 Google Research 的 Christian Szegedy 及其同事发明的，赢得了 ILSVRC 2014 冠军，top-5 误差率降低到了 7% 以内。能取得这么大的进步，很大的原因是它的网络比之前的 CNN 更深（见图 14-14）。这归功于被称为创始模块（inception module）的子网络，它可以让 GoogLeNet 可以用更高的效率使用参数：实际上，GoogLeNet 的参数量比 AlexNet 小 10 倍（大约是 600 万，而不是 AlexNet 的 6000 万）。

图 14-13 展示了一个创始模块的架构。 $3 \times 3 + 1(S)$  的意思是层使用的核是  $3 \times 3$ ，步长是 1，“same”填充。先复制输入信号，然后输入给 4 个不同的层。所有卷积层使用 ReLU 激活函数。注意，第二套卷积层使用了不同的核大小（ $1 \times 1$ 、 $3 \times 3$ 、 $5 \times 5$ ），可以让其捕捉不同程度的图案。还有，每个单一层的步长都是 1，都是零填充（最大池化层也同样），因此它们的输出和输入有同样的高度和宽度。这可以让所有输出在最终深度连接层，可以沿着深度方向连起来（即，将四套卷积层的所有特征映射堆叠起来）。这个连接层可以使用用 `tf.concat()` 实现，其 `axis=3`（深度方向的轴）。

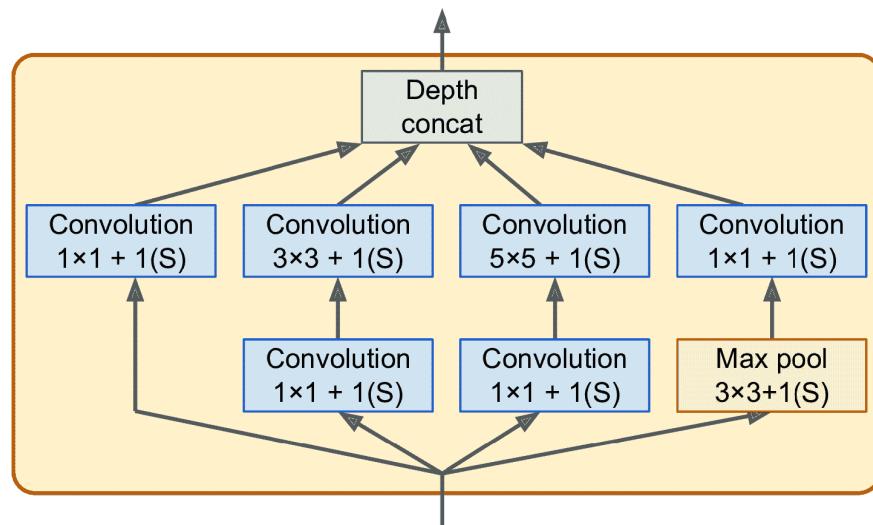


图 14-13 创始模块

为什么创始模块有核为  $1 \times 1$  的卷积层呢？这些层捕捉不到任何图案，因为只能观察一个像素？事实上，这些层有三个目的：

- 尽管不能捕捉空间图案，但可以捕捉沿深度方向的图案。
- 这些曾输出的特征映射比输入少，是作为瓶颈层来使用的，意味它们可以降低维度。这样可以减少计算和参数量、加快训练，提高泛化能力。

- 每一对卷积层 ( $[1 \times 1, 3 \times 3]$  和  $[1 \times 1, 5 \times 5]$ ) 就像一个强大的单一卷积层，可以捕捉到更复杂的图案。事实上，这对卷积层可以扫过两层神经网络。

总而言之，可以将整个创始模块当做一个卷积层，可以输出捕捉到不同程度、更多复杂图案的特征映射。

**警告：**每个卷积层的卷积核的数量是一个超参数。但是，这意味着每添加一个创始层，就多了 6 个超参数。

来看下 GoogLeNet 的架构（见图 14-14）。每个卷积层、每个池化层输出的特征映射的数量，展示在核大小的前面。因为比较深，只好摆成三列。GoogLeNet 实际是一列，一共包括九个创始模块（带有陀螺标志）。创始模块中的六个数表示模块中的每个卷积层输出的特征映射数（和图 14-13 的顺序相同）。注意所有卷积层使用 ReLU 激活函数。

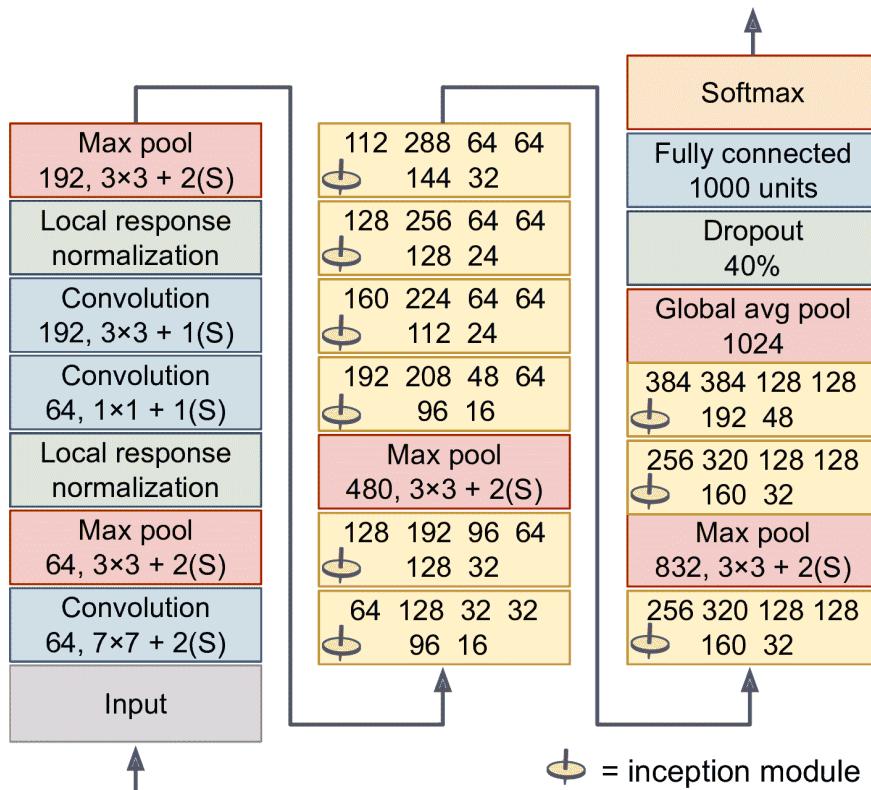


图 14-14 GoogLeNet 的架构

这个网络的结构如下：

- 前两个层将图片的高和宽除以了 4（所以面积除以了 16），以减少计算。第一层使用的核很大，可以保留大部分信息。
- 接下来，局部响应归一化层可以保证前面的层可以学到许多特征。
- 后面跟着两个卷积层，前面一层作为瓶颈层。可以将这两层作为一个卷积层。
- 然后，又是一个局部响应归一化层。
- 接着，最大池化层将图片的高度和宽度除以 2，以加快计算。
- 然后，是九个创始模块，中间插入了两个最大池化层，用来降维提速。

- 接着，全局平均池化层输出每个特征映射的平均值：可以丢弃任何留下的空间信息，可以这么做是因为此时留下的空间信息也不多了。事实上 GoogLeNet 的输入图片一般是  $224 \times 224$  像素的，经过 5 个最大池化层后，每个池化层将高和宽除以 2，特征映射降为  $7 \times 7$ 。另外，这是一个分类任务，不是定位任务，所以对象在哪无所谓。得益于该层降低了维度，就不用的网络的顶部（像 AlexNet 那样）加几个全连接层了，这么做可以极大减少参数数量，降低过拟合。
- 最后几层很明白：丢弃层用来正则，全连接层（因为有 1000 个类，所以有 1000 个单元）和 softmax 激活函数用来产生估计类的概率。

架构图经过轻微的简化：原始 GoogLeNet 架构还包括两个辅助的分类器，位于第三和第六创始模块的上方。它们都是由一个平均池化层、一个卷积层、两个全连接层和一个 softmax 激活层组成。在训练中，它们的损失（缩减 70%）被添加到总损失中。它们的目标是对抗梯度消失，对网络做正则。但是，后来的研究显示它们的作用很小。

Google 的研究者后来又提出了几个 GoogLeNet 的变体，包括 Inception-v3 和 Inception-v4，使用的创始模块略微不同，性能更好。

## VGGNet

ILSVRC 2014 年的亚军是 [VGGNet](#)，作者是来自牛津大学 Visual Geometry Group (VGG) 的 Karen Simonyan 和 Andrew Zisserman。VGGNet 的架构简单而经典，2 或 3 个卷积层和 1 个池化层，然后又是 2 或 3 个卷积层和 1 个池化层，以此类推（总共达到 16 或 19 个卷积层）。最终加上一个有两个隐藏层和输出层的紧密网络。VGGNet 只用  $3 \times 3$  的过滤器，但数量很多。

## ResNet

何凯明使用 [残差网络（或 ResNet）](#) 赢得了 ILSVRC 2015 的冠军，top-5 误差率降低到了 3.6% 以下。ResNet 的使用了极深的卷积网络，共 152 层（其它的变体有 1450 或 152 层）。反映了一个总体趋势：模型变得越来越深，参数越来越少。训练这样的深度网络的方法是使用跳连接（也被称为快捷连接）：输入信号添加到更高层的输出上。

当训练神经网络时，目标是使网络可以对目标函数  $h(x)$  建模。如果将输入  $x$  添加给网络的输出（即，添加一个跳连接），则网络就要对  $f(x) = h(x) - x$  建模，而不是  $h(x)$ 。这被称为残差学习（见图 14-15）。

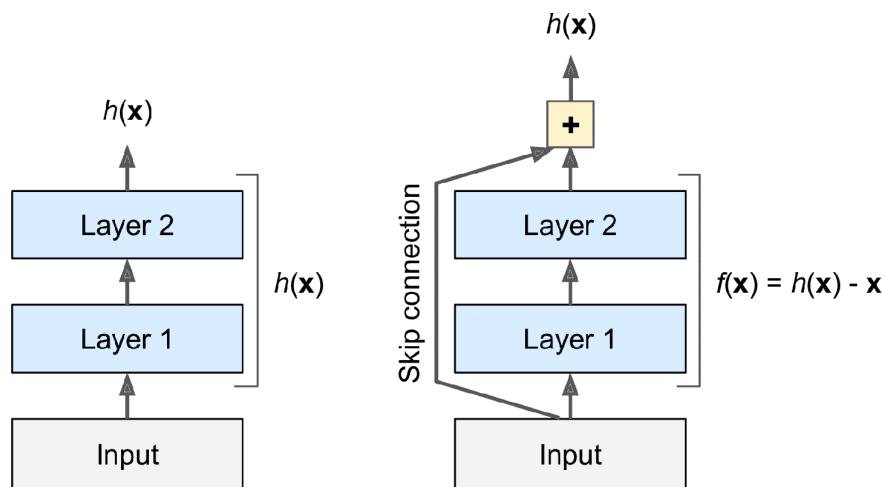


图 14-15 残差学习

初始化一个常规神经网络时，它的权重接近于零，所以输出值也接近于零。如果添加跳连接，网络就会输出一个输入的复制；换句话说，网络一开始是对恒等函数建模。如果目标函数与恒等函数很接近（通常会如此），就能极大的加快训练。

另外，如果添加多个跳连接，就算有的层还没学习，网络也能正常运作（见图 14-16）。多亏了跳连接，信号可以在整个网络中流动。深度残差网络，可以被当做残差单元（RU）的堆叠，其中每个残差单元是一个有跳连接的小神经网络。

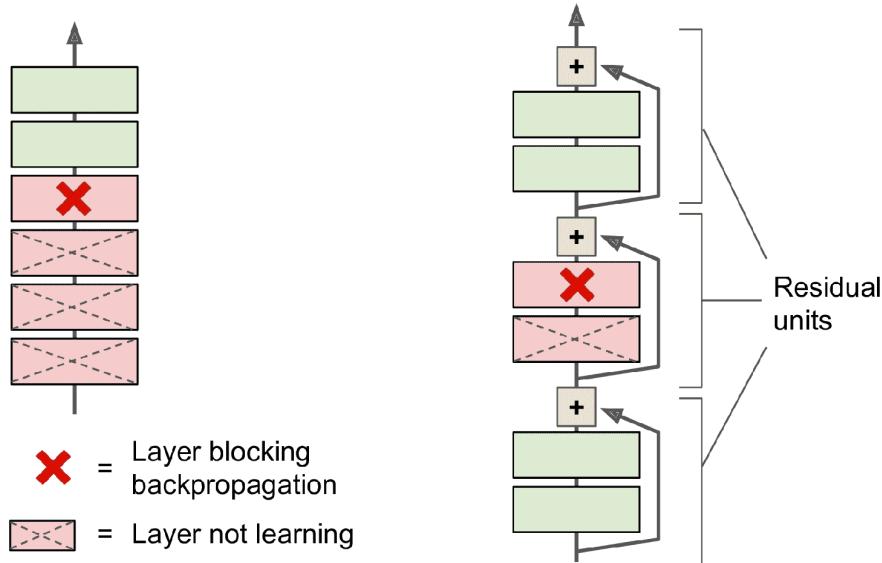


图 14-16 常规神经网络（左）和深度残差网络（右）

来看看 ResNet 的架构（见图 14-17）。特别简单。开头和结尾都很像 GoogLeNet（只是没有丢弃层），中间是非常深的残差单元的堆砌。每个残差单元由两个卷积层（没有池化层！）组成，有批归一化和 ReLU 激活，使用  $3 \times 3$  的核，保留空间维度（步长等于 1，零填充）。

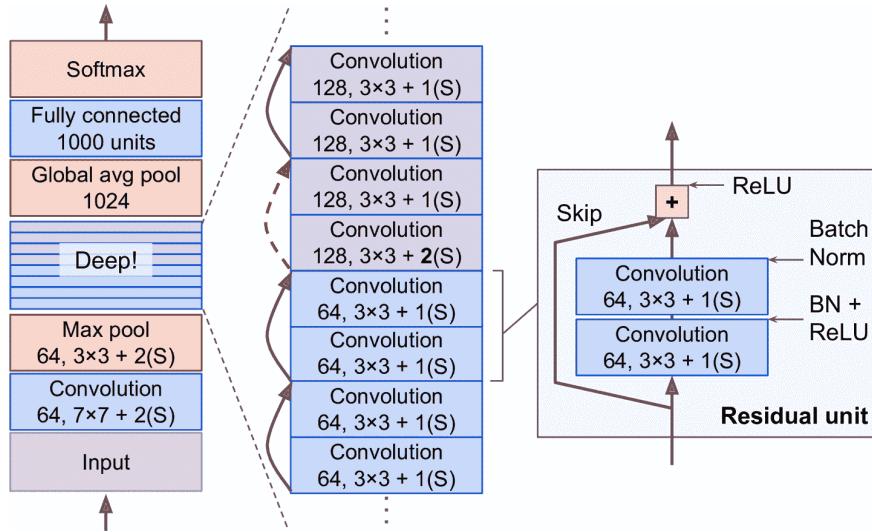


图 14-17 ResNet 架构

注意到，每经过几个残差单元，特征映射的数量就会翻倍，同时高度和宽度都减半  
( ) 卷积层的步长为 2。发生这种情况时，因为形状不同（见图 14-17 中虚线的跳连接），输入不能直接添加到残差单元的输出上。要解决这个问题，输入要经过一个  $1 \times 1$  的卷积层，步长为 2，特征映射数不变（见图 14-18）。

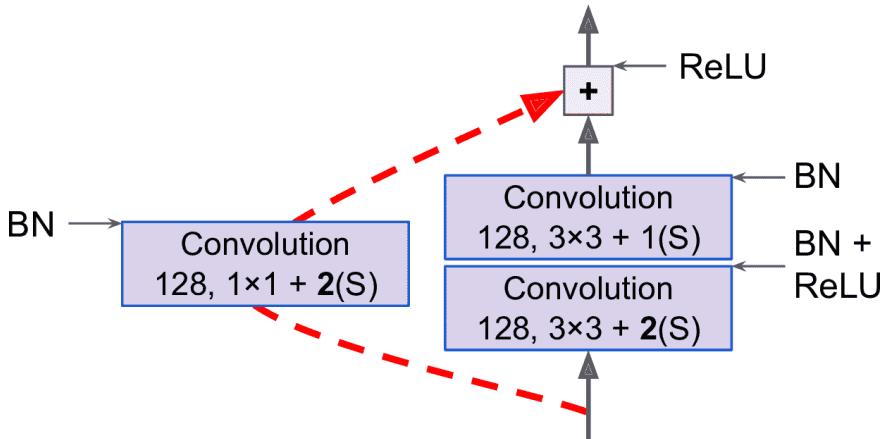


图 14-18 改变特征映射大小和深度时的跳连接

ResNet-34 是有 34 个层（只是计数了卷积层和全连接层）的 ResNet，有 3 个输出 64 个特征映射的残差单元，4 个输出 128 个特征映射的残差单元，6 个输出 256 个特征映射的残差单元，3 个输出 512 个特征映射的残差单元。本章后面会实现这个网络。

ResNet 通常比这个架构要深，比如 ResNet-152，使用了不同的残差单元。不是用  $3 \times 3$  的输出 256 个特征映射的卷积层，而是用三个卷积层：第一是  $1 \times 1$  的卷积层，只有 64 个特征映射（少 4 倍），作为瓶颈层使用；然后是  $1 \times 1$  的卷积层，有 64 个特征映射；最后是另一个  $1 \times 1$  的卷积层，有 256 个特征映射，恢复原始深度。ResNet-152 含有 3 个这样输出 256 个映射的残差单元，8 个输出 512 个映射的残差单元，36 个输出 1024 个映射的残差单元，最后是 3 个输出 2048 个映射的残差单元。

笔记：Google 的 Inception-v4 融合了 GoogLeNet 和 ResNet，使 ImageNet 的 top-5 误差率降低到接近 3%。

## Xception

另一个 GoogLeNet 架构的变体是 [Xception](#) ([Xception](#) 的意思是极限创始，Extreme Inception)。它是由 François Chollet (Keras 的作者) 在 2016 年提出的，Xception 在大型视觉任务 (3.5 亿张图、1.7 万个类) 上超越了 Inception-v3。和 Inception-v4 很像，Xception 融合了 GoogLeNet 和 ResNet，但将创始模块替换成一个特殊类型的层，称为深度可分卷积层 (或简称为可分卷积层)。深度可分卷积层在以前的 CNN 中出现过，但不像 Xception 这样处于核心。常规卷积层使用过滤器同时获取空间图案 (比如，椭圆) 和交叉通道图案 (比如，嘴+鼻子+眼睛=脸)，可分卷积层的假设是空间图案和交叉通道图案可以分别建模 (见图 14-19)。因此，可分卷积层包括两部分：第一个部分对于每个输入特征映射使用单空间过滤器，第二个部分只针对交叉通道图案——就是一个过滤器为  $1 \times 1$  的常规卷积层。

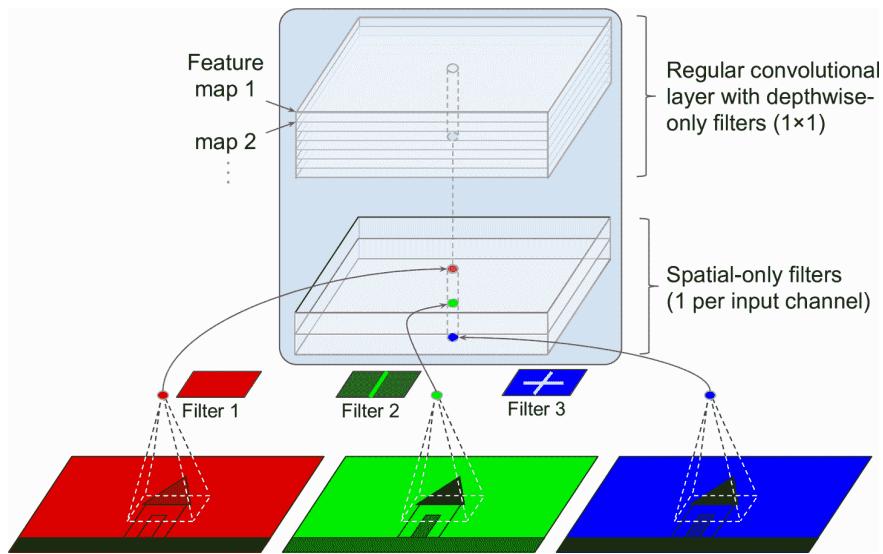


图 14-19 深度可分卷积层

因为可分卷积层对每个输入通道只有一个空间过滤器，要避免在通道不多的层之后使用可分卷积层，比如输入层 (这就是图 14-19 要展示的)。出于这个原因，Xception 架构一开始有 2 个常规卷积层，但剩下的架构都使用可分卷积层 (共 34 个)，加上一些最大池化层和常规的末端层 (全局平均池化层和紧密输出层)。

为什么 Xception 是 GoogLeNet 的变体呢，因为它并没有创始模块？正像前面讨论的，创始模块含有过滤器为  $1 \times 1$  的卷积层：只针对交叉通道图案。但是，它们上面的常规卷积层既针对空间、也针对交叉通道图案。所以可以将创始模块作为常规卷积层和可分卷积层的中间状态。在实际中，可分卷积层表现更好。

**提示：**相比于常规卷积层，可分卷积层使用的参数、内存、算力更少，性能也更好，所以应默认使用后者（除了通道不多的层）。

ILSVRC 2016 的冠军是香港中文大学的 CUImage 团队。他们结合使用了多种不同的技术，包括复杂的对象识别系统，称为 [GBD-Net](#)，top-5 误差率达到 3% 以下。尽管结果很经验，但方案相对于 ResNet 过于复杂。另外，一年后，另一个简单得多的架构取得了更好的结果。

## SENet

ILSVRC 2017 年的冠军是挤压-激活网络 (Squeeze-and-Excitation Network (SENet))。这个架构拓展了之前的创始模块和 ResNet，提高了性能。SENet 的 top-5 误差率达到了惊人的 2.25%。经过拓展之后的版本分别称为 SE-创始模块和 SE-ResNet。性能提升来自于 SENet 在原始架构的每个单元（比如创始模块或残差单元）上添加了一个小的神经网络，称为 SE 块，见图 14-20。

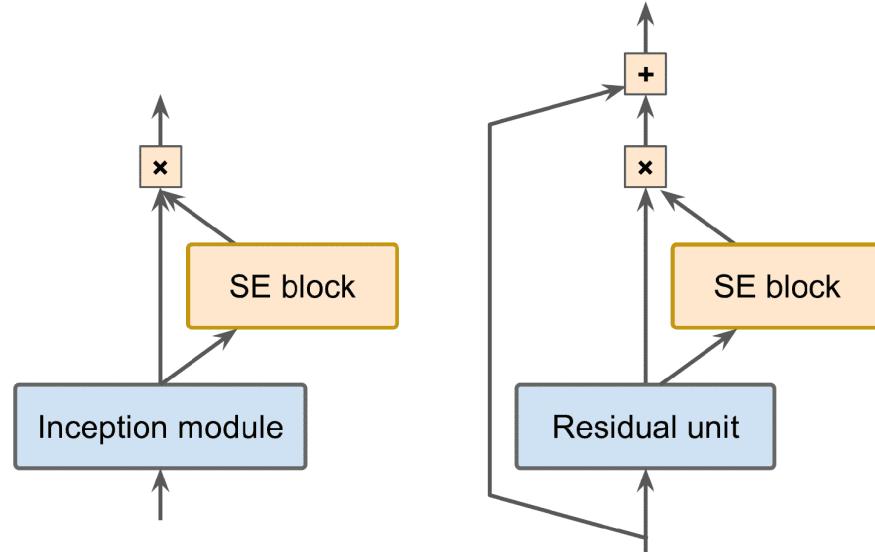


图 14-20 SE-创始模块（左）和 SE-ResNet（右）

SE 分析了单元输出，只针对深度方向，它能学习到哪些特征总是一起活跃的。然后根据这个信息，重新调整特征映射，见图 14-21。例如，SE 可以学习到嘴、鼻子、眼睛经常同时出现在图片中：如果你看见了嘴和鼻子，通常是期待看见眼睛。所以，如果 SE 块发向嘴和鼻子的特征映射有强激活，但眼睛的特征映射没有强激活，就会提升眼睛的特征映射（更准确的，会降低无关的特征映射）。如果眼睛和其他东西搞混了，特征映射重调可以解决模糊性。

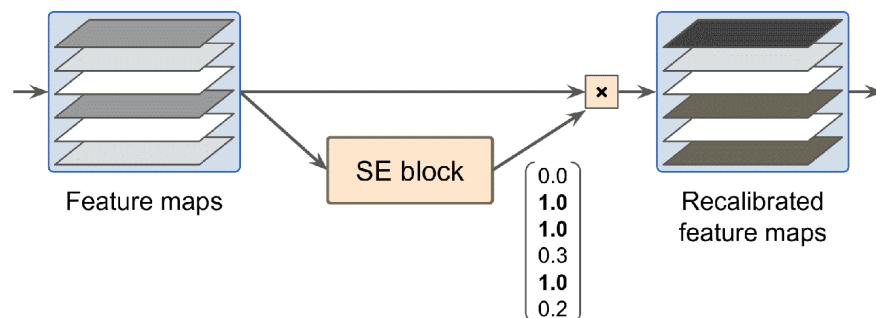


图 14-21 SE 快做特征重调

SE 块由三层组成：一个全局平均池化层、一个使用 ReLU 的隐含紧密层、一个使用 sigmoid 的紧密输出层（见图 14-22）。

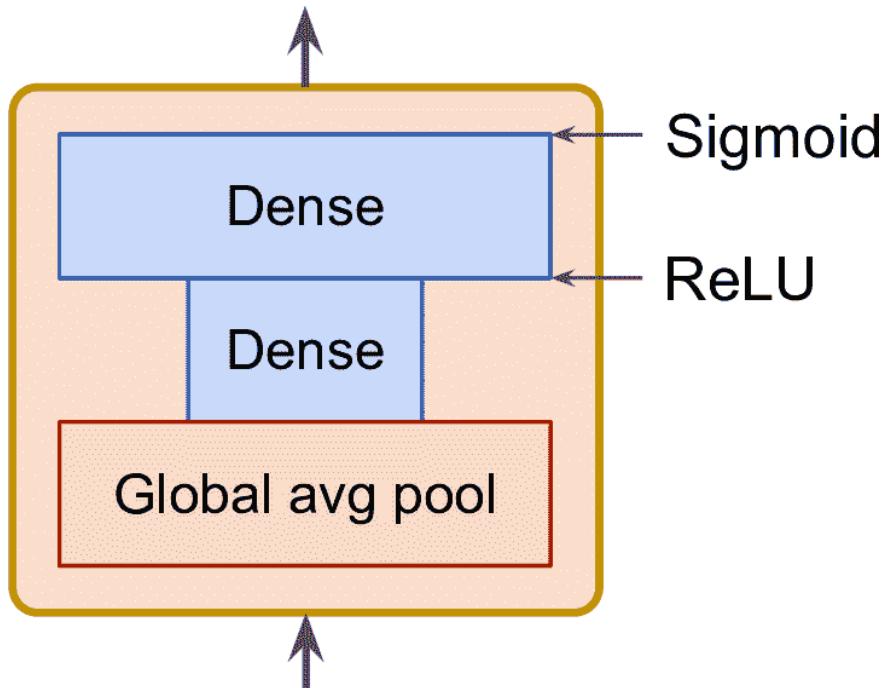


图 14-22 SE 块的结构

和之前一样，全局平均池化层计算每个特征映射的平均激活：例如，如果它的输入包括 256 个特征映射，就会输出 256 个数，表示对每个过滤器的整体响应水平。下一个层是“挤压”步骤：这个层的神经元数远小于 256，通常是小于特征映射数的 16 倍（比如 16 个神经元）——因此 256 个数被压缩成一个向量中（16 维）。这是特征响应的地位向量表征（即，嵌入）。这一步作为瓶颈，能让 SE 块强行学习特征组合的通用表征（第 17 章会再次接触这个原理）。最后，输出层使用这个嵌入，输出一个重调向量，每个特征映射（比如，256）包含一个数，都位于 0 和 1 之间。然后，特征映射乘以这个重调向量，所以无关特征（其重调分数小）就被弱化了，就剩下相关特征（重调分数接近于 1）了。

## 用 Keras 实现 ResNet-34 CNN

目前为止介绍的大多数 CNN 架构的实现并不难（但经常需要加载预训练网络）。

接下来用 Keras 实现 ResNet-34。首先，创建 `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(),
            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)

```

可以看到，这段代码和图 14-18 很接近。在构造器中，创建了所有需要的层：主要的层位于图中右侧，跳跃层位于左侧（只有当步长大于 1 时需要）。在 `call()` 方法中，我们让输入经过主层和跳跃层，然后将输出相加，再应用激活函数。

然后，使用 `Sequential` 模型搭建 ResNet-34，ResNet-34 就是一连串层的组合（将每个残差单元作为一个单一层）：

```

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"))

```

这段代码中唯一麻烦的地方，就是添加 `ResidualUnit` 层的循环部分：前 3 个 RU 有 64 个过滤器，接下来的 4 个 RU 有 128 个过滤器，以此类推。如果过滤器数和前一 RU 层相同，则步长为 1，否则为 2。然后添加 `ResidualUnit`，然后更新 `prev_filters`。

不到 40 行代码就能搭建出 ILSVRC 2015 年冠军模型，既体现出 ResNet 的优美，也展现了 Keras API 的表达力。实现其他 CNN 架构也不困难。但是 Keras 内置了其中一些架构，一起尝试下。

## 使用 Keras 的预训练模型

通常来讲，不用手动实现 GoogLeNet 或 ResNet 这样的标准模型，因为 `keras.applications` 中已经包含这些预训练模型了，只需一行代码就成。例如，要加载在 ImageNet 上预训练的 ResNet-50 模型，使用下面的代码就行：

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
```

仅此而已！这样就能穿件一个 ResNet-50 模型，并下载在 ImageNet 上预训练的权重。要使用它，首先要保证图片有正确的大小。ResNet-50 模型要用  $224 \times 224$  像素的图片（其它模型可能是  $299 \times 299$ ），所以使用 TensorFlow 的 `tf.image.resize()` 函数来缩放图片：

```
images_resized = tf.image.resize(images, [224, 224])
```

提示：`tf.image.resize()` 不会保留宽高比。如果需要，可以裁剪图片为合适的宽高比之后，再进行缩放。两步可以通過 `tf.image.crop_and_resize()` 来实现。

预训练模型的图片要经过特別的预处理。在某些情况下，要求输入是 0 到 1，有时是 -1 到 1，等等。每个模型提供了一个 `preprocess_input()` 函数，来对图片做预处理。这些函数假定像素值的范围是 0 到 255，因此需要乘以 255（因为之前将图片缩减到 0 和 1 之间）：

```
inputs = keras.applications.resnet50.preprocess_input(images_resized * 255)
```

现在就可以用预训练模型做预测了：

```
Y_proba = model.predict(inputs)
```

和通常一样，输出 `Y_proba` 是一个矩阵，每行是一张图片，每列是一个类（这个例子中有 1000 类）。如果想展示 top K 预测，要使用 `decode_predictions()` 函数，将每个预测出的类的名字和概率包括进来。对于每张图片，返回 top K 预测的数组，每个预测表示为包含类标识符、名字和置信度的数组：

```
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()
```

输出如下：

```
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%
```

正确的类（`monastery` 和 `daisy`）出现在 top3 的结果中。考慮到，这是从 1000 个类中挑出来的，结果相当不错。

可以看到，使用预训练模型，可以非常容易的创建出一个效果相当不错的图片分类器。`keras.applications` 中其它视觉模型还有几种 ResNet 的变体，GoogLeNet 的变体（比如 Inception-v3 和 Xception），VGGNet 的变体，MobileNet 和 MobileNetV2（移动设备使用的轻量模型）。

如果要使用的图片分类器不是给 ImageNet 图片做分类的呢？这时，还是可以使用预训练模型来做迁移学习。

## 使用预训练模型做迁移学习

如果想创建一个图片分类器，但没有足够的训练数据，使用预训练模型的低层通常是不错的主意，就像第 11 章讨论过的那样。例如，使用预训练的 Xception 模型训练一个分类花的图片的模型。首先，使用 TensorFlow Datasets 加载数据集（见 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
```

可以通过设定 `with_info=True` 来获取数据集信息。这里，获取到了数据集的大小和类名。但是，这里只有 "train" 训练集，没有测试集和验证集，所以需要分割训练集。TF Datasets 提供了一个 API 来做这项工作。比如，使用数据集的前 10% 作为测试集，接着的 15% 来做验证集，剩下的 75% 来做训练集：

```
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)
```

然后，必须要预处理图片。CNN 的要求是  $224 \times 224$  的图片，所以需要缩放。还要使用 Xception 的 `preprocess_input()` 函数来预处理图片：

```
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
```

对三个数据集使用这个预处理函数，打散训练集，给所有的数据集添加批次和预提取：

```
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)
```

如果想做数据增强，可以修改训练集的预处理函数，给训练图片添加一些转换。例如，使用 `tf.image.random_crop()` 随机裁剪图片，使用 `tf.image.random_flip_left_right()` 做随机水平翻转，等等（参考笔记本的“使用预训练模型做迁移学习”部分）。

**提示：** `keras.preprocessing.image.ImageDataGenerator` 可以方便地从硬盘加载图片，并用多种方式来增强：偏移、旋转、缩放、翻转、裁剪，或使用任何你想做的转换。对于简单项目，这么做很方便。但是，使用 `tf.data` 管道的好处更多：从任何数据源高效读取图片（例如，并行）；操作数据集；如果基于 `tf.image` 运算编写预处理函数，既可以用在 `tf.data` 管道中，也可以用在生产部署的模型中（见第 19 章）。

然后加载一个在 ImageNet 上预训练的 Xception 模型。通过设定 `include_top=False`，排除模型的顶层：排除了全局平均池化层和紧密输出层。我们然后根据基本模型的输出，添加自己的全局平均池化层，然后添加紧密输出层

(没有一个类就有一个单元，使用 softmax 激活函数)。最后，创建 Keras 模型：

```
base_model = keras.applications.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)
```

第 11 章介绍过，最好冻结预训练层的权重，至少在训练初期如此：

```
for layer in base_model.layers:
    layer.trainable = False
```

笔记：因为我们的模型直接使用了基本模型的层，而不是 `base_model` 对象，设置 `base_model.trainable=False` 没有任何效果。

最后，编译模型，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)
```

警告：训练过程非常慢，除非使用 GPU。如果没有 GPU，应该在 Colab 中运行本章的笔记本，使用 GPU 运行时（是免费的！）。见[这里](#)。

模型训练几个周期之后，它的验证准确率应该可以达到 75-80%，然后就没什么提升了。这意味着上层训练的差不多了，此时可以解冻所有层（或只是解冻上边的层），然后继续训练（别忘在冷冻和解冻层是编译模型）。此时使用小得多的学习率，以避免破坏预训练的权重：

```
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(...)
```

训练要花不少时间，最终在测试集上的准确率可以达到 95%。有个模型，就可以训练出惊艳的图片分类器了！计算机视觉除了分类，还有其它任务，比如，想知道花在图片中的位置，该怎么做呢？

## 分类和定位

第 10 章讨论过，定位图片中的物体可以表达为一个回归任务：预测物体的范围框，一个常见的方法是预测物体中心的水平和垂直坐标，和其高度和宽度。不需要大改模型，只要再添加一个有四个单元的紧密输出层（通常是在全局平均池化层的上面），可以用 MSE 损失训练：

```
base_model = keras.applications.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"])
```

但现在有一个问题：花数据集中没有围绕花的边框。因此，我们需要自己加上。这通常是机器学习任务中最难的部分：获取标签。一个好主意是花点时间来找合适的工具。给图片加边框，可供使用的开源图片打标签工具包括 VGG Image Annotator,、LabelImg,、OpenLabeler 或 ImgLab，或是商业工具，比如 LabelBox 或 Supervisely。还可以考虑众包平台，比如如果有很多图片要标注的话，可以使用 Amazon Mechanical Turk。但是，建立众包平台、准备数据格式、监督、保证质量，要做不少工作。如果只有几千张图片要打标签，又不是频繁来做，最好选择自己来做。Adriana Kovashka 等人写了一篇实用的计算机视觉方面的关于众包的[论文](#)，建议读一读。

假设你已经给每张图片的花都获得了边框。你需要创建一个数据集，它的项是预处理好的图片的批次，加上类标签和边框。每项应该是一个元组，格式是 `(images, (class_labels, bounding_boxes))`。然后就可以准备训练模型了！

提示：边框应该做归一化，让中心的横坐标、纵坐标、宽度和高度的范围变成 0 到 1 之间。另外，最好是预测高和宽的平方根，而不是直接预测高和宽：大边框的 10 像素的误差，相比于小边框的 10 像素的误差，不会惩罚那么大。

MSE 作为损失函数来训练模型效果很好，但不是评估模型预测边框的好指标。最常见的指标是交并比（Intersection over Union (IoU)）：预测边框与目标边框的重叠部分，除以两者的并集（见图 14-23）。在 `tf.keras` 中，交并比是用 `tf.keras.metrics.MeanIoU` 类来实现的。



图 14-23 交并比指标

完成了分类并定位单一物体，但如果图片中有多个物体该怎么办呢（常见于花数据集）？

## 目标检测

分类并定位图片中的多个物体的任务被称为目标检测。几年前，使用的方法还是用定位单一目标的 CNN，然后将其在图片上滑动，见图 14-24。在这个例子中，图片被分成了  $6 \times 8$  的网格，CNN（粗黑实线矩形）的范围是  $3 \times 3$ 。当 CNN 查看图片的左上部分时，检测到了最左边的玫瑰花，向右滑动一格，检测到的还是同样的花。又滑动一格，检测到了最上的花，再向右一格，检测到的还是最上面的

花。你可以继续滑动 CNN，查看所有  $6 \times 8$  的区域。另外，因为目标的大小不同，还需要用不同大小的 CNN 来观察。例如，检测完了所有  $6 \times 8$  的区域，可以继续用  $6 \times 8$  的区域来检测。

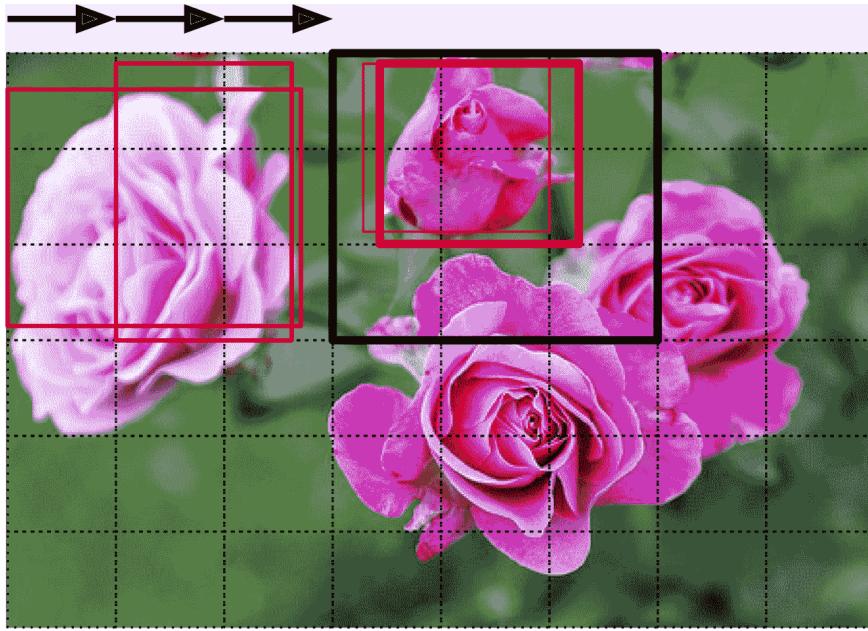


图 14-24 通过滑动 CNN 来检测多个目标

这个方法非常简单易懂，但是也看到了，它会在不同位置、多次检测到同样的目标。需要后处理，去除没用的边框，常见的方法是非极大值抑制（non-max suppression）。步骤如下：

1. 首先，给 CNN 添加另一个对象性输出，来估计花确实出现在图片中的概率（或者，可以添加一个“没有花”的类，但通常不好使）。必须要使用 sigmoid 激活函数，可以用二元交叉熵损失函数来训练。然后删掉对象性分数低于某阈值的所有边框：这样能删掉所有不包含花的边框。
2. 找到对象性分数最高的边框，然后删掉所有其它与之大面积重叠的边框（例如，IoU 大于 60%）。例如，在图 14-24 中，最大对象性分数的边框出现在最上面花的粗宾匡（对象性分数用边框的粗细来表示）。另一个边框和这个边框重合很多，所以将其删除。
3. 重复这两个步骤，直到没有可以删除的边框。

用这个简单的方法来做目标检测的效果相当不错，但需要运行 CNN 好几次，所以很慢。幸好，有一个更快的方法来滑动 CNN：使用全卷积网络（fully convolutional network, FCN）。

## 全卷积层

FCN 是 Jonathan Long 在 2015 年的一篇[论文](#)汇总提出的，用于语义分割（根据所属目标，对图片中的每个像素点进行分类）。作者指出，可以用卷积层替换 CNN 顶部的紧密层。要搞明白，看一个例子：假设一个 200 个神经元的紧密层，位于卷积层的上边，卷积层输出 100 个特征映射，每个大小是  $7 \times 7$ （这是特征映射的大小，不是核大小）。每个神经元会计算卷积层的  $100 \times 7 \times 7$  个激活结果的加权和（加上偏置项）。现在将紧密层替换为卷积层，有 200 个过滤器，每个大小为  $7 \times 7$ ，“valid” 填充。这个层能输出 200 个特征映射，每个是  $7 \times 7$ （因为

核大小等于输入特征映射的大小，并且使用的是 "valid" 填充）。换句话说，会产生 200 个数，和紧密层一样；如果仔细观察卷积层的计算，会发现这些数和紧密层输出的数一模一样。唯一不同的地方，紧密层的输出的张量形状是 [批次大小, 200]，而卷积层的输出的张量形状是 [批次大小, 1, 1, 200]。

提示：要将紧密层变成卷积层，卷积层中的过滤器的数量，必须等于紧密层的神经元数，过滤器大小必须等于输入特征映射的大小，必须使用 "valid" 填充。步长可以是 1 或以上。

为什么这点这么重要？紧密层需要的是一个具体的输入大小（因为它的每个输入特征都有一个权重），卷积层却可以处理任意大小的图片（但是，它也希望输入有一个确定的通道数，因为每个核对每个输入通道包含一套不同的权重集合）。因为 FCN 只包含卷积层（和池化层，属性相同），所以可以在任何大小的图片上训练和运行。

举个例子，假设已经训练好了一个用于分类和定位的 CNN。图片大小是  $224 \times 224$ ，输出 10 个数：输出 0 到 4 经过 softmax 激活函数，给出类的概率；输出 5 经过逻辑激活函数，给出对象性分数；输出 6 到 9 不经过任何激活函数，表示边框的中心坐标、高和宽。

现在可以将紧密层转换为卷积层。事实上，不需要再次训练，只需将紧密层的权重复制到卷积层中。另外，可以在训练前，将 CNN 转换成 FCN。

当输入图片为  $224 \times 224$  时（见图 14-25 的左边），假设输出层前面的最后一个卷积层（也被称为瓶颈层）输出  $224 \times 224$  的特征映射。如果 FCN 的输入图片是  $448 \times 448$ （见图 14-25 的右边），瓶颈层会输出  $224 \times 224$  的特征映射。因为紧密输出层被替换成了 10 个使用大小为  $224 \times 224$  的过滤器的卷积层，"valid" 填充，步长为 1，输出会有 10 个特征映射，每个大小为  $448 \times 448$ （因为  $14 - 7 + 1 = 8$ ）。换句话说，FCN 只会处理整张图片一次，会输出  $224 \times 224$  的网格，每个格子有 10 个数（5 个类概率，1 个对象性分数，4 个边框参数）。就像之前滑动 CNN 那样，每行滑动 8 步，每列滑动 8 步。再形象的讲一下，将原始图片切分成  $224 \times 224$  的网格，然后用  $224 \times 224$  的窗口在上面滑动，窗口会有  $8 \times 8 = 64$  个可能的位置，也就是 64 个预测。但是，FCN 方法又非常高效，因为只需观察图片一次。事实上，“只看一次”（You Only Look Once，YOLO）是一个非常流行的目标检测架构的名字，下面介绍。

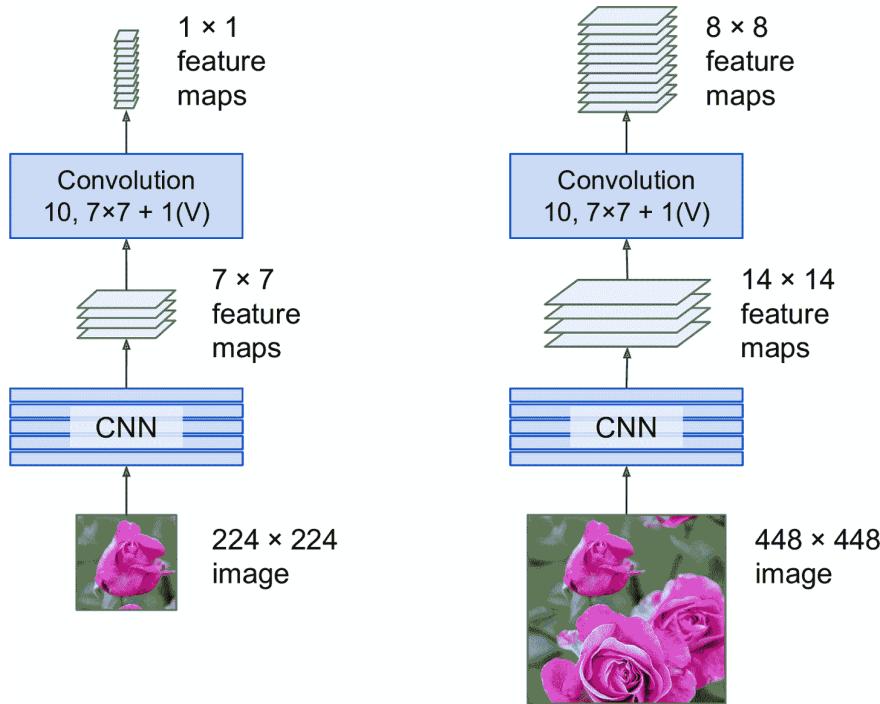


图 14-25 相同的 FCN 处理小图片（左）和大图片（右）

## 只看一次 (YOLO)

YOLO 是一个非常快且准确的目标检测框架，是 Joseph Redmon 在 2015 年的一篇[论文](#)中提出的，2016 年优化为 [YOLOv2](#)，2018 年优化为 [YOLOv3](#)。速度快到甚至可以在实时视频中运行，可以看 Redmon 的这个例子（[要翻墙](#)）。

YOLOv3 的架构和之前讨论过的很像，只有一些重要的不同点：

- 每个网格输出 5 个边框（不是 1 个），每个边框都有一个对象性得分。每个网格还输出 20 个类概率，是在 PASCAL VOC 数据集上训练的，这个数据集有 20 个类。每个网格一共有 45 个数：5 个边框，每个 4 个坐标参数，加上 5 个对象性分数，加上 20 个类概率。
- YOLOv3 不是预测边框的绝对坐标，而是预测相对于网格坐标的偏置量， $(0, 0)$  是网格的左上角， $(1, 1)$  是网格的右下角。对于每个网格，YOLOv3 是被训练为只能预测中心位于网格的边框（边框通常比网格大得多）。YOLOv3 对边框坐标使用逻辑激活函数，以保证其在 0 到 1 之间。
- 开始训练神经网络之前，YOLOv3 找了 5 个代表性边框维度，称为锚定框（anchor box）（或称为前边框）。它们是通过 K-Means 算法（见第 9 章）对训练集边框的高和宽计算得到的。例如，如果训练图片包含许多行人，一个锚定框就会获取行人的基本维度。然后当神经网络对每个网格预测 5 个边框时，实际是预测如何缩放每个锚定框。比如，假设一个锚定框是 100 个像素高，50 个像素宽，神经网络可能的预测是垂直放大到 1.5 倍，水平缩小为 0.9 倍。结果是  $150 \times 45$  的边框。更准确的，对于每个网格和每个锚定框，神经网络预测其垂直和水平缩放参数的对数。有了锚定框，可以更容易预测出边框，因为可以更快的学到边框的样子，速度也会更快。

- 神经网络是用不同规模的图片来训练的：每隔几个批次，网络就随机调训新照片维度（从  $330 \times 330$  到  $330 \times 330$  像素）。这可以让网络学到不同的规模。  
另外，还可以在不同规模上使用 YOLOv3：小图比大图快但准确性差。

还可能有些有意思的创新，比如使用跳连接来恢复一些在 CNN 中损失的空间分辨率，后面讨论语义分割时会讨论。在 2016 年的这篇论文中，作者介绍了使用层级分类的 YOLO9000 模型：模型预测视觉层级（称为词树，WordTree）中的每个节点的概率。这可以让网络用高置信度预测图片表示的是什么，比如狗，即便不知道狗的品种。建议阅读这三篇论文：不仅文笔不错，还给出不少精彩的例子，介绍深度学习系统是如何一点一滴进步的。

### 平均精度均值 (mean Average Precision, mAP)

目标检测中非常常见的指标是平均精度均值。“平均均值”听起来啰嗦了。要弄明白这个指标，返回到第3章中的两个分类指标：精确率和召回率。取舍关系：召回率越高，精确率就越低。可以在精确率/召回率曲线上看到。将这条曲线归纳为一个数，可以计算曲线下面积（AUC）。但精确率/召回率曲线上有些部分，当精确率上升时，召回率也上升，特别是当召回率较低时（可以在图3-5的顶部看到）。这就是产生mAP的激励之一。

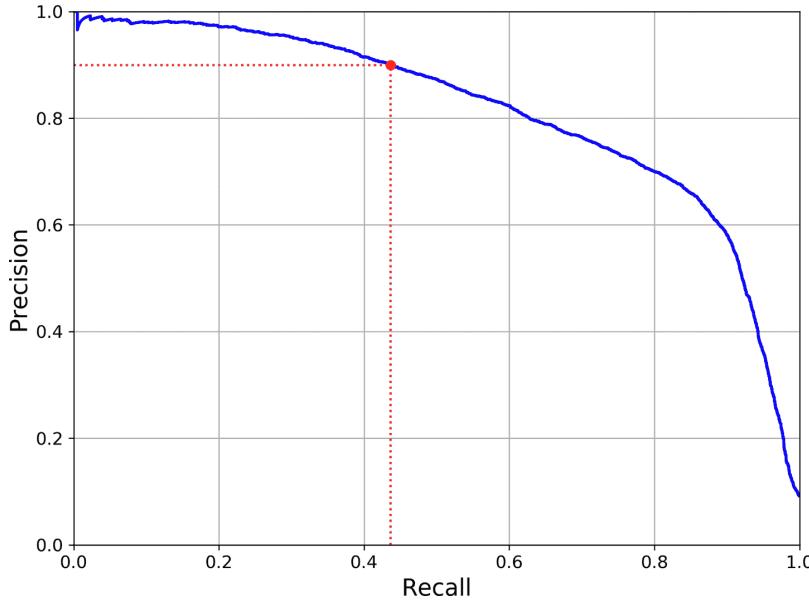


图3-5 精确率 vs 召回率

假设当召回率为10%时，分类器的精确率是90%，召回率为20%时，精确率是96%。这里就没有取舍关系：使用召回率为20%的分类器就好，因为此时精确率更高。所以当召回率至少有10%时，需要找到最高精确率，即96%。因此，一个衡量模型性能的方法是计算召回率至少为0%时，计算最大精确率，再计算召回率至少为10%时的最大精确率，再计算召回率至少为20%时的最大精确率，以此类推。最后计算这些最大精确率的平均值，这个指标称为平均精确率（Average Precision, AP）。当有超过两个类时，可以计算每个类的AP，然后计算平均AP（即，mAP）。就是这样！

在目标检测中，还有另外一个复杂度：如果系统检测到了正确的类，但是定位错了（即，边框不对）？当然不能将其作为正预测。一种方法是定义IOU阈值：例如，只有当IOU超过0.5时，预测才是正确的。相应的mAP表示为mAP@0.5（或mAP@50%，或AP50）。在一些比赛中（比如PASCAL VOC竞赛），就是这么做的。在其它比赛中（比如，COCO），mAP是用不同IOU阈值（0.50, 0.55, 0.60, ..., 0.95）计算的。最终指标是所有这些mAP的均值（表示为AP@[.50:.95]或AP@[.50:0.05:.95]），这是均值的均值。

一些YOLO的TensorFlow实现可以在GitHub上找到。可以看看[Zihao Zang用TensorFlow 2实现的项目](#)。TensorFlow Models项目中还有其它目标检测模型；一些还传到了TF Hub，比如SSD和Faster-RCNN，这两个都很流行。SSD也是一个“一次”检测模型，类似于YOLO。Faster R-CNN复杂一些：图片先经过CNN，然后输出经过区域提议网络（Region Proposal Network, RPN），RPN对边框做处理，更容易圈住目标。根据CNN的裁剪输出，每个边框都运行这一个分类器。

检测系统的选择取决于许多因素：速度、准确率、预训练模型是否可用、训练时间、复杂度，等等。论文中有许多指标表格，但测试环境的变数很多。技术进步也很快，很难比较出哪个更适合大多数人，并且有效期可以长过几个月。

## 语义分割

在语义分割中，每个像素根据其所属的目标来进行分类（例如，路、汽车、行人、建筑物，等等），见图 14-26。注意，相同类的不同目标是不做区分的。例如，分割图片的右侧的所有自行车被归类为一坨像素。这个任务的难点是当图片经过常规 CNN 时，会逐渐丢失空间分辨率（因为有的层的步长大于 1）；因此，常规的 CNN 可以检测出图片的左下有一个人，但不知道准确的位置。

和目标检测一样，有多种方法来解决这个问题，其中一些比较复杂。但是，之前说过，Jonathan Long 等人在 2015 年的一篇论文中提出乐意简单的方法。作者先将预训练的 CNN 转变为 FCN，CNN 使用 32 的总步长（即，将所有大于 1 的步长相加）作用到输入图片上，最后一层的输出特征映射比输入图片小 32 倍。这样过于粗糙，所以添加了一个单独的上采样层，将分辨率乘以 32。



图 14-26 语义分割

有几种上采样（增加图片大小）的方法，比如双线性插值，但只在  $\times 4$  或  $\times 8$  时好用。Jonathan Long 等人使用了转置卷积层：等价于，先在图片中插入空白的行和列（都是 0），然后做一次常规卷积（见图 14-27）。或者，有人将其考虑为常规卷积层，使用分数步长（比如，图 14-27 中是  $1/2$ ）。转置卷积层一开始的表现和线性插值很像，但因为是可训练的，在训练中会变得更好。在 `tf.keras` 中，可以使用 `Conv2DTranspose` 层。

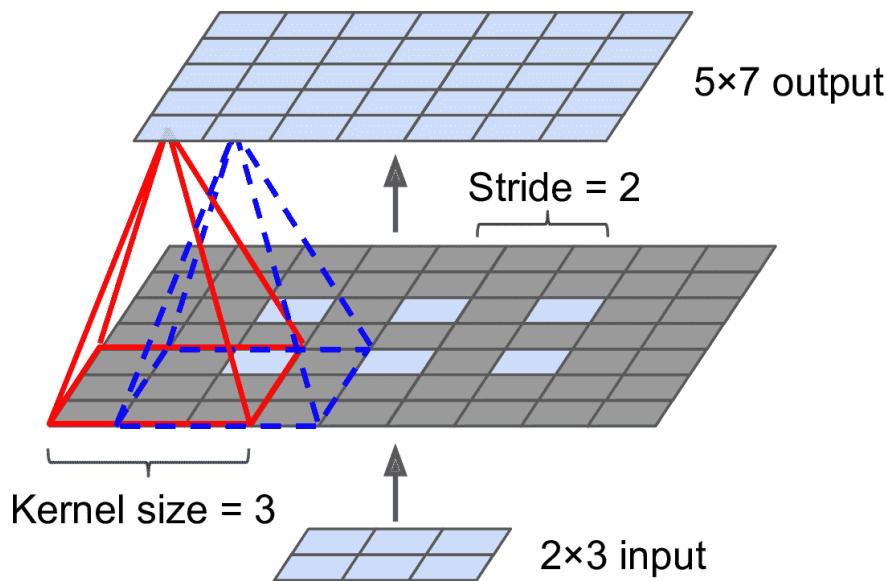


图 14-27 使用转置卷积层做上采样

笔记：在转置卷积层中，步长定义为输入图片被拉伸的倍数，而不是过滤器步长。所以步长越大，输出也就越大（和卷积层或池化层不同）。

TensorFlow 卷积运算

TensorFlow 还提供了一些其它类型的卷积层：

`keras.layers.Conv1D`：为 1D 输入创建卷积层，比如时间序列或文本，第 15 章会见到。

`keras.layers.Conv3D`：为 3D 输入创建卷积层，比如 3D PET 扫描。

`dilation_rate`：将任何卷积层的 `dilation_rate` 超参数设为 2 或更大，可以创建有孔卷积层。等价于常规卷积层，加上一个膨胀的、插入了空白行和列的过滤器。例如，一个  $1 \times 3$  的过滤器 `[[1, 2, 3]]`，膨胀 4 倍，就变成了 `[[1, 0, 0, 0, 2, 0, 0, 0, 3]]`。这可以让卷积层有一个更大的感受野，却没有增加计算量和额外的参数。

`tf.nn.depthwise_conv2d()`：可以用来创建深度方向卷积层（但需要自己创建参数）。它将每个过滤器应用到每个独立的输入通道上。因此，因此，如果有 `f[n]` 个过滤器和 `f[n']` 个输入通道，就会输出 `f[n] x f[n']` 个特征映射。

这个方法行得通，但还是不够准确。要做的更好，作者从低层开始就添加了跳连接：例如，他们使用因子 2（而不是 32）对输出图片做上采样，然后添加一个低层的输出。然后对结果做因子为 16 的上采样，总的上采样因子为 32（见图 14-28）。这样可以恢复一些在早期池化中丢失的空间分辨率。在他们的最优架构中，他们使用了两个相似的跳连接，以从更低层恢复更小的细节。

总之，原始 CNN 的输出又经过了下面的步骤：上采样  $\times 2$ ，加上一个低层的输出（形状相同），上采样  $\times 2$ ，加上一个更低层的输出，最后上采样  $\times 8$ 。甚至可以放大，超过原图大小：这个方法可以用来提高图片的分辨率，这个技术成为超分辨率。

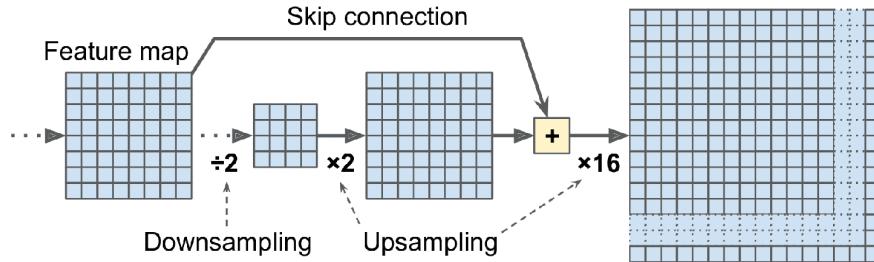


图 14-28 跳连接可以从低层恢复一些空间分辨率

许多 GitHub 仓库提供了语义分割的 TensorFlow 实现，还可以在 TensorFlow Models 中找到预训练的实例分割模型。实例分割和语义分割类似，但不是将相同类的所有物体合并成一坨，而是将每个目标都分开（可以将每辆自行车都分开）。目前，TensorFlow Models 中可用的实例分割时基于 Mask R-CNN 架构的，是在 2017 年的一篇[论文](#)中提出的：通过给每个边框做一个像素罩，拓展 Faster R-CNN 模型。所以不仅能得到边框，还能获得边框中像素的像素罩。

可以发现，深度计算机视觉领域既宽广又发展迅速，每年都会产生新的架构，都是基于卷积神经网络的。最近几年进步惊人，研究者们现在正聚焦于越来越难的问题，比如对抗学习（可以让网络对具有欺骗性的图片更有抵抗力），可解释性（理解为什么网络做出这样的分类），实时图像生成（见第 17 章），一次学习（观察一次，就能认出目标呢系统）。一些人在探索全新的架构，比如 Geoffrey Hinton 的[胶囊网络](#)（见[视频](#)，笔记本中有对应的代码）。下一章会介绍如何用循环神经网络和卷积神经网络来处理序列数据，比如时间序列。

## 练习

1. 对于图片分类，CNN 相对于全连接 DNN 的优势是什么？
2. 考虑一个 CNN，有 3 个卷积层，每个都是  $3 \times 3$  的核，步长为 2，零填充。最低的层输出 100 个特征映射，中间的输出 200 个特征映射，最上面的输出 400 个。输入图片是  $3 \times 3$  像素的 RGB 图。这个 CNN 的总参数量是多少？如果使用 32 位浮点数，做与测试需要多少内存？批次是 50 张图片，训练时的内存消耗是多少？
3. 如果训练 CNN 时 GPU 内存不够，解决该问题的 5 种方法是什么？
4. 为什么使用最大池化层，而不是同样步长的卷积层？
5. 为什么使用局部响应归一化层？
6. AlexNet 想对于 LeNet-5 的创新在哪里？GoogLeNet、ResNet、SENet、Xception 的创新又是什么？
7. 什么是全卷积网络？如何将紧密层转变为卷积层？
8. 语义分割的主要技术难点是什么？
9. 从零搭建你的 CNN，并在 MNIST 上达到尽可能高的准确率。
10. 使用迁移学习来做大图片分类，经过下面步骤：
  - a. 创建每个类至少有 100 张图片的训练集。例如，你可以用自己的图片基于地点来分类（沙滩、山、城市，等等），或者使用现成的数据集（比如从 TensorFlow Datasets）。

- b. 将其分成训练集、验证集、训练集。
  - c. 搭建输入管道，包括必要的预处理操作，最好加上数据增强。
  - d. 在这个数据集上，微调预训练模型。
1. 尝试下 TensorFlow 的[风格迁移教程](#)。用深度学习生成艺术作品很有趣。

## 十五、使用 RNN 和 CNN 处理序列

译者：[@SeanCheney](#)

击球手击出垒球，外场手会立即开始奔跑，并预测球的轨迹。外场手追踪球，不断调整移动步伐，最终在观众的掌声中抓到它。无论是在听完朋友的话还是早餐时预测咖啡的味道，你时刻在做的事就是在预测未来。在本章中，我们将讨论循环神经网络，一类可以预测未来的网络（当然，是到某一点为止）。它们可以分析时间序列数据，比如股票价格，并告诉你什么时候买入和卖出。在自动驾驶系统中，他们可以预测行车轨迹，避免发生事故。更一般地说，它们可在任意长度的序列上工作，而不是截止目前我们讨论的只能在固定长度的输入上工作的网络。举个例子，它们可以将语句，文件，以及语音范本作为输入，应用在自动翻译，语音到文本的自然语言处理应用中。

在本章中，我们将学习循环神经网络的基本概念，如何使用时间反向传播训练网络，然后用来预测时间序列。然后，会讨论 RNN 面对的两大难点：

- 不稳定梯度（换句话说，在第 11 章中讨论的梯度消失/爆炸），可以使用多种方法缓解，包括循环丢弃和循环层归一化。
- 有限的短期记忆，可以通过 LSTM 和 GRU 单元延长。

RNN 不是唯一能处理序列数据的神经网络：对于小序列，常规紧密网络也可以；对于长序列，比如音频或文本，卷积神经网络也可以。我们会讨论这两种方法，本章最后会实现一个 WaveNet：这是一种 CNN 架构，可以处理上万个时间步的序列。在第 16 章，还会继续学习 RNN，如何使用 RNN 来做自然语言处理，和基于注意力机制的新架构。

### 循环神经元和层

到目前为止，我们主要关注的是前馈神经网络，激活仅从输入层到输出层的一个方向流动（附录 E 中的几个网络除外）。循环神经网络看起来非常像一个前馈神经网络，除了它也有连接指向后方。让我们看一下最简单的 RNN，由一个神经元接收输入，产生一个输出，并将输出发送回自己，如图 15-1（左）所示。在每个时间步  $t$ （也称为一个帧），这个循环神经元接收输入  $x[t]$  以及它自己的前一时间步长  $y[t - 1]$  的输出。因为第一个时间步骤没有上一次的输出，所以是 0。可以用时间轴来表示这个微小的网络，如图 15-1（右）所示。这被称为随时间展开网络。

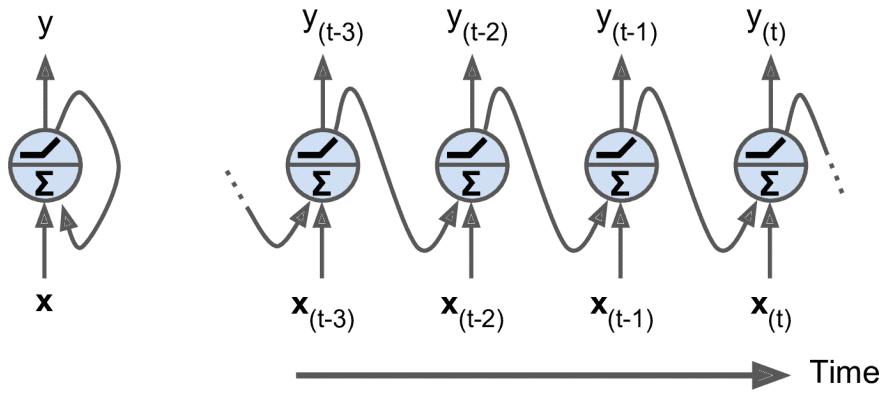


图 15-1 循环神经网络（左），随时间展开网络（右）

你可以轻松创建一个循环神经元层。在每个时间步  $t$ ，每个神经元都接收输入向量  $x[t]$  和前一个时间步  $y[t - 1]$  的输出向量，如图 15-2 所示。注意，输入和输出都是向量（当只有一个神经元时，输出是一个标量）。

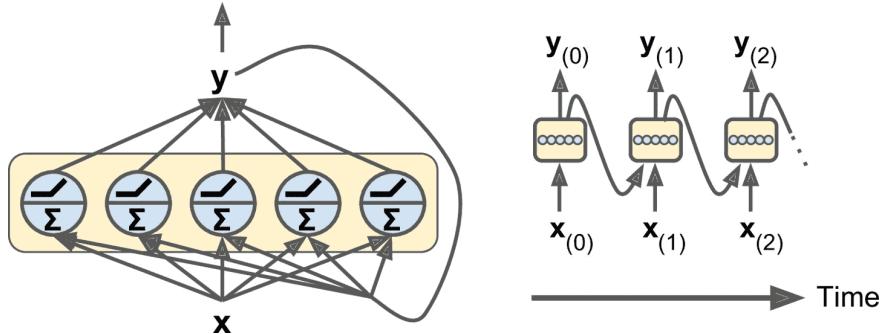


图 15-2 一层循环神经元（左），及其随时间展开（右）

每个循环神经元有两组权重：一组用于输入  $x[t]$ ，另一组用于前一时间步长  $y[t - 1]$  的输出。我们称这些权重向量为  $w[x]$  和  $w[y]$ 。如果考虑的是整个循环神经元层，可以将所有权重向量放到两个权重矩阵中， $w[x]$  和  $w[y]$ 。整个循环神经元层的输出可以用公式 15-1 表示（ $b$  是偏差项， $\phi(\cdot)$  是激活函数，例如 ReLU）。

$$\mathbf{y}(t) = \phi(\mathbf{W}_x^\top \mathbf{x}(t) + \mathbf{W}_y^\top \mathbf{y}(t-1) + \mathbf{b})$$

公式 15-1 单个实例的循环神经元层的输出

就像前馈神经网络一样，可以将所有输入和时间步  $t$  放到输入矩阵  $\mathbf{x}[t]$  中，一次计算出整个小批次的输出：（见公式 15-2）。

$$\begin{aligned} \mathbf{Y}(t) &= \phi(\mathbf{X}(t) \mathbf{W}_x + \mathbf{Y}(t-1) \mathbf{W}_y + \mathbf{b}) \\ &= \phi([\mathbf{X}(t) \quad \mathbf{Y}(t-1)] \mathbf{W} + \mathbf{b}) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix} \end{aligned}$$

公式 15-2 小批次实例的循环层输出

在这个公式中：

- $Y[t]$  是  $m \times n_{\text{neurons}}$  矩阵，包含在小批次中每个实例在时间步  $t$  的层输出（ $m$  是小批次中的实例数， $n_{\text{neurons}}$  是神经元数）。
- $X[t]$  是  $m \times n_{\text{inputs}}$  矩阵，包含所有实例的输入（ $n_{\text{inputs}}$  是输入特征的数量）。
- $W[x]$  是  $n_{\text{inputs}} \times n_{\text{neurons}}$  矩阵，包含当前时间步的输入的连接权重。
- $W[y]$  是  $n_{\text{neurons}} \times n_{\text{neurons}}$  矩阵，包含上一个时间步的输出的连接权重。
- $b$  是大小为  $n_{\text{neurons}}$  的向量，包含每个神经元的偏置项。
- 权重矩阵  $W[x]$  和  $W[y]$  通常纵向连接成一个权重矩阵  $W$ ，形状为  $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$ （见公式 15-2 的第二行）

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，以此类推。这使得  $Y[t]$  是从时间  $t = 0$  开始的所有输入（即  $x[0], x[1], \dots, x[t]$ ）的函数。在第一个时间步， $t = 0$ ，没有以前的输出，所以它们通常被假定为全零。

## 记忆单元

由于时间  $t$  的循环神经元的输出，是由所有先前时间步骤计算出来的函数，你可以说它有一种记忆形式。神经网络的一部分，保留一些跨越时间步长的状态，称为存储单元（或简称为单元）。单个循环神经元或循环神经元层是非常基本的单元，只能学习短期规律（取决于具体任务，通常是 10 个时间步）。本章后面我们将介绍一些更为复杂和强大的单元，可以学习更长时间步的规律（也取决于具体任务，大概是 100 个时间步）。

一般情况下，时间步  $t$  的单元状态，记为  $h[t]$ （ $h$  代表“隐藏”），是该时间步的某些输入和前一时间步状态的函数： $h[t] = f(h[t - 1], x[t])$ 。其在时间步  $t$  的输出，表示为  $y[t]$ ，也和前一状态和当前输入的函数有关。我们已经讨论过的基本单元，输出等于单元状态，但是在更复杂的单元中并不总是如此，如图 15-3 所示。

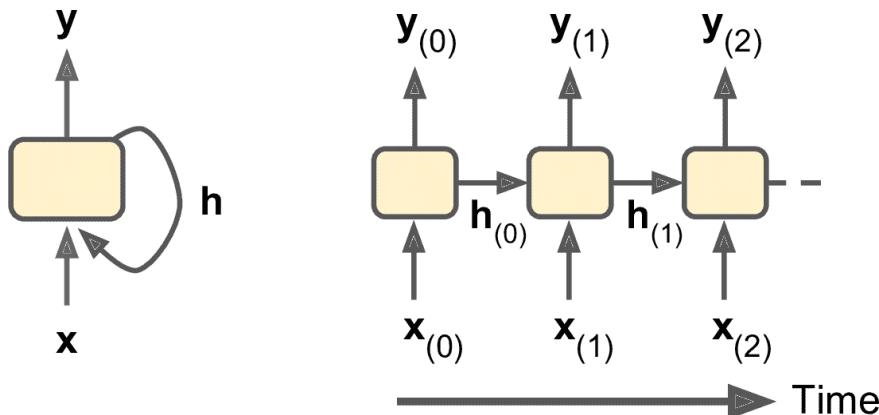


图 15-3 单元的隐藏状态和输出可能不同

## 输入和输出序列

RNN 可以同时输入序列并输出序列（见图 15-4，左上角的网络）。这种序列到序列的网络可以有效预测时间序列（如股票价格）：输入过去  $N$  天价格，则输出向未来移动一天的价格（即，从  $N - 1$  天前到明天）。

或者，你可以向网络输入一个序列，忽略除最后一项之外的所有输出（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，你可以向网络输入与电影评论相对应的单词序列，网络输出情感评分（例如，从 -1 [讨厌] 到 +1 [喜欢]）。

相反，可以向网络一遍又一遍输入相同的向量（见图 15-4 的左下角），输出一个序列。这是一个向量到序列的网络。例如，输入可以是图像（或是 CNN 的结果），输出是该图像的标题。

最后，可以有一个序列到向量的网络，称为编码器，后面跟着一个称为解码器的向量到序列的网络（见图 15-4 右下角）。例如，这可以用于将句子从一种语言翻译成另一种语言。给网络输入一种语言的一句话，编码器会把这个句子转换成单一的向量表征，然后解码器将这个向量解码成另一种语言的句子。这种称为编码器-解码器的两步模型，比用单个序列到序列的 RNN 实时地进行翻译要好得多，因为句子的最后一个单词可以影响翻译的第一句话，所以你需要等到听完整个句子才能翻译。第 16 章还会介绍如何实现编码器-解码器（会比图 15-4 中复杂）

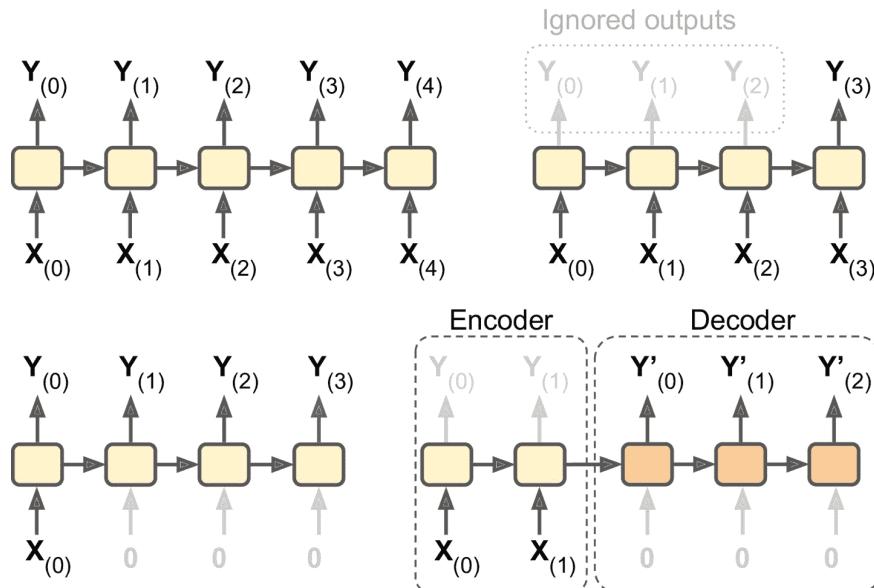


图 15-4 序列到序列（左上），序列到向量（右上），向量到序列（左下），延迟序列到序列（右下）

## 训练 RNN

训练 RNN 诀窍是在时间上展开（就像我们刚刚做的那样），然后只要使用常规反向传播（见图 15-5）。这个策略被称为时间上的反向传播（BPTT）。

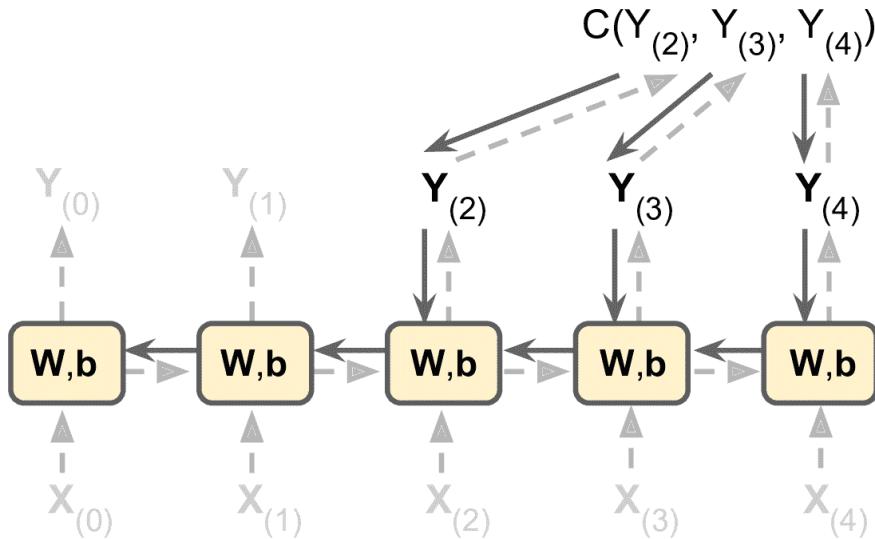


图 15-5 随时间反向传播

就像在正常的反向传播中一样，展开的网络（用虚线箭头表示）中先有一个正向传播（虚线）。然后使用损失函数  $C(Y[0], Y[1], \dots, Y[T])$  评估输出序列（其中  $T$  是最大时间步）。这个损失函数会忽略一些输出，见图 15-5（例如，在序列到向量的 RNN 中，除了最后一项，其它的都被忽略了）。损失函数的梯度通过展开的网络反向传播（实线箭头）。最后使用在 BPTT 期间计算的梯度来更新模型参数。注意，梯度在损失函数所使用的所有输出中反向流动，而不仅仅通过最终输出（例如，在图 15-5 中，损失函数使用网络的最后三个输出  $Y[2]$ ， $Y[3]$  和  $Y[4]$ ，所以梯度流经这三个输出，但不通过  $Y[0]$  和  $Y[1]$ 。而且，由于在每个时间步骤使用相同的参数  $w$  和  $b$ ，所以反向传播将做正确的事情并对所有时间步求和。

幸好，`tf.keras` 处理了这些麻烦。

## 预测时间序列

假设你在研究网站每小时的活跃用户数，或是所在城市的每日气温，或公司的财务状况，用多种指标做季度衡量。在这些任务中，数据都是一个序列，每步有一个或多个值。这被称为时间序列。在前两个任务中，每个时间步只有一个值，它们是单变量时间序列。在财务状况的任务中，每个时间步有多个值（利润、欠账，等等），所以是多变量时间序列。典型的任务是预测未来值，称为“预测”。另一个任务是填空：预测（或“后测”）过去的缺失值，这被称为“填充”。例如，图 15-6 展示了 3 个单变量时间序列，每个都有 50 个时间步，目标是预测下一个时间步的值（用  $x$  表示）。

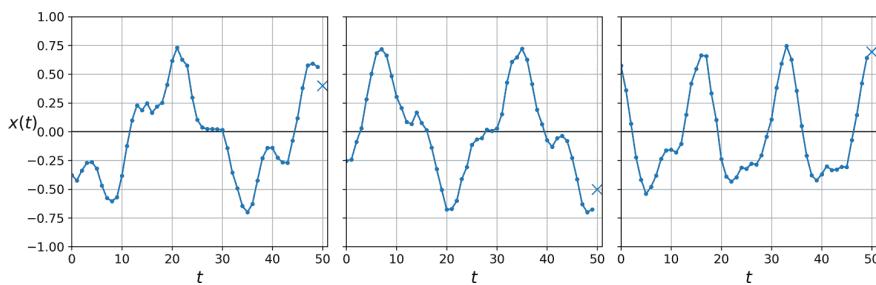


图 15-6 时间序列预测

简单起见，使用函数 `generate_time_series()` 生成的时间序列，如下：

```
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)
```

这个函数可以根据要求创建出时间序列（通过 `batch_size` 参数），长度为 `n_steps`，每个时间步只有 1 个值。函数返回 NumPy 数组，形状是 [批次大小, 时间步数, 1]，每个序列是两个正弦波之和（固定强度+随机频率和相位），加一点噪音。

笔记：当处理时间序列时（和其它类型的时间序列），输入特征通常用 3D 数组来表示，其形状是 [批次大小, 时间步数, 维度]，对于单变量时间序列，其维度是 1，多变量时间序列的维度是其维度数。

用这个函数来创建训练集、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，形状是 [7000, 50, 1]），`X_valid` 有 2000 个，`X_test` 有 1000 个。因为预测的是单一值，目标值是列向量（`y_train` 的形状是 [7000, 1]）。

## 基线模型

使用 RNN 之前，最好有基线指标，否则做出来的模型可能比基线模型还糟。例如，最简单的方法，是预测每个序列的最后一个值。这个方法被称为朴素预测，有时很难被超越。在这个例子中，它的均方误差为 0.020:

```
>>> y_pred = X_valid[:, -1]
>>> np.mean(keras.losses.mean_squared_error(y_valid, y_pred))
0.020211367
```

另一个简单的方法是使用全连接网络。因为结果要是打平的特征列表，需要加一个 `Flatten` 层。使用简单线性回归模型，使预测值是时间序列中每个值的线性组合：

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])
```

使用 MSE 损失、Adam 优化器编译模型，在训练集上训练 20 个周期，用验证集评估，最终得到的 MSE 值为 0.004。比朴素预测强多了！

## 实现一个简单 RNN

搭建一个简单 RNN 模型：

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

这是能实现的最简单的 RNN。只有 1 个层，1 个神经元，如图 15-1。不用指定输入序列的长度（和之前的模型不同），因为循环神经网络可以处理任意的时间步（这就是为什么将第一个输入维度设为 `None`）。默认时，`SimpleRNN` 使用双曲正切激活函数。和之前看到的一样：初始状态 `h[init]` 设为 0，和时间序列的第一个值 `x[0]` 一起传递给神经元。神经元计算这两个值的加权和，对结果使用双曲正切激活函数，得到第一个输出 `y[0]`。在简单 RNN 中，这个输出也是新状态 `h[0]`。这个新状态和下一个输入值 `x[1]`，按照这个流程，直到输出最后一个值，`y[49]`。所有这些都是同时对每个时间序列进行的。

**笔记：**默认时，Keras 的循环层只返回最后一个输出。要让其返回每个时间步的输出，必须设置 `return_sequences=True`。

用这个模型编译、训练、评估（和之前一样，用 Adam 训练 20 个周期），你会发现它的 MSE 只有 0.014。击败了朴素预测，但不如简单线性模型。对于每个神经元，线性简单模型中每个时间步骤每个输入都有一个参数（前面用过的简单线性模型一共有 51 个参数）。相反，对于简单 RNN 中每个循环神经元，每个输入每个隐藏状态只有一个参数（在简单 RNN 中，就是每层循环神经元的数量），加上一个偏置项。在这个简单 RNN 中，只有三个参数。

#### 趋势和季节性

还有其它预测时间序列的模型，比如权重移动平均模型或自动回归集成移动平均（ARIMA）模型。某些模型需要先移出趋势和季节性。例如，如果要研究网站的活跃用户数，它每月会增长 10%，就需要去掉这个趋势。训练好模型之后，在做预测时，你可以将趋势加回来做最终的预测。相似的，如果要预测防晒霜的每月销量，会观察到明显的季节性：每年夏天卖的多。需要将季节性从时间序列去除，比如计算每个时间步和前一年的差值（这个方法被称为差分）。然后，当训练好模型，做预测时，可以将季节性加回来，来得到最终结果。

使用 RNN 时，一般不需要做这些，但在有些任务中可以提高性能，因为模型不是非要学习这些趋势或季节性。

很显然，这个简单 RNN 过于简单了，性能不成。下面就来添加更多的循环层！

## 深度 RNN

将多个神经元的层堆起来，见图 15-7。就形成了深度 RNN。

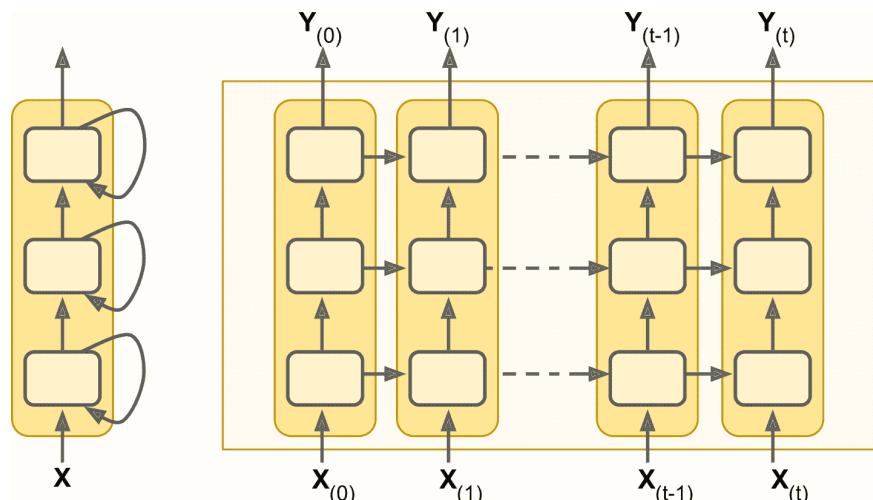


图 15-7 深度 RNN（左）和随时间展开的深度 RNN（右）

用 `tf.keras` 实现深度 RNN 相当容易：将循环层堆起来就成。在这个例子中，我们使用三个 `SimpleRNN` 层（也可以添加其它类型的循环层，比如 LSTM 或 GRU）：

```
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)
])
```

**警告：**所有循环层一定要设置 `return_sequences=True`（除了最后一层，因为最后一层只关心输出）。如果没有设置，输出的是 2D 数组（只有最终时间步的输出），而不是 3D 数组（包含所有时间步的输出），下一个循环层就接收不到 3D 格式的序列数据。

如果对这个模型做编译，训练和评估，其 MSE 值可以达到 0.003。总算打败了线性模型！

最后一层不够理想：因为要预测单一值，每个时间步只能有一个输出值，最终层只能有一个神经元。但是一个神经元意味着隐藏态只有一个值。RNN 大部分使用其他循环层的隐藏态的所有信息，最后一层的隐藏态不怎么用到。另外，因为 `SimpleRNN` 层默认使用 `tanh` 激活函数，预测值位于 -1 和 1 之间。想使用另一个激活函数该怎么办呢？出于这些原因，最好使用紧密层：运行更快，准确率差不多，可以选择任何激活函数。如果做了替换，要将第二个循环层的 `return_sequences=True` 删掉：

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])
```

如果训练这个模型，会发现它收敛更快，效果也不错。

## 提前预测几个时间步

目前为止我们只是预测下一个时间步的值，但也可以轻易地提前预测几步，只要改变目标就成（例如，要提前预测 10 步，只要将目标变为 10 步就成）。但如果想预测后面的 10 个值呢？

第一种方法是使用训练好的模型，预测出下一个值，然后将这个值添加到输入中（假设这个预测值真实发生了），使用这个模型再次预测下一个值，依次类推，见如下代码：

```
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:]
```

想象的到，第一个预测值比后面的更准，因为错误可能会累积（见图 15-8）。如果在验证集上评估这个方法，MSE 值为 0.029。MSE 比之前高多了，但因为任务本身难，这个对比意义不大。将其余朴素预测（预测时间序列可以恒定 10 个步骤）

或简单线性模型对比的意义更大。朴素方法效果很差（MSE 值为 0.223），线性简单模型的 MSE 值为 0.0188：比 RNN 的预测效果好，并且还快。如果只想在复杂任务上提前预测几步的话，这个方法就够了。

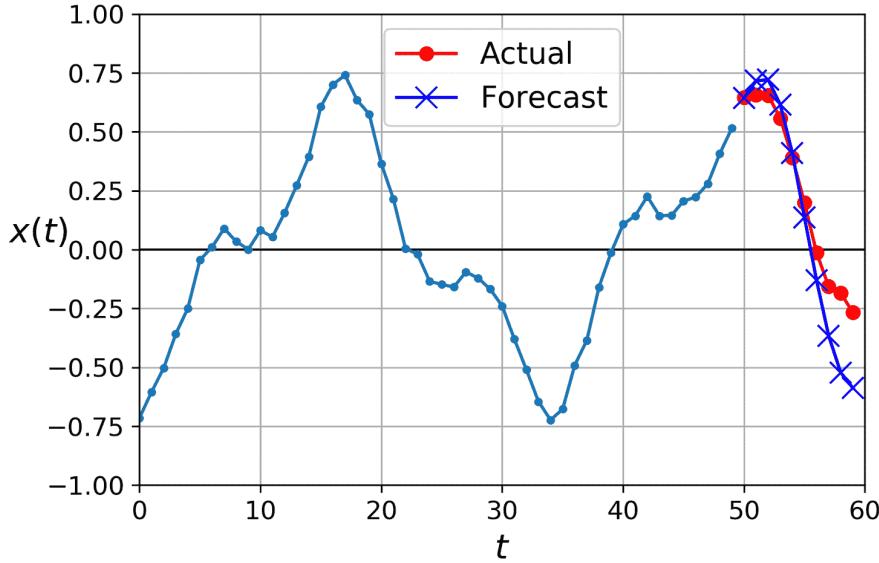


图 15-8 提前预测 10 步，每次 1 步

第二种方法是训练一个 RNN，一次性预测出 10 个值。还可以使用序列到向量模型，但输出的是 10 个值。但是，我们先需要修改向量，时期含有 10 个值：

```
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]
```

然后使输出层有 10 个神经元：

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])
```

训练好这个模型之后，就可以一次预测出后面的 10 个值了：

```
Y_pred = model.predict(X_new)
```

这个模型的效果不错：预测 10 个值的 MSE 值为 0.008。比线性模型强多了。但还有继续改善的空间，除了在最后的时间步用训练模型预测接下来的 10 个值，还可以在每个时间步预测接下来的 10 个值。换句话说，可以将这个序列到向量的 RNN 变成序列到序列的 RNN。这种方法的优势，是损失会包含 RNN 的每个时间步的输出项，不仅是最后时间步的输出。这意味着模型中会流动着更多的误差梯度，梯度不必只通过时间流动；还可以从输出流动。这样可以稳定和加速训练。

更加清楚一点，在时间步 0，模型输出一个包含时间步 1 到 10 的预测向量，在时间步 1，模型输出一个包含时间步 2 到 11 的预测向量，以此类推。因此每个目标必须是一个序列，其长度和输入序列长度相同，每个时间步包含一个 10 维向量。

先准备目标序列：

```

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:]

```

笔记：目标要包含出现在输入中的值（`x_train` 和 `Y_train` 有许多重复），听起来很奇怪。这不是作弊吗？其实不是：在每个时间步，模型只知道过去的时间步，不能向前看。这个模型被称为因果模型。

要将模型变成序列到序列的模型，必须给所有循环层（包括最后一个）设置 `return_sequences=True`，还必须在每个时间步添加紧密输出层。出于这个目的，Keras 提供了 `TimeDistributed` 层：它将任意层（比如，紧密层）包装起来，然后在输入序列的每个时间步上使用。通过变形输入，将每个时间步处理为独立实例（即，将输入从 [批次大小, 时间步数, 输入维度] 变形为 [批次大小 × 时间步数, 输入维度]；在这个例子中，因为前一 `SimpleRNN` 有 20 个神经元，输入的维度数是 20），这个层的效率很高。然后运行紧密层，最后将输出变形为序列（即，将输出从 [批次大小 × 时间步数, 输出维度] 变形为 [批次大小, 时间步数, 输出维度]；在这个例子中，输出维度数是 10，因为紧密层有 10 个神经元）。下面是更新后的模型：

```

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))
])

```

紧密层实际上是支持序列（和更高维度的输入）作为输入的：如同 `TimeDistributed(Dense(...))` 一样处理序列，意味着只应用在最后的输入维度上（所有时间步独立）。因此，因此可以将最后一层替换为 `Dense(10)`。但为了能够清晰，我们还是使用 `TimeDistributed(Dense(10))`，因为清楚的展示了紧密层独立应用在了每个时间上，并且模型会输出一个序列，不仅仅是一个单向量。

训练时需要所有输出，但预测和评估时，只需最后时间步的输出。因此尽管训练时依赖所有输出的 MSE，评估需要一个自定义指标，只计算最后一个时间步输出值的 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])

```

得到的 MSE 值为 0.006，比前面的模型提高了 25%。可以将这个方法和第一个结合起来：先用这个 RNN 预测接下来的 10 个值，然后将结果和输入序列连起来，再用模型预测接下来的 10 个值，以此类推。使用这个方法，可以预测任意长度的序列。对长期预测可能不那么准确，但用来生成音乐和文字是足够的，第 16 章有例子。

提示：当预测时间序列时，最好给预测加上误差条。要这么做，一个高效的方法是用 MC 丢弃，第 11 章介绍过：给每个记忆单元添加一个 MC 丢弃层丢失部分输入和隐藏状态。训练之后，要预测新的时间序列，可以多次使用模型计算每一步预测值的平均值和标准差。

简单 RNN 在预测时间序列或处理其它类型序列时表现很好，但在长序列上表现不佳。接下来就探究其原因和解决方法。

## 处理长序列

在训练长序列的 RNN 模型时，必须运行许多时间步，展开的 RNN 变成了一个很深的网络。正如任何深度神经网络一样，它面临不稳定梯度问题（第 11 章讨论过），使训练无法停止，或训练不稳定。另外，当 RNN 处理长序列时，RNN 会逐渐忘掉序列的第一个输入。下面就来看看这两个问题，先是第一个问题。

### 应对不稳定梯度

很多之前讨论过的缓解不稳定梯度的技巧都可以应用在 RNN 中：好的参数初始化方式，更快的优化器，丢弃，等等。但是非饱和激活函数（如 ReLU）的帮助不大；事实上，它会导致 RNN 更加不稳定。为什么呢？假设梯度下降更新了权重，可以令第一个时间步的输出提高。因为每个时间步使用的权重相同，第二个时间步的输出也会提高，这样就会导致输出爆炸——不饱和激活函数不能阻止这个问题。要降低爆炸风险，可以使用更小的学习率，更简单的方法是使用一个饱和激活函数，比如双曲正切函数（这就解释了为什么 tanh 是默认选项）。同样的道理，梯度本身也可能爆炸。如果观察到训练不稳定，可以监督梯度的大小（例如，使用 TensorBoard），看情况使用梯度裁剪。

另外，批归一化也没什么帮助。事实上，不能在时间步骤之间使用批归一化，只能在循环层之间使用。更加准确点，技术上可以将 BN 层添加到记忆单元上（后面会看到），这样就可以应用在每个时间步上了（既对输入使用，也对前一步的隐藏态使用）。但是，每个时间步用 BN 层相同，参数也相同，与输入和隐藏态的大小和偏移无关。在实践中，César Laurent 等人在 2015 年的一篇论文展示，这么做的效果不好：作者发现 BN 层只对输入有用，而对隐藏态没用。换句话说，在循环层之间使用 BN 层时，效果只有一点（即在图 15-7 中垂直使用），在循环层之内使用，效果不大（即，水平使用）。在 Keras 中，可以在每个循环层之前添加 BatchNormalization 层，但不要期待太高。

另一种归一化的形式效果好些：层归一化。它是由 Jimmy Lei Ba 等人在 2016 年的一篇论文中提出的：它跟批归一化很像，但不是在批次维度上做归一化，而是在特征维度上归一化。这么做的一个优势是可以独立对每个实例，实时计算所需的统计量。这还意味着训练和测试中的行为是一致的（这点和 BN 相反），且不需要使用指数移动平均来估计训练集中所有实例的特征统计。和 BN 一样，层归一化会学习每个输入的比例和偏移参数。在 RNN 中，层归一化通常用在输入和隐藏态的线型组合之后。

使用 `tf.keras` 在一个简单记忆单元中实现层归一化。要这么做，需要定义一个自定义记忆单元。就像一个常规层一样，`call()` 接收两个参数：当前时间步的 `inputs` 和上一时间步的隐藏 `states`。`states` 是一个包含一个或多个张量的列表。在简单 RNN 单元中，`states` 包含一个等于上一时间步输出的张量，但其它单元可能包含多个状态张量（比如 `LSTMCell` 有长期状态和短期状态）。单元还必须有一个 `state_size` 属性和一个 `output_size` 属性。在简单 RNN 中，这两个属性等于神经元的数量。下面的代码实现了一个自定义记忆单元，作用类似于 `SimpleRNNCell`，但会在每个时间步做层归一化：

```

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]

```

代码不难。和其它自定义类一样，`LNSimpleRNNCell` 继承自 `keras.layers.Layer`。构造器接收单元的数量、激活函数、设置 `state_size` 和 `output_size` 属性，创建一个没有激活函数的 `SimpleRNNCell`（因为要在线性运算之后、激活函数之前运行层归一化）。然后构造器创建 `LayerNormalization` 层，最终拿到激活函数。`call()` 方法先应用简单 RNN 单元，计算当前输入和上一隐藏态的线性组合，然后返回结果两次（事实上，在 `SimpleRNNCell` 中，输入等于隐藏状态：换句话说，`new_states[0]` 等于 `outputs`，因此可以放心地在剩下的 `call()` 中忽略 `new_states`）。然后，`call()` 应用层归一化，然后使用激活函数。最后，返回去输出两次（一次作为输出，一次作为新的隐藏态）。要使用这个自定义单元，需要做的是创建一个 `keras.layers.RNN` 层，传给其单元实例：

```

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))
])

```

相似地，可以创建一个自定义单元，在时间步之间应用丢弃。但有一个更简单的方法：Keras 提供的所有循环层（除了 `keras.layers.RNN`）和单元都有一个 `dropout` 超参数和一个 `recurrent_dropout` 超参数：前者定义丢弃率，应用到所有输入上（每个时间步），后者定义丢弃率，应用到隐藏态上（也是每个时间步）。无需在 RNN 中创建自定义单元来应用丢弃。

有了这些方法，就可以减轻不稳定梯度问题，高效训练 RNN 了。下面来看如何处理短期记忆问题。

## 处理短期记忆问题

由于数据在 RNN 中流动时会经历转换，每个时间步都损失了一定信息。一定时间后，第一个输入实际上会在 RNN 的状态中消失。就像一个搅局者。比如《寻找尼莫》中的多莉想翻译一个长句：当她读完这句话时，就把开头忘了。为了解决这个问题，涌现出了各种带有长期记忆的单元。首先了解一下最流行的一种：长短时记忆神经单元 LSTM。

## LSTM 单元

长短时记忆单元在 1997 年由 Sepp Hochreiter 和 Jürgen Schmidhuber 首次提出，并在接下来的几年内经过 Alex Graves、Hasim Sak、Wojciech Zaremba 等人的改进，逐渐完善。如果把 LSTM 单元看作一个黑盒，可以将其当做基本单元一样来使用，但 LSTM 单元比基本单元性能更好：收敛更快，能够感知数据的长时依赖。在 Keras 中，可以将 `SimpleRNN` 层，替换为 `LSTM` 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，可以使用通用的 `keras.layers.RNN layer`，设置 `LSTMCell` 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，当在 GPU 运行时，LSTM 层使用了优化的实现（见第 19 章），所以更应该使用 LSTM 层（`RNN` 大多用来自定义层）。

LSTM 单元的工作机制是什么呢？图 15-9 展示了 LSTM 单元的结构。

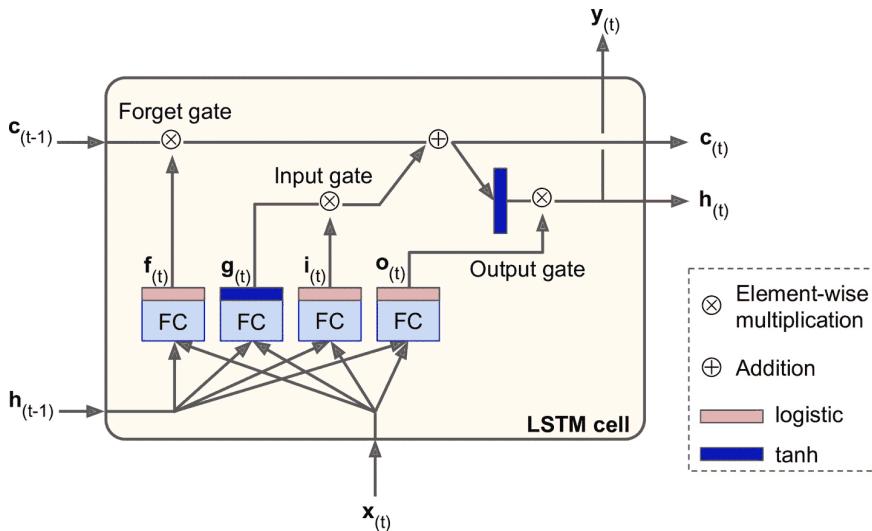


图 15-9 LSTM 单元

如果不观察黑箱的内部，LSTM 单元跟常规单元看起来差不多，除了 LSTM 单元的状态分成了两个向量： $h[t]$  和  $c[t]$ （ $c$  代表 cell）。可以认为  $h[t]$  是短期记忆状态， $c[t]$  是长期记忆状态。

现在打开黑箱。LSTM 单元的核心思想是它能从长期状态中学习该存储什么、丢掉什么、读取什么。当长期状态  $c[t-1]$  从左向右在网络中传播，它先经过遗忘门（forget gate），丢弃一些记忆，之后通过添加操作增加一些记忆（从输入门中选择一些记忆）。结果  $c[t]$  不经任何转换直接输出。因此，在每个时间步，都有一些记忆被抛弃，也有新的记忆添加进来。另外，添加操作之后，长时状态复制后经过 tanh 激活函数，然后结果被输出门过滤。得到短时状态  $h[t]$ （它等于这一时间步的单元输出， $y[t]$ ）。接下来讨论新的记忆如何产生，门是如何工作的。

首先，当前的输入向量  $x[t]$  和前一时刻的短时状态  $h[t-1]$  作为输入，传给四个不同的全连接层，这四个全连接层有不同的目的：

- 输出  $g[t]$  的层是主要层。它的常规任务是分析当前的输入  $x[t]$  和前一时刻的短时状态  $h[t-1]$ 。基本单元中与这种结构一样，直接输出了  $h[t]$  和  $y[t]$ 。相反的，LSTM 单元中的该层的输出不会直接出去，而是将最重要的部分保存在长期状态中（其余部分丢掉）。

- 其它三个全连接层被是门控制器（gate controller）。其采用 Logistic 作为激活函数，输出范围在 0 到 1 之间。可以看到，这三个层的输出提供了逐元素乘法操作，当输入为 0 时门关闭，输出为 1 时门打开。具体讲：
  - 遗忘门（由  $f[t]$  控制）决定哪些长期记忆需要被删除；
  - 输入门（由  $i[t]$  控制）决定哪部分  $g[t]$  应该被添加到长时状态中。
  - 输出门（由  $o[t]$  控制）决定长时状态的哪些部分要读取和输出为  $h[t]$  和  $y[t]$ 。

总而言之，LSTM 单元能够学习识别重要输入（输入门的作用），存储进长时状态，并保存必要的时间（遗忘门功能），并在需要时提取出来。这解释了为什么 LSTM 单元能够如此成功地获取时间序列、长文本、录音等数据中的长期模式。

公式 15-3 总结了如何计算单元的长时状态，短时状态，和单个实例的在每个时间步的输出（小批次的公式和这个公式很像）。

$$\begin{aligned}
 i_{(t)} &= \sigma(\mathbf{W}_{xi}^T \mathbf{x}_{(t)} + \mathbf{W}_{hi}^T \mathbf{h}_{(t-1)} + \mathbf{b}_i) \\
 f_{(t)} &= \sigma(\mathbf{W}_{xf}^T \mathbf{x}_{(t)} + \mathbf{W}_{hf}^T \mathbf{h}_{(t-1)} + \mathbf{b}_f) \\
 o_{(t)} &= \sigma(\mathbf{W}_{xo}^T \mathbf{x}_{(t)} + \mathbf{W}_{ho}^T \mathbf{h}_{(t-1)} + \mathbf{b}_o) \\
 g_{(t)} &= \tanh(\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \mathbf{h}_{(t-1)} + \mathbf{b}_g) \\
 c_{(t)} &= f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \\
 y_{(t)} &= h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)})
 \end{aligned}$$

### 公式 15-3 LSTM 计算

在这个公式中，

- $\mathbf{W}_{xi}$ ， $\mathbf{W}_{xf}$ ， $\mathbf{W}_{xo}$ ， $\mathbf{W}_{xg}$  是四个全连接层连接输入向量  $x[t]$  的权重。
- $\mathbf{W}_{hi}$ ， $\mathbf{W}_{hf}$ ， $\mathbf{W}_{ho}$ ， $\mathbf{W}_{hg}$  是四个全连接层连接上一时刻的短时状态  $h[t - 1]$  的权重。
- $\mathbf{b}_i$ ， $\mathbf{b}_f$ ， $\mathbf{b}_o$ ， $\mathbf{b}_g$  是全连接层的四个偏置项。需要注意的是 TensorFlow 将  $\mathbf{b}_f$  初始化为全 1 向量，而非全 0。这样可以保证在训练状态开始时，忘掉所有东西。

## 窥孔连接

在基本 LSTM 单元中，门控制器只能观察当前输入  $x[t]$  和前一时刻的短时状态  $h[t - 1]$ 。不妨让各个门控制器窥视一下长时状态，获取一些上下文信息。[该想法](#)由 Felix Gers 和 Jürgen Schmidhuber 在 2000 年提出。他们提出了一个 LSTM 的变体，带有叫做窥孔连接的额外连接：把前一时刻的长时状态  $c[t - 1]$  输入给遗忘门和输入门，当前时刻的长时状态  $c[t]$  输入给输出门。这么做时常可以提高性能，但不一定每次都能有效，也没有清晰的规律显示哪种任务适合添加窥孔连接。

Keras 中，LSTM 层基于 `keras.layers.LSTMCell` 单元，后者目前还不支持窥孔。但是，试验性的 `tf.keras.experimental.PeepholeLSTMCell` 支持，所以可以创建一个 `keras.layers.RNN` 层，向构造器传入 `PeepholeLSTMCell`。

LSTM 有多种其它变体，其中特别流行的是 GRU 单元。

## GRU 单元

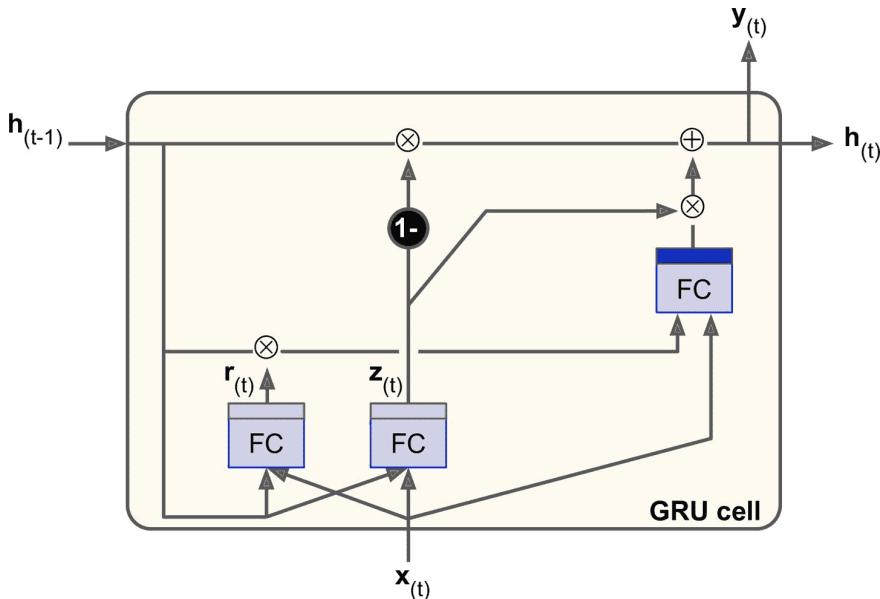


图 15-10 GRU 单元

门控循环单元（图 15-10）在 2014 年的 Kyunghyun Cho 的论文中提出，并且此文也引入了前文所述的编码器-解码器网络。

GRU 单元是 LSTM 单元的简化版本，能实现同样的性能（这也说明了为什么它能越来越流行）。简化主要在以下几个方面：

- 长时状态和短时状态合并为一个向量  $h[t]$ 。
- 用一个门控制器  $z[t]$  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，如果当有记忆要存储，那么就必须先在其存储位置删掉该处记忆。这构成了 LSTM 本身的常见变体。
- GRU 单元取消了输出门，每个时间步输出全态向量。但是，增加了一个控制门  $r[t]$  来控制前一状态的哪些部分呈现给主层  $g[t]$ 。

公式 15-4 总结了如何计算单元对单个实例在每个时间步的状态。

$$\begin{aligned} z_{(t)} &= \sigma(\mathbf{W}_{xz}^T \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \mathbf{h}_{(t-1)} + \mathbf{b}_z) \\ r_{(t)} &= \sigma(\mathbf{W}_{xr}^T \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \mathbf{h}_{(t-1)} + \mathbf{b}_r) \\ g_{(t)} &= \tanh (\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T (r_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g) \\ \mathbf{h}_{(t)} &= z_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - z_{(t)}) \otimes g_{(t)} \end{aligned}$$

公式 15-4 GRU 计算

Keras 提供了 `keras.layers.GRU` 层（基于 `keras.layers.GRUCell` 记忆单元）；使用时，只需将 `SimpleRNN` 或 `LSTM` 替换为 `GRU`。

`LSTM` 和 `GRU` 是 `RNN` 取得成功的主要原因之一。尽管它们相比于简单 `RNN` 可以处理更长的序列了，还是有一定程度的短时记忆，序列超过 100 时，比如音频、长时间序列或长序列，学习长时模式就很困难。应对的方法之一，是使用缩短输入序列，例如使用 1D 卷积层。

## 使用 1D 卷积层处理序列

在第 14 章中，我们使用 2D 卷积层，通过在图片上滑动几个小核（或过滤器），来产生多个 2D 特征映射（每个核产生一个）。相似的，1D 卷积层在序列上滑动几个核，每个核可以产生一个 1D 特征映射。每个核能学到一个非常短序列模式（不会超过核的大小）。如果你是用 10 个核，则输出会包括 10 个 1 维的序列（长度相同），或者可以将输出当做一个 10 维的序列。这意味着，可以搭建一个由循环层和 1D 卷积层（或 1 维池化层）混合组成的神经网络。如果 1D 卷积层的步长是 1，填充为零，则输出序列的长度和输入序列相同。但如果使用 “`valid`” 填充，或大于 1 的步长，则输出序列会比输入序列短，所以一定要按照目标作出调整。例如，下面的模型和之前的一样，除了开头是一个步长为 2 的 1D 卷积层，用因子 2 对输入序列降采样。核大小比步长大，所以所有输入会用来计算层的输出，所以模型可以学到保存有用的信息、丢弃不重要信息。通过缩短序列，卷积层可以帮助 `GRU` 检测长模式。注意，必须裁剪目标中的前三个时间步（因为核大小是 4，卷积层的第一个输出是基于输入时间步 0 到 3），并用因子 2 对目标做降采样：

```
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]))
```

如果训练并评估这个模型，你会发现它是目前最好的模型。卷积层确实发挥了作用。事实上，可以只使用 1D 卷积层，不用循环层！

## WaveNet

在一篇 2016 年的[论文](#)中，Aaron van den Oord 和其它 DeepMind 的研究者，提出了一个名为 WaveNet 的架构。他们将 1D 卷积层叠起来，每一层膨胀率（如何将每个神经元的输入分开）变为 2 倍：第一个卷积层一次只观察两个时间步，，接下来的一层观察四个时间步（感受野是 4 个时间步的长度），下一层观察八个时间步，以此类推（见图 15-11）。用这种方式，底下的层学习短时模式，上面的层学习长时模式。得益于翻倍的膨胀率，这个网络可以非常高效地处理极长的序列。

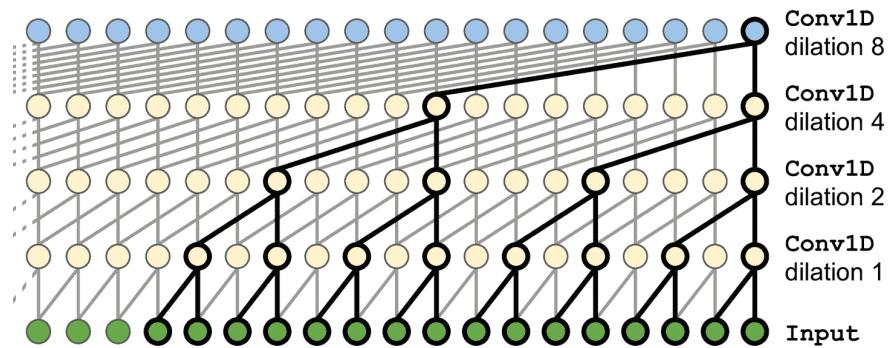


图 15-11 WaveNet 架构

在 WaveNet 论文中，作者叠了 10 个卷积层，膨胀率为 1, 2, 4, 8, ..., 256, 512, 然后又叠了一组 10 个相同的层（膨胀率还是 1, 2, 4, 8, ..., 256, 512）, 然后又是 10 个相同的层。作者解释到，一摞这样的 10 个卷积层，就像一个超高效的核大小为 1024 的卷积层（只是更快、更强、参数更少），所以同样的结构叠了三次。他们还给输入序列左填充了一些 0，以满足每层的膨胀率，使序列长度不变。下面的代码实现了简化的 WaveNet，来处理前面的序列：

```
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))
```

Sequential 模型开头是一个输入层（比只在第一个层上设定 `input_shape` 简单的多）；然后是一个 1D 卷积层，使用 "causal" 填充：这可以保证卷积层在做预测时，不会窥视到未来值（等价于在输入序列的左边用零填充填充合适数量的 0）。然后添加相似的成对的层，膨胀率为 1、2、4、8，接着又是 1、2、4、8。最后，添加输出层：一个有 10 个大小为 1 的过滤器的卷积层，没有激活函数。得益于填充层，每个卷积层输出的序列长度都和输入序列一样，所以训练时的目标可以是完整序列：无需裁剪或降采样。

最后两个模型的序列预测结果最好！在 WaveNet 论文中，作者在多种音频任务（WaveNet 名字正是源于此）中，包括文本转语音任务（可以输出多种语言极为真实的语音），达到了顶尖的表现。他们还用这个模型生成音乐，每次生成一段音频。每段音频包含上万个时间步（LSTM 和 GRU 无法处理如此长的序列），这是相当了不起的。

第 16 章，我们会继续探索 RNN，会看到如何用 RNN 处理各种 NLP 任务。

## 练习

1. 你能说出序列到序列 RNN 的几个应用吗？序列到向量的应用？向量到序列的应用？
2. RNN 层的输入要有多少维？每一维表示什么？输出呢？
3. 如果搭建深度序列到序列 RNN，哪些 RNN 层要设置 `return_sequences=True`？序列到向量 RNN 又如何？

4. 假如有一个每日单变量时间序列，想预测接下来的七天。要使用什么 RNN 架构？
5. 训练 RNN 的困难是什么？如何应对？
6. 画出 LSTM 单元的架构图？
7. 为什么在 RNN 中使用 1D 卷积层？
8. 哪种神经网络架构可以用来分类视频？
9. 为 SketchRNN 数据集（TensorFlow Datasets 中有），训练一个分类模型。
10. 下载 [Bach chorales](#) 数据集，并解压。它含有 382 首巴赫作曲的赞美歌。每首的长度是 100 到 640 时间步，每个时间步包含 4 个整数，每个整数对应一个钢琴音符索引（除了 0，表示没有音符）。训练一个可以预测下一个时间步（四个音符）的模型，循环、卷积、或混合架构。然后使用这个模型来生成类似巴赫的音乐，每个时间一个音符：可以给模型一首赞美歌的开头，然后让其预测接下来的时间步，然后将输出加到输入上，再让模型继续预测。或者查看 [Google 的 Coconet 模型](#)，它是 Google 来做巴赫曲子的。

参考答案见附录 A。

## 十六、使用 RNN 和注意力机制进行自然语言处理

译者：[@SeanCheney](#)

当阿兰·图灵在 1950 年设计[图灵机](#)时，他的目标是用人的智商来衡量机器。他本可以用其它方法来测试，比如看图识猫、下棋、作曲或逃离迷宫，但图灵选择了一个语言任务。更具体的，他设计了一个聊天机器人，试图迷惑对话者将其当做真人。这个测试有明显的缺陷：一套硬编码的规则可以愚弄粗心人（比如，机器可以针对一些关键词，做出预先定义的模糊响应；机器人可以假装开玩笑或喝醉；或者可以通过反问侥幸过关），忽略了人类的多方面的智力（比如非语言交流，比如面部表情，或是学习动手任务）。但图灵测试强调了一个事实，语言能力是智人最重要的认知能力。我们能创建一台可以读写自然语言的机器吗？

自然语言处理的常用方法是循环神经网络。所以接下来会从字符 RNN 开始（预测句子中出现的下一个字符），继续介绍 RNN，这可以让我们生成一些原生文本，在过程中，我们会学习如何在长序列上创建 TensorFlow Dataset。先使用的是无状态 RNN（每次迭代中学习文本中的随机部分），然后创建一个有状态 RNN（保留训练迭代之间的隐藏态，可以从断点继续，用这种方法学习长规律）。然后，我们会搭建一个 RNN，来做情感分析（例如，读取影评，提取评价者对电影的感情），这次是将句子当做词的序列来处理。然后会介绍用 RNN 如何搭建编码器-解码器架构，来做神经网络机器翻译（NMT）。我们会使用 TensorFlow Addons 项目中的 seq2seq API。

本章的第二部分，会介绍注意力机制。正如其名字，这是一种可以选择输入指定部分，模型在每个时间步都得聚焦的神经网络组件。首先，会介绍如何使用注意力机制提升基于 RNN 的编码器-解码器架构的性能，然后会完全摒弃 RNN，介绍只使用注意力的架构，被称为 Transformer（转换器）。最后，会介绍 2018、2019 两年 NLP 领域的进展，包括强大的语言模型，比如 GPT-2 和 Bert，两者都是基于 Transformer 的。

先从一个简单有趣的模型开始，它能写出莎士比亚风格的文字。

### 使用 Character RNN 生成莎士比亚风格的文本

在 2015 年一篇著名的、名为《The Unreasonable Effectiveness of Recurrent Neural Networks》博客中，Andrej Karpathy 展示了如何训练 RNN，来预测句子中的下一个字符。这个 Char-RNN 可以用来生成小说，每次一个字符。下面是一段简短的、由 Char-RNN 模型（在莎士比亚全部著作上训练而成）生成的文本：

```
PANDARUS:  
Alas, I think he shall be come approached and the day  
When little strain would be attain'd into being never fed,  
And who is but a chain and subjects of his death,  
I should not sleep.
```

虽然文笔一般，但只是通过学习来预测一句话中的下一个字符，模型在单词、语法、断句等等方面做的很好。接下来一步一步搭建 Char-RNN，从创建数据集开始。

## 创建训练数据集

首先，使用 Keras 的 `get_file()` 函数，从 Andrej Karpathy 的 [Char-RNN 项目](#)，下载所有莎士比亚的作品：

```
shakespeare_url = "https://homl.info/shakespeare" # shortcut URL
filepath = keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

然后，将每个字符编码为一个整数。方法之一是创建一个自定义预处理层，就像之前在第 13 章做的那样。但在这里，使用 Keras 的 `Tokenizer` 会更加简单。首先，将一个将分词器拟合到文本：分词器能从文本中发现所有的字符，并将所有字符映射到不同的字符 ID，映射从 1 开始（注意不是从 0 开始，0 是用来做遮挡的，后面会看到）：

```
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)
tokenizer.fit_on_texts([shakespeare_text])
```

设置 `char_level=True`，以得到字符级别的编码，而不是默认的单词级别的编码。这个分词器默认将所有文本转换成了小写（如果不想这样，可以设置 `lower=False`）。现在分词器可以将一整句（或句子列表）编码为字符 ID 列表，这可以告诉我们文本中有多少个独立的字符，以及总字符数：

```
>>> 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
```

现在对完整文本做编码，将每个字符都用 ID 来表示（减 1 使 ID 从 0 到 38，而不是 1 到 39）：

```
[encoded] = np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1
```

继续之前，需要将数据集分成训练集、验证集和测试集。不能大论字符，该怎么处理这种序列式的数据集呢？

## 如何切分序列数据集

避免训练集、验证集、测试集发生重合非常重要。例如，可以取 90% 的文本作为训练集，5% 作为验证集，5% 作为测试集。在这三个数据之间留出空隙，以避免段落重叠也是非常好的主意。

当处理时间序列时，通常按照时间切分：例如，可以将从 2000 到 2012 的数据作为训练集，2013 年到 2015 年作为验证集，2016 年到 2018 年作为测试集。但是，在另一些任务中，可以按照其它维度来切分，可以得到更长的时间周期进行训练。例如，10000 家公司从 2000 年到 2018 年的金融健康数据，可以按照不同公司来切分。但是，很可能其中一些公司是高度关联的（比如，经济领域的公司涨落相同），如果训练集和测试集中有关联的公司，则测试集的意义就不大，泛化误差会存在偏移。

因此，在时间维度上切分更加安全 —— 但这实际是默认 RNN 可以（在训练集）从过去学到的规律也适用于将来。换句话说，我们假设时间序列是静态的（至少是在一个较宽的区间内）。对于时间序列，这个假设是合理的（比如，化学反应就是这样，化学定理不会每天发生改变），但其它的就不是（例如，金融市场就不是静态的，一旦交易员发现规律并从中牟利，规律就会改变）。要保证时间序列确实是静态的，可以在验证集上画出模型随时间的误差：如果模型在验证集的前端表现优于后段，则时间序列可能就不够静态，最好是在一个更短的时间区间内训练。

总而言之，将时间序列切分成训练集、验证集和测试集不是简单的工作，怎么做要取决于具体的任务。

回到莎士比亚！这里将前 90% 的文本作为训练集（剩下的作为验证集和测试集），创建一个 `tf.data.Dataset`，可以从这个集和一个个返回每个字符：

```
train_size = dataset_size * 90 // 100
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
```

## 将序列数据集切分成多个窗口

现在训练集包含一个单独的长序列，超过 100 万的任务，所以不能直接在这个训练集上训练神经网络：现在的 RNN 等同于一个有 100 万层的深度网络，只有一个超长的单实例来训练。所以，得使用数据集的 `window()` 方法，将这个长序列转化为许多小窗口文本。每个实例都是完整文本的相对短的子字符串，RNN 只在这些子字符串上展开。这被称为截断沿时间反向传播。调用 `window()` 方法创建一个短文本窗口的数据集：

```
n_steps = 100
window_length = n_steps + 1 # target = input 向前移动 1 个字符
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
```

**提示：**可以调节 `n_steps`：用短输入序列训练 RNN 更为简单，但肯定的是 RNN 学不到任何长度超过 `n_steps` 的规律，所以 `n_steps` 不要太短。

默认情况下，`window()` 方法创建的窗口是不重叠的，但为了获得可能的最大训练集，我们设定 `shift=1`，好让第一个窗口包含字符 0 到 100，第二个窗口包含字符 1 到 101，等等。为了确保所有窗口是准确的 101 个字符长度（为了不做填充而创建批次），设置 `drop_remainder=True`（否则，最后的 100 个窗口会包含 100 个字符、99 个字符，一直到 1 个字符）。

`window()` 方法创建了一个包含窗口的数据集，每个窗口也是数据集。这是一个嵌套的数据集，类似于列表的列表。当调用数据集方法处理（比如、打散或做批次）每个窗口时，这样会很方便。但是，不能直接使用嵌套数据集来训练，因为模型要的输入是张量，不是数据集。因此，必须调用 `flat_map()` 方法：它能将嵌套数据集转换成打平的数据集。例如，假设 `{1, 2, 3}` 表示包含张量 1、2、3 的序列。如果将嵌套数据集 `[[1, 2], [3, 4, 5, 6]]` 打平，就会得到 `[1, 2, 3, 4, 5, 6]`。另外，`flat_map()` 方法可以接收函数作为参数，可以处理嵌套数据集的每个数据集。例如，如果将函数 `lambda ds: ds.batch(2)` 传递给 `flat_map()`，它能将 `[[1, 2], [3, 4, 5, 6]]` 转变为 `[[[1, 2], [3, 4], [5, 6]]]`：这是一个张量大小为 2 的数据集。

有了这些知识，就可以打平数据集了：

```
dataset = dataset.flat_map(lambda window: window.batch(window_length))
```

我们在每个窗口上调用了 `batch(window_length)`：因为所有窗口都是这个长度，对于每个窗口，都能得到一个独立的张量。现在的数据集包含连续的窗口，每个有 101 个字符。因为梯度下降在训练集中的实例独立同分布时的效果最好，需要打散这些窗口。然后我们可以对窗口做批次，分割输入（前 100 个字符）和目标（最后一个字符）：

```
batch_size = 32
dataset = dataset.shuffle(10000).batch(batch_size)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))


```

图 16-1 总结了数据集准备步骤（窗口长度是 11，不是 101，批次大小是 3，不是 32）。

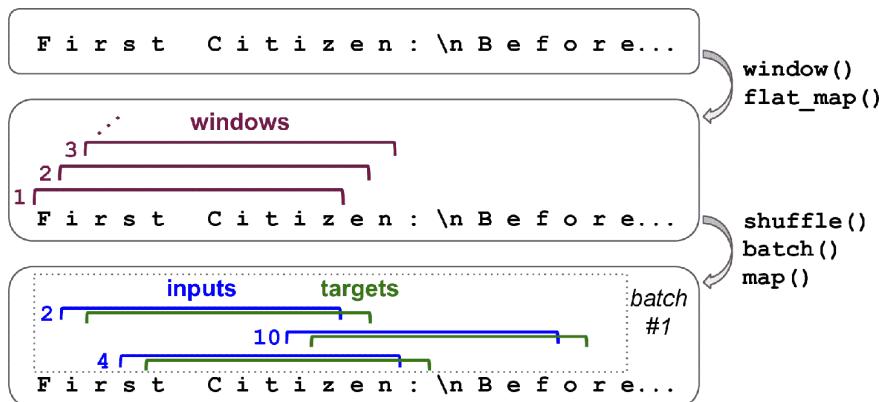


图 16-1 准备打散窗口的数据集

第 13 章讨论过，类型输入特征通常都要编码，一般是独热编码或嵌入。这里，使用独热编码，因为独立字符不多（只有 39）：

```
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
```

最后，加上预提取：

```
dataset = dataset.prefetch(1)
```

就是这样！准备数据集是最麻烦的部分。下面开始搭建模型。

## 搭建并训练 Char-RNN 模型

根据前面的 100 个字符预测下一个字符，可以使用一个 RNN，含有两个 GRU 层，每个 128 个单元，每个单元对输入（`dropout`）和隐藏态（`recurrent_dropout`）的丢弃率是 20%。如果需要的话，后面可以微调这些超参数。输出层是一个时间分布的紧密层，有 39 个单元（`max_id`），因为文本中有 39 个不同的字符，需要输出每个可能字符（在每个时间步）的概率。输出概率之后应为 1，所以使用 `softmax` 激活很熟。然后可以使⽤ "sparse\_categorical\_crossentropy" 损失和 Adam 优化器，编译模型。最后，就可以训练模型几个周期了（训练过程可能要几个小时，取决于硬件）：

```

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)

```

## 使用 Char-RNN 模型

现在就有了可以预测莎士比亚要写的下一个人物的模型了。输入数据之前，先要像之前那样做预处理，因此写个小函数来做预处理：

```

def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)

```

现在，用这个模型预测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)[-1] # 1st sentence, last char
'u'

```

预测成功！接下来用这个模型生成文本。

## 生成假莎士比亚文本

要使用 Char-RNN 生成新文本，我们可以给模型输入一些文本，让模型预测出下一个字母，将字母添加到文本的尾部，再将延长后的文本输入给模型，预测下一个字母，以此类推。但在实际中，这会导致相同的单词不断重复。相反的，可以用 `tf.random.categorical()` 函数，随机挑选下一个字符，概率等同于估计概率。这样就能生成一些多样且有趣的文本。根据类的对数概率

`(logits)`，`categorical()` 函数随机从类索引采样。为了对生成文本的多样性更可控，我们可以用一个称为“温度”的可调节的数来除以对数概率：温度接近 0，会利于高概率字符，而高温度会是所有字符概率相近。下面的 `next_char()` 函数使用这个方法，来挑选添加进文本中的字符：

```

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]

```

然后，可以写一个小函数，重复调用 `next_char()`：

```

def complete_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text

```

现在就可以生成一些文本了！先尝试下不同的温度数：

```
>>> 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，当温度数接近 1 时，我们的莎士比亚模型效果最好。为了生成更有信服力的文字，可以尝试用更多 GRU 层、每层更多的神经元、更长的训练时间，添加正则（例如，可以在 GRU 层中设置 `recurrent_dropout=0.3`）。另外，模型不能学习长度超过 `n_steps`（只有 100 个字符）的规律。你可以使用更大的窗口，但也会让训练更为困难，甚至 LSTM 和 GRU 单元也不能处理长序列。另外，还可以使用有状态 RNN。

## 有状态 RNN

到目前为止，我们只使用了无状态 RNN：在每个训练迭代中，模型从全是 0 的隐藏状态开始训练，然后在每个时间步更新其状态，在最后一个时间步，隐藏态就被丢掉，以后再也不用了。如果让 RNN 保留这个状态，供下一个训练批次使用如何呢？这么做的话，尽管反向传播只在短序列传播，模型也可以学到长时规律。这被称为有状态 RNN。

首先，有状态 RNN 只在前一批次的序列离开，后一批次中的对应输入序列开始的情况下才有意义。所以第一件要做的事情是使用序列且没有重叠的输入序列（而不是用来训练无状态 RNN 时的打散和重叠的序列）。当创建 `Dataset` 时，调用 `window()` 必须使用 `shift=n_steps`（而不是 `shift=1`）。另外，不能使用 `shuffle()` 方法。但是，准备有状态 RNN 数据集的批次会麻烦些。事实上，如果调用 `batch(32)`，32 个连续的窗口会放到一个相同的批次中，后面的批次不会接着这些窗口。第一个批次含有窗口 1 到 32，第二个批次含有窗口 33 到 64，因此每个批次中的第一个窗口（窗口 1 和 33），它们是不连续的。最简单办法是使用只包含一个窗口的“批次”：

```
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
dataset = dataset.window(window_length, shift=n_steps, drop_remainder=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 展示了处理的第一步。

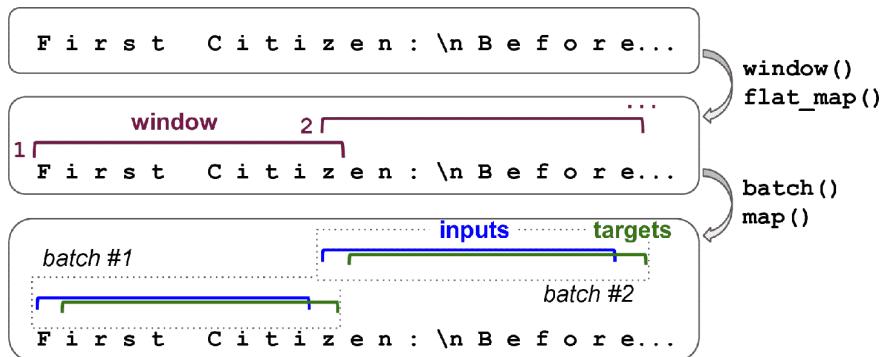


图 16-2 为有状态 RNN 准备连续序列片段的数据集

做批次虽然麻烦，但可以实现。例如，我们可以将莎士比亚作品切分成 32 段等长的文本，每个做成一个连续序列的数据集，最后使用 `tf.train.Dataset.zip(datasets).map(lambda *windows: tf.stack(windows))` 来创建合适的连续批次，批次中的 `n` 输入序列紧跟着 `n` 结束的地方（笔记本中有完整代码）。

现在创建有状态 RNN。首先，创建每个循环层时需要设置 `stateful=True`。第二，有状态 RNN 需要知道批次大小（因为要为批次中的输入序列保存状态），所以要在第一层中设置 `batch_input_shape` 参数。不用指定第二个维度，因为不限制序列的长度：

```
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"))
])
```

在每个周期之后，回到文本开头之前，需要重设状态。要这么做，可以使用一个小回调：

```
class ResetStatesCallback(keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

现在可以编译、训练模型了（周期数更多，是因为每个周期比之前变短了，每个批次只有一个实例）：

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="Adam")
model.fit(dataset, epochs=50, callbacks=[ResetStatesCallback()])
```

提示：训练好模型之后，只能预测训练时相同大小的批次。为了避免这个限制，可以创建一个相同的无状态模型，将有状态模型的参数复制到里面。

创建了一个字符层面的模型，接下来看看词层面的模型，并做一个常见的自然语言处理任务：情感分析。我们会学习使用遮掩来处理变化长度的序列。

## 情感分析

如果说 MNIST 是计算机视觉的“hello world”，那么 IMDb 影评数据集就是自然语言处理的“hello world”：这个数据集包含 50000 条英文影评，25000 条用于训练，25000 条用于测试，是从 IMDb 网站提取的，并带有影评标签，负 (0) 或正 (1)。和 MNIST 一样，IMDb 影评数据集的流行是有原因的：笔记本电脑上就可以跑起来，不会耗时太长，也具有一定挑战。Keras 提供了一个简单的函数加载数据集：

```
>>> (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]
```

影评在哪里？可以看到，数据集已经经过预处理了：`X_train` 包括列表形式的影评，每条都是整数 NumPy 数组，每个整数代表一个词。所有标点符号都被去掉了，单词转换为小写，用空格隔开，最后用频次建立索引（小整数对应常见词）。

整数 0、1、2 是特殊的：它们表示填充标记、序列开始（SSS）标记、和未知单词。如果想看到影评，可以如下解码：

```
>>> 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
...
'<sos> this film was just brilliant casting location scenery story'
```

在真实的项目中，必须要自己预处理文本。你可以使用前面用过的 `Tokenizer`，但要设置 `char_level=False`（其实是默认的）。当编码单词时，`Tokenizer` 会过滤掉许多字符，包括多数标点符号、换行符、制表符（可以通过 `filters` 参数控制）。最重要的，`Tokenizer` 使用空格确定单词的边界。这对于英语和其它用空格隔开单词的语言是行得通的，但并不是所有语言都有空格。中文不使用空格，越南语甚至在单词里也有空格，德语经常将几个单词不用空格连在一起。就算在英语中，空格也不总是标记文本的最好方法：比如 San Francisco 或 #ILoveDeepLearning。

幸好，有更好的方法。Taku Kudo 在 [2018 年的一篇论文](#) 中介绍了一种无监督学习方法，在亚词层面分词和取消分词文本，与所属语言独立，空格和其它字符等同处理。使用这种方法，就算模型碰到一个之前没见过的单词，模型还是能猜出它的意思。例如，模型在训练期间没见过单词 `smartest`，但学过 `est` 词尾是最的意思，然后就可以推断 `smartest` 的意思。Google 的 [SentencePiece](#) 项目提供了开源实现，见 Taku Kudo 和 John Richardson 的[论文](#)。

另一种方法，是 Rico Sennrich 在更早的[论文](#)中提出的，探索了其它创建亚单词编码的方法（比如，使用字节对编码）。最后同样重要的，TensorFlow 团队在 2019 年提出了 [TF.Text](#) 库，它实现了多种分词策略，包括 [WordPiece](#)（字节对编码的变种）。

如果你想将模型部署到移动设备或网页中，又不想每次都写一个不同的预处理函数，最好只使用 TensorFlow 运算，它可以融进模型中。看看怎么做。首先，使用 TensorFlow Datasets 加载原始 IMDb 评论，为文本（字节串）：

```
import tensorflow_datasets as tfds
datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
```

然后，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
```

预处理函数先裁剪影评，只保留前 300 个字符：这么做可以加速训练，并且不会过多影响性能，因为大多数时候只要看前一两句话，就能判断是正面或侧面的了。然后使用正则表达式替换 `<br />` 标签为空格，然后将所有非字母字符替换为空格。例如，文本 "Well, I can't<br />" 变成 "Well I can't"。最后，`preprocess()` 函数用空格分隔影评，返回一个嵌套张量，然后将嵌套张量转变为紧密张量，给所有影评填充上 "`<pad>`"，使其长度相等。

然后，构建词典。这需要使用 `preprocess()` 函数再次处理训练集，并使用 `Counter` 统计每个单词的出现次数：

```
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()))
```

看看最常见的词有哪些：

```
>>> vocabulary.most_common()[:3]
[(b'<pad>', 215797), (b'the', 61137), (b'a', 38564)]
```

但是，并不需要让模型知道词典中的所有词，所以裁剪词典，只保留 10000 个最常见的词：

```
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]
```

现在需要加上预处理步骤将每个单词替换为单词 ID（即它在词典中的索引）。就像第 13 章那样，创建一张查找表，使用 1000 个未登录词（oov）桶：

```
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)
```

用这个词表查找几个单词的 ID：

```
>>> table.lookup(tf.constant([b"This movie was faaaaaantastic".split()]))
<tf.Tensor: [...], dtype=int64, numpy=array([[ 22,   12,   11, 10054]])>
```

因为 `this`、`movie`、`was` 是在词表中的，所以它们的 ID 小于 10000，而 `faaaaaantastic` 不在词表中，所以将其映射到一个 oov 桶，其 ID 大于或等于 10000。

**提示**：TF Transform 提供了一些实用的函数来处理词典。例如，`tft.compute_and_apply_vocabulary()` 函数：它可以遍历数据集，找到所有不同的词，创建词典，还能生成 TF 运算，利用词典编码每个单词。

现在，可以创建最终的训练集。对影评做批次，使用 `preprocess()` 将其转换为词的短序列，然后使用一个简单的 `encode_words()` 函数，利用创建的词表来编码这些词，最后预提取下一个批次：

```
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，创建模型并训练：

```

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)

```

第一个层是一个嵌入层，它将所有单词 ID 变为嵌入。每有一个单词 ID (`vocab_size + num_oov_buckets`)，嵌入矩阵就有一行，每有一个嵌入维度，嵌入矩阵就有一列（这个例子使用了 128 个维度，这是一个可调的超参数）。模型输入是 2D 张量，形状为 [批次大小, 时间步]，嵌入层的输出是一个 3D 张量，形状为 [批次大小, 时间步, 嵌入大小]。

模型剩下的部分就很简单了：有两个 GRU 层，第二个只返回最后时间步的输出。输出层只有一个神经元，使用 sigmoid 激活函数，输出评论是正或负的概率。然后编译模型，利用前面准备的数据集来训练几个周期。

## 遮掩

在训练过程中，模型会学习到填充标记要被忽略掉。但这其实是已知的。为什么不告诉模型直接忽略填充标记，将精力集中在真正重要的数据中呢？只需一步就好：创建嵌入层时加上 `mask_zero=True`。这意味着填充标记（其 ID 为 0）可以被接下来的所有层忽略。

其中的原理，是嵌入层创建了一个等于 `K.not_equal(inputs, 0)`（其中 `K = keras.backend`）遮掩张量：这是一个布尔张量，形状和输入相同，只要词 ID 有 0，它就等于 `False`，否则为 `True`。模型自动将这个遮掩张量向前传递给所有层，只要时间维度保留着。所以在例子中，尽管两个 GRU 都接收到了遮掩张量，但第二个 GRU 层不返回序列（只返回最后一个时间步），遮掩张量不会传递到紧密层。每个层处理遮掩的方式不同，但通常会忽略被遮掩的时间步（遮掩为 `False` 的时间步）。例如，当循环神经层碰到被遮掩的时间步时，就只是从前一时间步复制输出而已。如果遮掩张量一直传递到输出（输出为序列的模型），则遮掩也会作用到损失上，所以遮掩时间步不会贡献到损失上（它们的损失为 0）。

警告：基于英伟达的 cuDNN 库，LSTM 层和 GRU 层针对 GPU 有优化实现。但是，这个实现不支持遮挡。如果你的模型使用了遮挡，则这些曾会回滚到（更慢的）默认实现。注意优化实现还需要使用几个超参数的默认值：`activation`、`recurrent_activation`、`recurrent_dropout`、`unroll`、`use_bias`、`reset_after`。

所有接收遮挡的层必须支持遮挡（否则会抛出异常）。包括所有的循环层、`TimeDistributed` 层和其它层。所有支持遮挡的层必须有等于 `True` 的属性 `supports_masking`。如果想实现自定义的支持遮挡的层，应该给 `call()` 方法添加 `mask` 参数。另外，要在构造器中设定 `self.supports_masking = True`。如果第一个层不是嵌入层，可以使用 `keras.layers.Masking` 层：它设置遮挡为 `K.any(K.not_equal(inputs, 0), axis=-1)`，意思是最后一维都是 0 的时间步，会被后续层遮挡。

对于 `Sequential` 模型，使用遮挡层，并自动向前传递遮挡是最佳的。但复杂模型上不能这么做，比如将 `Conv1D` 层与循环层混合使用时。对这种情况，需要使用函数式 API 或子类化 API 显式计算遮挡张量，然后将其传给需要的层。例如，下面的模型等价于前一个模型，除了使用函数式 API 手动处理遮挡张量：

```
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])
```

训练几个周期之后，这个模型的表现就相当不错了。如果使用 `TensorBoard()` 调回，可以可视化 `TensorBoard` 中的嵌入是怎么学习的：可以看到 `awesome` 和 `amazing` 这样的词渐渐聚集于嵌入空间的一边，而 `awful`、`terrible` 这样的词聚集到另一边。一些词可能不会像预期那样是正面的，比如 `good`，可能所有负面评论含有 `not good`。模型只基于 25000 个词就能学会词嵌入，让人印象深刻。如果训练集有几十亿的规模，效果就更好了。但可惜没有，但可以利用在其它大语料（比如，维基百科文章）上训练的嵌入，就算不是影评也可以？毕竟，`amazing` 这个词在哪种语境的意思都差不多。另外，甚至嵌入是在其它任务上训练的，也可能有益于情感分析：因为 `awesome` 和 `amazing` 有相似的意思，即使对于其它任务（比如，预测句子中的下一个词），它们也倾向于在嵌入空间聚集，所以对情感分析也是有用的。所以看看能否重复利用预训练好的词嵌入。

## 复用预训练的词嵌入

在 `TensorFlow Hub` 上可以非常方便的找到可以复用的预训练模型组件。这些模型组件被称为模块。只需浏览 [TF Hub 仓库](#)，找到需要的模型，复制代码到自己的项目中就行，模块可以自动下载下来，包含预训练权重，到自己的模型中。

例如，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.KerasLayer` 从给定的 URL 下载模块。这个特殊的模块是“句子编码器”：它接收字符串作为输入，将每句话编码为一个独立向量（这个例子中是 50 维度的向量）。在内部，它将字符串解析（空格分隔），然后使用预训练（训练语料是 Google News 7B，一共有 70 亿个词）的嵌入矩阵来嵌入每个词。然后计算所有词嵌入的平均值，结果是句子嵌入。我们接着可以添加两个简单的紧密层来创建一个出色的情感分析模型。默认，`hub.KerasLayer` 是不可训练的，但创建时可以设定 `trainable=True`，就可以针对自己的任务微调了。

警告：不是所有的 TF Hub 模块都支持 TensorFlow 2。

然后，就可以加载 IMDb 影评数据集了，不需要预处理（但要做批次和预提取），直接训练模型就成：

```

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)

```

注意到，TF Hub 模块的 URL 的末尾指定了是模型的版本 1。版本号可以保证当有新的模型版本发布时，不会破坏自己的模型。如果在浏览器中输入这个 URL，能看到这个模块的文档。TF Hub 会默认将下载文件缓存到系统的临时目录。你可能想将文件存储到固定目录，以免每次系统清洗后都要下载。要这么做的话，设置环境变量 `TFHUB_CACHE_DIR` 就成（比如，`os.environ["TFHUB_CACHE_DIR"] = "./my_tfhub_cache"`）。

截至目前，我们学习了时间序列、用 Char-RNN 生成文本、用 RNN 做情感分析、训练自己的词嵌入或复用预训练词嵌入。接下来看看另一个重要的 NLP 任务：神经网络机器翻译（NMT），我们先使用纯粹的编码器-解码器模型，然后使用注意力机制，最后看看 Transformer 架构。

## 用编码器-解码器做机器翻译

看一个简单的神经网络机器翻译模型，它能将英语翻译为法语（见图 16-3）。

简而言之，英语句子输入进编码器，解码器输出法语。注意，法语翻译也作为解码器的输入，但向后退一步。换句话说，解码器将前一步的输出再作为输入（不管它输出什么）。对于第一个词，给它加上一个序列开始（SOS）标记，序列结尾加上序列结束（EOS）标记。

英语句子在输入给编码器之前，先做了翻转。例如，`I drink milk` 翻转为 `milk drink I`。这样能保证英语句子的第一个词是最后一个输入给编码器的，通常也是解码器要翻译的第一个词。

每个单词首先用它的 ID 来表示（例如，288 代表 `milk`）。然后，嵌入层返回单词嵌入。单词嵌入才是输入给编码器和解码器的。

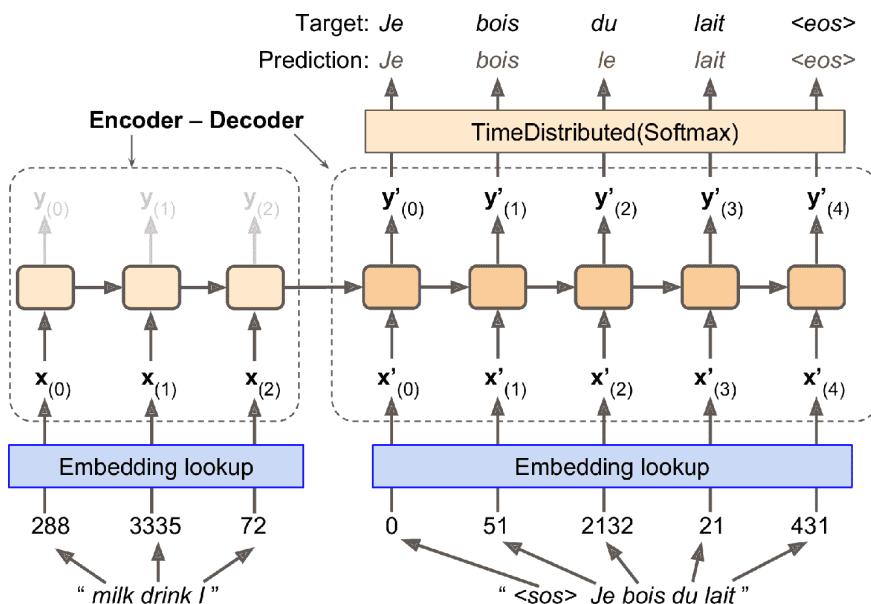


图 16-3 一个简单的机器翻译模型

在每一步，解码器输出一个输出词典中每个单词的分数，然后 softmax 层将分数变为概率。例如，在第一步，`Je` 的概率可能为 20%，`Tu` 的概率可能为 1%，等等。概率最高的词作为输出。这特别像一个常规分类任务，所以可以用 `"sparse_categorical_crossentropy"` 损失训练模型，跟前面的 Char-RNN 差不多。

在做推断时，没有目标语句输入进解码器。相反的，只是输入解码器前一步的输出，见图 16-4（这需要一个嵌入查找表，图中没有展示）。

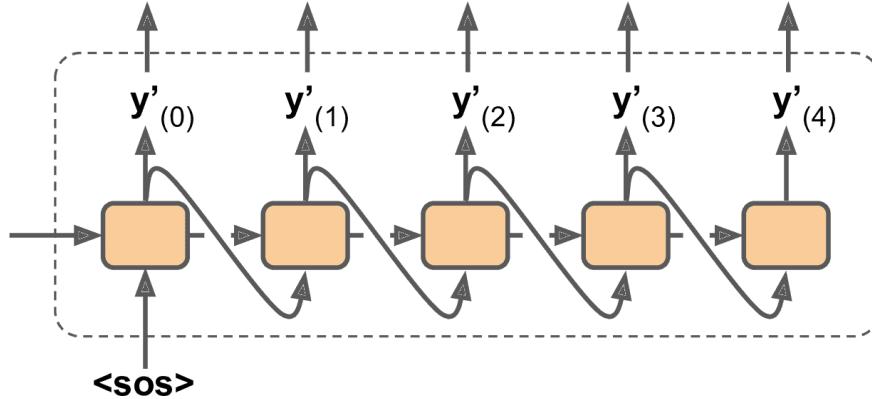


图 16-4 在推断时，将前一步的输出作为输入

好了，现在知道整体的大概了。但要实现模型的话，还有几个细节要处理：

- 目前假定所有（编码器和解码器的）输入序列的长度固定。但很显然句子长度是变化的。因为常规张量的形状固定，它们只含有相同长度的句子。可以用遮挡来处理；但如果句子的长度非常不同，就不能像之前情感分析那样截断（因为想要的是完整句子的翻译）。可以将句子放进长度相近的桶里（一个桶放 1 个词到 6 个词的句子，一个桶放 7 个词到 12 个词的句子，等等），给短句子加填充，使同一个桶中的句子长度相同  
(见 `tf.data.experimental.bucket_by_sequence_length()` 函数)。例如，`I drink milk` 变为 `<pad> <pad> <pad> milk drink I`。
- 要忽略所有在 EOS 标记后面的输出，这些输出不能影响损失（遮挡起来）。例如，如果模型输出 `Je bois du lait <eos> oui`，忽略最后一个词对损失的影响。
- 如果输出词典比较大（这个例子就是这样），输出每个词的概率会非常慢。如果目标词典有 50000 个发语词，则解码器要输出 50000 维的向量，在这个向量上计算 softmax 非常耗时。一个方法是只查看模型对正确词和非正确词采样的对数概率输出，然后根据这些对数概率计算一个大概的损失。这个采样 softmax 方法是 Sébastien Jean 在 2015 年提出的。在 TensorFlow 中，你可以在训练时使用 `tf.nn.sampled_softmax_loss()`，在推断时使用常规 softmax 函数（推断时不能使用采样 softmax，因为需要知道目标）。

TensorFlow Addons 项目涵盖了许多序列到序列的工具，可以创建准生产的编码器-解码器。例如，下面的代码创建了一个基本的编码器-解码器模型，相似于图 16-3：

```

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])

```

这个代码很简单，但有几点要注意。首先，创建 LSTM 层时，设置 `return_state=True`，以便得到最终隐藏态，并将其传给解码器。因为使用的是 LSTM 单元，它实际返回两个隐藏态（短时和长时）。`TrainingSampler` 是 TensorFlow Addons 中几个可用的采样器之一：它的作用是在每一步告诉解码器，前一步的输出是什么。在推断时，采样器是实际输出的标记嵌入。在训练时，是前一个目标标记的嵌入：这就是为什么使用 `TrainingSampler` 的原因。在实际中，一个好方法是，一开始用目标在前一时间步的嵌入训练，然后逐渐过渡到实际标记在前一步的输出。这个方法是 Samy Bengio 在 [2015 年的一篇论文](#) 中提出的。`ScheduledEmbeddingTrainingSampler` 可以随机从目标或实际输出挑选，你可以在训练中逐渐调整概率。

## 双向 RNN

在每个时间步，常规循环层在产生输出前，只会查看过去和当下的输入。换句话说，循环层是遵循因果关系的，它不能查看未来。这样的 RNN 在预测时间序列时是合理的，但对于许多 NLP 任务，比如机器翻译，在编码给定词时，最好看看后面的词是什么。比如，对于这几个短

语 `the Queen of the United Kingdom`、`the queen of hearts`、`the queen bee`：要正确编码 `queen`，需要向前看。要实现的话，可以对于相同的输入运行两个循环层，一个从左往右读，一个从右往左读。然后将每个时间步的输出结合，通常是连起来。这被称为双向循环层（见图 16-5）。

要在 Keras 中实现双向循环层，可以在 `keras.layers.Bidirectional` 层中包一个循环层。例如，下面的代码创建了一个双向 GRU 层：

```
keras.layers.Bidirectional(keras.layers.GRU(10, return_sequences=True))
```

**笔记：** `Bidirectional` 层会创建一个 GRU 层的复制（但方向相反），会运行两个层，并将输出连起来。因此 GRU 层有 10 个神经元，`Bidirectional` 层在每个时间步会输出 20 个值。

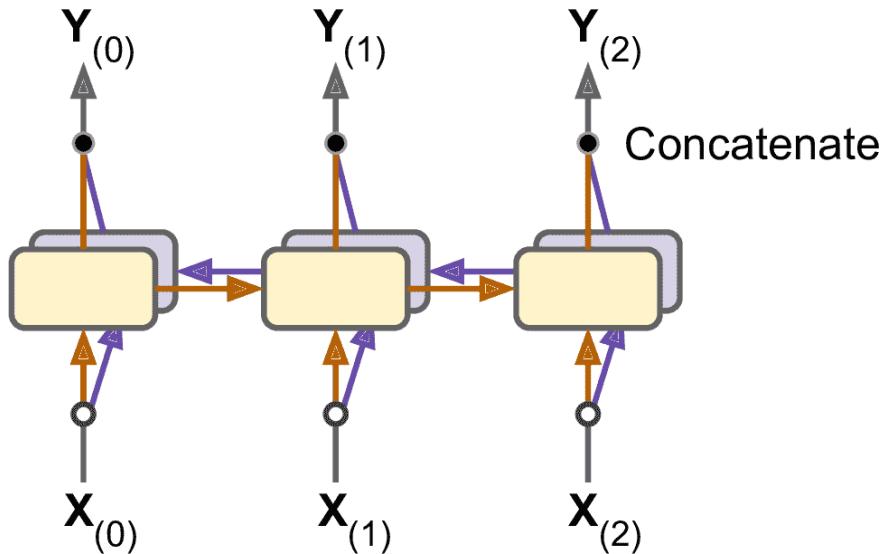


图 16-5 双向循环层

## 集束搜索

假设你用编码器-解码器模型将法语 `Comment vas-tu?` 翻译为英语。正确的翻译应该是 `How are you?`，但得到的结果是 `How will you?`。查看训练集，发现许多句子，比如 `Comment vas-tu jouer?` 翻译成了 `How will you play?`。所以模型看到 `Comment vas` 之后，将其翻译为 `How will` 并不那么荒唐。但在这个例子中，这就是一个错误，并且模型还不能返回修改，模型只能尽全力完成句子。如果每步都是最大贪心地输出结果，只能得到次优解。如何能让模型返回到之前的错误并改错呢？最常用的方法之一，是使用集束搜索：它跟踪  $k$  个最大概率的句子列表，在每个解码器步骤延长一个词，然后再关注其中  $k$  个最大概率的句子。参数  $k$  被称为集束宽度。

例如，假设使用宽度为 3 的集束搜索，用模型来翻译句子 `Comment vas-tu?`。在第一个解码步骤，模型会输出每个可能词的估计概率。假设前 3 个词的估计概率是 `How`（估计概率是 75%）、`What`（3%）、`You`（1%）。这是目前的句子列表。然后，创建三个模型的复制，预测每个句子的下一个词。第一个模型会预测 `How` 后面的词，假设结果是 36% 为 `will`、32% 为 `are`、16% 为 `do`，等等。注意，这是条件概率。第二个模型会预测 `What` 后面的词：50% 为 `are`，等等。假设词典有 10000 个词，每个模型会输出 10000 个概率。

然后，计算 30000 个含有两个词的句子的概率。将条件概率相乘。例如，`How will` 的概率是  $75\% \times 36\% = 27\%$ 。计算完 30000 个概率之后，只保留概率最大的 3 个。假设是 `How will` (27%)、`How are` (24%)、`How do` (12%)。现在 `How will` 的概率最大，但 `How are` 并没有被删掉。

接着，重复同样的过程：用三个模型预测这三个句子的接下来的词，再计算 30000 个含有三个词的句子的概率。假设前三名是 `How are you` (10%)、`How do you` (8%)、`How will you` (2%)。再下一步的前三名是 `How do you do` (7%)、`How are you <eos>` (6%)、`How are you doing` (3%)。注意，`How will` 被淘汰了。没有使用额外的训练，只是在使用层面做了改动，就提高了模型的性能。

TensorFlow Addons 可以很容易实现集束搜索：

```
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)
```

首先创建 `BeamSearchDecoder`，它包装所有的解码器的克隆（这个例子中有 10 个）。然后给每个解码器克隆创建一个编码器的最终状态的复制，然后将状态传给解码器，加上开始和结束标记。

有了这些，就能得到不错的短句的翻译了（如果使用预训练词嵌入，效果更好）。但是这个模型翻译长句子的效果很糟。这又是 RNN 的短时记忆问题。注意力机制的出现，解决了这一问题。

## 注意力机制

图 16-3 中，从 `milk` 到 `lait` 的路径非常长。这意味着这个单词的表征（还包括其它词），在真正使用之前，要经过许多步骤。能让这个路径短点吗？

这是 Dzmitry Bahdanau 在 2014 年的突破性[论文](#)中的核心想法。他们引入了一种方法，可以让解码器在每个时间步关注特别的（被编码器编码的）词。例如，在解码器需要输出单词 `lait` 的时间步，解码器会将注意力关注在单词 `milk` 上。这意味着从输入词到其翻译结果的路径变的短得多了，所以 RNN 的短时记忆的限制就减轻了很多。注意力机制革新了神经网络机器翻译（和 NLP 的常见任务），特别是对于长句子（超过 30 个词），带来了非凡的进步。

图 16-6 展示了注意力机制的架构（稍微简化过，后面会说明）。左边是编码器和解码器。不是将编码器的最终隐藏态传给解码器（其实是传了，但图中没有展示），而是将所有的输出传给解码器。在每个时间步，解码器的记忆单元计算所有这些输出的加权和：这样可以确定这一步关注哪个词。权重  $\alpha[t, i]$  是第  $i$  个编码器输出在第  $t$  解码器时间步的权重。例如，如果权重  $\alpha[3, 2]$  比  $\alpha[3, 0]$  和  $\alpha[3, 1]$  大得多，则解码器会用更多注意力关注词 2（`milk`），至少是在这个时间步。剩下的解码器就和之前一样工作：在每个时间步，记忆单元接收输入，加上上一个时间步的隐藏态，最后（这一步图上没有画出）加上上一个时间步的目标词（或推断时，上一个时间步的输出）。

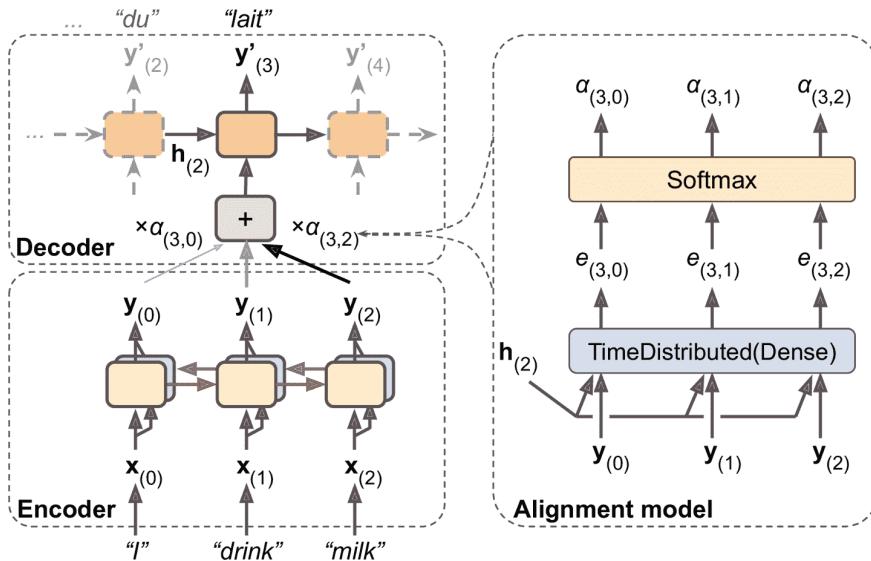


图 16-6 使用了注意力模型的编码器-解码器结构

权重  $\alpha[t, i]$  是从哪里来的呢？其实很简单：是用一种小型的、被称为对齐模型（或注意力层）的神经网络生成的，注意力层与模型的其余部分联合训练。对齐模型展示在图的右边：一开始是一个时间分布紧密层，其中有一个神经元，它接收所有编码器的输出，加上解码器的上一个隐藏态（即  $h[2]$ ）。这个层输出对每个编码器输出，输出一个分数（或能量）（例如， $e[3, 2]$ ）：这个分数衡量每个输出和解码器上一个隐藏态的对齐程度。最后，所有分数经过一个 softmax 层，得到每个编码器输出的最终权重（例如， $\alpha[3, 2]$ ）。给定解码器时间步的所有权重相加等于 1（因为 softmax 层不是时间分布的）。这个注意力机制称为 Bahdanau 注意力。因为它将编码器输出和解码器的上一隐藏态连了起来，也被称为连接注意力（或相加注意力）。

笔记：如果输入句子有  $n$  个单词，假设输出也是这么多单词，则要计算  $n^2$  个权重。幸好，平方计算的复杂度不高，因为即使是特别长的句子，也不会有数千个单词。

另一个常见的注意力机制是不久之后，由 Minh-Thang Luong 在 2015 年的[论文](#)中提出的。因为注意力机制的目标是衡量编码器的输出，和解码器上一隐藏态的相似度，Minh-Thang Luong 提出，只要计算这两个向量的点积，因为点积是有效衡量相似度的手段，并且计算起来很快。要计算的话，两个向量的维度必须相同。这被称为 Luong 注意力，或相乘注意力。和 Bahdanau 注意力一样，点积的结果是一个分数，所有分数（在特定的解码器时间步）通过 softmax 层，得到最终权重。Luong 提出的另一个简化方法是使用解码器在当前时间步的隐藏态，而不是上一时间步，然后使用注意力机制的输出（标记为  $h_{\hat{t}}[t]$ ），直接计算解码器的预测（而不是计算解码器的当前隐藏态）。他还提出了一个点击的变体，编码器的输出先做线性变换（即，时间分布紧密层不加偏置项），再做点积。这被称为“通用”点积方法。作者比较了点积方盒和连接注意力机制（加上一个缩放参数  $v$ ），观察到点积方法的变体表现的更好。因为这个原因，如今连接注意力很少使用了。公式 16-1 总结了这三种注意力机制。

$$\tilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t,i)} \mathbf{y}_{(i)}$$

with  $\alpha_{(t,i)} = \frac{\exp(e_{(t,i)})}{\sum_{i'} \exp(e_{(t,i')})}$

and  $e_{(t,i)} = \begin{cases} \mathbf{h}_{(t)}^\top \mathbf{y}_{(i)} & dot \\ \mathbf{h}_{(t)}^\top \mathbf{W} \mathbf{y}_{(i)} & general \\ \mathbf{v}^\top \tanh(\mathbf{W} [\mathbf{h}_{(t)}; \mathbf{y}_{(i)}]) & concat \end{cases}$

公式 16-1 注意力机制

使用 TensorFlow Addons 将 Luong 注意力添加到编码器-解码器模型的方法如下：

```
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)
```

只是将解码器单元包装进 `AttentionWrapper`，然后使用了想用的注意力机制（这里用的是 Luong 注意力）。

## 视觉注意力

注意力机制如今应用的非常广泛。最先用途之一是利用视觉注意力生成图片标题：卷积神经网络首先处理图片，生成一些特征映射，然后用带有注意力机制的解码器 RNN 来生成标题，每次生成一个词。在每个解码器时间步（每个词），解码器使用注意力模型聚焦于图片的一部分。例如，对于图 16-7，模型生成的标题是“一个女人正在公园里扔飞盘”，可以看到解码器要输出单词“飞盘”时，注意力关注的图片的部分：显然，注意力大部分聚焦于飞盘。

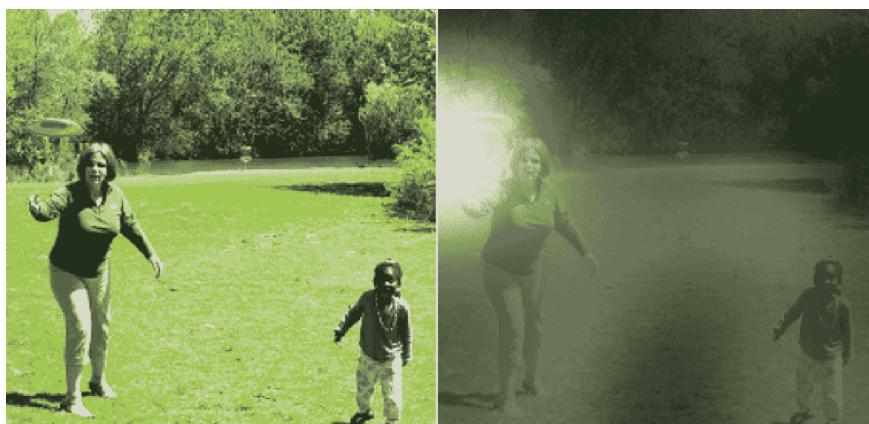


图 16-7 视觉注意力：输入图片（左）和模型输出“飞盘”时模型的关注点（右）

### 解释性

注意力机制的一个额外的优点，是它更容易使人明白是什么让模型产生输出。这被称为可解释性。当模型犯错时，可解释性非常有帮助：例如，如果一张狗在雪中行走的图，被打上了“狼在雪中行走”的标签，你就可以回去查看当模型输出“狼”时，模型聚焦于什么。你可能看到模型不仅关注于狗，还关注于雪地，暗示了一种可能的解释：可能模型判断是根据有很多雪，来判断是狗还是狼。然后可以通过用更多没有雪的狼的图片进行训练，来修复模型。这个例子来自于 Marco Tulio Ribeiro 在 2016 年的[论文](#)，他们使用了不同的可解释性：局部围绕分类器的预测，来学习解释性模型。

在一些应用中，可解释性不仅是调试模型的工具，而是正当的需求（比如一个判断是否进行放贷的需求）。

注意力机制如此强大，以至于只需要注意力机制就能创建出色的模型。

## Attention Is All You Need: Transformer 架构

在 2017 年一篇突破性[论文](#)中，谷歌的研究者提出了：Attention Is All You Need（只要注意力）。他们创建了一种被称为 Transformer（转换器）的架构，它极大的提升了 NMT 的性能，并且没有使用任何循环或卷积层，只用了注意力机制（加上嵌入层、紧密层、归一化层，和一些其它组件）。这个架构的另一个优点，是训练的更快，且更容易并行运行，花费的时间和精力比之前的模型少得多。

Transformer 架构见图 16-8。

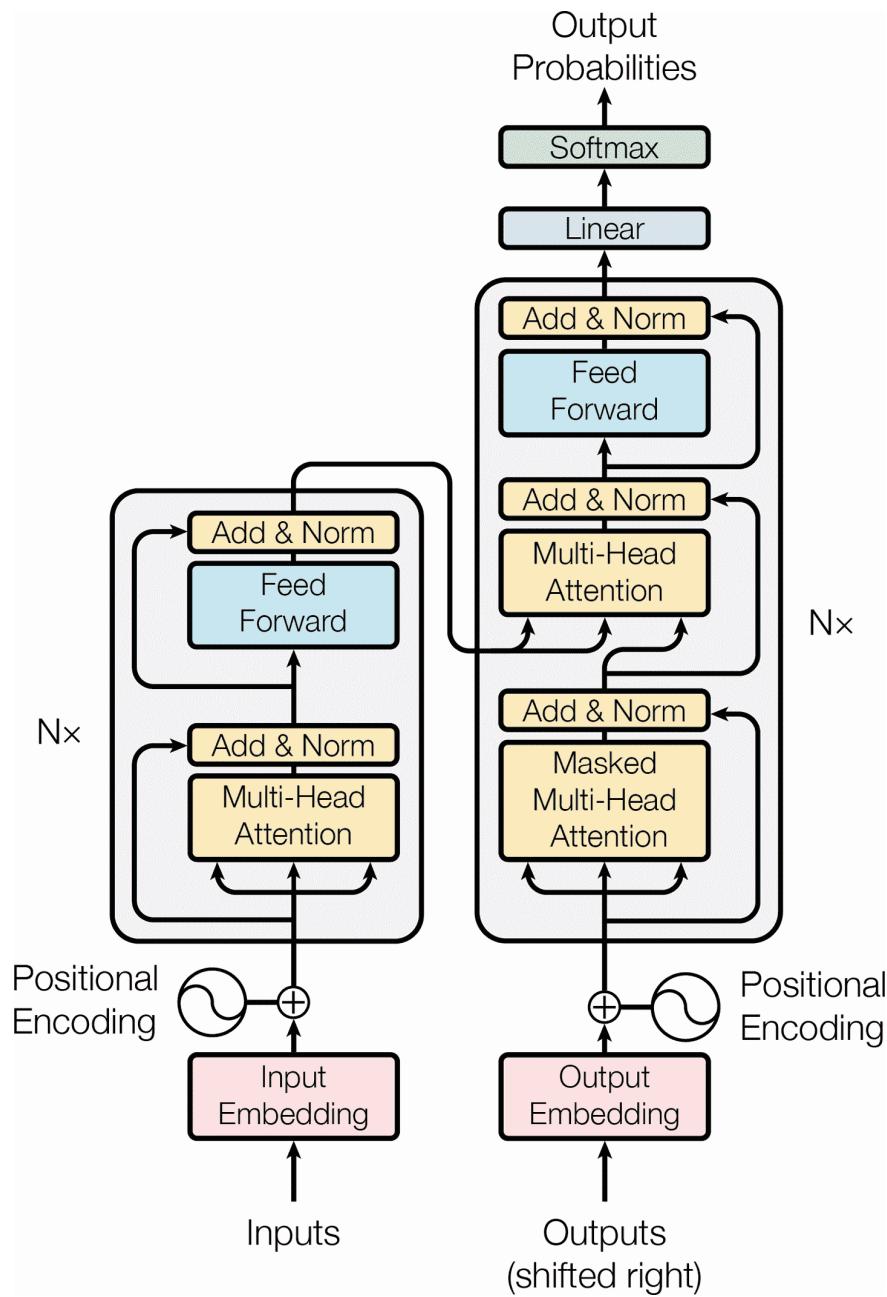


图 16-8 Transformer 架构

一起看下这个架构：

- 图的左边和以前一样是编码器，接收的输入是一个批次的句子，表征为序列化的单词 ID（输入的形状是 [批次大小, 最大输入句子长度]），每个单词表征为 512 维（所以编码器的输出形状是 [批次大小, 最大输入句子长度, 512]）。注意，编码器的头部叠加了  $N$  次（论文中， $N=6$ ）。
- 架构的右边是解码器。在训练中，它接收目标句子作为输入（也是表征为序列化的单词 ID），向右偏移一个时间步（即，在起点插入一个 SOS 标记）。它还接收编码器的输出（即，来自左边的箭头）。注意，解码器的头部也重叠了  $N$  次，编码器的最终输出，传入给解码器重叠层中的每一个部分。和以前一样，在每个时间步，解码器输出每个下一个可能词的概率（输出形状是 [批次大小, 最大输出句子长度, 词典长度]）。

- 在推断时，解码器不能接收目标，所以输入的是前面的输出词（起点用 SOS 标记）。因此模型需要重复被调用，每一轮预测一个词（预测出来的词在下一轮输入给解码器，直到输出 EOS 标记）。
- 仔细观察下，可以看到其实你已经熟悉其中大部分组件了：两个嵌入层， $5 \times N$  个跳连接，每个后面是一个归一化层， $2 \times N$  个“前馈”模块（由两个紧密层组成（第一个使用 ReLU 激活函数，第二个不使用激活函数），输出层是使用 softmax 激活函数的紧密层）。所有这些层都是时间分布的，因此每个词是独立处理的。但是一次只看一个词，该如何翻译句子呢？这时就要用到新组件了：
  - 编码器的多头注意力层，编码每个词与句子中其它词的关系，对更相关的词付出更多注意力。例如，输出句子 They welcomed the Queen of the United Kingdom 中的词 Queen 的层的输出，会取决于句子中的所有词，但更多注意力会在 United 和 Kingdom 上。这个注意力机制被称为自注意力（句子对自身注意）。后面会讨论它的原理。解码器的遮挡多头注意力层做的事情一样，但每个词只关注它前面的词。最后，解码器的上层多头注意力层，是解码器用于在输入句子上付出注意力的。例如，当解码器要输出 Queen 的翻译时，解码器会对输入句子中的 Queen 这个词注意更多。
  - 位置嵌入是紧密向量（类似词嵌入），表示词在句子中的位置。第  $n$  个位置嵌入，添加到每个句子中的第  $n$  个词上。这可以让模型知道每个词的位置，这是因为多头注意力层不考虑词的顺序或位置，它只看关系。因为所有其它层都是时间分布的，它们不知道每个词的（相对或绝对）位置。显然，相对或绝对的词的位置非常重要，因此需要将位置信息以某种方式告诉 Transformer，位置嵌入是行之有效的方法。

下面逐一详细介绍 Transformer 中的新组件，从位置嵌入开始。

## 位置嵌入

位置嵌入是一个紧密向量，它对词在句子中的位置进行编码：第  $i$  个位置嵌入添加到句子中的第  $i$  个词。模型可以学习这些位置嵌入，但在论文中，作者倾向使用固定位置嵌入，用不同频率的正弦和余弦函数来定义。公式 16-2 定义了位置嵌入矩阵  $P$ ，见图 16-9 的底部（做过转置），其中  $P[p, i]$  是单词在句子的第  $p$  个位置的第  $i$  个嵌入的组件。

$$\begin{aligned} P_{p,2i} &= \sin \left( p / 10000^{2i/d} \right) \\ P_{p,2i+1} &= \cos \left( p / 10000^{2i/d} \right) \end{aligned}$$

公式 16-2 正弦/余弦位置嵌入

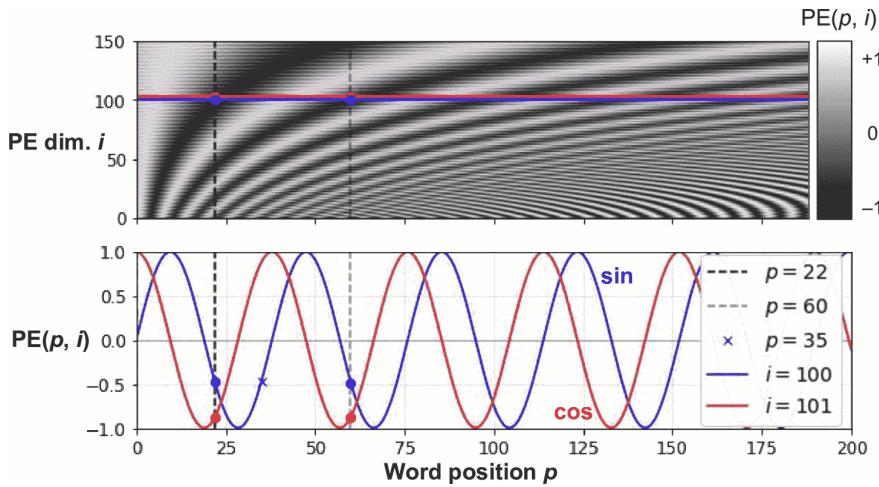


图 16-9 正弦/余弦位置嵌入矩阵（经过转置，上），关注  $i$  的两个值（下）

这个方法的效果和学习过的位置嵌入相同，但可以拓展到任意长度的句子上，这是它受欢迎的原因。给词嵌入加上位置嵌入之后，模型剩下的部分就可以访问每个词在句子中的绝对位置了，因为每个值都有一个独立的位置嵌入（比如，句子中第 22 个位置的词的位置嵌入，表示为图 16-9 中的左下方的垂直虚线，可以看到位置嵌入对这个位置是一对一的）。另外，振动函数（正弦和余弦）选择也可以让模型学到相对位置。例如，相隔 38 个位置的词（例如，在位置  $p=22$  和  $p=60$ ）总是在嵌入维度  $i=100$  和  $i=101$  有相同的位置嵌入值，见图 16-9。这解释了对于每个频率，为什么需要正弦和余弦两个函数：如果只使用正弦（蓝线， $i=100$ ），模型不能区分位置  $p=25$  和  $p=35$ （叉子标记）。

TensorFlow 中没有 `PositionalEmbedding` 层，但创建很容易。出于效率的考量，在构造器中先计算出位置嵌入（因此需要知道最大句子长度，`max_steps`，每个词表征的维度，`max_dims`）。然后调用 `call()` 方法裁剪嵌入矩阵，变成输入的大小，然后添加到输入上。因为创建位置嵌入矩阵时，添加了一个大小为 1 的维度，广播机制可以确保位置矩阵添加到输入中的每个句子上：

```
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[:, :, ::2] = np.sin(p / 10000**2 * i / max_dims).T
        pos_emb[:, :, 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]]
```

然后可以创建 Transformer 的前几层：

```
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)
```

接下来看看 Transformer 的核心：多头注意力层。

## 多头注意力

要搞懂多头注意力层的原理，必须先搞懂收缩点积注意力层（Scaled Dot-Product Attention），多头注意力是基于它的。假设编码器分析输入句子 `They played chess`，编码器分析出 `They` 是主语，`played` 是动词，然后用词的表征编码这些信息。假设解码器已经翻译了主语，接下来要翻译动词。要这么做的话，它需要从输入句子取动词。这有点像查询字典：编码器创建了字典 `{"subject": "They", "verb": "played", ...}`，解码器想查找键 `verb` 对应的值是什么。但是，模型没有离散的标记来表示键（比如 `subject` 或 `verb`）；它只有这些（训练中学到的）信息的向量化表征所以用来查询的键，不会完美对应前面字典中的键。解决的方法是计算查询词和键的相似度，然后用 `softmax` 函数计算概率权重。如果表示动词的键和查询词很相似，则键的权重会接近于 1。然后模型可以计算对应值的加权和，如果 `verb` 键的权重接近 1，则加权和会接近于词 `played` 的表征。总而言之，可以将整个过程当做字典查询。`Transformer` 使用点积做相似度计算，和 Luong 注意力一样。实际上，公式和 Luong 注意力一样，除了有缩放参数，见公式 16-3，是向量的形式。

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{keys}}}\right)\mathbf{V}$$

公式 16-3 缩放点积注意力

在这个公式中：

- `Q` 矩阵每行是一个查询词。它的形状是 `[n_queries, d_keys]`，`n_queries` 是查询数，`d_keys` 是每次查询和每个键的维度数。
- `K` 矩阵每行是一个键。它的形状是 `[n_keys, d_keys]`，`n_keys` 是键和值的数量。
- `V` 矩阵每行是一个值。它的形状是 `[n_keys, d_values]`，`d_values` 是每个值的数。
- `Q K^T` 的形状是 `[n_queries, n_keys]`：它包含这每个查询/键对的相似分数。`softmax` 函数的输出有相同的形状，且所有行的和是 1。最终的输出形状是 `[n_queries, d_values]`，每行代表一个查询结果（值的加权和）。
- 缩放因子缩小了相似度分数，防止 `softmax` 函数饱和（饱和会导致梯度变小）。
- 在计算 `softmax` 之前，通过添加一些非常大的负值，到对应的相似度分上，可以遮挡一些键值对。这在遮挡多头机制层中很有用。

在编码器中，这个公式应用到批次中的每个句子，`Q`、`K`、`V` 等于输入句中的词列表（所以，句子中的每个词会和相同句中的每个词比较，包括自身）。相似的，在解码器的遮挡注意力层中，这个公式会应用到批次中每个目标句上，但要用遮挡，防止每个词和后面的词比较（因为在推断时，解码器只能访问已经输出的词，所以训练时要遮挡后面的输出标记）。在解码器的上边的注意力层，键 `K` 矩阵和值 `V` 矩阵是斌吗器生成的此列表，查询 `Q` 矩阵是解码器生成的词列表。

`keras.layers.Attention` 层实现了缩放点积注意力，它的输入是 `Q`、`K`、`V`，除此之外，还有一个批次维度（第一个维度）。

提示：在 TensorFlow 中，如果  $A$  和  $B$  是两个维度大于 2 的张量 —— 比如，分别是  $[2, 3, 4, 5]$  和  $[2, 3, 5, 6]$  —— 则 `tf.matmul(A, B)` 会将这两个张量当做  $2 \times 3$  的数组，每个单元都是一个矩阵，它会乘以对应的矩阵。 $A$  中第  $i$  行、第  $j$  列的矩阵，会乘以  $B$  的第  $i$  行、第  $j$  列的矩阵。因为  $2 \times 3$  矩阵乘以  $2 \times 3$  矩阵，结果是  $2 \times 3$  矩阵，所以 `tf.matmul(A, B)` 的结果数组的形状是  $[2, 3, 4, 6]$ 。

如果忽略跳连接、归一化层、前馈块，且这是缩放点积注意力，不是多头注意力，则 Transformer 可以如下实现：

```
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` 参数可以让层学会如何缩小相似度分数。这是和 Transformer 的一个区别，后者总是用相同的因子 () 缩小相似度分数。`causal=True` 参数，可以让注意力层的每个输出标记只注意前面的输出标记。

下面来看看多头注意力层是什么？它的架构见图 16-10。

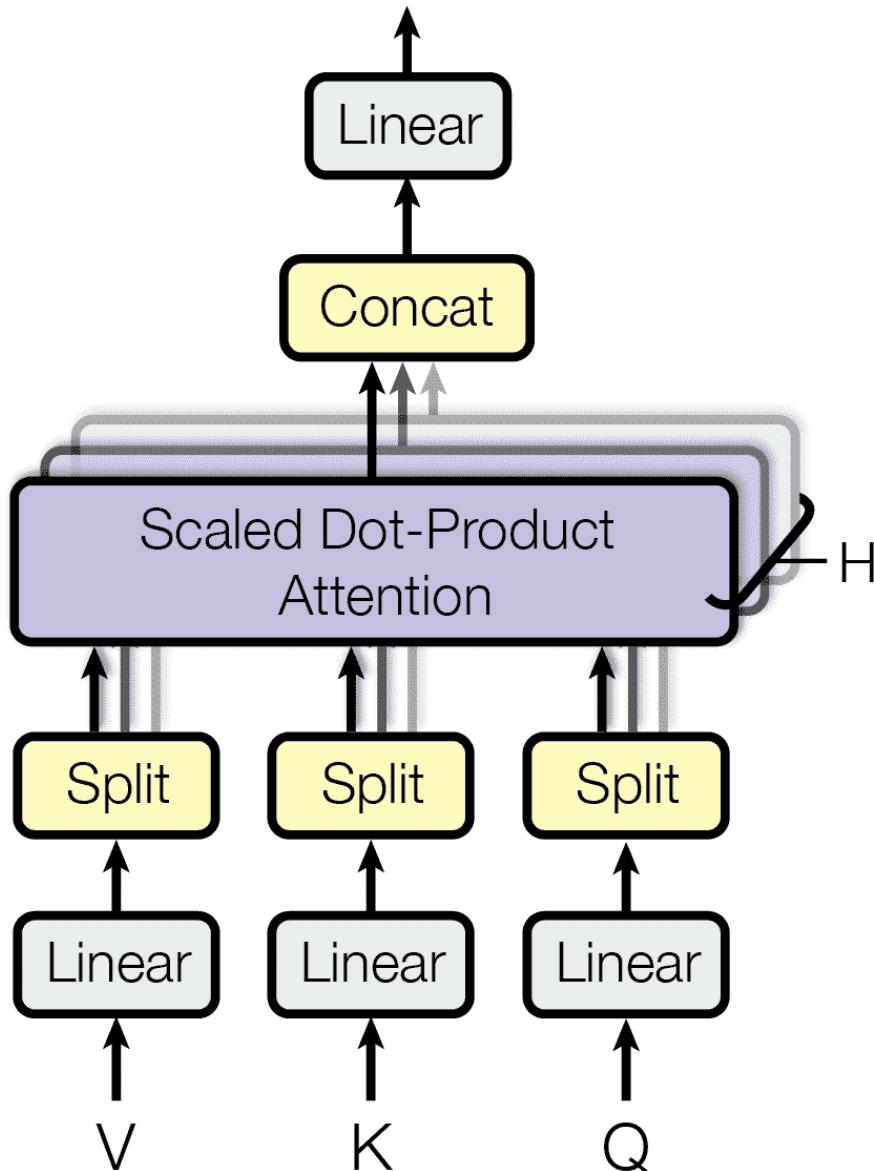


图 16-10 多头注意力层架构

可以看到，它包括一组缩放点积注意力层，每个前面有一个值、键、查询的线性变换（即，时间分布紧密层，没有激活函数）。所有输出简单连接起来，再通过一个最终的线性变换。为什么这么做？这个架构的背后意图是什么？考虑前面讨论过的单词 `played`。编码器可以将它是动词的信息做编码。同时，词表征还包含它在文本中的位置（得益于位置嵌入），除此之外，可能还包括了其它有用的信息，比如时态。总之，词表征编码了词的许多特性。如果只用一个缩放点积注意力层，则只有一次机会来查询所有这些特性。这就是为什么多头注意力层使用了多个不同的值、键、查询的线性变换：这可以让模型将词表征投影到不同的亚空间，每个关注于词特性的一个子集。也许一个线性层将词表征投影到一个亚空间，其中的信息是该词是个动词，另一个线性层会提取它是一个过去式，等等。然后缩放点积注意力做查询操作，最后将所有结果串起来，在投射到原始空间。

在写作本书时，TensorFlow 2 还没有 `Transformer` 类或 `MultiHeadAttention` 类。但是，可以查看 TensorFlow 的这个教程：[创建语言理解的 Transformer 模型](#)。另外，TF Hub 团队正向 TensorFlow 2 移植基于 `Transformer` 的模块，很快就可以用

了。同时，我希望我向你展示了自己实现 Transformer 并不难，这是一个很好的练习！

## 语言模型的最新进展

2018 年被称为“NLP 的 ImageNet 时刻”：成果惊人，产生了越来越大的基于 LSTM 和 Transformer、且在大数据集上训练过的架构。建议你看看下面的论文，都是 2018 年发表的：

- Matthew Peters 的 [ELMo 论文](#)，介绍了语言模型的嵌入（Embeddings from Language Models (ELMo)）：学习深度双向语言模型的内部状态，得到的上下文词嵌入。例如，词 queen 在 Queen of the United Kingdom 和 queen bee 中的嵌入不同。
- Jeremy Howard 和 Sebastian Ruder 的 [ULMFiT 论文](#)，介绍了无监督预训练对 NLP 的有效性：作者用海量语料，使用自监督学习（即，从数据自动生成标签）训练了一个 LSTM 语言模型，然后在各种任务上微调模型。他们的模型在六个文本分类任务上取得了优异的结果（将误差率降低了 18-24%）。另外，他们证明，通过在 100 个标签样本上微调预训练模型，可以达到在 10000 个样本上训练的效果。
- Alec Radford 和其他 OpenAI 人员的 [GPT 论文](#)，也展示了无监督训练的有效性，但他们使用的是类似 Transformer 的架构。作者预训练了一个庞大但简单的架构，由 12 个 Transformer 模块组成（只使用了遮挡多头注意力机制），也是用自监督训练的。然后在多个语言任务上微调，只对每个任务做了小调整。任务种类很杂：包括文本分类、衔接（句子 A 是否跟着句子 B），相似度（例如，Nice weather today 和 It is sunny 很像），还有问答（通过阅读几段文字，让模型来回答多选题）。几个月之后，在 2019 年的二月，Alec Radford、Jeffrey Wu 和其它 OpenAI 的人员发表了 [GPT-2 论文](#)，介绍了一个相似的架构，但是更大（超过 15 亿参数），他们展示了这个架构可以在多个任务上取得优异的表现，且不需要微调。这被称为零次学习（zero-shot learning (ZSL)）。[这个页面](#)上是一个 GPT-2 模型的带有预训练权重的小型版本，“只有”1.17 亿个参数。
- Jacob Devlin 和其它 Google 人员的 [BERT 论文](#)，也证明了在海量语料上做自监督预训练的有效性，使用的是类似 GPT 的架构，但用的是无遮挡多头注意力层（类似 Transformer 的编码器）。这意味着模型实际是双向的这就是 BERT (Bidirectional Encoder Representations from Transformers) 中的 B 的含义。最重要的，作者提出了两个预训练任务，用以测试模型能力：

遮挡语言模型 (MLM) 句子中的词有 15 的概率被遮挡。训练模型来预测被遮挡的词。例如，如果原句是 She had fun at the birthday party，模型的输入是 She <mask> fun at the <mask> party，让模型来预测 had 和 birthday（忽略其它输出）。更加准确些，每个选出的单词有 80% 的概率被遮挡，10% 的概率被替换为随机词（降低预训练和微调的差异，因为模型在微调时看不到 <mask> 标记），10% 的概率不变（使模型偏向正确答案）。

预测下一句 (NSP) 训练模型预测两句话是否是连续的。例如，模型可以预测 The dog sleeps 和 It snores loudly 是连续的，但是 The dog sleeps 和 The Earth orbits the Sun 是不连续的。这是一个有挑战的任务，可以在微调任务，比如问答和衔接上，极大提高模型的性能。

可以看到，2018 年和 2019 年的创新是亚词层面的分词，从 LSTM 转向 Transformer，使用自监督学习预训练语言模型，做细微的架构变动（或不变动）来微调模型。因为进展非常快，每人说得清明年流行的是什么。如今，流行的是 Transformer，但明天可能是 CNN（Maha Elbayad 在 [2018 年的论文](#)，使用了遮挡的 2D 卷积层来做序列到序列任务）。如果卷土重来的话，也有可能是 RNN（例如，Shuai Li 在 [2018 年的论文](#)展示了，通过让给定 RNN 层中的单元彼此独立，可以训练出更深的 RNN，能学习更长的序列）。

下一章，我们会学习用自编码器，以无监督的方式学习深度表征，并用生成对抗网络生成图片及其它内容！

## 练习

1. 有状态 RNN 和无状态 RNN 相比，优点和缺点是什么？
2. 为什么使用编码器-解码器 RNN，而不是普通的序列到序列 RNN，来做自动翻译？
3. 如何处理长度可变的输入序列？长度可变的输出序列怎么处理？
4. 什么是集束搜索，为什么要用集束搜索？可以用什么工具实现集束搜索？
5. 什么是注意力机制？用处是什么？
6. Transformer 架构中最重要的层是什么？它的目的是什么？
7. 什么时候需要使用采样 softmax？
8. Hochreiter 和 Schmidhuber 在关于 LSTM 的[论文](#)中使用了嵌入 Reber 语法。这是一种人工的语法，用来生成字符串，比如 BPBTSXXVPSEPE。查看 Jenny Orr 对它的[介绍](#)。选择一个嵌入 Reber 语法（比如 Jenny Orr 的论文中展示的），然后训练一个 RNN 来判断字符串是否符合语法。你需要先写一个函数来生成训练批次，其中 50% 符合语法，50% 不符合语法。
9. 训练一个编码器-解码器模型，它可以将日期字符串从一个格式变为另一个格式（例如，从 April 22, 2019 变为 2019-04-22）。
10. 阅读 TensorFlow 的 [《Neural Machine Translation with Attention tutorial》](#)。
11. 使用一个最近的语言模型（比如，BERT），来生成一段更具信服力的莎士比亚文字。

参考答案见附录 A。

## 十七、使用自编码器和 GAN 做表征学习和生成式学习

译者：[@SeanCheney](#)

自编码器是能够在无监督（即，训练集是未标记）的情况下学习输入数据的紧密表征（叫做潜在表征或编码）的人工神经网络。这些编码通常具有比输入数据低得多的维度，使得自编码器对降维有用（参见第 8 章）。自编码器还可以作为强大的特征检测器，它们可以用于无监督的深度神经网络预训练（正如我们在第 11 章中讨论过的）。最后，一些自编码器是生成式模型：他们能够随机生成与训练数据非常相似的新数据。例如，您可以在脸图片上训练自编码器，然后可以生成新脸。但是生成出来的图片通常是模糊且不够真实。

相反，用对抗生成网络（GAN）生成的人脸可以非常逼真，甚至让人认为他们是真实存在的人。你可以去[这个网址](#)，这是用 StyleGAN 生成的人脸，自己判断一下（还可以去[这里](#)，看看 GAN 生成的卧室图片），GAN 现在广泛用于超清图片涂色，图片编辑，将草图变为照片，增强数据集，生成其它类型的数据（比如文本、音频、时间序列），找出其它模型的缺点并强化，等等。

自编码器和 GAN 都是无监督的，都可以学习紧密表征，都可以用作生成模型，有许多相似的应用，但原理非常不同：

- 自编码器是通过学习，将输入复制到输出。听起来很简单，但内部结构会使其相当困难。例如，你可以限制潜在表征的大小，或者可以给输入添加噪音，训练模型恢复原始输入。这些限制组织自编码器直接将输入复制到输出，可以强迫模型学习数据的高效表征。总而言之，编码是自编码器在一些限制下学习恒等函数的副产品。
- GAN 包括两个神经网络：一个生成器尝试生成和训练数据相似的数据，一个判别器来区分真实数据和假数据。特别之处在于，生成器和判别器在训练过程中彼此竞争：生成器就像一个制造伪钞的罪犯，而判别器就像警察一样，要把真钱挑出来。对抗训练（训练竞争神经网络），被认为是近几年的一大进展。在 2016 年，Yann LeCun 甚至说 GAN 是过去 10 年机器学习领域最有趣的发明。

本章中，我们先探究自编码器的工作原理开始，如何做降维、特征提取、无监督预训练将、如何用作生成式模型。然后过渡到 GAN。先用 GAN 生成假图片，可以看到训练很困难。会讨论对抗训练的主要难点，以及一些解决方法。先从自编码器开始。

### 有效的数据表征

以下哪一个数字序列更容易记忆？

- 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，因为它要短得多。但是，如果仔细观察第二个序列，就会发现它是从 50 到 14 的偶数。一旦你注意到这个规律，第二个序列比第一个更容易记忆，因为你只需要记住规律就成，开始的数字和结尾的数字。请注意，如果您可以快速轻松地记住非常长的序列，则不会在意第二个序列中存在的规律。只要记住每一个数字，就够了。事实上，很难记住长序列，因此识别规律非常有用，并且希望能够澄清为什么在训练过程中限制自编码器会促使它发现并利用数据中的规律。

记忆、感知和模式匹配之间的关系在 20 世纪 70 年代早期由 William Chase 和 Herbert Simon 研究。他们观察到，专业棋手能够通过观看棋盘 5 秒钟就能记住所有棋子的位置，这是大多数人认为不可能完成的任务。然而，只有当这些棋子被放置在现实位置（来自实际比赛）时才是这种情况，而不是随机放置棋子。国际象棋专业棋手没有比你更好的记忆，他们只是更容易看到国际象棋的规律，这要归功于他们的比赛经验。观察规律有助于他们有效地存储信息。

就像这个记忆实验中的象棋棋手一样，一个自编码器会查看输入信息，将它们转换为高效的潜在表征，然后输出一些（希望）看起来非常接近输入的东西。自编码器总是由两部分组成：将输入转换为潜在表征的编码器（或识别网络），然后是将潜在表征转换为输出的解码器（或生成网络）（见图 17-1）。

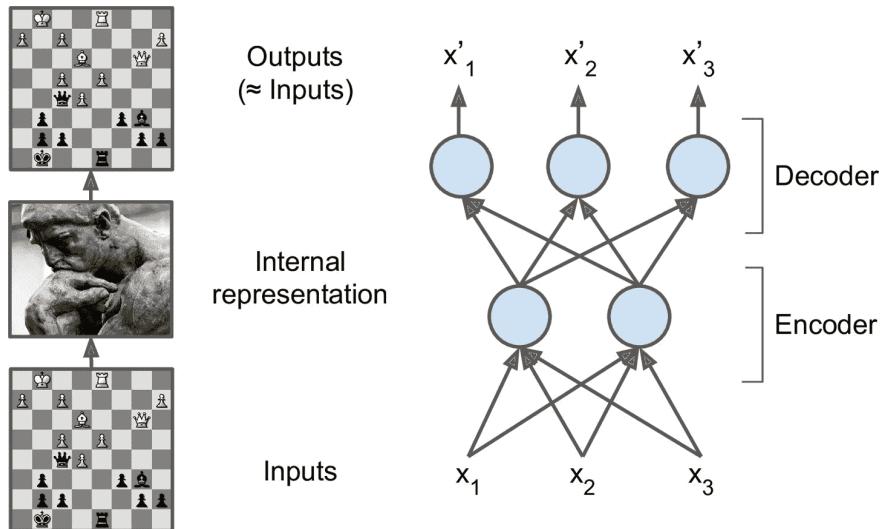


图 17-1 记忆象棋试验（左）和一个简单的自编码器（右）

如你所见，自编码器通常具有与多层感知器（MLP，请参阅第 10 章）相同的体系结构，但输出层中的神经元数量必须等于输入数量。在这个例子中，只有一个由两个神经元（编码器）组成的隐藏层和一个由三个神经元（解码器）组成的输出层。由于自编码器试图重构输入，所以输出通常被称为重建，并且损失函数包含重建损失，当重建与输入不同时，重建损失会对模型进行惩罚。

由于内部表征具有比输入数据更低的维度（它是 2D 而不是 3D），所以自编码器被认为是不完整的。不完整的自编码器不能简单地将其输入复制到编码，但它必须找到一种方法来输出其输入的副本。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）。

我们来看看如何实现一个非常简单的不完整的自编码器，以降低维度。

## 用不完整的线性自编码器来做 PCA

如果自编码器仅使用线性激活并且损失函数是均方误差（MSE），最终其实是做了主成分分析（参见第 8 章）。

以下代码创建了一个简单的线性自编码器，以在 3D 数据集上执行 PCA，并将其投影到 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))
```

这段代码与我们在前面章节中创建的所有 MLP 没有什么大不同。只有以下几点要注意：

- 自编码器由两部分组成：编码器和解码器。两者都是常规的 `Sequential` 模型，每个含有一个紧密层，自编码器是一个编码器和解码器连起来的 `Sequential` 模型（模型可以用作其它模型中的层）。
- 自编码器的输出等于输入。
- 简单 PCA 不需要激活函数（即，所有神经元是线性的），且损失函数是 MSE。后面会看到更复杂的自编码器。

现在用生成出来的 3D 数据集训练模型，并用模型编码数据集（即将其投影到 2D）：

```
history = autoencoder.fit(X_train, X_train, epochs=20)
codings = encoder.predict(X_train)
```

注意，`X_train` 既用来做输入，也用来做目标。图 17-2 显示了原始 3D 数据集（左侧）和自编码器隐藏层的输出（即编码层，右侧）。可以看到，自编码器找到了投影数据的最佳二维平面，保留了数据的尽可能多的差异（就像 PCA 一样）。

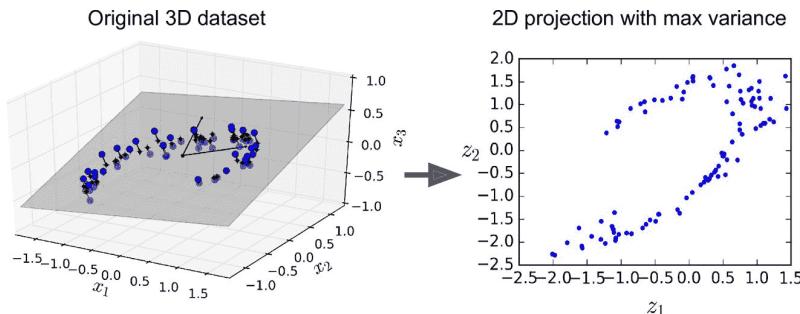


图 17-2 用不完整的线性自编码器实现 PCA

笔记：可以将自编码器当做某种形式的自监督学习（带有自动生成标签功能的监督学习，这个例子中标签等于输入）

## 栈式自编码器

就像我们讨论过的其他神经网络一样，自编码器可以有多个隐藏层。在这种情况下，它们被称为栈式自编码器（或深度自编码器）。添加更多层有助于自编码器了解更复杂的编码。但是，必须注意不要让自编码器功能太强大。设想一个编码器

非常强大，只需学习将每个输入映射到一个任意数字（并且解码器学习反向映射）即可。很明显，这样的自编码器将完美地重构训练数据，但它不会在过程中学习到任何有用的数据表征（并且它不可能很好地泛化到新的实例）。

栈式自编码器的架构以中央隐藏层（编码层）为中心通常是对称的。简单来说，它看起来像一个三明治。例如，一个用于 MNIST 的自编码器（在第 3 章中介绍）可能有 784 个输入，其次是一个隐藏层，有 100 个神经元，然后是一个中央隐藏层，有 30 个神经元，然后是另一个隐藏层，有 100 个神经元，输出层有 784 个神经元。这个栈式自编码器如图 17-3 所示。

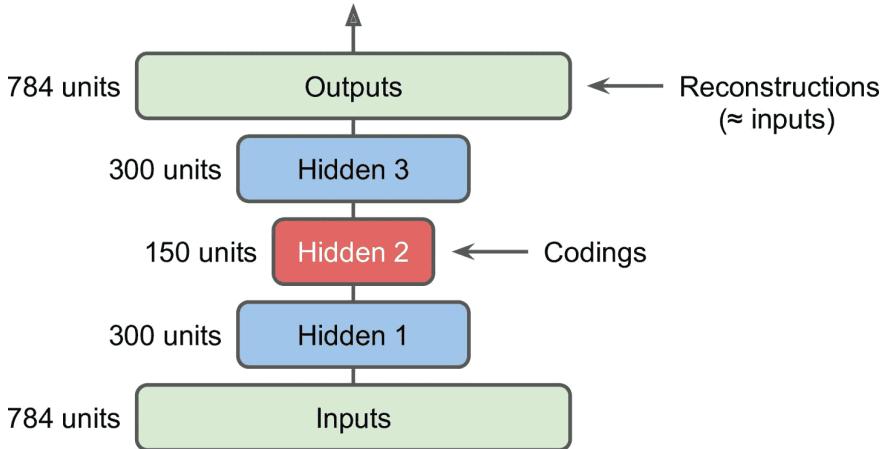


图 17-3 栈式自编码器

## 用 Keras 实现栈式自编码器

你可以像常规深度 MLP 一样实现栈式自编码器。特别是，我们在第 11 章中用于训练深度网络的技术也可以应用。例如，下面的代码使用 SELU 激活函数为 Fashion MNIST 创建了一个栈式自编码器：

```

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])
  
```

逐行看下这个代码：

- 和之前一样，自编码器包括两个子模块：编码器和解码器。
- 编码器接收  $28 \times 28$  像素的灰度图片，打平为大小等于 784 的向量，用两个紧密层来处理，两个紧密层都是用 SELU 激活函数（还可以加上 LeCun 归一初始化，但因为网络不深，效果不大）。对于每张输入图片，编码器输出的向量大小是 30。
- 解码器接收大小等于 30 的编码（编码器的输出），用两个紧密层来处理，最后的向量转换为  $28 \times 28$  的数组，使解码器的输出和编码器的输入形状相同。

- 编译时，使用二元交叉熵损失，而不是 MSE。将重建任务当做多标签分类问题：每个像素强度表示像素应该为黑色的概率。这么界定问题（而不是当做回归问题），可以使模型收敛更快。
- 最后，使用 `X_train` 既作为输入，也作为目标，来训练模型（相似的，使用 `X_valid` 既作为验证的输入也作为目标）。

## 可视化重建

确保自编码器训练得当的方式之一，是比较输入和输出：差异不应过大。画一些验证集的图片，及其重建：

```
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 展示了比较结果。



图 17-4 原始图片（上）及其重建（下）

可以认出重建，但图片有些失真。需要再训练模型一段时间，或使编码器和解码器更深，或使编码更大。但如果使网络太强大，就学不到数据中的规律。

## 可视化 Fashion MNIST 数据集

训练好栈式自编码器之后，就可以用它给数据集降维了。可视化的话，结果不像（第 8 章其它介绍的）其它降维方法那么好，但自编码器的优势是可以处理带有很多实例多个特征的大数据集。所以一个策略是利用自编码器将数据集降维到一个合理的水平，然后使用另外一个降维算法做可视化。用这个策略来可视化 Fashion MNIST。首先，使用栈式自编码器的编码器将维度降到 30，然后使用 Scikit-Learn 的 t-SNE 算法实现，将维度降到 2 并做可视化：

```
from sklearn.manifold import TSNE
X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE()
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

对数据集作图：

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
```

图 17-5 展示了结果的散点图（并展示了一些图片）。t-SNE 算法区分除了几类，比较符合图片的类别（每个类的颜色不一样）。

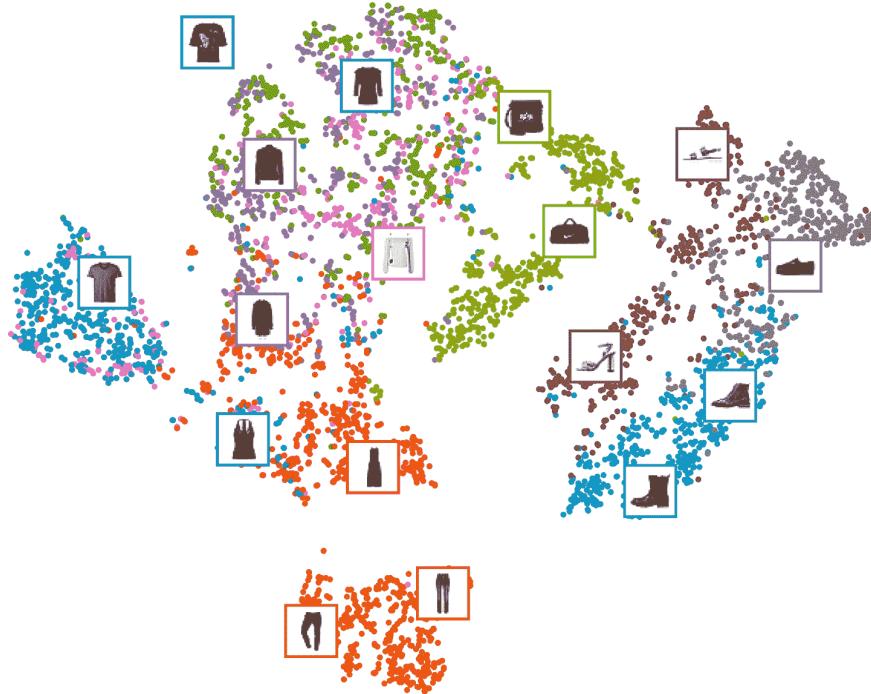


图 17-5 使用自编码器和 t-SNE 对 Fashion MNIST 做可视化

自编码器的另一个用途是无监督预训练。

## 使用栈式自编码器做无监督预训练

第 11 章讨论过，如果要处理一个复杂的监督任务，但又缺少标签数据，解决的方法之一，是找一个做相似任务的神经网络，复用它的底层。这么做就可以使用少量训练数据训练出高性能的模型，因为模型不必学习所有低层次特征；模型可以复用之前的特征探测器。

相似的，如果有一个大数据集，但大部分实例是无标签的，可以用全部数据训练一个栈式自编码器，然后使用其底层创建一个神经网络，再用有标签数据来训练。例如，图 17-6 展示了如何使用栈式自编码器来做分类的无监督预训练。当训练分类器时，如果标签数据不足，可以冻住预训练层（底层）。

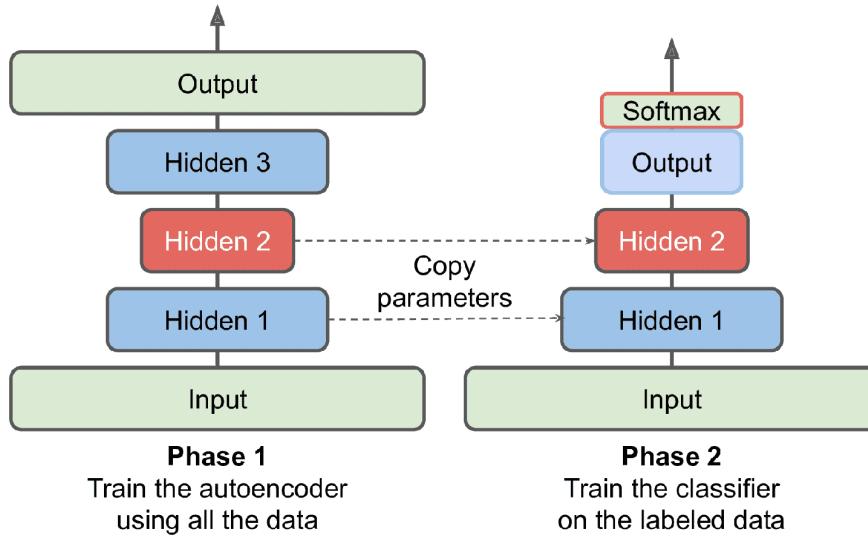


图 17-6 使用自编码器做无监督预训练

笔记：无标签数据很多，有标签数据数据很少，非常普遍。搭建一个大无便签数据集很便宜（比如，一段小脚本可以从网上下载许多图片），但是给这些图片打标签（比如，将其标签为可爱或不可爱）只有人做才靠谱。打标签又耗时又耗钱，所以人工标注实例有几千就不错了。

代码实现没有特殊之处：用所有训练数据训练自编码器，然后用编码器层创建新的神经网络（本章有练习题例子）。

接下来，看看关联权重的方法。

## 关联权重

当自编码器整齐地对称时，就像我们刚刚构建的那样，一种常用方法是将解码器层的权重与编码器层的权重相关联。这样减半了模型中的权重数量，加快了训练速度，并限制了过度拟合的风险。具体来说，如果自编码器总共具有  $N$  个层（不计输入层），并且  $w[L]$  表示第  $L$  层的连接权重（例如，层 1 是第一隐藏层，则层  $N/2$  是编码层，而层  $N$  是输出层），则解码器层权重可以简单地定义为： $w[N-L+1] = w[L]^T$ （其中  $L = 1, 2, \dots, N/2$ ）。

使用 Keras 将层的权重关联起来，先定义一个自定义层：

```
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)
```

自定义层的作用就像一个常规紧密层，但使用了另一个紧密层的权重，并且做了转置（设置 `transpose_b=True` 等同于转置第二个参数，但在 `matmul()` 运算中实时做转置更为高效）。但是，要使用自己的偏置向量。然后，创建一个新的栈式自编码器，将解码器的紧密层和编码器的紧密层关联起来：

```

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])

```

这个模型的重建误差小于前一个模型，且参数量只有一半。

## 一次训练一个自编码器

不是一次完成整个栈式自编码器的训练，而是一次训练一个浅自编码器，然后将所有这些自编码器堆叠到一个栈式自编码器（因此名称）中，通常要快得多，如图 17-7 所示。这个方法如今用的不多了，但偶尔还会撞见谈到“贪婪层级训练”的论文，所以还是看一看。

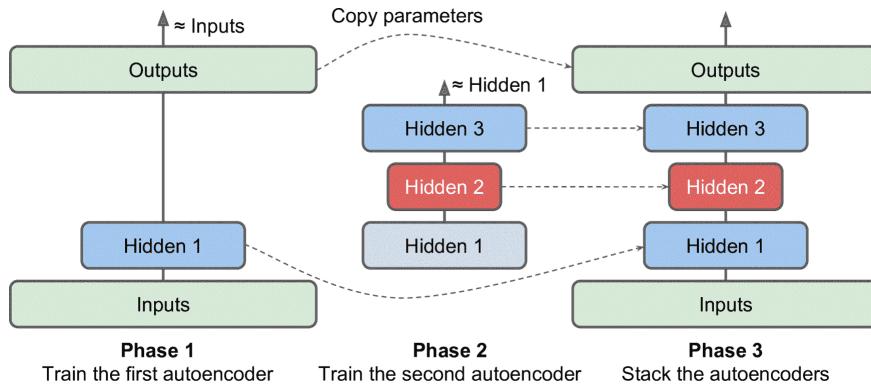


图 17-7 一次训练一个自编码器

在训练的第一阶段，第一个自编码器学习重构输入。然后，使用整个训练集训练第一个自编码器，得到一个新的（压缩过的）训练集。然后用这个数据集训练第二个自编码器。这是第二阶段的训练。最后，我们用所有这些自编码器创建一个三明治结构，见图 17-7（即，先把每个自编码器的隐藏层叠起来，再加上输出层）。这样就得到了最终的栈式自编码器（见笔记本）。我们可以用这种方式训练更多的自编码器，搭建非常深的栈式自编码器。

正如前面讨论过的，现在的一大趋势是 Geoffrey Hinton 等人在 2006 年发现的，靠这种贪婪层级方法，可以用无监督方式训练神经网络。他们还使用了受限玻尔兹曼机（RBM，见附录 E）。但在 2007 年，Yoshua Bengio 发现只用自编码器也可以达到不错的效果。在这几年间，自编码器是唯一的有效训练深度网络的方法，知道出现第 11 章介绍过的方法。

自编码器不限于紧密网络：还有卷积自编码器和循环自编码器。

## 卷积自编码器

如果处理的是图片，则前面介绍的自编码器的效果可能一般（除非图片非常小）。

第 14 章介绍过，对于图片任务，卷积神经网络比紧密网络的效果更好。所以如果想用自编码器来处理图片的话（例如，无监督预训练或降维），你需要搭建一个卷积自编码器。编码器是一个包含卷积层和池化层的常规 CNN。通常降低输入的空间维度（即，高和宽），同时增加深度（即，特征映射的数量）。解码器的工作相反（放大图片，压缩深度），要这么做的话，可以转置卷积层（或者，可以将上采样层和卷积层合并）。下面是一个卷积自编码器处理 Fashion MNIST 的例子：

```
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])
```

## 循环自编码器

如果你想用自编码器处理序列，比如对时间序列或文本无监督学习和降维，则循环神经网络要优于紧密网络。搭建循环自编码器很简单：编码器是一个序列到向量的 RNN，而解码器是向量到序列的 RNN：

```
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])
```

这个循环自编码器可以处理任意长度的序列，每个时间步有 28 个维度。这意味着，可以将 Fashion MNIST 的图片作为几行序列来处理。注意，解码器第一层用的是 RepeatVector，以保证在每个时间步将输入向量传给解码器。

我们现在已经看过了多种自编码器（基本的、栈式的、卷积的、循环的），学习了训练的方法（一次性训练或逐层训练）。还学习了两种应用：视觉可视化和无监督学习。

为了让自编码学习特征，我们限制了编码层的大小（使它处于不完整的状态）。还可以使用许多其他的限制方法，可以让编码层和输入层一样大，甚至更大，得到一个过完成的自编码器。下面就是其中一些方法。

## 降噪自编码

另一种强制自编码器学习特征的方法是为其输入添加噪声，对其进行训练以恢复原始的无噪声输入。自 20 世纪 80 年代以来，使用自编码器消除噪音的想法已经出现（例如，在 Yann LeCun 的 1987 年硕士论文中提到过）。在 2008 年的一篇论文中，帕斯卡尔文森特等人。表明自编码器也可用于特征提取。在 2010 年的一篇炉温中，Vincent 等人引入了栈式降噪自编码器。

噪音可以是添加到输入的纯高斯噪声，或者可以随机关闭输入，就像丢弃（在第 11 章介绍）。图 17-8 显示了这两种方法。

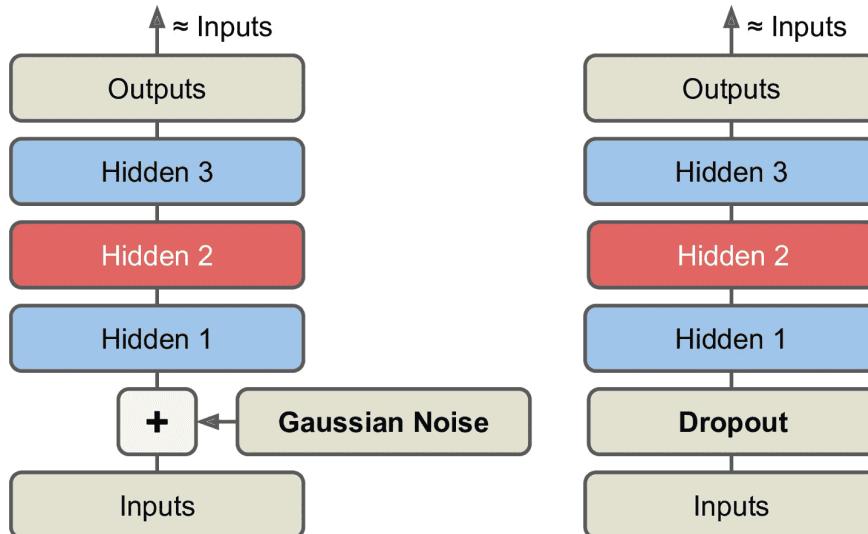


图 17-8 高斯噪音（左）和丢弃（右）的降噪自编码器

实现很简单：常规的栈式自编码器中有一个应用于输入的 Dropout 层（或使用 GaussianNoise 层）。Dropout 层只在训练中起作用（GaussianNoise 层也只在训练中起作用）：

```
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 展示了一些带有造影的图片（有一半像素被丢弃），重建图片是用基于丢弃的自编码器实现的。注意自编码器是如何猜测图片中不存在的细节的，比如四张图片的领口。

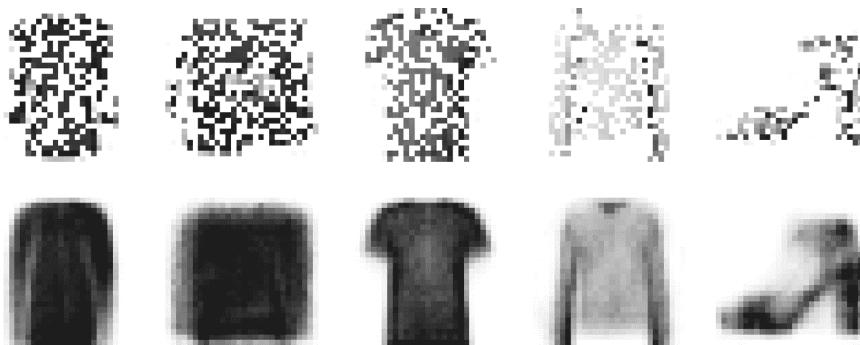


图 17-9 噪音图片（上）和重建图片（下）

## 稀疏自编码器

通常良好特征提取的另一种约束是稀疏性：通过向损失函数添加适当的项，让自编码器减少编码层中活动神经元的数量。例如，可以让编码层中平均只有 5% 的活跃神经元。这迫使自编码器将每个输入表示为少量激活的组合。因此，编码层中的每个神经元通常都会代表一个有用的特征（如果每个月只能说几个字，你会说的特别精炼）。

使用 sigmoid 激活函数可以实现这个目的。添加一个编码层（比如，有 300 个神经元），给编码层的激活函数添加  $\ell_1$  正则（解码器就是一个常规解码器）：

```
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])
```

`ActivityRegularization` 只是返回输入，但副作用是新增了训练损失，大小等于输入的绝对值之和（这个层只在训练中起作用）。等价的，可以移出 `ActivityRegularization`，并在前一层设置 `activity_regularizer=keras.regularizers.l1(1e-3)`。这项惩罚可以让神经网络产生接近 0 的编码，如果没有正确重建输入，还是会有损失，仍然会产生一些非 0 值。不使用  $\ell_2$ ，而使用  $\ell_1$ ，可以让神经网络保存最重要的编码，同时消除输入图片不需要的编码（而不是压缩所有编码）。

另一种结果更好的方法是在每次训练迭代中测量编码层的实际稀疏度，当偏移目标值，就惩罚模型。我们通过计算整个训练批次中编码层中每个神经元的平均激活来实现。批量大小不能太小，否则平均数不准确。

一旦我们对每个神经元进行平均激活，我们希望通过向损失函数添加稀疏损失来惩罚太活跃的神经元，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值为 0.3，但目标稀疏度为 0.1，那么它必须受到惩罚才能激活更少。一种方法可以简单地将平方误差  $(0.3-0.1)^2$  添加到损失函数中，但实际上更好的方法是使用 Kullback-Leibler 散度（在第 4 章中简要讨论），它具有比均方误差更强的梯度，如图 17-10 所示。

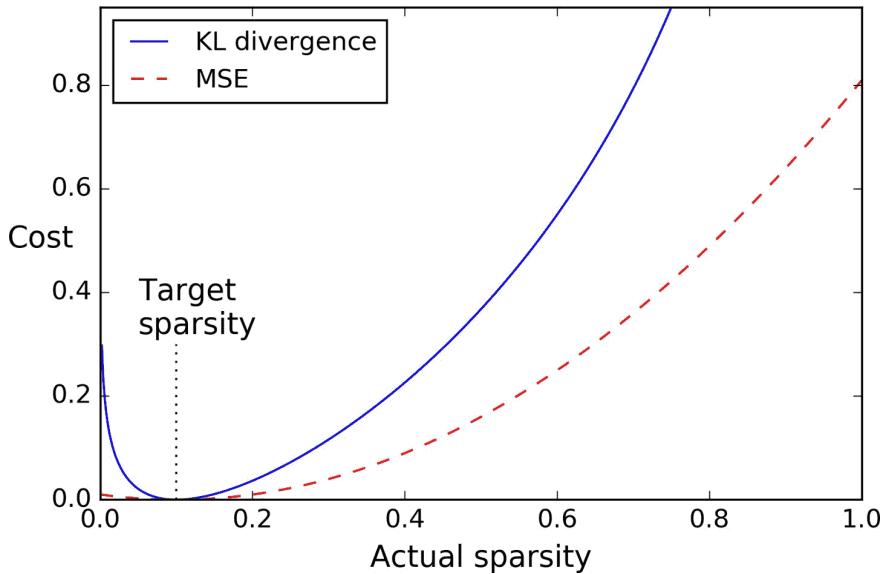


图 17-10 稀疏损失

给定两个离散的概率分布  $P$  和  $Q$ ，这些分布之间的 KL 散度，记为  $D_{\text{KL}}(P \parallel Q)$ ，可以使用公式 17-1 计算。

$$D_{\text{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

公式 17-1 Kullback–Leibler 散度

在我们的例子中，我们想要测量编码层中的神经元将激活的目标概率  $p$  与实际概率  $q$ （即，训练批次上的平均激活）之间的差异。所以 KL 散度简化为公式 17-2。

$$D_{\text{KL}}(p \parallel q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}$$

公式 17-2 目标稀疏度  $p$  和实际稀疏度  $q$  之间的 KL 散度

一旦我们已经计算了编码层中每个神经元的稀疏损失，就相加这些损失，并将结果添加到损失函数中。为了控制稀疏损失和重构损失的相对重要性，我们可以用稀疏权重超参数乘以稀疏损失。如果这个权重太高，模型会紧贴目标稀疏度，但它可能无法正确重建输入，导致模型无用。相反，如果它太低，模型将大多忽略稀疏目标，它不会学习任何有趣的功能。

现在就可以实现基于 KL 散度的稀疏自编码器了。首先，创建一个自定义正则器来实现 KL 散度正则：

```

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))

```

使用 `KLDivergenceRegularizer` 作为编码层的激活函数，创建稀疏自编码器：

```

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_regularizer=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 上训练好稀疏自编码器之后，编码层中的神经元的激活大部分接近 0（70% 的激活小于 0.1） ，所有神经元的平均值在 0.1 附近（90% 的平均激活在 0.1 和 0.2 之间）见图 17-11。

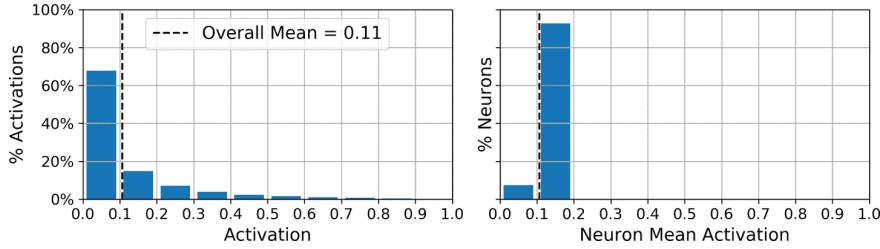


图 17-11 编码层的所有激活的分布（左）和每个神经元平均激活的分布（右）

## 变分自编码器（VAE）

Diederik Kingma 和 Max Welling 于 2013 年推出了另一类重要的自编码器，并迅速成为最受欢迎的自编码器类型之一：变分自编码器。

它与我们迄今为止讨论的所有自编码器非常不同，特别是：

- 它们是概率自编码器，意味着即使在训练之后，它们的输出部分也是偶然确定的（相对于仅在训练过程中使用随机性的自编码器的去噪）。
- 最重要的是，它们是生成自编码器，这意味着它们可以生成看起来像从训练集中采样的新实例。

这两个属性使它们与 RBM 非常相似（见附录 E），但它们更容易训练，并且取样过程更快（在 RBM 之前，您需要等待网络稳定在“热平衡”之后才能进行取样一个新的实例）。正如其名字，变分自编码器要做变分贝叶斯推断（第 9 章介绍过），这是估计变微分推断的一种有效方式。

我们来看看他们是如何工作的。图 17-12（左）显示了一个变分自编码器。当然，您可以认识到所有自编码器的基本结构，编码器后跟解码器（在本例中，它们都有两个隐藏层），但有一个转折点：不是直接为给定的输入生成编码，编码器产生平均编码  $\mu$  和标准差  $\sigma$ 。然后从平均值  $\mu$  和标准差  $\sigma$  的高斯分布随机采样实际编码。之后，解码器正常解码采样的编码。该图的右侧部分显示了一个训练实例通过此自编码器。首先，编码器产生  $\mu$  和  $\sigma$ ，随后对编码进行随机采样（注意它不是完全位于  $\mu$  处），最后对编码进行解码，最终的输出与训练实例类似。

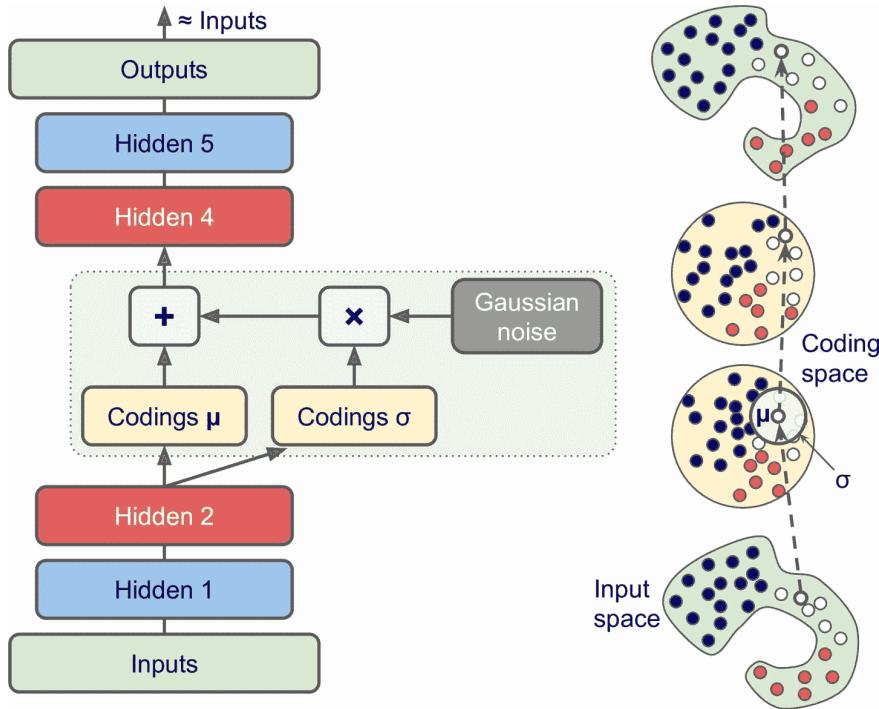


图 17-12 变分自编码器（左）和一个执行中的实例（右）

从图中可以看出，尽管输入可能具有非常复杂的分布，但变分自编码器倾向于产生编码，看起来好像它们是从简单的高斯分布采样的：在训练期间，损失函数（将在下面讨论）推动 编码在编码空间（也称为潜在空间）内逐渐迁移以占据看起来像高斯点集成的云的大致（超）球形区域。一个重要的结果是，在训练了一个变分自编码器之后，你可以很容易地生成一个新的实例：只需从高斯分布中抽取一个随机编码，对它进行解码就可以了！

再来看看损失函数。它由两部分组成。首先是通常的重建损失，推动自编码器重现其输入（我们可以使用交叉熵来解决这个问题，如前所述）。第二种是潜在的损失，推动自编码器使编码看起来像是从简单的高斯分布中采样，为此我们使用目标分布（高斯分布）与编码实际分布之间的 KL 散度。数学比以前复杂一点，特别是因为高斯噪声，它限制了可以传输到编码层的信息量（从而推动自编码器学习有用的特征）。幸好，这些方程经过简化，可以用公式 17-3 计算潜在损失：

$$\mathcal{L} = -\frac{1}{2} \sum_{i=1}^K 1 + \log (\sigma_i^2) - \sigma_i^2 - \mu_i^2$$

公式 17-3 变分自编码器的潜在损失

在这个公式中， $L$  是潜在损失， $n$  是编码维度， $\mu[i]$  和  $\sigma[i]$  是编码的第  $i$  个成分的平均值和标准差。向量  $\mu$  和  $\sigma$  是编码器的输出，见图 17-12 的左边。

一种常见的变体是训练编码器输出  $\gamma = \log(\sigma^2)$  而不是  $\sigma$ 。可以用公式 17-4 计算潜在损失。这个方法的计算更稳定，且可以加速训练。

$$\mathcal{L} = -\frac{1}{2} \sum_{i=1}^K 1 + \gamma_i - \exp(\gamma_i) - \mu_i^2$$

公式 17-4 变分自编码器的潜在损失，使用  $\gamma = \log(\sigma^2)$

给 Fashion MNIST 创建一个自编码器（见图 17-12，使用  $\gamma$  变体）。首先，需要一个自定义层从编码采样，给定  $\mu$  和  $\gamma$ ：

```
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
```

这个 `Sampling` 层接收两个输入：`mean` ( $\mu$ ) 和 `log_var` ( $\gamma$ )。使用函数 `K.random_normal()` 根据正态分布随机采样向量（形状为  $\gamma$ ）平均值为 0 标准差为 1。然后乘以  $\exp(\gamma / 2)$ （这个值等于  $\sigma$ ），最后加上  $\mu$  并返回结果。这样就能从平均值为 0 标准差为 1 的正态分布采样编码向量。

然后，创建编码器，因为模型不是完全顺序的，所以要使用函数式 API：

```
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` ( $\mu$ ) 和 `codings_log_var` ( $\gamma$ ) 的 `Dense` 层，有同样的输入（即，第二个紧密层的输出）。然后将 `codings_mean` 和 `codings_log_var` 传给 `Sampling` 层。最后，`variational_encoder` 模型有三个输出，可以用来检查 `codings_mean` 和 `codings_log_var` 的值。真正使用的是最后一个（`codings`）。下面创建解码器：

```
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])
```

对于解码器，因为是简单栈式结构，可以不使用函数式 API，而使用顺序 API。最后，创建变分自编码器：

```
_, _, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = keras.Model(inputs=[inputs], outputs=[reconstructions])
```

注意，我们忽略了编码器的前两个输出。最后，必须将潜在损失和重建损失加起来：

```
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")
```

我们首先用公式 17-4 计算批次中每个实例的潜在损失。然后计算所有实例的平均损失，然后除以，使其量纲与重建损失一致。实际上，变分自编码器的重建损失是像素重建误差的和，但当 Keras 计算 "binary\_crossentropy" 损失时，它计算的是 784 个像素的平均值，而不是和。因此，重建损失比真正要的值小 784 倍。我们可以定义一个自定义损失来计算误差和，但除以 784 更简单。

注意，这里使用了 RMSprop 优化器。最后，我们可以训练自编码器。

```
history = variational_ae.fit(X_train, X_train, epochs=50, batch_size=128,
                             validation_data=[X_valid, X_valid])
```

## 生成 Fashion MNIST 图片

接下来用上面的变分自编码器生成图片。我们要做的只是从高斯分布随机采样编码，然后做解码：

```
codings = tf.random.normal(shape=[12, codings_size])
images = variational_decoder(codings).numpy()
```

图 17-13 展示了 12 张生成的图片。

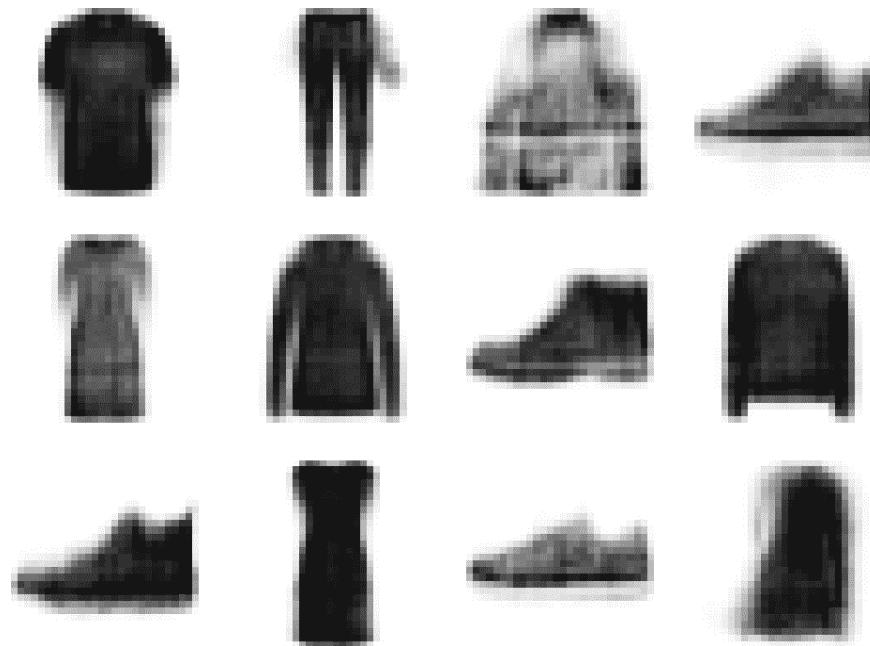


图 17-13 用变分自编码器生成的 Fashion MNIST 图片

大多数生成的图片很逼真，就是有些模糊。其它的效果一般，这是因为自编码器只学习了几分钟。经过微调和更长时间的训练，效果就能编号。

变分自编码器也可以做语义插值：不是对两张图片做像素级插值（结果就像是两张图重叠），而是在编码级插值。先用编码层运行两张图片，然后对两个编码层插值，然后解码插值编码，得到结果图片。结果就像一个常规的 Fashion MINIST 图片，但还是介于原始图之间。在接下来的代码中，将 12 个生成出来的编码，排列成  $3 \times 4$  的网格，然后用 TensorFlow 的 `tf.image.resize()` 函数，将其缩放为  $5 \times 7$ 。默认条件下，`resize()` 函数会做双线性插值，所以每两个行或列都会包含插值编码。然后用解码器生成所有图片：

```
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 展示了结果。画框的是原始图，其余是根据附近图片做出的语义插值图。注意，第 4 行第 5 列的鞋，是上下两张图的完美融合。

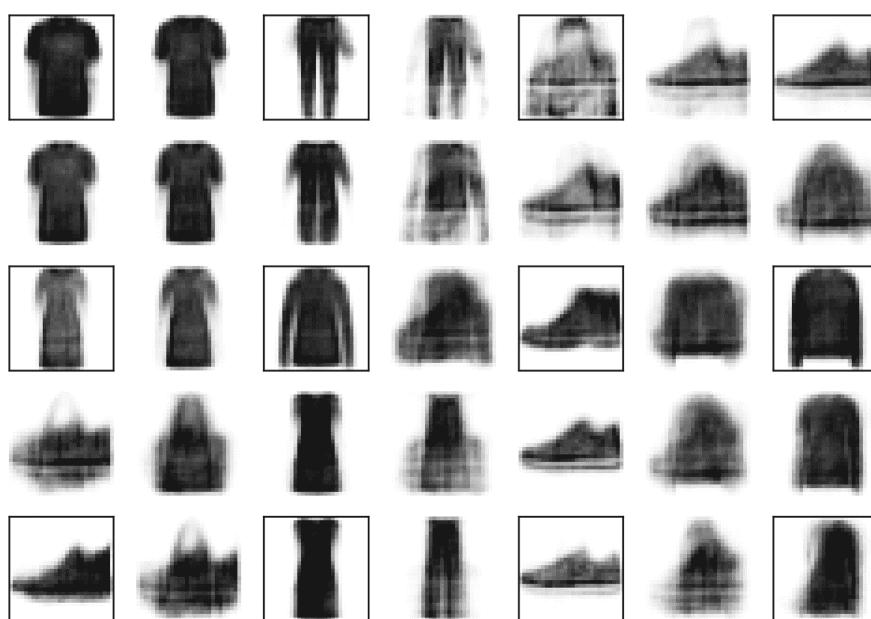


图 17-14 语义插值

变分自编码器流行几年之后，就被 GAN 超越了，后者可以生成更为真实的图片。

## 对抗生成网络 (GAN)

对抗生成网络是 Ian Goodfellow 在 2014 年的一篇[论文](#)中提出的，尽管一开始就引起了众人的兴趣，但用了几年时间才客服了训练 GAN 的一些难点。和其它伟大的想法一样，GAN 的本质很简单：让神经网络互相竞争，让其在竞争中进步。见图 17-15，GAN 包括两个神经网络：

- 生成器 使用随机分布作为输入（通常为高斯分布），并输出一些数据，比如图片。可以将随机输入作为生成文件的潜在表征（即，编码）。生成器的作用和变分自编码器中的解码器差不多，可以用同样的方式生成图片（只要输入一些高斯噪音，就能输出全新的图片）。但是，生成器的训练过程很不一样。
- 判别器 从训练集取出一张图片，判断图片是真是假。

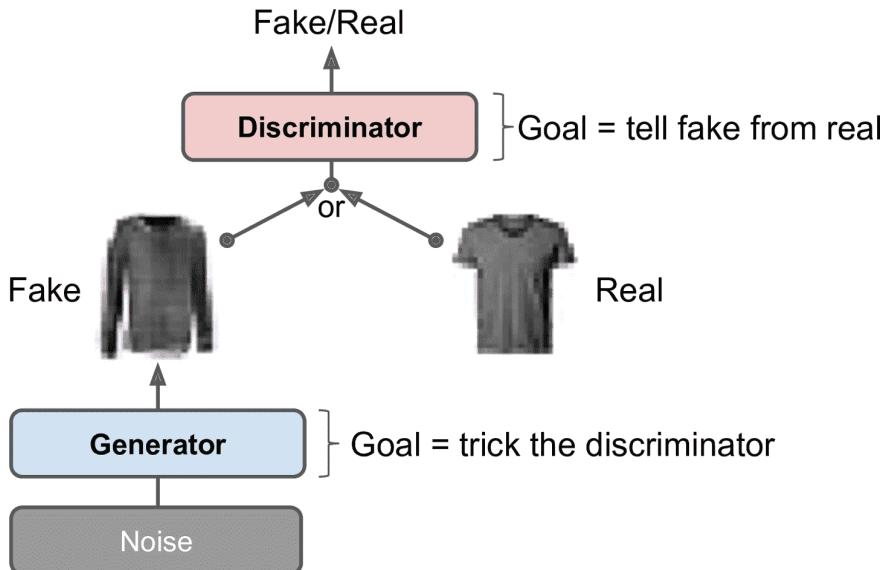


图 17-15 一个对抗生成网络

在训练中，生成器和判别器的目标正好相反：判别器判断图片的真假，生成器尽力生成看起来像真图的图片。因为 GAN 由这两个目的不同的网络组成，所以不能像常规网络那样训练。每次训练迭代分成两个阶段：

- 第一个阶段，训练判别器。从训练集取样一批真实图片，数量与假图片相同。假图片的标签设为 0，真图片的标签设为 1，判别器用这个有标签的批次训练一步，使用二元交叉熵损失。反向传播在这一阶段只优化判别器的权重。
- 第二个阶段，训练生成器。首先用生成器产生另一个批次的假图片，再用判别器来判断图片是真是假。这一次不添加真图片，但所有标签都设为 1（真）：换句话说，我们想让生成器产生可以让判别器信以为真的图片。判别器的权重在这一步是冷冻的，所以反向传播只影响生成器。

笔记：生成器看不到真图，但却逐渐生成出逼真的不骗。它只是使用了经过判别器返回的梯度。幸好，随着判别器的优化，这些二手梯度中包含的关于真图的信息也越来越多，所以生成器才能进步。

接下来为 Fashion MNIST 创建一个简单的 GAN 模型。

首先，创建生成器和判别器。生成器很像自编码器的解码器，判别器就是一个常规的二元分类器（图片作为输入，输出是包含一个神经元的紧密层，使用 sigmoid 激活函数）。对于每次训练迭代中的第二阶段，需要完整的 GAN 模型：

```

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])
  
```

然后，我们需要编译这些模型。因为判别器是一个二元分类器，我们可以使用二元交叉熵损失。生成器只能通过 GAN 训练，所以不需要编译生成器。gan 模型也是一个二元分类器，所以可以使用二元交叉熵损失。重要的，不能在第二个阶段训练判别器，所以编译模型之前，使其不可训练：

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

笔记：Keras 只有在编译模型时才会考虑 trainable 属性，所以运行这段代码后，如果调用 fit() 方法或 train\_on\_batch() 方法， discriminator 就是可训练的了。但在 gan 模型上调用这些方法，判别器是不可训练的。

因为训练循环是非常规的，我们不能使用常规的 fit() 方法。但我们可以写一个自定义的训练循环。要这么做，需要先创建一个 Dataset 迭代这些图片：

```
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

现在就可以来写训练循环了。用 train\_gan() 函数来包装：

```
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)
```

和前面讨论的一样，每次迭代都有两个阶段：

- 在第一阶段，向生成器输入高斯噪音来生成假图片，然后再补充同等数量的真图片。假图片的目标 y1 设为 0，真图片的目标 y1 设为 1。然后用这个批次训练判别器。注意，将判别器的 trainable 属性设为 True：这是为了避免 Keras 检查到现在是 False 而在训练时为 True，显示警告。
- 在第二阶段，向 GAN 输入一些高斯噪音。它的生成器会开始假图片，然后判别器会判断其真假。我们希望判别器判断图片是真的，所以 y2 设为 1。注意，为了避免警告，将 trainable 属性设为 False。

这样就好了！如果展示生成出来的图片（见图 17-16），可以看到在第一个周期的后期，图片看起来已经接近 Fashion MNIST 的图片了。



图 17-16 GAN 训练一个周期后，生成的图片

不过，再怎么训练，图片的质量并没有提升，还发现在有的周期 GAN 完全忘了学到了什么。为什么会这样？貌似训练 GAN 很有挑战。接下来看看原因。

## 训练 GAN 的难点

在训练中，生成器和判别器不断试图超越对方，这是一个零和博弈。随着训练的进行，可能会达成博弈学家称为纳什均衡的状态：每个选手都不改变策略，并认为对方也不会改变策略。例如，当所有司机都靠左行驶时，就达到了纳什均衡：没有司机会选择换边。当然，也有第二种可能：每个人都靠右行驶。不同的初始状态和动力学会导致不同的均衡。在这个例子中，达到均衡时，只有一种最优策略，但纳什均衡包括多种竞争策略（比如，捕食者追逐猎物，猎物试图逃跑，两者都要改变策略）。

如何将博弈论应用到 GAN 上呢？论文作者证明，GAN 只能达到一种均衡状态：生成器产生完美的真实图片，同时让判别器来判断（50% 为真，50% 为假）。这是件好事：看起来只要训练 GAN 足够久，就会达到均衡，获得完美的生成器。不过，并没有这么简单：没有人能保证一定能达到均衡。

最大的困难是模式坍塌：生成器的输出逐渐变得不那么丰富。为什么会这样？假设生成器产生的鞋子图片比其它类的图片更让人信服，假鞋子图片就会更多的欺骗判别器，就会导致生成更多的鞋子图片。逐渐的，生成器会忘掉如何生成其它类的图片。同时，判别器唯一能看到的就是鞋子图片，所以判别器也会忘掉如何判断其它类的图片。最终，当判别器想要区分假鞋和真鞋时，生成器会被迫生成其它类。生成器可能变成善于衬衫，而忘了鞋子，判别器也会发生同样的转变。GAN 会逐渐在一些类上循环，从而对哪一类都不擅长。

另外，因为生成器和判别器不断试探对方，它们的参数可能不断摇摆。训练可能一开始正常，但因为不稳定性，会突然发散。又因为多种因素可能会影响动力学，GAN 会对超参数特别敏感：微调超参数会特别花费时间。

这些问题自从 2014 年就一直困扰着人们：人们发表了许多论文，一些论文提出新的损失函数、或稳定化训练的手段、或避免模式坍塌。例如，经验接力：将生成器在每个迭代产生的图片存储在接力缓存中（逐次丢弃旧的生成图），使用真实图片和从缓存中取出的图片训练判别器。这样可以降低判别器对生成器的最后一个输出过拟合的几率。另外一个方法是小批次判别：测量批次中图片的相似度，然后将数据传给判别器，判别器就可以删掉缺乏散度的假图片。这可以鼓励生成器产生更多类的图片，避免模式坍塌。

总而言之，这是一个非常活跃的研究领域，GAN 的动力学仍然没有彻底搞清。好消息是人们已经取得了一定成果，效果不俗。接下来看看一些成功的架构，从深度卷积 GAN 开始，这是几年前的前沿成果。然后再看两个新近的（更复杂的）架构。

## 深度卷积 GAN

2014 年的原始 GAN 论文是用卷积层实验的，但只用来生成小图片。不久之后，许多人使用深度卷积网络为大图片创建 GAN。过程艰难，因为训练不稳定，但最终 Alec Radford 等人试验了许多不同的架构和超参数，在 2015 年取得了成功。他们将最终架构称为深度卷积 GAN (DCGAN)。他们提出的搭建稳定卷积 GAN 的建议如下：

- (判别器中) 用卷积步长 (strided convolutions) 、(生成器中) 用转置卷积，替换池化层。
- 生成器和判别器都使用批归一化，除了生成器的输出层和判别器的输入层。
- 去除深层架构中的全连接隐藏层。
- 生成器的输出层使用 tanh 激活，其它层使用 ReLU 激活。
- 判别器的所有层使用 leaky ReLU 激活。

这些建议在许多任务中有效，但存在例外，所以你还是需要尝试不同的超参数（事实上，改变随机种子，再训练模型，可能就成功了）。例如，下面是一个小型的 DCGAN，在 Fashion MNIST 上效果不错：

```
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])
```

生成器的编码大小为 100，将其投影到 6272 个维度上 ( $7 \times 7 \times 128$ )，将结果变形为  $7 \times 7 \times 128$  的张量。这个张量经过批归一化，然后输入给步长为 2 的转置卷积层，从  $7 \times 7$  上采样为  $14 \times 14$ ，深度从 128 降到 64。结果再做一次批归一化，传给另一个步长为 2 的转置卷积层，从  $7 \times 7$  上采样为  $14 \times 14$ ，深度从 64 降到 1。这个层使用 tanh 激活函数，输出范围是 -1 到 1。因为这个原因，在训练 GAN 之前，需要收缩训练集到相同的范围。还需要变形，加上通道维度：

```
X_train = X_train.reshape(-1, 28, 28, 1) * 2 / . - 1 / . # 变形和收缩
```

判别器看起来很像英语二元分类的常规 CNN，除了使用的不是最大池化层降采样图片，而是使用卷积步长。另外，使用的激活函数是 leaky ReLU。

总之，我们遵守了 DCGAN 的建议，除了将判别器中的 BatchNormalization 替换成了 Dropout 层（否则训练会变得不稳定），生成器的 ReLU 替换为 SELU。你可以随意调整这个架构：可以看到对超参数（特别是学习率）的敏感度。

最后，要创建数据集，然后编译训练模型，使用和之前一样的代码。经过 50 个周期的训练，生成器的图片见图 17-17。还是不怎么完美，但一些图片已经很逼真了。



图 17-17 DCGAN 经过 50 个周期的训练，生成的图片

如果扩大这个架构，然后用更大的面部数据集训练，可以得到相当逼真的图片。事实上，DCGAN 可以学习到许多有意义的潜在表征，见图 17-18：从生成的诸多图片中手动选取了九张（左上），包括三张戴眼镜的男性，三张不戴眼镜的男性，和三张不戴眼镜的女性。对于每一类，对其编码做平均，用平均的结果再生成一张图片（放在下方）。总之，下方的图片是上方图片的平均。但不是简单的像素平均，而是潜在空间的平均，所以看起来仍是正常的人脸。如果用戴眼镜的男性，减去不戴眼镜的男性，加上不戴眼镜的女性，使用平均编码，就得到了右边  $3 \times 3$  网格的正中的图片，一个戴眼镜的女性！其它八张是添加了一些噪声的结果，用于解释 DCGAN 的语义插值能力。可以用人脸做加减法就像科幻小说一样！

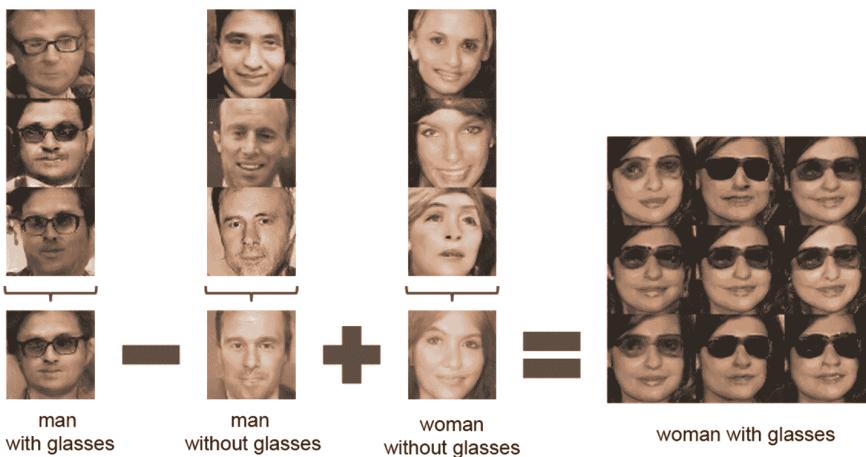


图 17-18 面部的向量运算（来自 DCGAN 论文的图 7）

提示：如果将图片的类作为另一个输入，输入给生成器和判别器，它们都能学到每个类的样子，你就可以控制生成器产生图片的类。这被称为条件GAN (CGAN)。

但是，DCGAN 并不完美。比如，当你使用 DCGAN 生成非常大的图片时，通常是局部逼真，但整体不协调（比如 T 恤的一个袖子比另一个长很多）。如何处理这种问题呢？

## GAN 的渐进式变大

Nvidia 研究员 Tero Karras 等人在 2018 年发表了一篇[论文](#)，提出了一个重要方法：他们建议在训练时，先从生成小图片开始，然后逐步给生成器和判别器添加卷积层，生成越来越大的图片

( $4 \times 4, 8 \times 8, 16 \times 16, \dots, 512 \times 512, 1,024 \times 1,024$ )。这个方法和栈式自编码器的贪婪层级训练很像。余下的层添加到生成器的末端和判别器的前端，之前训练好的层仍然可训练。

例如，当生成器的输出从  $4 \times 4$  变为  $4 \times 4$  时（见图 17-19），在现有的卷积层上加上一个上采样层（使用近邻过滤），使其输出  $4 \times 4$  的特征映射。再接着传给一个新的卷积层（使用 same 填充，步长为 1，输出为  $8 \times 8$ ）。接着是一个新的输出卷积层：这是一个常规卷积层，核大小为 1，将输出投影到定好的颜色通道上（比如 3）。为了避免破坏第一个训练好的卷积层的权重，最后的输出是原始输出层（现在的输出是  $4 \times 4$  的特征映射）的权重之和。新输出的权重是  $\alpha$ ，原始输出的权重是  $1 - \alpha$ ， $\alpha$  逐渐从 0 变为 1。换句话说，新的卷积层（图 17-19 中的虚线）是淡入的，而原始输出层淡出。向判别器（跟着平均池化层做降采样）添加新卷积层时，也是用相似的淡入淡出方法。

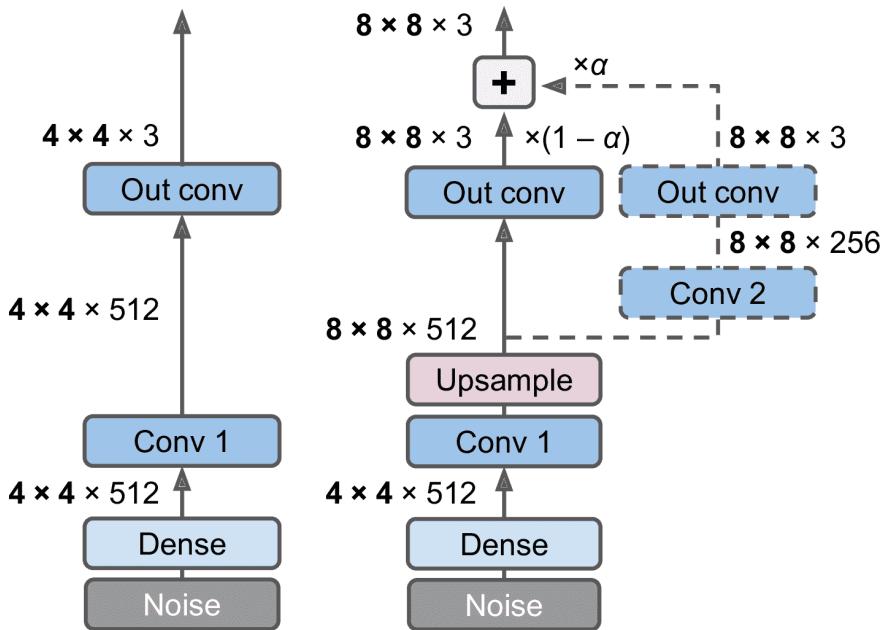


图 17-19 GAN 的渐进式变大：GAN 生成器输出  $4 \times 4$  的彩色图片（左）；将其扩展为  $4 \times 4$  的图片（右）

这篇文章还提出了一些其它的方法，用于提高输出的散度（避免模式坍塌），使训练更稳定：

- 小批次标准差层

添加在判别器的靠近末端的位置。对于输入的每个位置，计算批次 `( s = tf.math.reduce_std(inputs, axis=[0, -1]) )` 中，所有通道所有实例的标准差。接着，这些标准差对所有点做平均，得到一个单值 `( v = tf.reduce_mean(s) )`。最后，给批次中的每个实例添加一个额外的特征映射，填入计算得到的单值 `( tf.concat([inputs, tf.fill([batch_size, height, width, 1], v)], axis=-1)`。这样又什么用呢？如果生成器产生的图片没有什么偏差，则判别器的特征映射的标准差会特别小。有了这个层，判别器就可以做出判断。可以让生成器产生高散度的输出，降低模式坍塌的风险。

- 相等的学习率

使用一个简单的高斯分布（平均值为 0，标准差为 1）初始化权重，而不使用 He 初始化。但是，权重在运行时（即，每次执行层）会变小：会除以  $\sqrt{2/n\_inputs}$ ，`n_inputs` 是层的输入数。这篇论文说，使用这个方法可以显著提升 GAN 使用 RMSProp、Adam 和其它适应梯度优化器时的性能。事实上，这些优化器用估计标准差（见第 11 章）归一化了梯度更新，所以有较大动态范围的参数需要更长时间训练，而较小动态范围的参数可能更新过快，会导致不稳定。通过缩放模型的部分参数，可以保证参数的动态范围在训练过程中一致，可以用相同的速度学习。这样既加速了训练，也做到了稳定。

- 像素级归一化层

生成器的每个卷积层之后添加。它能归一化每个激活函数，基于相同图片相同位置的所有激活，而且跨通道（除以平均激活平方的平方根）。在 TensorFlow 的代码中，这是 `inputs / tf.sqrt(tf.reduce_mean(tf.square(X), axis=-1, keepdims=True) + 1e-8)`（平滑项 `1e-8` 用于避免零除）。这种方法可以避免生成器和判别器的过分竞争导致的激活爆炸。

使用所有这些方法，作者制作出了[非常逼真的人脸图片](#)。但如何给“逼真”下定义呢？GAN 的评估时一大挑战：尽管可以自动评估生成图片的散度，判断质量要棘手和主观的多。一种方法是让人来打分，但成本高且耗时。因此作者建议比较生成图和训练图的局部图片结构，在各个层次比较。这个想法使他们创造出了另一个突破性的成果：StyleGAN。

## StyleGAN

相同的 Nvidia 团队在 2018 年的一篇[论文](#)中提出了高性能的高清图片生成架构，StyleGAN。作者在生成器中使用了风格迁移方法，使生成的图片和训练图片在每个层次，都有相同的局部结构，极大提升了图片的质量。判别器和损失函数没有变动，只修改了生成器。StyleGAN 包含两个网络（见图 17-20）：

- 映射网络

一个八层的 MLP，将潜在表征 `z`（即，编码）映射为向量 `w`。向量然后传给仿射变换（即，没有激活函数的紧密层，用图 17-20 中的框 A 表示），输出许多向量。这些向量在不同级别控制着生成图片的风格，从细粒度纹理（比如，头发颜色）到高级特征（比如，成人或孩子）。总而言之，映射网络将编码变为许多风格向量。

- 合成网络

负责生成图片。它有一个固定的学好的输入（这个输入在训练之后是不变的，但在训练中被反向传播更新）。和之前一样，合成网络使用多个卷积核上采样层处理输入，但有两处不同：首先，输入和所有卷积层的输出（在激活函数之前）都添加了噪音。第二，每个噪音层的后面是一个适应实例归一化

(AdaIN) 层：它独立标准化每个特征映射（减去平均值，除以标准差），然后使用风格向量确定每个特征映射的缩放和偏移（风格向量对每个特征映射包含一个缩放和一个偏置项）。

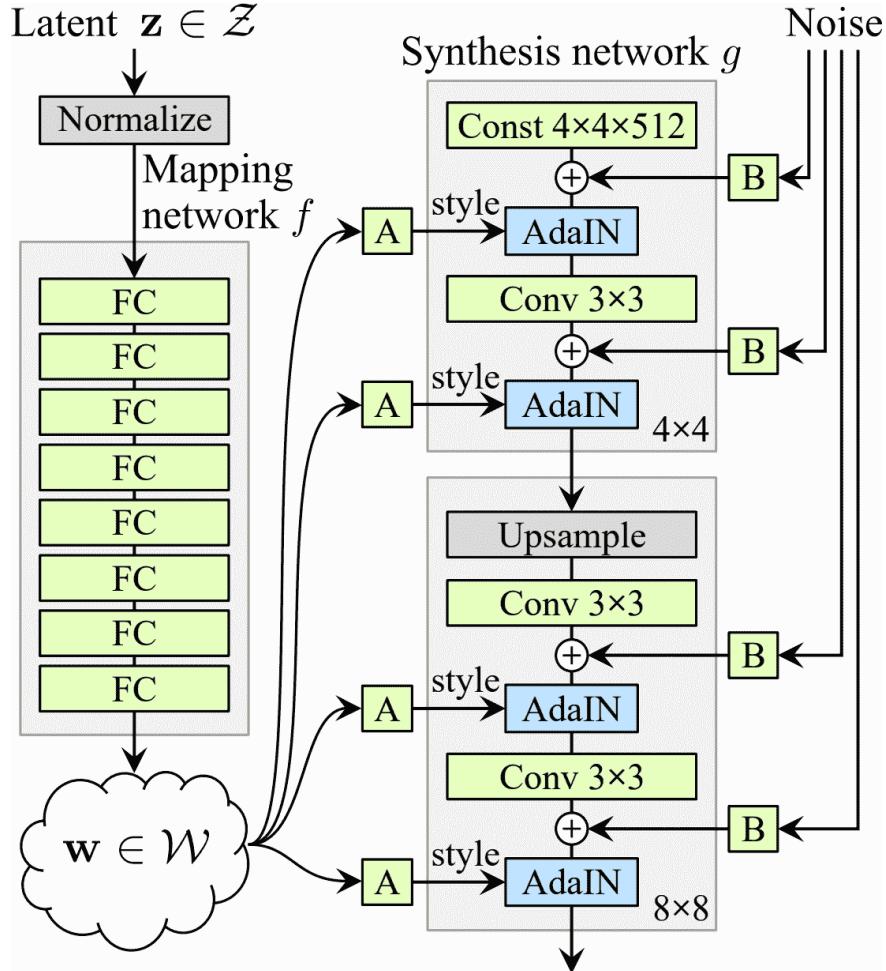


图 17-20 StyleGAN 的生成器架构 (StyleGAN 论文的图 1 的一部分)

在编码层独立添加噪音非常重要。图片的一些部分是很随机的，比如雀斑和头发的确切位置。在早期的 GAN 中，这个随机性要么来自编码，要么是生成器的一些伪噪音。如果来自编码，意味着生成器要用编码的很重要的一部分来存储噪音：这样会非常浪费。另外，噪音会在网络中流动，直到生成器的最后一层：这是一种没有必要的约束，会显著减慢训练。最后，因为噪音的存在，会出现一些视觉伪影。如果是生成器来制造伪噪音，噪音可能不够真实，造成更多的视觉伪影。另外，用生成器的一部分权重来生成伪噪音，这也是一种浪费。通过添加额外的噪音输入，可以避免所有这些问题；GAN 可以利用噪音，给图片的每个部分添加随机量。

添加的噪音在每个级别都不同。每个噪音输入包含一个单独的包含高斯噪音的特征映射，广播到所有特征映射上（给定级别），然后在添加前用每个特征的缩放因子缩放（这是图 17-20 的框 B）。

最后，StyleGAN 使用了一种称为混合正则（或风格混合）的方法，生成图的一定比例使用两个编码来生成。特别的，编码  $c[1]$  和  $c[2]$  发送给映射网络，得到两个风格向量  $w[1]$  和  $w[2]$ 。然后合成网络使用风格  $w[1]$  生成第一级，用  $w[2]$  生成其余的。级的选取是随机的。这可以防止模型认为临近的级是有关联的，会导致 GAN 的局部性，每个风格向量只会影响生成图的有限数量的特性。

GAN 的种类如此之多，用一本书才能介绍全。希望这里的内容可以告诉你 GAN 的主要观点，以及继续学习的动力。如果你对数学概念掌握不好，可以看看网上的博客。然后就可以创建自己的 GAN 了，如果一开始碰到问题，千万别气馁：有问题是很正常的，通常要好好练习，才能掌握好。如果对实现细节不明白，可以看看别人的 Keras 和 TensorFlow 实现。事实上，如果你只是想快速获得一些经验的结果，可以使用预训练模型（例如，存在适用于 Keras 的 StyleGAN 预训练模型）。

下一章会介绍深度学习的另一领域：深度强化学习。

## 练习

1. 自编码器主要用来做什么？
2. 假设你想训练一个分类器，有许多未打标签的训练数据，只有一千多打了标签的数据。如何使用自编码器来解决这个问题？
3. 如果自编码器完美重建了输入，它一定是个好的自编码器吗？如何评估自编码器的表现？
4. 自编码器的欠完成和过完成是什么？超欠完成的风险是什么？过完成的风险是什么？
5. 如何将栈式自编码器的权重连起来？这么做的意义是什么？
6. 什么是生成式模型？可以举出生成式自编码器的例子吗？
7. GAN 是什么？可以用于什么任务？
8. 训练 GAN 的难点是什么？
9. 用去噪音自编码器预训练一个图片分类器。可以使用 MNIST，或是更复杂的图片数据集，比如 CIFAR10。不管用的是什么数据集，遵循下面的步骤：
  - 将数据集分成训练集和测试集。在完整训练集上，训练一个深度去噪音自编码器。
  - 检查图片正确重建了。可视化最激活编码层神经元的图片。
  - 搭建一个分类 DNN，使用自编码器的浅层。用训练集中的 500 张图片来训练。然后判断预训练是否提升了性能？
10. 用刚才选择的数据集，训练一个变分自编码器。用它来生成图片。或者，用一个没有标签的数据集，来生成新样本。
11. 训练一个 DCGAN 来处理选择的数据集，生成新图片。添加经验接力，看看它是否有作用。再将其变为一个条件 GAN，可以控制生成的类。

参考答案见附录 A。

## 十八、强化学习

译者：[@SeanCheney](#)

强化学习（RL）如今是机器学习的一大令人激动的领域，也是最老的领域之一。自从 1950 年被发明出来后，它被用于一些有趣的应用，尤其是在游戏（例如 TD-Gammon，一个西洋双陆棋程序）和机器控制领域，但是从未弄出什么大新闻。直到 2013 年一个革命性的发展：来自英国的研究者发起了 Deepmind 项目，这个项目可以学习去玩任何从头开始的 Atari 游戏，在多数游戏中，比人类玩的还好，它仅使用像素作为输入而没有使用游戏规则的任何先验知识。这是一系列令人惊叹的壮举中的第一个，并在 2016 年 3 月以他们的系统 AlphaGo 战胜了世界围棋冠军李世石而告终。从未有程序能勉强打败这个游戏的大师，更不用说世界冠军了。今天，RL 的整个领域正在沸腾着新的想法，其都具有广泛的应用范围。DeepMind 在 2014 被谷歌以超过 5 亿美元收购。

DeepMind 是怎么做到的呢？事后看来，原理似乎相当简单：他们将深度学习运用到强化学习领域，结果却超越了他们最疯狂的设想。在本章中，我们将首先解释强化学习是什么，以及它擅长于什么，然后我们将介绍两个在深度强化学习领域最重要的技术：策略梯度和深度 Q 网络（DQN），包括讨论马尔可夫决策过程

(MDP)。我们将使用这些技术来训练一个模型来平衡移动车上的杆子；然后，我会介绍 TF-Agents 库，这个库利用先进的算法，可以大大简化创建 RL 系统，然后我们会用这个系统来玩 Breakout，一个著名的 Atari 游戏。本章最后，会介绍强化学习领域的最新进展。

### 学习优化奖励

在强化学习中，智能体在环境（environment）中观察（observation）并且做出决策（action），随后它会得到奖励（reward）。它的目标是去学习如何行动能最大化期望奖励。如果你不在意拟人化的话，可以认为正奖励是愉快，负奖励是痛苦（这样的话奖励一词就有点误导了）。总之，智能体在环境中行动，并且在实验和错误中去学习最大化它的愉快，最小化它的痛苦。

这是一个相当广泛的设置，可以适用于各种各样的任务。以下是几个例子（详见图 16-1）：

1. 智能体可以是控制一个机器人的程序。在此例中，环境就是真实的世界，智能体通过许多的传感器例如摄像机或者触觉传感器来观察，它可以通过给电机发送信号来行动。它可以被编程设置为如果到达了目的地就得到正奖励，如果浪费时间，或者走错方向，或摔倒了就得到负奖励。
2. 智能体可以是控制 Ms.Pac-Man 的程序。在此例中，环境是 Atari 游戏的模拟器，行为是 9 个操纵杆位（上下左右中间等等），观察是屏幕，回报就是游戏点数。
3. 相似地，智能体也可以是棋盘游戏的程序，例如围棋。
4. 智能体也可以不用去控制一个实体（或虚拟的）去移动。例如它可以是一个智能恒温器，当它调整到目标温度以节能时会得到正奖励，当人们需要自己去调节温度时它会得到负奖励，所以智能体必须学会预见人们的需要。

5. 智能体也可以去观测股票市场价格以实时决定买卖。奖励的依据为挣钱或者赔钱。

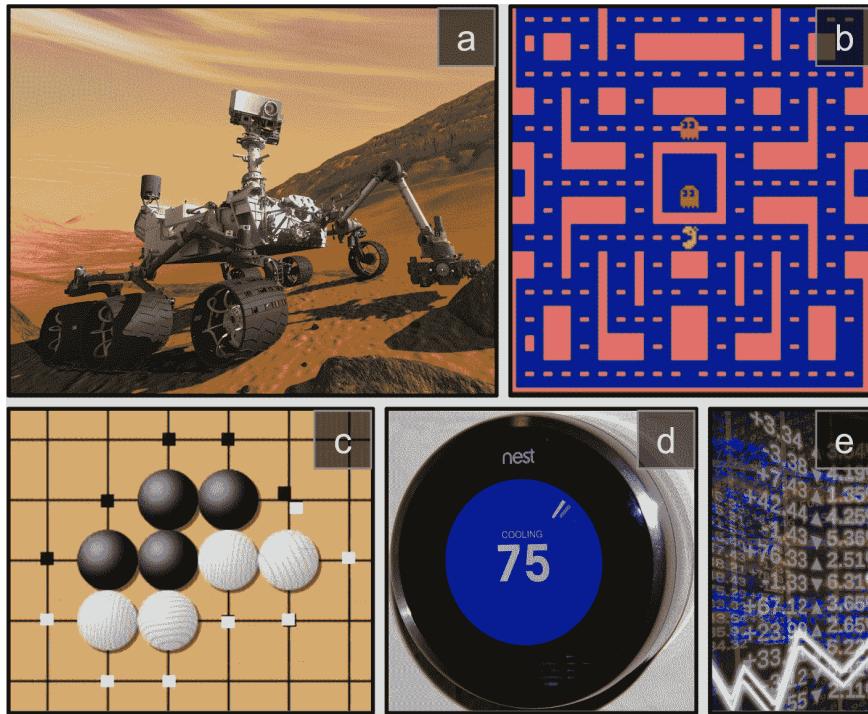


图 18-1 强化学习案例：(a) 行走机器人，(b) Ms.Pac-Man 游戏，(c) 围棋玩家，(d) 恒温器，(e) 自动交易员

其实没有正奖励也是可以的，例如智能体在迷宫内移动，它每分每秒都得到一个负奖励，所以它要尽可能快的找到出口！还有很多适合强化学习的领域，例如自动驾驶汽车，推荐系统，在网页上放广告，或者控制一个图像分类系统让它明白它应该关注于什么。

## 策略搜索

智能体用于改变行为的算法称为策略。例如，策略可以是一个把观测当输入，行为当做输出的神经网络（见图 16-2）。

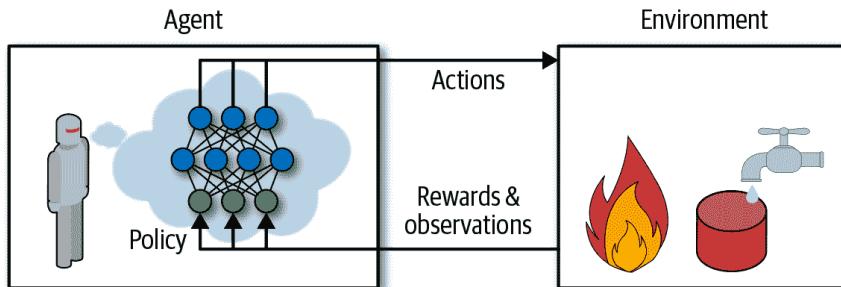


图 18-2 用神经网络策略做加强学习

这个策略可以是你能想到的任何算法，它甚至可以是非确定性的。事实上，在某些任务中，策略根本不必观察环境！举个例子，例如，考虑一个真空吸尘器，它的奖励是在 30 分钟内捡起的灰尘数量。它的策略可以是每秒以概率  $p$  向前移动，或者以概率  $1-p$  随机地向左或向右旋转。旋转角度将是  $-r$  和  $+r$  之间的随机角度，因

为该策略涉及一些随机性，所以称为随机策略。机器人将有一个不确定的轨迹，它保证它最终会到达任何可以到达的地方，并捡起所有的灰尘。问题是：30 分钟后它会捡起多少灰尘？

怎么训练这样的机器人？你能调整的策略参数只有两个：概率  $p$  和角度范围  $r$ 。一个想法是这些参数尝试许多不同的值，并选择执行最佳的组合（见图 18-3）。这是一个策略搜索的例子，在这种情况下使用暴力方法。然而，当策略空间太大（通常情况下），以这样的方式找到一组好的参数就像是大海捞针。

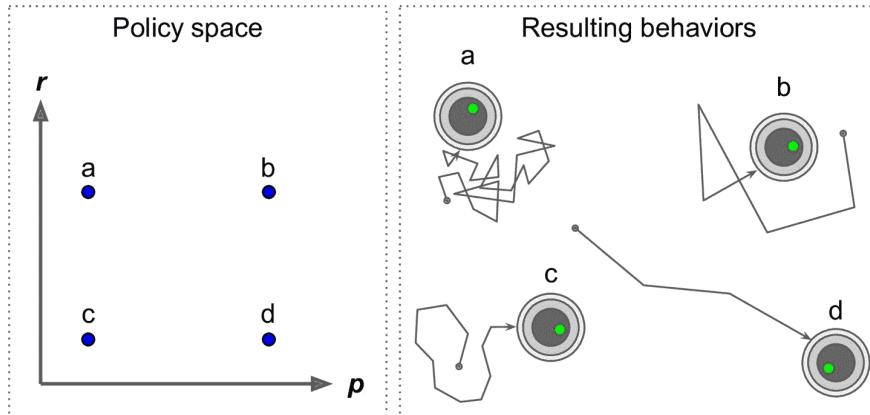


图 18-3 策略空间中的四个点以及机器人的对应行为

另一种搜寻策略空间的方法是遗传算法。例如你可以随机创造一个包含 100 个策略的第一代基因，随后杀死 80 个糟糕的策略，随后让 20 个幸存策略繁衍 4 代。一个后代只是它父辈基因的复制品加上一些随机变异。幸存的策略加上他们的后代共同构成了第二代。你可以继续以这种方式迭代，直到找到一个好的策略。

另一种方法是使用优化技术，通过评估奖励关于策略参数的梯度，然后通过跟随梯度向更高的奖励（梯度上升）调整这些参数。这种方法被称为策略梯度（policy gradient, PG），我们将在本章后面详细讨论。例如，回到真空吸尘器机器人，你可以稍微增加概率  $P$  并评估这是否增加了机器人在 30 分钟内拾起的灰尘的量；如果确实增加了，就相对应增加  $p$ ，否则减少  $p$ 。我们将使用 Tensorflow 来实现 PG 算法，但是在这之前我们需要为智能体创造一个生存的环境，所以现在是介绍 OpenAI Gym 的时候了。

## OpenAI Gym 介绍

强化学习的一个挑战是，为了训练对象，首先需要有一个工作环境。如果你想设计一个可以学习 Atari 游戏的程序，你需要一个 Atari 游戏模拟器。如果你想设计一个步行机器人，那么环境就是真实的世界，你可以直接在这个环境中训练你的机器人，但是这有其局限性：如果机器人从悬崖上掉下来，你不能仅仅点击“撤消”。你也不能加快时间；增加更多的计算能力不会让机器人移动得更快。一般来说，同时训练 1000 个机器人是非常昂贵的。简而言之，训练在现实世界中是困难和缓慢的，所以你通常需要一个模拟环境，至少需要引导训练。例如，你可以使用 PyBullet 或 MuJoCo 来做 3D 物理模拟。

OpenAI Gym 是一个工具包，它提供各种各样的模拟环境（Atari 游戏，棋盘游戏，2D 和 3D 物理模拟等等），所以你可以训练，比较，或开发新的 RL 算法。

安装之前，如果你是用虚拟环境创建的独立的环境，需要先激活：

```
$ cd $ML_PATH          # 工作目录 (e.g., $HOME/ml)
$ source my_env/bin/activate # Linux or MacOS
$ .\my_env\Scripts\activate # Windows
```

接下来安装 OpenAI gym。可通过 pip 安装：

```
$ python3 -m pip install --upgrade gym
```

取决于系统，你可能还要安装 Mesa OpenGL Utility (GLU) 库（比如，在 Ubuntu 18.04 上，你需要运行 `apt install libglu1-mesa`）。这个库用来渲染第一个环境。接着，打开一个 Python 终端或 Jupyter 笔记本，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])
```

这里创建了一个 CartPole 环境。这是一个 2D 模拟，其中推车可以被左右加速，以平衡放置在它上面的平衡杆（见图 18-4）。你可以用 `gym.envs.registry.all()` 获得所有可用的环境。在创建环境之后，需要使用 `reset()` 初始化。这会返回第一个观察结果。观察取决于环境的类型。对于 CartPole 环境，每个观测是包含四个浮点数的 1D Numpy 向量：这些浮点数代表推车的水平位置（0.0 为中心）、速度（正是右）、杆的角度（0.0 为垂直）及角速度（正为顺时针）。

用 `render()` 方法展示环境（见图 18-4）。在 Windows 上，这需要安装 X Server，比如 VcXsrv 或 Xming：

```
>>> env.render()
True
```

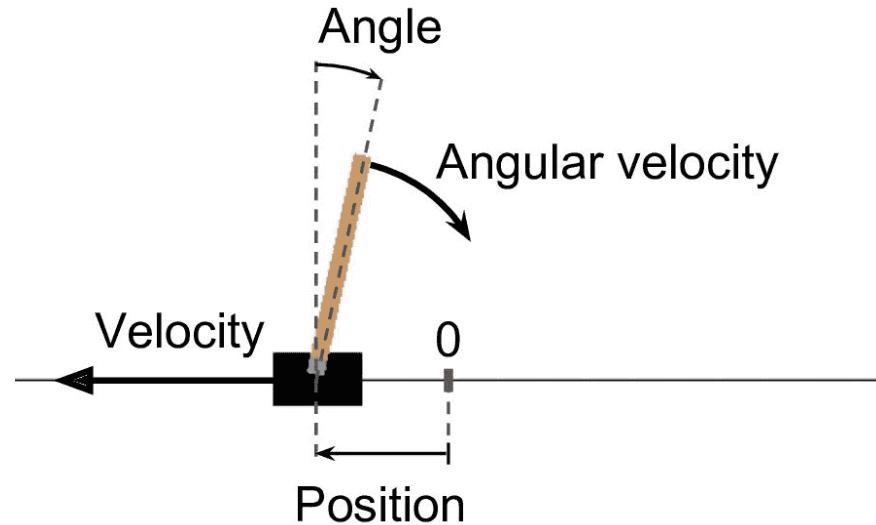


图 18-4 CartPole 环境

提示：如果你在使用无头服务器（即，没有显示器），比如云上的虚拟机，渲染就会失败。解决的唯一方法是使用假 X server，比如 Xvfb 或 Xdummy。例如，装好 Xvfb 之后（Ubuntu 或 Debian 上运行 `apt install xvfb`），用这条命令启动 Python：`xvfb-run -s "-screen 0 1400x900x24" python3`。或者，安装 Xvfb 和 `pyvirtualdisplay` 库（这个库包装了 Xvfb），在程序启动处运行 `pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()`。

如果你想让 `render()` 让图像以一个 Numpy 数组格式返回，可以将 `mode` 参数设置为 `rgb_array`（注意，这个环境会渲染环境到屏幕上）：

```
>>> img = env.render(mode="rgb_array")
>>> img.shape # height, width, channels (3=RGB)
(800, 1200, 3)
```

询问环境，可以采取的可能行动：

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` 的意思是可能的行动是整数 0 和 1，表示向左（0）或向右（1）加速。其它的环境可能有其它离散的行动，或其它种类的行动（例如，连续性行动）。因为棍子是向右偏的（`obs[2] > 0`），让车子向右加速：

```
>>> 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四个值：

`obs`：

这是新的观测，小车现在正在向右走（`obs[1]>0`，注：当前速度为正，向右为正）。平衡杆仍然向右倾斜（`obs[2]>0`），但是他的角速度现在为负（`obs[3]<0`），所以它在下一步后可能会向左倾斜。

`reward`：

在这个环境中，无论你做什么，每一步都会得到 1.0 奖励，所以游戏的目标就是尽可能长的运行。

`done`：

当游戏结束时这个值会为 `True`。当平衡杆倾斜太多、或越过屏幕、或超过 200 步时会发生这种情况。之后，必须重新设置环境才能重新使用。

`info`：

该字典可以在其他环境中提供额外信息用于调试或训练。例如，在一些游戏中，可以指示智能体还剩多少条命。

提示：使用完环境后，应当调用它的 `close()` 方法释放资源。

让我们硬编码一个简单的策略，当杆向左倾斜时向左边加速，当杆向右倾斜时加速向右边加速。我们使用这个策略来获得超过 500 步的平均回报：

```
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)
```

这段代码不难。让我们看看结果：

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(41.718, 8.858356280936096, 24.0, 68.0)
```

即使有 500 次尝试，这一策略从未使平衡杆在超过 68 个连续的步骤里保持直立。结果太好。如果你看一下 Jupyter 笔记本中的模拟，你会发现，推车越来越强烈地左右摆动，直到平衡杆倾斜过度。让我们看看神经网络是否能提出更好的策略。

## 神经网络策略

让我们创建一个神经网络策略。就像之前我们编码的策略一样，这个神经网络将把观察作为输入，输出要执行的动作。更确切地说，它将估计每个动作的概率，然后我们将根据估计的概率随机地选择一个动作（见图 18-5）。在 CartPole 环境中，只有两种可能的动作（左或右），所以我们只需要一个输出神经元。它将输出动作 0（左）的概率  $p$ ，动作 1（右）的概率显然将是  $1 - p$ 。例如，如果它输出 0.7，那么我们将以 70% 的概率选择动作 0，以 30% 的概率选择动作 1。

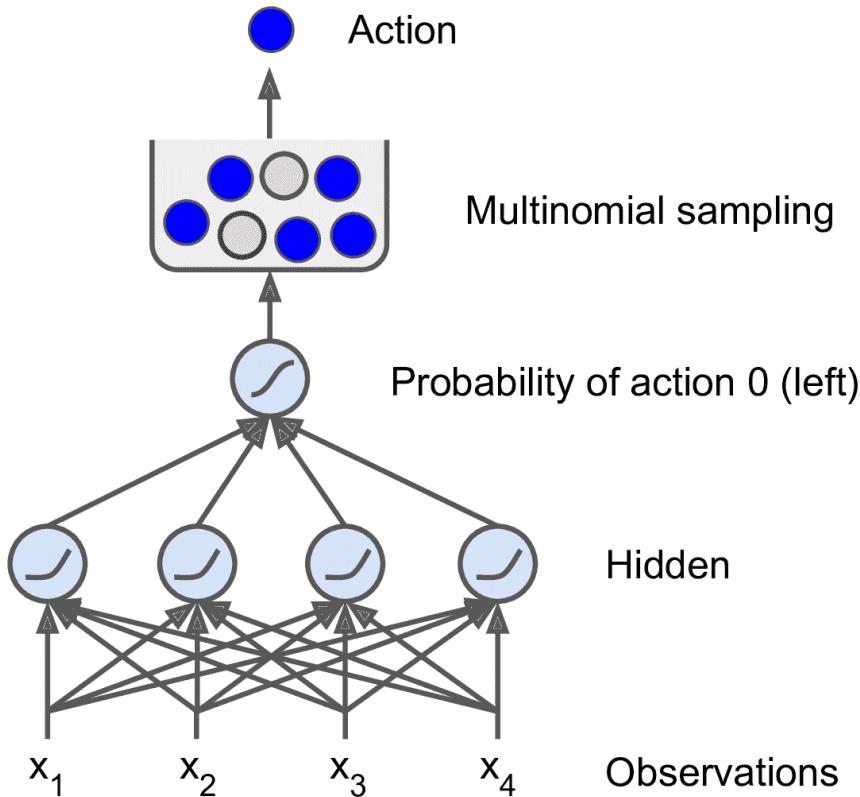


图 18-5 神经网络策略

你可能奇怪为什么我们根据神经网络给出的概率来选择随机的动作，而不是选择最高分数的动作。这种方法使智能体在探索新的行为和利用那些已知可行的行动之间找到正确的平衡。举个类比：假设你第一次去餐馆，所有的菜看起来同样吸引人，所以你随机挑选一个。如果菜好吃，你可以增加下一次点它的概率，但是你不应该把这个概率提高到 100%，否则你将永远不会尝试其他菜肴，其中一些甚至比你尝试的更好。

还要注意，在这个特定的环境中，过去动作和观察可以被放心地忽略，因为每个观察都包含环境的完整状态。如果有一些隐藏状态，那么你也需要考虑过去的行为和观察。例如，如果环境仅仅揭示了推车的位置，而不是它的速度，那么你不仅要考虑当前的观测，还要考虑先前的观测，以便估计当前的速度。另一个例子是当观测是有噪声的，在这种情况下，通常你想用过去的观察来估计最可能的当前状态。因此，CartPole 问题是简单的；观测是无噪声的，而且它们包含环境的全状态。

下面是用 `tf.keras` 创建这个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"),
])
```

在导入之后，我们使用 `Sequential` 模型定义策略网络。输入的数量是观测空间的大小（在 CartPole 的情况下是 4 个），我们只有 5 个隐藏单元，并且我们只有一个输出概率（向左的概率），所以输出层只需一个使用 `sigmoid` 的神经元就成。如

果超过两个动作，每个动作就要有一个神经元，然后使用 softmax 激活函数。

好了，现在我们有一个可以观察和输出动作的神经网络了，那我们怎么训练它呢？

## 评价行为：信用分配问题

如果我们知道每一步的最佳动作，我们可以像通常一样训练神经网络，通过最小化估计概率和目标概率之间的交叉熵。这只是通常的监督学习。然而，在强化学习中，智能体获得的指导的唯一途径是通过奖励，奖励通常是稀疏的和延迟的。例如，如果智能体在 100 个步骤内设法平衡杆，它怎么知道它采取的 100 个行动中的哪一个是好的，哪些是坏的？它所知道的是，在最后一次行动之后，杆子坠落了，但最后一次行动肯定不是负全责的。这被称为信用分配问题：当智能体得到奖励时，很难知道哪些行为应该被信任（或责备）。如果一只狗在表现优秀几小时后才得到奖励，它会明白它做对了什么吗？

为了解决这个问题，一个通常的策略是基于这个动作后得分的总和来评估这个动作，通常在每个步骤中应用衰减因子  $r$ 。例如（见图 18-6），如果一个智能体决定连续三次向右，在第一步之后得到 +10 奖励，第二步后得到 0，最后在第三步之后得到 -50，然后假设我们使用衰减率  $r=0.8$ ，那么第一个动作将得到  $10 + r \times 0 + r^2 \times (-50) = -22$  的分数。如果衰减率接近 0，那么与即时奖励相比，未来的奖励不会有多大意义。相反，如果衰减率接近 1，那么对未来的奖励几乎等于即时回报。典型的衰减率通常从 0.9 到 0.99 之间。如果衰减率为 0.95，那么未来 13 步的奖励大约是即时奖励的一半 ( $0.95^{13} \times 0.5$ )，而当衰减率为 0.99，未来 69 步的奖励是即时奖励的一半。在 CartPole 环境下，行为具有相当短期的影响，因此选择 0.95 的折扣率是合理的。

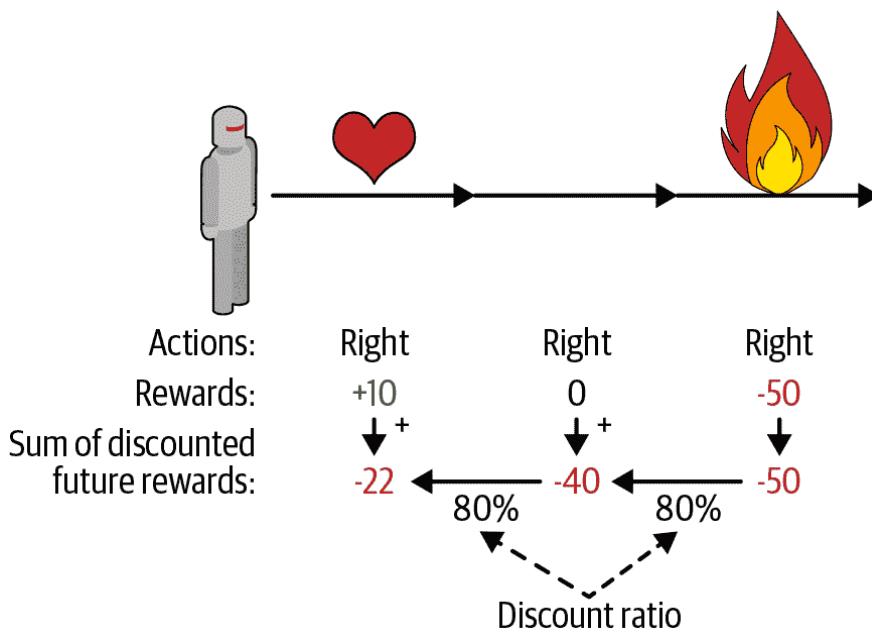


图 18-6 计算行动的回报：未来衰减求和

当然，一个好的动作可能会紧跟着一串坏动作，这些动作会导致平衡杆迅速下降，从而导致一个好的动作得到一个低分数（类似的，一个好行动者有时会在一部烂片中扮演主角）。然而，如果我们花足够多的时间来训练游戏，平均下来好的行为会得到比坏的更好的分数。因此，为了获得相当可靠的动作分数，我们必须运行很多次并将所有动作分数归一化（通过减去平均值并除以标准偏差）。之后，我们可以

合理地假设消极得分的行为是坏的，而积极得分的行为是好的。现在我们有一个方法来评估每一个动作，我们已经准备好使用策略梯度来训练我们的第一个智能体。让我们看看如何做。

## 策略梯度

正如前面所讨论的，PG 算法通过遵循更高回报的梯度来优化策略参数。一种流行的 PG 算法，称为增强算法，在 1929 由 Ronald Williams 提出。这是一个常见的变体：

1. 首先，让神经网络策略玩几次游戏，并在每一步计算梯度，这使得智能体更可能选择行为，但不应用这些梯度。
2. 运行几次后，计算每个动作的得分（使用前面段落中描述的方法）。
3. 如果一个动作的分数是正的，这意味着动作是好的，可应用较早计算的梯度，以便将来有更大的概率选择这个动作。但是，如果分数是负的，这意味着动作是坏的，要应用相反梯度来使得这个动作在将来采取的可能性更低。我们的方法就是简单地将每个梯度向量乘以相应的动作得分。
4. 最后，计算所有得到的梯度向量的平均值，并使用它来执行梯度下降步骤。

让我们使用 `tf.keras` 实现这个算法。我们将训练我们早先建立的神经网络策略，让它学会平衡车上的平衡杆。首先，需要一个能执行一步的函数。假定做出的动作都是对的，激素亲戚损失和梯度（梯度会保存一会，根据动作的结果再对其修改）：

```
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
```

逐行看代码：

- 在 `GradientTape` 代码块内，先调用模型，传入一个观察（将观察变形为包含单个实例的批次）。输出是向左的概率。
- 然后，选取一个 0 到 1 之间的浮点数，检查是否大于 `left_proba`。概率为 `left_proba` 时，`action` 是 `False`；概率为 `1-left_proba` 时，`action` 是 `True`。当将这个布尔值转变为数字时，动作是 0（左）或 1（右）及对应的概率。
- 接着，定义向左的目标概率：1 减去动作（浮点值）。如果动作是 0（左），则向左的目标概率等于 1。如果动作是 1（右），则目标概率等于 0。
- 然后使用损失函数计算损失，使用记录器计算模型可训练变量的损失梯度。这些梯度会在后面应用前，根据动作的结果做微调。
- 最后，执行选择的动作，无论是否结束，返回新的观察、奖励，和刚刚计算的梯度。

现在，创建另一个函数基于 `play_one_step()` 的多次执行函数，返回所有奖励和每个周期和步骤的梯度：

```
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, 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
```

这段代码返回了奖励列表（每个周期一个奖励列表，每个步骤一个奖励），还有一个梯度列表（每个周期一个梯度列表，每个步骤一个梯度元组，每个元组每个变脸有一个梯度张量）。

算法会使用 `play_multiple_episodes()` 函数，多次执行游戏（比如，10 次），然后会检查所有奖励，做衰减，然后归一化。要这么做，需要多个函数：第一个计算每个步骤的未来衰减奖励的和，第二个归一化所有这些衰减奖励（减去平均值，除以标准差）：

```
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]
```

检测其是否有效：

```
>>> 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, 1.07277777 ])]
```

调用 `discount_rewards()`，返回了我们想要的结果（见图 18-6）。可以确认函数 `discount_and_normalize_rewards()` 返回了每个周期每个步骤的归一化的行动的结果。可以看到，第一个周期的表现比第二个周期的表现糟糕，所以归一化的结果都是负的；第一个周期中的动作都是不好的，而第二个周期中的动作被认为是好的。

可以准备运行算法了！现在定义超参数。运行 150 个训练迭代，每次迭代完成 10 次周期，每个周期最多 200 个步骤。衰减因子是 0.95：

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

还需要一个优化器和损失函数。优化器用普通的 Adam 就成，学习率用 0.01，因为是二元分类器，使用二元交叉熵损失函数：

```
optimizer = keras.optimizers.Adam(lr=0.01)
loss_fn = keras.losses.binary_crossentropy
```

接下来创建和运行训练循环。

```
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))
```

逐行看下代码：

- 在每次训练迭代，循环调用 `play_multiple_episodes()`，这个函数玩 10 次游戏，返回每个周期和步骤的奖励和梯度。
- 然后调用 `discount_and_normalize_rewards()` 计算每个动作的归一化结果（代码中是 `final_reward`）。这样可以测量每个动作的好坏结果。
- 接着，循环每个可训练变量，计算每个变量的梯度加权平均，权重是 `final_reward`。
- 最后，将这些平均梯度应用于优化器：微调模型的变量。

就是这样。这段代码可以训练神经网络策略，模型可以学习保持棍子的平衡（可以尝试笔记本中的“策略梯度”部分）。每个周期的平均奖励会非常接近 200（200 是环境默认的最大值）。成功！

**提示：**研究人员试图找到一种即使当智能体最初对环境一无所知时也能很好工作的算法。然而，除非你正在写论文，否则你应该尽可能多地将先前的知识注入到智能体中，因为它会极大地加速训练。例如，因为知道棍子要尽量垂直，你可以添加与棍子角度成正比的负奖励。这可以让奖励不那么分散，是训练加速。此外，如果你已经有一个相当好的策略，你可以训练神经网络模仿它，然后使用策略梯度来改进它。

尽管它相对简单，但是该算法是非常强大的。你可以用它来解决更难的问题，而不仅仅是平衡一辆手推车上的平衡杆。事实上，因为样本不足，必须多次玩游戏，才能取得更大进展。但这个算法是更强大算法的基础，比如演员评论家算法（后面会介绍）。

现在我们来看看另一个流行的算法族。与 PG 算法直接尝试优化策略以增加奖励相反，我们现在看的算法不那么直接：智能体学习去估计每个状态的未来衰减奖励的期望总和，或者在每个状态中的每个行为未来衰减奖励的期望和。然后，使用这些知识来决定如何行动。为了理解这些算法，我们必须首先介绍马尔可夫决策过程 (MDP)。

## 马尔可夫决策过程

在二十世纪初，数学家 Andrey Markov 研究了没有记忆的随机过程，称为马尔可夫链。这样的过程具有固定数量的状态，并且在每个步骤中随机地从一个状态演化到另一个状态。它从状态  $s$  演变为状态  $s'$  的概率是固定的，它只依赖于  $(s, s')$  对，而不是依赖于过去的状态（系统没有记忆）。

图 18-7 展示了一个具有四个状态的马尔可夫链的例子。假设该过程从状态  $s_0$  开始，并且在下一步骤中有 70% 的概率保持在该状态不变中。最终，它必然离开那个状态，并且永远不会回来，因为没有其他状态回到  $s_0$ 。如果它进入状态  $s_1$ ，那么它很可能会进入状态  $s_2$  (90% 的概率)，然后立即回到状态  $s_1$  (以 100% 的概率)。它可以在这两个状态之间交替多次，但最终它会落入状态  $s_3$  并永远留在那里 (这是一个终端状态)。马尔可夫链可以有非常不同的动力学，它们在热力学、化学、统计学等方面有着广泛的应用。

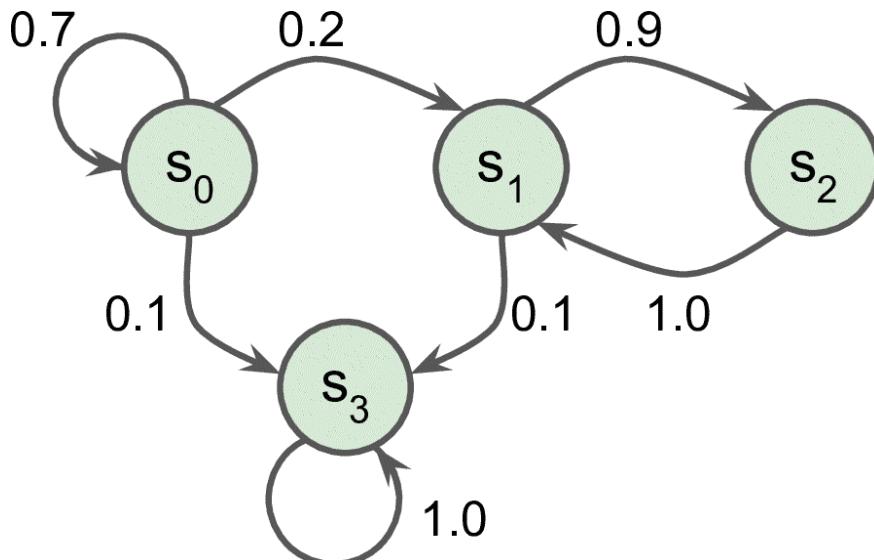


图 18-7 马尔科夫链案例

马尔可夫决策过程最初是在 20 世纪 50 年代由 Richard Bellman 描述的。它们类似于马尔可夫链，但有一个不同：在状态转移的每一步中，一个智能体可以选择几种可能的动作中的一个，并且过渡概率取决于所选择的动作。此外，一些状态过渡返回一些奖励（正或负），智能体的目标是找到一个策略，随着时间的推移将最大限度地提高奖励。

例如，图 18-8 中所示的 MDP 在每个步骤中具有三个状态（用圆圈表示）和三个可能的离散动作（用菱形表示）。

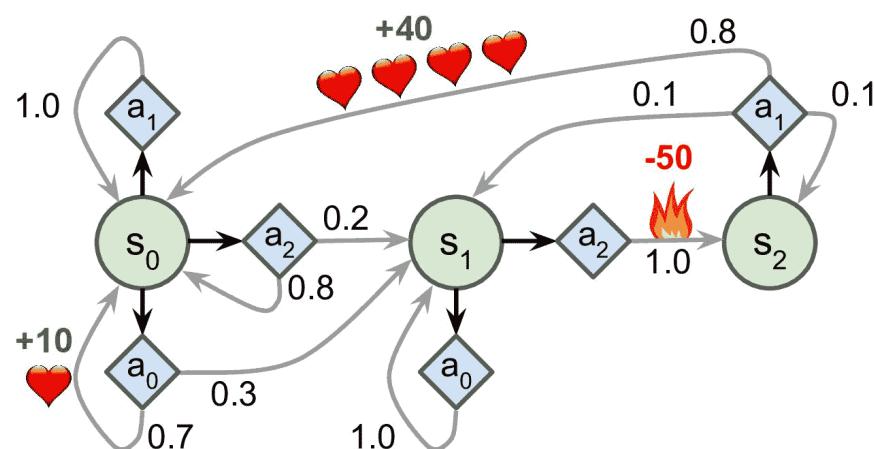


图 18-8 马尔科夫决策过程案例

如果从状态  $s_0$  开始，可以在动作  $A_0$ 、 $A_1$  或  $A_2$  之间进行选择。如果它选择动作  $A_1$ ，它就保持在状态  $s_0$  中，并且没有任何奖励。因此，如果愿意的话，它可以决定永远呆在那里。但是，如果它选择动作  $A_0$ ，它有 70% 的概率获得 +10 奖励，并保持在状态  $s_0$ 。然后，它可以一次又一次地尝试获得尽可能多的奖励。但它将在状态  $s_1$  中结束这样的行为。在状态  $s_1$  中，它只有两种可能的动作： $A_0$  或  $A_2$ 。它可以通过反复选择动作  $A_0$  来选择停留，或者它可以選擇动作  $A_2$  移动到状态  $s_2$  并得到 -50 奖励。在状态  $s_2$  中，除了采取行动  $A_1$  之外，别无选择，这将最有可能引导它回到状态  $s_0$ ，在途中获得 +40 的奖励。通过观察这个 MDP，你能猜出哪一个策略会随着时间的推移而获得最大的回报吗？在状态  $s_0$  中， $A_0$  是最好的选择，在状态  $s_2$  中，智能体别无选择，只能采取行动  $A_1$ ，但是在状态  $s_1$  中，智能体否应该保持不动（ $A_0$ ）或通过火（ $A_2$ ），这是不明确的。

贝尔曼找到了一种估计任何状态  $s$  的最佳状态值的方法，记作  $v(s)$ ，它是智能体在其采取最佳行为达到状态  $s$  后所有衰减未来奖励的总和的平均期望。他证明，如果智能体的行为最佳，那么就适用于贝尔曼最优性公式（见公式 18-1）。这个递归公式表示，如果智能体最优地运行，那么当前状态的最优值等于在采取一个最优动作之后平均得到的奖励，加上该动作可能导致的所有可能的下一个状态的期望最优值。

$$V^*(s) = \max_a \sum_s T(s, a, s')[R(s, a, s') + \gamma \cdot V^*(s')] \quad \text{for all } s$$

公式 18-1 贝尔曼最优性公式

其中：

- $T(s, a, s')$  为智能体选择动作  $a$  时从状态  $s$  到状态  $s'$  的概率。例如，图 18-8 中， $T(s_2, a_1, s_0) = 0.8$ 。
- $R(s, a, s')$  为智能体选择以动作  $a$  从状态  $s$  到状态  $s'$  的过程中得到的奖励。例如图 18-8 中， $R(s_2, a_1, s_0) = +40$ 。
- $\gamma$  为衰减率。

这个等式直接引出了一种算法，该算法可以精确估计每个可能状态的最优状态值：首先将所有状态值估计初始化为零，然后用数值迭代算法迭代更新它们（见公式 18-2）。一个显著的结果是，给定足够的时间，这些估计保证收敛到最优状态值，对应于最优策略。

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \cdot V_k(s')] \quad \text{for all } s$$

公式 18-2 数值迭代算法

在这个公式中， $v[k](s)$  是在  $k$  次算法迭代对状态  $s$  的估计。

**笔记：**该算法是动态规划的一个例子，它将了一个复杂的问题（在这种情况下，估计潜在的未来衰减奖励的总和）变为可处理的子问题，可以迭代地处理（在这种情况下，找到最大化平均报酬与下一个衰减状态值的和的动作）

了解最佳状态值可能是有用的，特别是评估策略，但它没有明确地告诉智能体要做什么。幸运的是，贝尔曼发现了一种非常类似的算法来估计最优状态-动作值（state-action values），通常称为 Q 值。状态行动  $(s, A)$  对的最优 Q 值，记

为  $Q^*(s, a)$ ，是智能体在到达状态  $s$ ，然后选择动作  $a$  之后平均衰减未来奖励的期望的总和。但是在它看到这个动作的结果之前，假设它在该动作之后的动作是最优的。

下面是它的工作原理：再次，通过初始化所有的 Q 值估计为零，然后使用 Q 值迭代算法更新它们（参见公式 18-3）。

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \cdot \max_{a'} Q_k(s', a') \right]$$

for all  $(s, a)$

公式 18-3 Q 值迭代算法

一旦你有了最佳的 Q 值，定义最优的策略  $\pi^*(s)$ ，就没什么作用了：当智能体处于状态  $s$  时，它应该选择具有最高 Q 值的动作，用于该状态：

$$\pi^*(s) = \operatorname{argmax}_a Q^*(s, a)$$

让我们把这个算法应用到图 18-8 所示的 MDP 中。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，要想知道经过动作  $a_1$ ，从  $s_2$  到  $s_0$  的过渡概率，我们需要查询  $transition\_probabilities[2][1][0]$ （等于 0.8）。相似的，要得到奖励，需要查询  $rewards[2][1][0]$ （等于 +40）。要得到  $s_2$  的可能的动作，需要查询  $possible\_actions[2]$ （结果是  $a_1$ ）。然后，必须将 Q 值初始化为 0（对于不可能的动作，Q 值设为  $-\infty$ ）：

```
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
```

现在运行 Q 值迭代算法。它反复对 Q 值的每个状态和可能的动作应用公式 18-3：

```
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)])
```

Q 值的结果如下：

```
>>> Q_values
array([[ 18.91891892,  17.02702702,  13.62162162],
       [ 0.,          -inf,          -inf,          -4.87971488],
       [ 0.,          -inf,  50.13365013,          -inf]])
```

例如，当智能体处于状态  $s_0$ ，选择动作  $a_1$ ，衰减未来奖励的期望和大约是 17.0。

对于每个状态，查询拥有最高 Q 值的动作：

```
>>> np.argmax(Q_values, axis=1) # optimal action for each state
array([0, 0, 1])
```

这样就得到了衰减因子等于 0.9 时，这个 MDP 的最佳策略是什么：状态  $s_0$  时选择动作  $a_0$ ；在状态  $s_1$  时选择动作  $a_0$ ；在状态  $s_2$  时选择动作  $a_1$ 。有趣的是，如果将衰减因子提高到 0.95，最佳策略发生了改变：在状态  $s_1$  时，最佳动作变为  $a_2$ （通过火！）。道理很明显，如果未来期望越高，忍受当前的痛苦是值得的。

## 时间差分学习

具有离散动作的强化学习问题通常可以被建模为马尔可夫决策过程，但是智能体最初不知道转移概率是什么（它不知道  $T(s, a, s')$ ），并且它不知道奖励会是什么（它不知道  $R(s, a, s')$ ）。它必须经历每一个状态和每一次转变并且至少知道一次奖励，并且如果要对转移概率进行合理的估计，就必须经历多次。

时间差分学习（TD 学习）算法与数值迭代算法非常类似，但考虑到智能体仅具有 MDP 的部分知识。一般来说，我们假设智能体最初只知道可能的状态和动作，没有更多了。智能体使用探索策略，例如，纯粹的随机策略来探索 MDP，并且随着它的发展，TD 学习算法基于实际观察到的转换和奖励来更新状态值的估计（见公式 18-4）。

$$V_{k+1}(s) \leftarrow (1 - \alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

or, equivalently:

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$$

with  $\delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$

公式 18-4 TD 学习算法

在这个公式中：

- $\alpha$  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。

公式的第一种形式的更为准确的表达，是使用：

$$a \xleftarrow[\alpha]{} b$$

它的意思是  $a_{k+1} \leftarrow (1 - \alpha) \cdot a_k + \alpha \cdot b_k$ ，公式 18-4 的第一行可以重写为：

$$V(s) \leftarrow r + \gamma \cdot V(s')$$

提示：TD 学习和随机梯度下降有许多相似点，特别是 TD 学习每次只处理一个样本。另外，和随机梯度下降一样，如果逐渐降低学习率，是能做到收敛的（否则，会在最佳 Q 值附近反复跳跃）。

对于每个状态  $s$ ，该算法只跟踪智能体离开该状态时立即获得的奖励的平均值，再加上它期望稍后得到的奖励（假设它的行为最佳）。

## Q 学习

类似地，Q 学习算法是 Q 值迭代算法的改编版本，其适应转移概率和回报在初始未知的情况（见公式 18-5）。Q 学习通过观察智能体玩游戏，逐渐提高 Q 值的估计。一旦有了准确（或接近）的 Q 值估计，则选择具有最高 Q 值的动作（即，贪婪策略）。

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha)Q_k(s, a) + \alpha(r + \gamma \max_a Q_k(s', a'))$$

公式 18-5 Q 学习算法

对于每一个状态动作对  $(s, a)$ ，该算法跟踪智能体在以动作  $A$  离开状态  $s$  时获得的即时奖励平均值  $r$ ，加上它期望稍后得到的奖励。由于目标策略将最优地运行，所以我们取下一状态的 Q 值估计的最大值。

以下是如何实现 Q 学习算法。首先，需要让一个智能体探索环境。要这么做的话，我们需要一个步骤函数，好让智能体执行一个动作，并返回结果状态和奖励：

```
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，实现智能体的探索策略。因为状态空间很小，使用简单随机策略就可以。如果长时间运行算法，智能体会多次访问每个状态，也会多次尝试每个可能的动作：

```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

然后，和之前一样初始化 Q 值，使用学习率递降的方式运行 Q 学习算法（使用第 11 章介绍过的指数调度算法）：

```

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

```

算法会覆盖最优 Q 值，但会经历多次迭代，可能有许多超参数调节。见图 18-9，Q 值迭代算法（左）覆盖速度很快，只用了不到 20 次迭代，而 Q 学习算法（右）用了 8000 次迭代才覆盖完。很明显，不知道过渡概率或奖励，使得找到最佳策略显著变难！

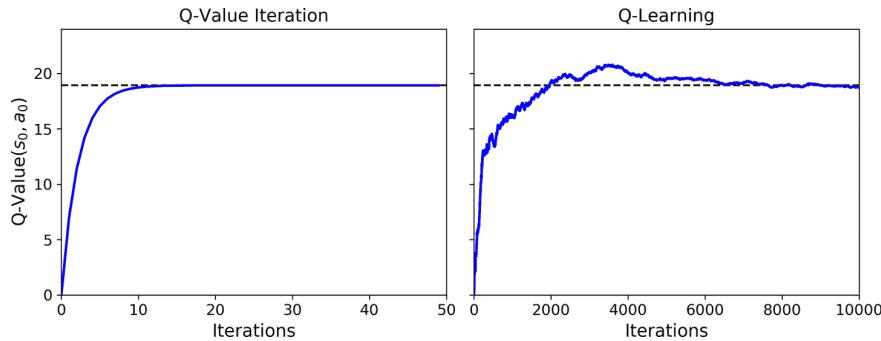


图 18-9 Q 值迭代算法（左）对比 Q 学习算法（右）

Q 学习被称为离线策略算法，因为正在训练的策略不是正在执行的策略：在前面的例子中，被执行的策略（探索策略）是完全随机的，而训练的算法总会选择具有最高 Q 值的动作。相反的，策略梯度下降算法是在线算法：使用训练的策略探索世界。令人惊讶的是，该算法能够通过观察智能体的随机行为（例如当你的老师是一个醉猴子时，学习打高尔夫球）学习最佳策略。我们能做得更好吗？

## 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。尽管一个纯粹的随机策略保证最终访问每一个状态和每个转换多次，但可能需要很长的时间这样做。因此，一个更好的选择是使用  $\epsilon$  贪婪策略：在每个步骤中，它以概率  $\epsilon$  随机地或以概率为  $1-\epsilon$  贪婪地选择具有最高 Q 值的动作。 $\epsilon$  贪婪策略的优点（与完全随机策略相比）是，它将花费越来越多的时间来探索环境中有趣的部分，因为 Q 值估计越来越好，同时仍花费一些时间访问 MDP 的未知区域。以  $\epsilon$  为很高的值（例如，1）开始，然后逐渐减小它（例如，下降到 0.05）是很常见的。

或者，不依赖于探索的可能性，另一种方法是鼓励探索策略来尝试它以前没有尝试过的行动。这可以被实现为加到 Q 值估计的奖励，如公式 18-6 所示。

$$Q(s, a) \leftarrow r + \gamma \cdot \max_{a'} f(Q(s', a'), N(s', a'))$$

公式 18-6 使用探索函数的 Q 学习

在这个公式中：

- $N(s', a')$  计算了在状态  $s$  时选择动作  $a$  的次数
- $f(Q, N)$  是一个探索函数，例如  $f(Q, N) = Q + \kappa/(1 + N)$ ，其中  $\kappa$  是一个好奇心超参数，它测量智能体被吸引到未知状态的程度。

## 近似 Q 学习和深度 Q 学习

Q 学习的主要问题是，它不能很好地扩展到具有许多状态和动作的大（甚至中等）的 MDP。例如，假如你想用 Q 学习来训练一个智能体去玩 Ms. Pac-Man（图 18-1）。Ms. Pac-Man 可以吃超过 150 粒粒子，每一粒都可以存在或不存在（即已经吃过）。因此，可能状态的数目大于  $2^{150} \approx 10^{45}$ 。空间大小比地球的总原子数要多得多，所以你绝对无法追踪每一个 Q 值的估计值。

解决方案是找到一个函数  $Q[\theta](s, a)$ ，使用可管理数量的参数（根据向量  $\theta$ ）来近似 Q 值。这被称为近似 Q 学习。多年来，人们都是手工在状态中提取并线性组合特征（例如，最近的鬼的距离，它们的方向等）来估计 Q 值，但在 2013 年，DeepMind 表明使用深度神经网络可以工作得更好，特别是对于复杂的问题。它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），并且使用近似 Q 学习的 DQN 被称为深度 Q 学习。

如何训练 DQN 呢？这里用 DQN 在给定的状态动作对  $(s, a)$ ，来估计 Q 值。感谢贝尔曼，我们知道这个近似 Q 值要接近在状态  $s$  执行动作  $a$  的奖励  $r$ ，加上之前的衰减奖励。要估计未来衰减奖励的和，我们只需在下一个状态  $s'$ ，对于所有可能的动作  $a'$  执行 DQN。针对每个可能的动作，获得了近似的 Q 值。然后挑选最高的，并做衰减，就得到了未来衰减奖励的和。通过将奖励  $r$  和未来衰减奖励估计相加，得到了状态动作对  $(s, a)$  的目标 Q 值  $y(s, a)$ ，见公式 18-7。

$$Q_{\text{target}}(s, a) = r + \gamma \cdot \max_{a'} Q_{\theta}(s', a')$$

公式 18-7 目标 Q 值

有了这个目标 Q 值，可以使用梯度下降运行一步训练算法。具体地，要最小化 Q 值  $Q(s, a)$  和目标 Q 值的平方根方差（或使用 Huber 损失降低算法对大误差的敏感度）。这就是基础的深度 Q 学习算法。下面用其处理平衡车问题。

## 实现深度 Q 学习

首先需要的是一个深度 Q 网络。理论上，需要一个输入是状态-动作对、输出是近似 Q 值的神经网络，但在实际中，使用输入是状态、输出是每个可能动作的近似 Q 值的神经网络，会更加高效。要处理 CartPole 环境，我们不需要非常复杂的神经网络；只要几个隐藏层就够了：

```
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)
])
```

使用这个 DQN 选择一个动作，选择 Q 值最大的动作。要保证智能体探索环境，使用的是 $\epsilon$  贪婪策略（即，选择概率为 $\epsilon$ 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])
```

不仅只根据最新的经验训练 DQN，将所有经验存储在接力缓存（或接力记忆）中，每次训练迭代，从中随机采样一个批次。这样可以降低训练批次中的经验相关性，可以极大的提高训练效果。如下，使用双端列表实现：

```
from collections import deque
replay_buffer = deque(maxlen=2000)
```

提示：双端列表是一个链表，每个元素指向后一个和前一个元素。插入和删除元素都非常快，但双端列表越长，随机访问越慢。如果需要一个非常大的接力缓存，可以使用环状缓存；见笔记本中的 Deque vs Rotating List 章节。

每个经验包含五个元素：状态，智能体选择的动作，奖励，下一个状态，一个知识是否结束的布尔值（done）。需要一个小函数从接力缓存随机采样。返回的是五个 NumPy 数组，对应五个经验：

```
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
```

再创建一个使用 $\epsilon$  贪婪策略的单次玩游戏函数，然后将结果经验存储在接力缓存中：

```
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，再创建最后一个批次采样函数，用单次梯度下降训练这个 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))
```

逐行看下代码：

- 首先定义一些超参数，并创建优化器和损失函数。
- 然后创建 `training_step()` 函数。先采样经验批次，然后使用 DQN 预测每个可能动作的每个经验的下一状态的 Q 值。因为假定智能体采取最佳行动，所以只保留下一状态的最大 Q 值。接着，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。
- 接着，使用 DQN 计算每个有经验的状态-动作对的 Q 值。但是，DQN 还会输出其它可能动作的 Q 值，不仅是智能体选择的动作。所以，必须遮掩不需要的 Q 值。`tf.one_hot()` 函数可以方便地将动作下标的数组转别为掩码。例如，如果前三个经验分别包含动作 1, 1, 0，则掩码会以 `[[0, 1], [0, 1], [1, 0], ...]` 开头。然后将 DQN 的输出乘以这个掩码，就可以排除所有不需要的 Q 值。然后，按列求和，去除所有的零，只保留有经验的状态-动作对的 Q 值。得到张量 `Q_values`，包含批次中每个经验的预测的 Q 值。
- 然后，计算损失：即有经验的状态-动作对的目标 Q 值和预测 Q 值的均方误差。
- 最后，对可训练变量，用梯度下降步骤减小损失。

这是最难的部分。现在，训练模型就简单了：

```
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)
```

跑 600 次游戏，每次最多 200 步。在每一步，先计算  $\epsilon$  贪婪策略的 `epsilon` 值：这个值在 500 个周期内，从 1 线性降到 0.01。然后调用 `play_one_step()` 函数，用  $\epsilon$  贪婪策略挑选动作，然后执行并在接力缓存中记录经验。如果周期结束，就退出循环。最后，如果超过了 50 个周期，就调用 `training_step()` 函数，用从接力缓存取出的批次样本训练模型。玩 50 个周期，而不训练的原因是给接力缓存一些时间来填充（如果等待的不够久，则接力缓存中的样本散度太小）。像上面这样，我们就实现了深度 Q 学习算法。

图 18-10 展示了智能体在每个周期获得的总奖励。

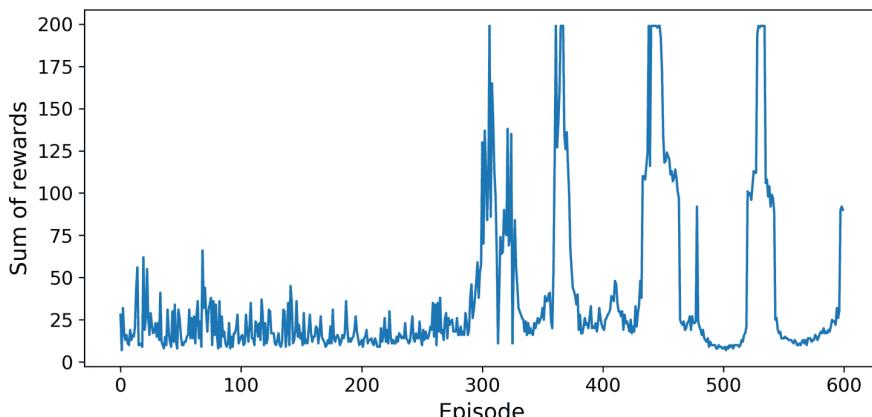


图 18-10 深度 Q 学习算法的学习曲线

可以看到，在前 300 个周期，算法的进步不大（部分是因为 $\epsilon$ 在一开始时非常高），然后表现突然提升到了 200（环境最高值）。这说明算法效果不错，并且比策略梯度算法快得多！但仅仅几个周期之后，性能就骤降到了 25。这被称为“灾难性遗忘”，这是所有 RL 算法都面临的大问题：随着智能体探索环境，不断更新策略，但是在环境的一部分学到的内容可能和之前学到的内容相悖。经验是关联的，学习环境不断改变——这不利于梯度下降！如果增加接力缓存的大小，可以减轻这个问题。但真实的情况是，强化学习很难：训练通常不稳定，需要尝试许多超参数值和随机种子。例如，如果改变每层神经元的数量，从 32 到 30 或 34，模型表现不会超过 100（DQN 只有一个隐藏层时，可能更稳定）。

**笔记：**强化学习非常困难，很大程度是因为训练的不稳定性，以及巨大的超参数和随机种子的不稳定性。就像 Andrej Karpathy 说的：“监督学习自己就能工作，强化学习被迫工作”。你需要时间、耐心、毅力，还有一点运气。这是为什么强化学习不是常用的深度学习算法的原因。除了 AlphaGo 和 Atari 游戏，还有一些其它应用：例如，Google 使用 RL 优化数据中心的费用，也用于一些机器人应用的超参数调节，和推荐系统。

你可能想为什么我们不画出损失。事实证明损失不是模型表现的好指标。就算损失下降，智能体的表现也可能更糟（例如，智能体困在了环境中，则 DQN 开始对区域过拟合）。相反的，损失可能变大，但智能体表现不错（例如，如果 DQN 知道 Q 值，就能提高预测的质量，智能体就能表现得更好，得到更多奖励，但因为 DQN 还设置了更大的目标，所以误差增加了）。

我们现在学的基本的深度 Q 学习算法，在玩 Atari 时太不稳定。DeepMind 是怎么做的呢？他们调节了算法。

## 深度 Q 学习的变体

下面看几个深度 Q 学习算法的变体，它们不仅训练稳定而且很快。

### 固定 Q 值目标

在基本的深度 Q 学习算法中，模型不仅做预测还自己设置目标。有点像一只狗追自己的尾巴。反馈循环使得网络不稳定：会发生分叉、摇摆、冻结，等等。要解决问题，DeepMind 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，它在每一步进行学习，并移动智能体；另一个是目标模型只定义目标。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，只需要变动一行，使用目标模型计算接下来状态的 Q 值：

```
next_Q_values = target.predict(next_states)
```

最后，在训练循环中，必须每隔一段周期（比如，每 50 个周期），将在线模型的权重新复制到目标模型中：

```
if episode % 50 == 0:
    target.set_weights(model.get_weights())
```

因为目标模型更新的没有在线模型频繁，Q 值目标更加稳定，前面讨论反馈循环减弱了。这个方法是 DeepMind 在 2013 年的论文中提出的方法之一，可以让智能体从零学习 Atari 游戏。要稳定训练，他们使用的学习率是 0.00025，很小，每隔 10000 步才更新目标模型，接力缓存的大小是 1 百万。并且 `epsilon` 降低的很慢，用 1 百万步从 1 降到 0.1，他们让算法运行了 5000 万步。

本章后面会用这些超参数，使用 TF-Agents 库训练 DQN 智能体来玩 Breakout。在此之前，再看另一个性能更好的 DQN 变体。

## 双 DQN

在 [2015 年的论文](#) 中，DeepMind 调节了他们的 DQN 算法，提高了性能，也稳定化了训练。他们称这个变体为双 DQN。算法更新的原因，是观察到目标网络倾向于高估 Q 值。事实上，假设所有动作都一样好：目标模型预测的 Q 值应该一样，但因为是估计值，其中一些可能存在更大的几率。目标模型会选择最大的 Q 值，最大的 Q 值要比平均 Q 值稍大，就像高估真正的 Q 值（就像在测量池塘深度时，测量随机水波的最高峰）。要修正这个问题，他们提出使用在线模型，而不是目标模型，来选择下一状态的最佳动作，只用目标模型估计这些最佳动作的 Q 值。下面是改善后的 `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
```

几个月之后，人们又提出了另一个改进的 DQN 算法。

## 优先经验接力

除了均匀地从接力缓存采样经验，如果更频繁地采样重要经验如何呢？这个主意被称为重要性采样（importance sampling, IS）或优先经验接力（prioritized experience replay, PER），是在 [2015 年的论文](#) 中由 DeepMind 发表的。

更具体的，可以导致快速学习成果的经验被称为重要经验。但如何估计呢？一个可行的方法是测量 TD 误差的大小  $\delta = r + \gamma \cdot V(s') - V(s)$ 。大 TD 误差说明过  $(s, r, s')$  很值得学习。当经验记录在接力缓存中，它的的重要性被设为非常大的值，保证可以快速采样。但是，一旦被采样（以及每次采样时），就计算 RD 误差  $\delta$ ，这个经验的优先度设为  $p = |\delta|$ （加上一个小常数，保证每个经验的采样概率不是零）。采样优先度为  $p$  的概率  $P$  正比于  $p[\zeta]$ ， $\zeta$  是调整采样贪婪度的超参数：当  $\zeta=0$  时，就是均匀采样， $\zeta=1$  时，就是完全的重要性采样。在论文中，作者使用的是  $\zeta=0.6$ ，最优值取决于任务。

但有一点要注意，因为样本偏向重要经验，必须要在训练时，根据重要性降低经验的重要性，否则模型会对重要经验过拟合。更加清楚的讲，重要经验采样更频繁，但训练时的权重小。要这么做，将每个经验的训练权重定义

为  $w = (n/P)^{(-\beta)}$ ， $n$  是接力缓存的经验数， $\beta$  是平衡重要性偏向的超参数（0是不偏向，1是完全偏向）。在论文中，作者一开始使用的是  $\beta=0.4$ ，在训练结束，提高到了  $\beta=1$ 。最佳值取决于任务，如果你提高了一个，也要提高其它的值。

接下来是最后一个重要的 DQN 算法的变体。

## 决斗 DQN

决斗 DQN 算法（DDQN，不要与双 DQN 混淆）是 DeepMind 在另一篇 [2015 年的论文](#) 中提出的。要明白原理，首先状态-动作对  $(s, a)$  的 Q 值，可以表示为  $Q(s, a) = V(s) + A(s, a)$ ，其中  $V(s)$  是状态  $s$  的值， $A(s, a)$  是状态  $s$  采取行动  $a$  的结果。另外，状态的值等于状态最佳动作  $a^*$  的 Q 值（因为最优策略会选最佳动作），因此  $V(s) = Q(s, a^*)$ ，即  $A(s, a^*) = 0$ 。在决斗 DQN 中，模型估计状态值和每个动作的结果。因为最佳动作的结果是 0，模型减去最大预测结果。下面是一个简单的决斗 DQN，用函数式 API 实现：

```
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])
```

算法的其余部分和之前一样。事实上，你可以创建一个双决斗 DQN，并结合优先经验队列！更为一般地，许多 RL 方法都可以结合起来，就像 DeepMind 在 [2017 年的论文](#) 展示的。论文的作者将六个不同的方法结合起来，训练了一个智能体，称为“彩虹”，表现很好。

不过，要实现所有这些方法，进行调试、微调，并且训练模型需要很多工作。因此，不要重新草轮子，最好的方法是复用可扩展的、使用效果好的库，比如 TF-Agents。

## TF-Agents 库

[TF-Agents 库](#)是基于 TensorFlow 实现的强化学习库，Google 开发并在 2018 年开源。和 OpenAI Gym 一样，它提供了许多现成的环境（包括了 OpenAI Gym 环境的包装），还支持 PyBullet 库（用于 3D 物理模拟），DeepMind 的 DM 控制库（基于 MuJoCo 的物理引擎），Unity 的 ML-Agents 库（模拟了许多 3D 环境）。它还使用了许多 RL 算法，包括 REINFORCE、DQN、DDQN，和各种 RL 组件，比如高效接力缓存和指标。TF-Agents 速度快、可扩展、便于使用、可自定义：你可以创建自己的环境和神经网络，可以对任意组件自定义。在这一节，我们使用 TF-Agents 训练一个智能体玩 Breakout，一个有名的 Atari 游戏（见图 18-11），使用的是 DQN 算法（可以换成任何你想用的算法）。

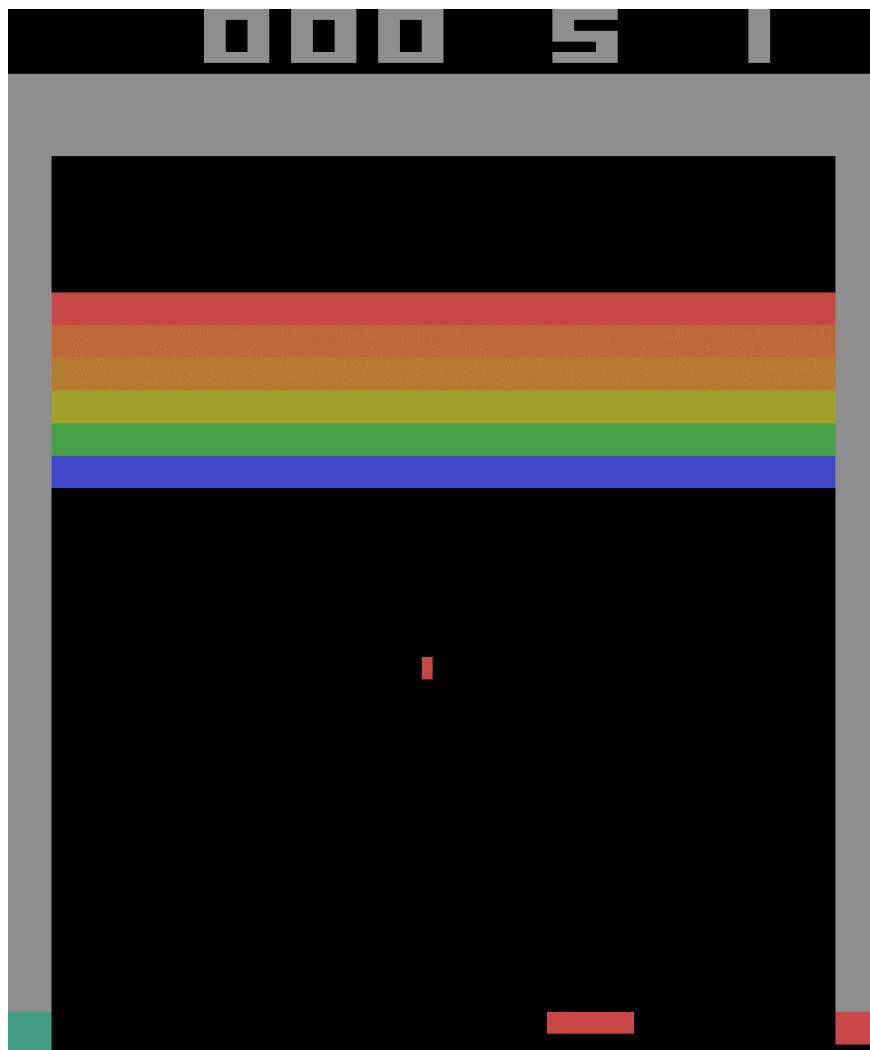


图 18-11 Breakout 游戏

## 安装 TF-Agents

先安装 TF-Agents。可以使用 PIP 安装（如果使用的是虚拟环境，一定要先激活；如果不激活，要使用选项 `--user`，或用管理员权限）：

```
$ python3 -m pip install --upgrade tf-agents
```

警告：写作本书时，TF-Agents 还很新，每天都有新进展，因此 API 可能会和现在有所不同 —— 但大体相同。如果代码不能运行，我会更新 Jupyter 笔记本。

然后，创建一个 TF-Agents 包装了 OpenAI Gym 的 Breakout 的环境。要这么做，需要先安装 OpenAI Gym 的 Atari 依赖：

```
$ python3 -m pip install --upgrade 'gym[atari]'
```

这条命令安装了 `atari-py`，这是 Arcade 学习环境的 Python 接口，这个学习环境是基于 Atari 2600 模拟器 Stella。

## TF-Agents 环境

如果一切正常，就能引入 TF-Agents，创建 Breakout 环境了：

```
>>> from tf_agents.environments import suite_gym
>>> env = suite_gym.load("Breakout-v4")
>>> env
<tf_agents.environments.wrappers.TimeLimit at 0x10c523c18>
```

这是 OpenAI Gym 环境的包装，可以通过属性 `gym` 访问：

```
>>> env.gym
<gym.envs.atari.atari_env.AtariEnv at 0x24dcab940>
```

TF-Agents 环境和 OpenAI Gym 环境非常相似，但有些差别。首先，`reset()` 方法不返回观察；返回的是 `TimeStep` 对象，它包装了观察，和一些其它信息：

```
>>> 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()` 方法返回的也是 `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)
```

属性 `reward` 和 `observation` 是奖励和观察，与 OpenAI Gym 相同（除了 `reward` 表示为 NumPy 数组）。对于周期的第一个时间步，属性 `step_type` 等于 0，1 是中间步，2 后最后一步。可以调用时间步的 `is_last()` 方法，检测是否是最后一步。最后，`discount` 属性指明了在这个时间步的衰减率。在这个例子中的值等于 1，所以没有任何衰减。可以通过在加载环境时设置 `discount` 参数，定义衰减因子。

笔记：任何时候，你可以通过调用方法 `current_time_step()` 方法，访问环境的当前时间步。

## 环境配置

TF-Agents 环境提供了配置，包括观察、动作、时间步，以及它们的形状、数据类型、名字，还有最小值和最大值：

```
>>> 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), ...))
```

可以看到，观察就是 Atari 屏幕的截图，用形状是 [210, 160, 3] 的 NumPy 数组表示。要渲染环境，可以调用 `env.render(mode="human")`，如果想用 NumPy 数组的形式返回图片，可以调用 `env.render(mode="rgb_array")`（与 OpenAI Gym 不同，这是默认模式）。

有四个可能的动作。Gym 的 Atari 环境有另一个方法，可以知道每个动作对应什么：

```
>>> env.gym.get_action_meanings()
['NOOP', 'FIRE', 'RIGHT', 'LEFT']
```

提示：配置是配置类的一个实例，可以是嵌套列表、字典。如果配置是嵌套的，则配置对象必须匹配配置的嵌套结构。例如，如果观察配置是 `{"sensors": ArraySpec(shape=[2]), "camera": ArraySpec(shape=[100, 100])}`，有效观察应该是 `{"sensors": np.array([1.5, 3.5]), "camera": np.array(...)}`。`tf.nest` 包提供了工具处理嵌套结构（即，`nests`）。

观察结果很大，所以需要做降采样，并转换成灰度。这样可以加速训练，减少内存使用。要这么做，要使用环境包装器。

## 环境包装器和 Atari 预处理

TF-Agents 在 `tf_agents.environments.wrappers` 中，提供了一些环境包装器。正如名字，它们可以包装环境，转发每个调用，还可以添加其它功能。以下是一些常见的包装器：

`ActionClipWrapper`

- 根据动作配置裁剪动作。

`ActionDiscretizeWrapper`

- 将连续动作空间量化到离散的动作空间。例如，如果原始环境的动作空间是 -1.0 到 +1.0 的连续范围，但是如果想用算法支持离散的动作空间，比如 DQN，就可以

用 `discrete_env = ActionDiscretizeWrapper(env, num_actions=5)` 包装环境，新的 `discrete_env` 有离散的可能动作空间：0、1、2、3、4。这些动作对应原始环境的动作 -1.0、-0.5、0.0、0.5、1.0。

`ActionRepeat`

- 将每个动作重复 `n` 次，并积累奖励。在许多环境中，这么做可以显著加速训练。

`RunStats`

- 记录环境数据，比如步骤数和周期数。

`TimeLimit`

- 超过最大的时间步数，则中断环境。

`VideoWrapper`

- 记录环境的视频。

要创建包装环境，需要先创建一个包装器，将包装过的环境传递给构造器。例如，下面的代码将一个环境包装在 `ActionRepeat` 中，让每个动作重复四次：

```
from tf_agents.environments.wrappers import ActionRepeat
repeating_env = ActionRepeat(env, times=4)
```

OpenAI Gym 在 `gym.wrappers` 中有一些环境包装器。但它们是用来包装 Gym 环境，不是 TF-Agents 环境，所以要使用的话，必须用 Gym 包装器包装 Gym 环境，再用 TF-Agents 包装器再包装起来。`suite_gym.wrap_env()` 函数可以实现，只要传入 Gym 环境和 Gym 包装器列表，和/或 TF-Agents 包装器的列表。另外，`suite_gym.load()` 函数既能创建 Gym 环境，如果传入包装器，也能做包装。每个包装器在包装时没有参数，所以如果想设置参数，必须传入 `lambda`。例如，下面的代码创建了一个 Breakout 环境，每个周期最多运行 10000 步，每个动作重复四次：

```
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，大多数论文使用了标准预处理步骤，TF-Agents 提供了便捷的 `AtariPreprocessing` 包装器做预处理。以下是支持的预处理：

#### 灰度和降采样

- 将观察转换为灰度，并降采样（默认是  $84 \times 84$  像素）

#### 最大池化

- 游戏的最后两帧使用  $1 \times 1$  过滤器做最大池化。是为了去除闪烁点。

#### 跳帧

- 智能体每隔  $n$  个帧做一次观察（默认是 4），对于每一帧，动作都要重复几次，并收集所有的奖励。这么做可以有效加速游戏，因为奖励延迟降低，训练也加速了。

#### 丢弃损失

在某些游戏中，奖励是基于得分的，所以智能体死掉的话，不会立即受到惩罚。一种方法是当死掉时，立即结束游戏。这种做法有些争议，所以默认是关掉的。

因为默认 Atari 环境已经应用了随机跳帧和最大池化，我们需要加载原生不跳帧的变体，`BreakoutNoFrameskip-v4`。另外，从 Breakout 游戏中的一帧并不能知道球的方向和速度，这会使得智能体很难玩好游戏（除非这是一个 RNN 智能体，它可以在步骤之间保存状态）。应对方法之一是使用一个环境包装器，沿着每个频道维度，将多个帧叠起来做输出。`FrameStack4` 包装器实现了这个策略，返回四个帧的栈式结果。下面就创建一个包装过的 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])

```

预处理的结果展示在图 18-12 中。可以看到解析度更低了，但足够玩游戏了。另外，帧沿着频道维度叠起来，所以红色表示的是三步之前到现在的帧，绿色是从两步之前，蓝色是前一帧，粉色是当前帧。根据这一帧的观察，智能体可以看到球是像左下角移动的，所以应该继续将板子向左移动（和前面一步相同）。

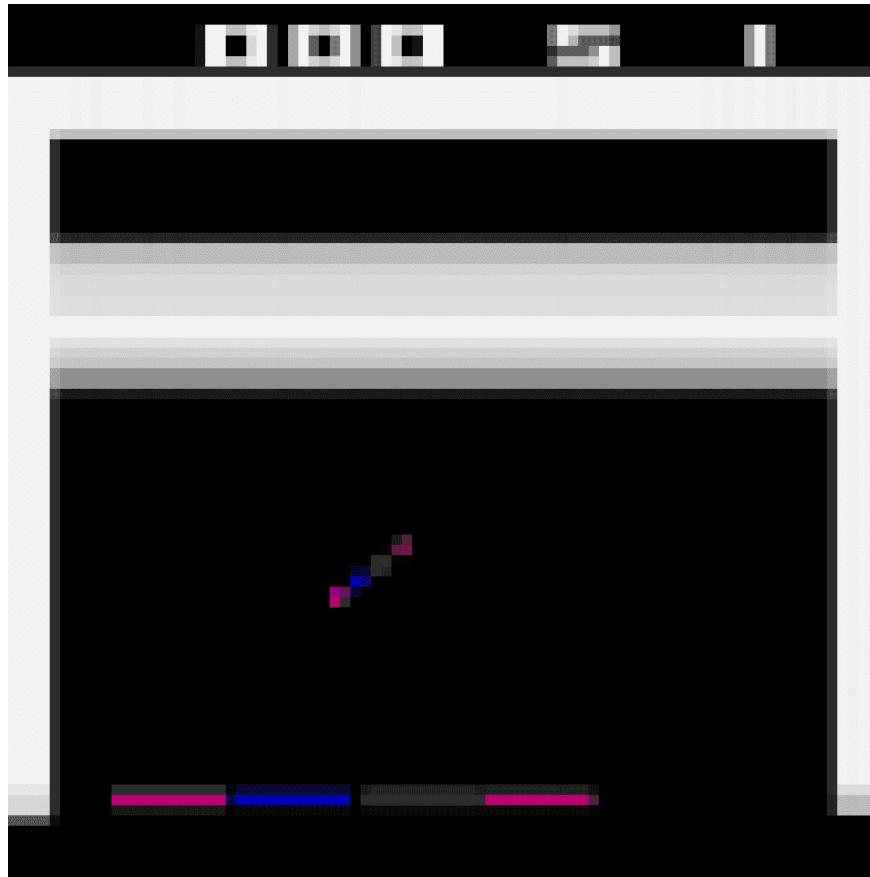


图 18-12 预处理 Breakout 观察

最后，可以将环境包装进 `TFPyEnvironment`：

```

from tf_agents.environments.tf_py_environment import TFPyEnvironment
tf_env = TFPyEnvironment(env)

```

这样就能在 TensorFlow 图中使用这个环境（在底层，这个类使用的 `tf.py_function()`，这可以让图调用任何 Python 代码）。有了 `TFPyEnvironment` 类，TF-Agents 支持纯 Python 环境和基于 TensorFlow 环境。更为一般的，TF-Agents 支持并提供了纯 Python 和基于 TensorFlow 的组件（智能体，接力缓存，指标，等等）。

有了 Breakout 环境，预处理和 TensorFlow 支持，我们必须创建 DQN 智能体，和其它要训练的组件。下面看看系统架构。

## 训练架构

TF-Agents 训练程序通常分为两个并行运行的部分，见图 18-13：左边，driver 使用收集策略探索环境选择动作，并收集轨迹（即，经验），将轨迹发送给观测器，观测器将轨迹存储到接力缓存中；右边，智能体从接力缓存中取出轨迹批次，然后训练网络。总而言之，左边的部分探索环境、收集轨迹，右边的部分学习更新收集策略。

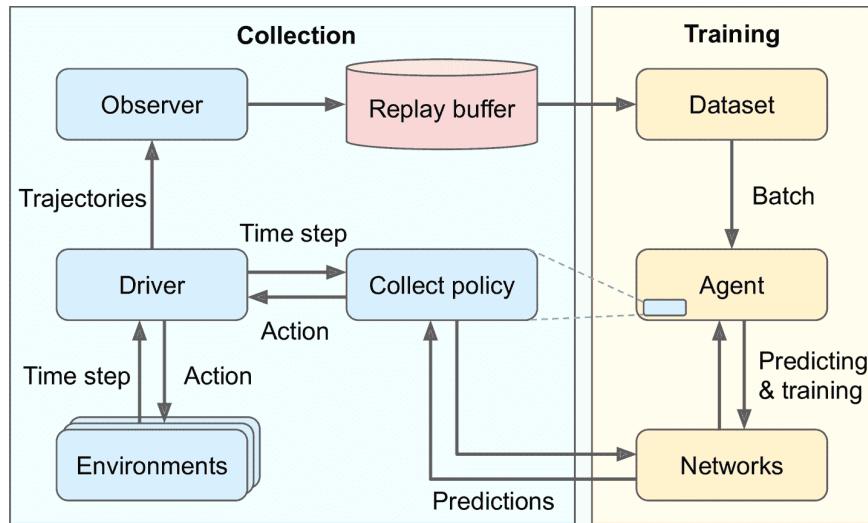


图 18-13 一个典型的 TF-Agents 训练架构

这张图有些疑惑点，回答如下：

- 为什么使用多个环境呢？这是为了让驱动并行探索多个环境的复制，发挥 CPU、GPU 的能力，给训练算法提供低关联的轨迹。
- 什么是轨迹？这是从一个时间步向下一个时间步过渡的简洁表征，或是一连串连续的从时间步  $n$  到时间步  $n+t$  的过渡。驱动收集的轨迹传给观测器，再将其存入接力缓存，接着再被采样用来训练。
- 为什么需要观测器？驱动不能直接保存轨迹吗？事实上，驱动可以直接保存轨迹，但这么做的话，会使得架构不够灵活。例如，如果不想使用接力缓存，该怎么做呢？如果想用轨迹做一些其它事情，比如计算指标，该怎么做呢？事实上，观测器是使用轨迹作为参数的任意函数。可以用观测器将轨迹存入接力缓存，或保存为 TFRecord 文件，或计算指标，或其它事情。另外，可以将多个观测器传给驱动，广播轨迹。

提示：尽管这个架构是最常见的，但是可以尽情自定义，可以更换成自己的组件。事实上，除非是研究新的 RL 算法，更适合使用自定义的环境来做自己的任务。要这么做，需要创建一个自定义类，继承自 `tf_agents.environments.py_environment` 包的 `PyEnvironment` 类，并重写一些方法，比如 `action_spec()`、`observation_spec()`、`_reset()`、`_step()`（见笔记本的章节 Creating a Custom TF\_Agents Environment）。

现在创建好了所有组件：先是深度 Q 网络，然后是 DQN 智能体（负责创建收集策略），然后是接力缓存和观测器，一些训练指标，驱动，最后是数据集。有了所有组件之后，先用一些轨迹填充接力缓存，然后运行主训练循环。因此，从创建深度 Q 网络开始。

## 创建深度 Q 网络

TF-Agents 库在 `tf_agents.networks` 包和子包中提供了许多网络。我们使用 `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` 的输入是观察，每个动作输出一个 Q 值，所以必须给出观察和动作的配置。先是预处理层：一个 `lambda` 层将观察转换为 32 位浮点数，并做归一化（范围落到 0.0 和 1.0 之间）。观察包含无符号字节，占用空间是 32 位浮点数的四分之一，这就是为什么不在前面将观察转换为 32 位浮点数；我们要节省接力缓存的内存空间。接着，网络使用三个卷积层：第一个有 32 个  $8 \times 8$  过滤器，步长是 4，第二个有 64 个  $8 \times 8$  过滤器，步长是 2，第三个层有 64 个  $8 \times 8$  的过滤器，步长为 1。最后，使用一个有 512 个神经元的紧密层，然后是一个有 4 个神经元的紧密输出层，输出是 Q 值（每个动作一个 Q 值）。所有卷积层和除了输出层的紧密层使用 ReLU 激活函数（可以通过设置参数 `activation_fn` 改变）。输出层不使用激活函数。

`QNetwork` 的底层包含两个部分：一个处理观察的编码网络，和一个输出 Q 值的输出层。TF-Agent 的 `EncodingNetwork` 类实现了多种智能体都使用了的神经网络架构（见图 18-14）。

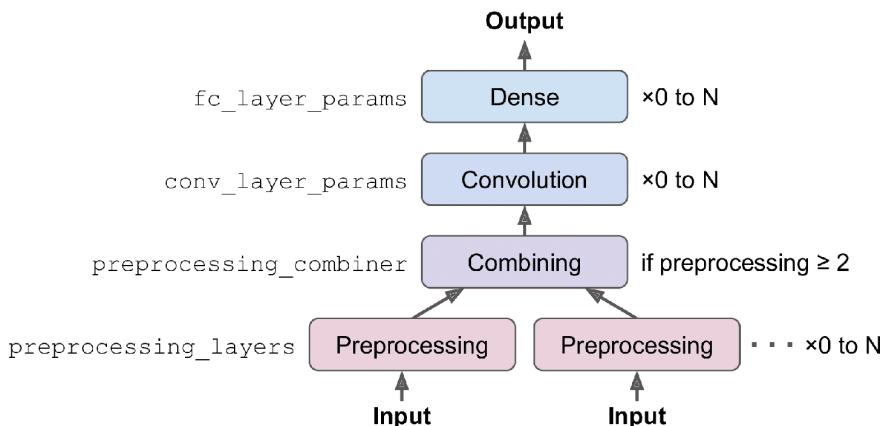


图 18-14 编码网络架构

可能有一个或多个输入。例如，如果每个观察包括传感器数据加上摄像头图片，就有两个输入。每个输入可能需要一些预处理步骤，你可以通过 `preprocessing_layers` 参数指定 Keras 层列表，每个输入有一个预处理层，网络

会将层应用到每个对应的输入上（如果输入需要多个预处理层，可以传入一个完整模型，因为 Keras 模型也可以用作层）。如果有两个或更多输入，必须通过参数 `preprocessing_combiner` 传入其它层，将预处理层的输出合并成一个输出。

然后，编码层会顺序应用一列卷积层，只要指定参数 `conv_layer_params`。这是一个包含 3 个元组的列表（每个卷积层一个元组），指明过滤器的数量、核大小，步长。卷积层之后，如果设置参数 `fc_layer_params`，编码网络可以应用一串紧密层：参数 `fc_layer_params` 是一个列表，包含每个紧密层的神经元数。另外，通过参数 `dropout_layer_params`，还可以传入丢弃率列表（每个紧密层一个），给每个紧密层设置丢弃。`QNetwork` 将编码网络的输出传入给紧密输出层（每个动作一个神经元）。

**笔记：** `QNetwork` 类非常灵活，可以创建许多不同的架构，如果需要更多的灵活性，还可以通过创建自己的类：扩展类 `tf_agents.networks.Network`，像常规自定义 Keras 层一样实现。`tf_agents.networks.Network` 类是 `keras.layers.Layer` 类的子类，前者添加了一些智能体需要的功能，比如创建网络的浅复制（即，只复制架构，不复制权重）。例如，`DQNAgent` 使用这个功能创建在线模型。

有了 DQN，接下来创建 DQN 智能体。

## 创建 DQN 智能体

利用 `tf_agents.agents` 包和它的子包，TF-Agents 库实现了多种类型的智能体。我们使用类 `tf_agents.agents.dqn.dqn_agent.DqnAgent`：

```
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()
```

逐行看下代码：

- 首先创建计算训练步骤数的变量。
- 然后创建优化器，使用 2015 DQN 论文相同的超参数。
- 接着，创建对象 `PolynomialDecay`，根据当前的训练步骤（用于降低学习率，也可以是其它值），用于计算  $\epsilon$  贪婪收集策略的  $\epsilon$  值。在 100 万 ALE 帧内（等于 250000 步骤，因为跳帧周期等于 4），将  $\epsilon$  值从 1 降到 0.01（也是 2015 DQN 论文的用值）。另外，每隔 4 步（即，16 个 ALE 帧），所以  $\epsilon$  值是在 62500 个训练步内下降的。

- 然后创建 `DQNAgent`，传入时间步和动作配置、`QNetwork`、优化器、目标模型更新间的训练步骤数、损失函数、衰减率、变量 `train_step`、返回  $\epsilon$  值的函数（不接受参数，这就是为什么使用匿名函数传入 `train_step` 的原因）。注意，损失函数对每个实例返回一个误差，不是平均误差，所以要设置 `reduction="none"`。
- 最后，启动智能体。

然后，创建接力缓存和观测器。

## 创建接力缓存和观测器

TF-Agents 库在 `tf_agents.replay_buffers` 包实现了多种接力缓存。一些是用纯 Python 写的（模块名开头是 `py_`），其它是基于 TensorFlow 的（开头是 `tf_`）。我们使用 `tf_agents.replay_buffers.tf_uniform_replay_buffer` 包追踪的 `TFUniformReplayBuffer` 类。它实现了高效均匀采样的接力缓存：

```
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)
```

看一下这些参数：

`data_spec`

- 数据的配置会存储在接力缓存中。DQN 智能体知道收集数据什么样，通过属性 `collect_data_spec` 做数据配置。

`batch_size`

- 轨迹数量添加到每个步骤。在这个例子中，轨迹数是 1，因为驱动每个步骤执行一个动作收集一个轨迹。如果环境是一个批次化的环境（环境在每个时间步接收批次动作，返回批次观察），则驱动必须在每个时间步保存批次的轨迹。因为使用的是 TensorFlow 接力缓存，需要知道批次大小（创建计算图）。批次化环境的一个例子是 `ParallelPyEnvironment`（出自包 `tf_agents.environments.parallel_py_environment`）：用独立进程并行运行多个环境（对于相同动作和观察配置，进程可以不同），每个步骤接收批次化的动作，并在环境中执行（每个环境一个动作），然后返回所有观察结果。

`max_length`

- 接力缓存的最大大小。我们创建一个可以存储 100 万个轨迹的接力缓存（和 2015 DQN 论文一样）。这需要不少内存。

提示：当存储两个连续的轨迹，它们包含两个连续的观察，每个观察有四个帧（因为包装器是 `FrameStack4`），但是第二个观察中的三个帧是多余的（第一个观察中已经存在了）。换句话说，使用的内存大小是必须的四倍。要避免这个问题，可以使用包 `tf_agents.replay_buffers.py_hashed_replay_buffer` 的 `PyHashedReplayBuffer`：它能沿着观察的最后一个轴对存储的轨迹去重。

现在创建向接力缓存写入轨迹的观测器。观测器就是一个接收轨迹参数的函数（或是调用对象），所以可以直接使用方法 `add_method()`（绑定 `replay_buffer` 对象）作为观测器：

```
replay_buffer_observer = replay_buffer.add_batch
```

如果想创建自己的观测器，可以一个包含参数 `trajectory` 的函数。如果必须有状态，可以写一个包含方法 `__call__(self, trajectory)` 的类。例如，下面是一个每次调用，计数器都会加 1 的观测器（除了轨迹表示周期间的边界，不算成一步），每隔 100 次累加，显示总数（`\r` 和 `end=""` 保证展示的计数器处于一条线）。

```
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="")
```

接下来创建一些训练指标。

## 创建训练指标

TF-Agents 库再 `tf_agents.metrics` 包中实现了几个 RL 指标，一些是基于纯 Python 的，一些是基于 TensorFlow 的。创建一些指标统计周期数、步骤数、周期的平均数、平均周期长度：

```
from tf_agents.metrics import tf_metrics
train_metrics = [
    tf_metrics.NumberOfEpisodes(),
    tf_metrics.EnvironmentSteps(),
    tf_metrics.AverageReturnMetric(),
    tf_metrics.AverageEpisodeLengthMetric(),
]
```

**笔记**：训练或实现策略时，对奖励做衰减是合理的，这是为了平衡当前奖励与未来奖励的平衡。但是，当周期结束时，通过对所有未衰减的奖励求和来做评估。出于这个原因，`AverageReturnMetric` 计算了每个周期未衰减奖励的和，并追踪平均值。

任何时候，可以调用 `result()` 方法获取指标（例如，`train_metrics[0].result()`）。或者，可以调用 `log_metrics(train_metrics)` 记录所有指标（这个函数位于 `tf_agents.eval.metric_utils` 包）：

```
>>> 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
```

接下来创建收集驱动。

## 创建收集驱动

正如图 18-13，驱动是使用给定策略探索环境的对象，收集经验，并广播给 `observer`。在每一步，发生的事情如下：

- 驱动将当前时间步传给收集策略，收集策略使用时间步选择动作，并返回包含动作的动作步对象。
- 驱动然后将动作传给环境，环境返回下一个时间步。
- 最后，驱动创建一个轨迹对象表示过渡，并广播给所有观察。

一些策略，比如 RNN 策略，是有状态的：策略根据给定的时间步和内部状态选择动作。有状态策略在动作步返回自己的状态，驱动会在下一个时间步将这个状态返回给策略。另外，驱动将策略状态保存到轨迹中（在字段 `policy_info` 中）：当智能体采样一条轨迹，它必须设置策略的状态设为采样时间步时的状态。

另外，就像前面讨论的，环境可能是批次化的环境，这种情况下，驱动将批次化的时间步传给策略（即，时间步对象包含批次观察、批次步骤类型、批次奖励、批次衰减，这四个批次的大小相同）。驱动还传递前一批次的策略状态。然后，策略返回去批次动作步，包含着批次动作和批次策略状态。最后，驱动创建批次化轨迹（即，轨迹包含批次步骤类型、批次观察、批次动作、批次奖励，更一般地，每个轨迹属性一个批次，所有批次大小相同）。

有两个主要的驱动类：`DynamicStepDriver` 和 `DynamicEpisodeDriver`。第一个收集给定数量步骤的经验，第二个收集给定数量周期数的经验。我们想收集每个训练迭代的四个步骤的经验（正如 2015 DQN 论文），所以创建一个 `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
```

传入环境、智能体的收集策略、观测器列表（包括接力缓存观测器和训练指标），最后是要运行的步骤数（这个例子中是 4）。现在可以调用方法 `run()` 来运行，但最好先用纯随机策略收集的经验先填充接力缓存。要这么做，可以使用类 `RandomTFPolicy` 创建第二个驱动，运行 20000 步这个策略（等于 80000 个模拟帧，正如 2015 DQN 论文）。可以用 `ShowProgress` 观测器展示进展：

```
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()
```

快要能运行训练循环了。只需要最后一个组件：数据集。

## 创建数据集

要从接力缓存采样批次的轨迹，可以调用 `get_next()` 方法。这返回了轨迹的批次，还返回了含有样本 id 和采样概率的 `BufferInfo` 对象（可能对有些算法有用，比如 PER）。例如，下面的代码采样了一个包含两条轨迹的批次（子周期），每个包含三个连续步。这些子周期见图 18-15（每行包含一个周期的三个连续步）：

```

>>> 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` 对象是一个命名元组，有 7 个字段。每个字段包含一个张量，前两个维度是 2 和 3（因为有两条轨迹，每个三个时间步）。这解释了为什么 `observation` 字段的形状是 `[2, 3, 84, 84, 4]`：这是两条轨迹，每条轨迹三个时间步，每步的观察是  $84 \times 84 \times 4$ 。相似的，`step_type` 张量的形状是 `[2, 3]`：在这个例子中，两条轨迹包含三个连续步骤，步骤是在周期的中部，（类型是 `1, 1, 1`）。在第二条轨迹中，看不到第一个观察中左下方的球，在接下来的两个观察中，球消失了，所以智能体会死，但周期不会马上结束，因为还剩几条命。

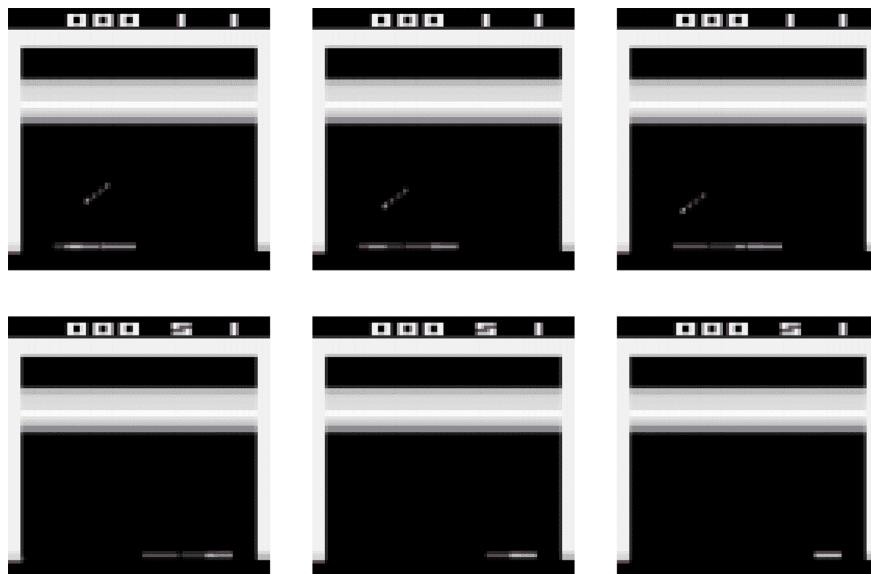
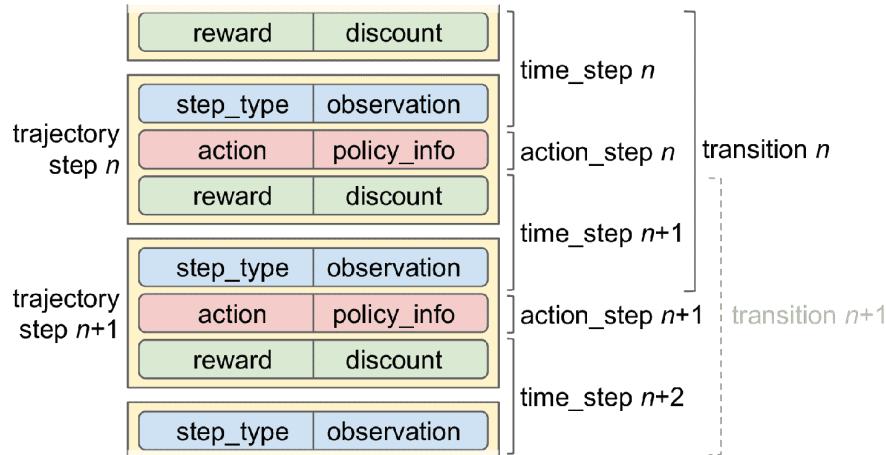


图 18-15 包含三个连续步骤的两条轨迹

每条轨迹是连续时间步和动作步的简洁表征，初衷是为了避免繁琐，怎么做呢？见图 18-16，过渡  $n$  由时间步  $n$ 、动作步  $n$ 、时间步  $n+1$  组成，而过渡  $n+1$  由时间步  $n+1$ 、动作步  $n+1$ 、时间步  $n+2$ 。如果将这两个过渡直接存入接力缓存，时间步`是重复的。为了避免重复，第  $n$  个轨迹步只包括时间步  $n$  的类型和观察（不是奖励和衰减），不包括`的观察（但是，不包括下一个时间步类型的复制）。



Note: trajectory step  $n$  also includes next\_step\_type  $n = \text{step\_type } n+1$

图 18-16 轨迹，过渡，时间步和动作步

因此，如果有批次轨迹，每个轨迹有  $t+1$  步骤（从时间步  $n$  到时间步  $n+t$ ），包含从时间步  $n$  到时间步  $n+t$  的所有数据，但没有奖励和时间步  $n$  的衰减（但包括时间步  $n+t+1$  的奖励和衰减）。这表示  $t$  过渡（ $n$  到  $n+1$ ,  $n+1$  到  $n+2$ , ...,  $n+t-1$  到  $n+t$ ）

模块 `tf_agents.trajectories.trajectory` 中的函数 `to_transition()` 将批次化的轨迹转变为包含批次 `time_step`、`action_step`、`next_time_step` 的列表。注意，第二个维度是 2，而不是 3，这是因为  $t+1$  个时间步之间有  $t$  个过渡：

```
>>> 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
```

笔记：采样的轨迹可能会将两个（或多个）周期重叠！这种情况下，会包含边界过渡，意味着过渡的 `step_type` 等于 2（结束），`next_step_type` 等于 0（开始）。当然，TF-Agents 可以妥善处理这些轨迹（例如，通过在碰到边界时重新设置策略状态）。轨迹的方法 `is_boundary()` 返回只是每一步是否是边界的张量。

对于主训练循环，不使用 `get_next()`，而是用 `tf.data.Dataset`。这样，就能借助 Data API 的高效（并行计算和预提取）。要这么做，可以调用接力缓存的 `as_dataset()` 方法：

```
dataset = replay_buffer.as_dataset(
    sample_batch_size=64,
    num_steps=2,
    num_parallel_calls=3).prefetch(3)
```

在每个训练步骤，提取包含 64 条轨迹的批次（和 2015 DQN 论文一样），每条轨迹有两步（即，2 步为 1 个完整过渡，包括下一步的观察）。这个数据集能并行处理三条轨迹，预提取三条轨迹。

笔记：对于策略算法，比如策略梯度，每个经验只需采样一次，训练完就可以丢掉。在这个例子中，你还可以使用一个接力缓存，但使用接力缓存的 `gather_all()` 方法，在每个训练迭代获取轨迹张量，训练完，再动过 `clear()` 方法清空接力缓存。

有了所有组件之后，就可以训练模型了。

## 创建训练循环

要加速训练，将主函数转换为 TensorFlow 函数。可以使用函数 `tf_agents.utils.common.function()`，它包装了 `tf.function()`，还有一些其它选项：

```
from tf_agents.utils.common import function
collect_driver.run = function(collect_driver.run)
agent.train = function(agent.train)
```

写一个小函数，可以 `n_iterations` 次运行主训练循环：

```
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)
```

这个函数先向收集策略询问初始状态（给定环境批次大小，这个例子中是 1）。因为策略是无状态的，返回的是空元组（所以可以写成 `policy_state = ()`）。然后，创建一个数据集的迭代器，并运行训练循环。在每个迭代，调用驱动的 `run()` 方法，传入当前的时间步（最初是 `None`）和当前的策略状态。运行收集策略，收集四步的经验，将收集到的轨迹广播给接力缓存和指标。然后，从数据集采样一个批次轨迹，传给智能体的 `train()` 方法。返回对象 `train_loss`，可能根据智能体的类型有变动。接着，展示迭代数和训练损失，每隔 1000 次迭代，输出所有指标的日志。现在可以调用 `train_agent()` 做一些迭代，智能体就能逐渐学会玩 Breakout 了。

```
train_agent(10000000)
```

训练需要大量算力和极大的耐心（根据硬件，可能需要几个小时甚至几天），可能还需要用不同的随机种子多次运行，以得到更好的结果，但是训练完成后，智能体在玩 Breakout 就比人厉害了。你还可以在其它 Atari 游戏上训练这个 DQN 智能体：智能体对于大多数动作游戏都可以超越人的表现，但是智能体对长故事线游戏不擅长。

## 流行 RL 算法概览

本章结束前，快速浏览一些流行的 RL 算法：

### 演员评论家算法

- 将策略梯度和深度 Q 网络结合而成 RL 算法族。演员评论家智能体包含两个神经网络：一个策略网络和一个 DQN。用智能体的经验正常训练 DQN。与常规 PG 相比，策略网络的学习有所不同：智能体（演员）依赖 DQN（评论家）估计的动作值。就像运动员（智能体）在教练（DQN）的帮助下学习。

### 异步优势演员评论家算法 (A3C)

- 这是 DeepMind 在 2016 年推出的重要的演员评论家算法的变体，其中多个智能体并行学习，探索环境的不同复制。每隔一段间隔，每个智能体异步更新主网权重，然后从网络拉取最新权重。每个智能体都对网络产生共现，也从其它智能体学习。另外，DQN 不估计 Q 值，而是估计每个动作的优势，这样可以稳定训练。

### 优势演员评论家算法 (A2C)

- A3C 算法的变体，去除了异步。所有模型更新是同步的，所以梯度更新倾向于大批次，可以让模型更好地利用 GPU。

### 软演员评论家算法 (SAC)

- Tuomas Haarnoja 和其它 UC Berkeley 研究员在 2018 年提出的演员评论家变体。这个算法不仅学习奖励，还最大化其动作的熵。换句话说，在尽可能获取更多奖励的同时，尽量不可预测。这样可以鼓励智能体探索环境，可以加速训练。在 DQN 的估计不好时，可以避免重复执行相同动作。这个算法采样非常高效（与前面的算法相反，前者采样慢）。TF-Agents 中有 SAC。

### 近似策略优化 (PPO)

- 基于 A2C 的算法，它能裁剪函数的损失，避免过量权重更新（会导致训练不稳定）。PPO 是信任区域策略优化 (TRPO) 的简化版本，作者是 John Schulman 和其它 OpenAI 研究员。OpenAI 在 2019 年四月弄了个大新闻，他们用基于 PPO 的 OpenAI Five 打败了多人游戏 Dota2 的世界冠军。TF-Agents 中有 PPO。

### 基于好奇探索

- RL 算法中反复出现的问题是奖励过于稀疏，这使得学习太慢且低效。Deepak Pathak 和其它 UC Berkeley 的研究员提出了解决方法：忽略奖励，让智能体极度好奇地探索环境？奖励变为了智能体的一部分，而不是来自环境。相似的，让孩子变得更好奇，比纯粹的奖励孩子，能取得更好的结果。怎么实现呢？智能体不断地预测动作的结果，并探索结果不匹配预测的环境。换句话说，智能体想得到惊喜。如果结果是可预测的（枯燥），智能体就去其它地方。但是，如果结果不可预测，智能体发现无法控制结果，也会变得无聊。只用好奇心，作者成功地训练智能体玩电子游戏：即使智能体失败不会受惩罚，游戏也会结束，智能体是玩腻了。

这一章学习了许多主题：策略梯度、马尔科夫链、马尔科夫决策过程、Q 学习、近似 Q 学习、深度 Q 学习及其变体（固定 Q 值目标、双 DQN、决斗 DQN、优先经验接力）。还讨论了如何使用 TF-Agents 训练智能体，最后浏览了一些流行的算法。强化学习是一个庞大且令人兴奋的领域，每天都有新主意和新算法冒出来，希望这章除能激发你的好奇心！

## 练习

- 如何定义强化学习？它与传统的监督和非监督学习有什么不同？
- 你能想到什么本章没有提到过的强化学习的应用？环境是什么？智能体是什么？什么是可能的动作，什么是奖励？

3. 什么是衰减率？如果你修改了衰减率那最优策略会变化吗？
4. 如何测量强化学习智能体的表现？
5. 什么是信用分配问题？它怎么出现的？怎么解决？
6. 使用接力缓存的目的是什么？
7. 什么是 off 策略 RL 算法？
8. 使用策略梯度处理 OpenAI gym 的“LunarLander-v2”环境。需要安装 Box2D 依赖（`python3 -m pip install gym[box2d]`）。
9. 用任何可行的算法，使用 TF-Agents 训练可以达到人类水平的可以玩 SpaceInvaders-v4 的智能体。
10. 如果你有大约 100 美元备用，你可以购买 Raspberry Pi 3 再加上一些便宜的机器人组件，在 Pi 上安装 TensorFlow，然后让我们嗨起来~！举个例子，看看 Lukas Biewald 的这个[有趣的帖子](#)，或者看看 GoPiGo 或 BrickPi。从简单目标开始，比如让机器人转向最亮的角度（如果有光传感器）或最近的物体（如果有声呐传感器），并移动。然后可以使用深度学习：比如，如果机器人有摄像头，可以实现目标检测算法，检测人并向人移动。还可以利用 RL 算法让智能体自己学习使用马达达到目的。

参考答案见附录 A。

## 十九、规模化训练和部署 TensorFlow 模型

译者：[@SeanCheney](#)

有了能做出惊人预测的模型之后，要做什么呢？当然是部署生产了。这只要用模型运行一批数据就成，可能需要写一个脚本让模型每夜都跑着。但是，现实通常会更复杂。系统基础组件都可能需要这个模型用于实时数据，这种情况需要将模型包装成网络服务：这样的话，任何组件都可以通过 REST API 询问模型。随着时间的推移，你需要用新数据重新训练模型，更新生产版本。必须处理好模型版本，平稳地过渡到新版本，碰到问题的话需要回滚，也许要并行运行多个版本做 AB 测试。如果产品很成功，你的服务可能每秒会有大量查询，系统必须提升负载能力。提升负载能力的方法之一，是使用 TF Serving，通过自己的硬件或通过云服务，比如 Google Cloud API 平台。TF Serving 能高效服务化模型，优雅处理模型过渡，等等。如果使用云平台，还能获得其它功能，比如强大的监督工具。

另外，如果有很多训练数据和计算密集型模型，则训练时间可能很长。如果产品需要快速迭代，这么长的训练时间是不可接受的（例如，新闻推荐系统总是推荐上个星期的新闻）。更重要的，过长的训练时间会让你没有时间试验新想法。在机器学习中（其它领域也是），很难提前知道哪个想法有效，所以应该尽量多、尽量快尝试。加速训练的方法之一是使用 GPU 或 TPU。要进一步加快，可以在多个机器上训练，每台机器上都有硬件加速。TensorFlow 的 Distribution Strategies API 可以轻松实现多机训练。

本章我们会介绍如何部署模型，先是 TF Serving，然后是 Google Cloud AI 平台。还会快速浏览如何将模型部署到移动 app、嵌入式设备和网页应用上。最后，会讨论如何用 GPU 加速训练、使用 Distribution Strategies API 做多机训练。

### TensorFlow 模型服务化

训练好 TensorFlow 模型之后，就可以在 Python 代码中使用了：如果是 `tf.keras` 模型，调用 `predict()` 模型就成。但随着基础架构扩张，最好是将模型包装在服务中，它的唯一目的是做预测，其它组件查询就成（比如使用 REST 或 gRPC API）。这样就将模型和其它组件解耦，可以方便地切换模型或扩展服务（独立于其它组件），做 AB 测试，确保所有组件都是依赖同一个模型版本。还可以简化测试和开发，等等。可以使用任何技术做微服务（例如，使用 Flask），但有了 TF Serving，为什么还要重复造轮子呢？

### 使用 TensorFlow Serving

TF Serving 是一个非常高效，经过实战检测的模型服务，是用 C++ 写成的。可以支持高负载，服务多个模型版本，并监督模型仓库，自动部署最新版本，等等（见 19-1）。

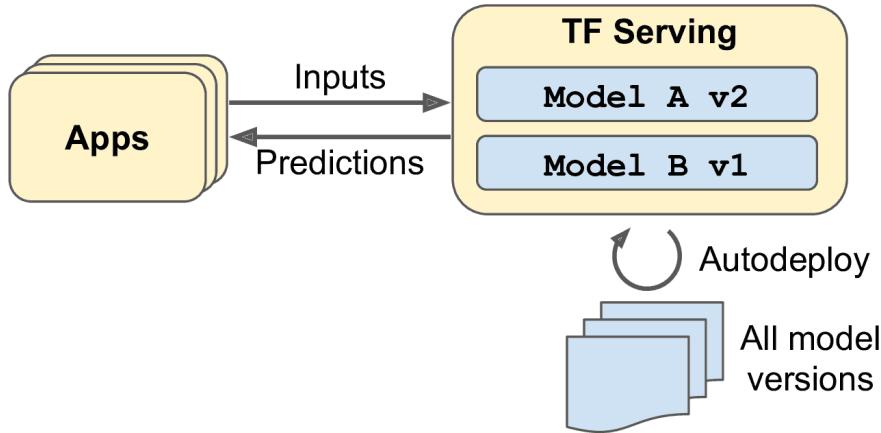


图 19-1 TF Serving 可以服务多个模型，并自动部署每个模型的最新版本

假设你已经用 `tf.keras` 训练了一个 MNIST 模型，要将模型部署到 TF Serving。第一件事是输出模型到 TensorFlow 的 `SavedModel` 格式。

## 输出 SavedModel

TensorFlow 提供了简便的函数 `tf.saved_model.save()`，将模型输出为 `SavedModel` 格式。只需传入模型，配置名字、版本号，这个函数就能保存模型的计算图和权重：

```
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)
```

通常将预处理层包含在最终模型里，这样部署在生产中，就能接收真实数据。这样可以避免在应用中单独做预处理。将预处理和模型绑定，还能防止两者不匹配。

**警告：**因为 `SavedModel` 保存了计算图，所以只支持基于 TensorFlow 运算的模型，不支持 `tf.py_function()` 运算（它包装了任意 Python 代码）。也不支持动态 `tf.keras` 模型（见附录 G），因为这些模型不能转换成计算图。动态模型需要用其它工具（例如，Flask）服务化。

`SavedModel` 表示了模型版本。它被保存为一个包含 `saved_model.pb` 文件的目录，它定义了计算图（表示为序列化协议缓存），变量子目录包含了变量值。对于含有大量权重的模型，这些变量值可能分割在多个文件中。`SavedModel` 还有一个 `assets` 子目录，包含着其余数据，比如词典文件、类名、一些模型的样本实例。目录结构如下（这个例子中，没有使用 `assets`）：

```
my_mnist_model
└── 0001
    ├── assets
    ├── saved_model.pb
    └── variables
        └── variables.data-00000-of-00001
            └── variables.index
```

可以使用函数 `tf.saved_model.load()` 加载 `SavedModel`。但是，返回的对象不是 Keras 模型：是 `SavedModel`，包括计算图和变量值。可以像函数一样做预测（输入是张量，还要设置参数 `training`，通常设为 `False`）：

```
saved_model = tf.saved_model.load(model_path)
y_pred = saved_model(X_new, training=False)
```

另外，可以将 SavedModel 的预测函数包装进 Keras 模型：

```
inputs = keras.layers.Input(shape=...)
outputs = saved_model(inputs, training=False)
model = keras.models.Model(inputs=[inputs], outputs=[outputs])
y_pred = model.predict(X_new)
```

TensorFlow 还有一个命令行工具 `saved_model_cli`，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
```

SavedModel 包含一个或多个元图。元图是计算图加上了函数签名定义（包括输入、输出名，类型和形状）。每个元图可以用一组标签做标识。例如，可以用一个元图包含所有的计算图，包括训练运算（例如，这个元图的标签是 "train"）。但是，当你将 `tf.keras` 模型传给函数 `tf.saved_model.save()`，默认存储的是一个简化的 SavedModel：保存一个元图，标签是 "serve"，包含两个签名定义，一个初始化函数（`__saved_model_init_op`）和一个默认的服务函数（`serving_default`）。保存 `tf.keras` 模型时，默认服务函数对应模型的 `call()` 函数。

`saved_model_cli` 也可以用来做预测（用于测试，不是生产）。假设有一个 NumPy 数组（`X_new`），包含三张用于预测的手写数字图片。首先将其输出为 NumPy 的 `npy` 格式：

```
np.save("my_mnist_tests.npy", X_new)
```

然后，如下使用 `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]]
```

输出包含 3 个实例的 10 个类的概率。现在有了可以工作的 SavedModel，下一步是安装 TF Serving。

## 安装 TensorFlow Serving

有多种方式安装 TF Serving：使用 Docker 镜像、使用系统的包管理器、从源代码安装，等等。我们使用 Docker 安装的方法，这是 TensorFlow 团队高度推荐的方法，不仅安装容易，不会扰乱系统，性能也很好。需要先安装 Docker。然后下载官方 TF Serving 的 Docker 镜像：

```
$ docker pull tensorflow/serving
```

创建一个 Docker 容器运行镜像：

```
$ 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 ...
```

这样，TF Serving 就运行起来了。它加载了 MNIST 模型（版本 1），通过 gRPC（端口 8500）和 REST（端口 8501）运行。下面是命令行选项的含义：

`-it`

使容器可交互（`ctrl-C` 关闭），展示服务器的输出。

`--rm`

停止时删除容器。但不删除镜像。

`-p 8500:8500`

将 Docker 引擎将主机的 TCP 端口 8500 转发到容器的 TCP 端口 8500。默认时，TF Serving 使用这个端口服务 gRPC API。

`-p 8501:8501`

将 Docker 引擎将主机的 TCP 端口 8501 转发到容器的 TCP 端口 8501。默认时，TF Serving 使用这个端口服务 REST API。

`-v "$ML_PATH/my_mnist_model:/models/my_mnist_model"`

使主机的 `$ML_PATH/my_mnist_model` 路径对容器的路径 `/models/mnist_model` 开放。在 Windows 上，可能需要将 `/` 替换为 `\`。

`-e MODEL_NAME=my_mnist_model`

将容器的 `MODEL_NAME` 环境变量，让 TF Serving 知道要服务哪个模型。默认时，它会在路径 `/models` 查询，并会自动服务最新版本。

`tensorflow/serving`

镜像名。

现在回到 Python 查询服务，先使用 REST API，然后使用 gRPC API。

## 用 REST API 查询 TF Serving

先创建查询。必须包含想要调用的函数签名的名字，和输入数据：

```

import json
input_data_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})

```

注意，json 格式是 100% 基于文本的，因此 `X_new` NumPy 数组要转换为 Python 列表，然后 json 格式化：

```

>>> 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]]]}'

```

通过发送 HTTP POST 请求，将数据发送给 TF Serving。使用 `requests` 就成：

```

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()

```

响应是一个字典，唯一的键是 `"predictions"`，它对应的值是预测列表。这是一个 Python 列表，将其转换为 NumPy 数组，小数点保留两位：

```

>>> y_proba = np.array(response["predictions"])
>>> y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. ,
       0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. ,
       0. , 0.96, 0.01, 0. , 0. , 0. , 0. , 0.01, 0.01,
       0. ]])

```

现在就有预测了。模型 100% 肯定第一张图是类 7，99% 肯定第二张图是类 2，96% 肯定第三章图是类 1。

REST API 既优雅又简单，当输入输出数据不大时，可以工作的很好。另外，客户端无需其它依赖就能做 REST 请求，其它协议不一定成。但是，REST 是基于 JSON 的，JSON 又是基于文本的，很冗长。例如，必须将 NumPy 数组转换为 Python 列表，每个浮点数都转换成了字符串。这样效率很低，序列化/反序列化很费时，负载大小也高：浮点数要表示为 15 个字符，32 位浮点数要超过 120 比特。这样在传输大 NumPy 数组时，会造成高延迟和高带宽消耗。所以转而使用 gRPC。

**提示：**当传输大量数据时，（如果客户端支持）最好使用 gRPC API，因为它是基于压缩二进制格式和高效通信协议（基于 HTTP/2 框架）。

## 用 gRPC API 查询 TF Serving

gRPC API 的输入是序列化的 `PredictRequest` 协议缓存，输出是序列化的 `PredictResponse` 协议缓存。这些协议缓存是 `tensorflow-serving-api` 库的一部分（通过 PIP 安装）。首先，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))

```

这段代码创建了 `PredictRequest` 协议缓存，填充了需求字段，包括模型名（之前定义的），想要调用的函数签名，最后是输入数据，形式是 `Tensor` 协议缓存。`tf.make_tensor_proto()` 函数创建了一个基于给定张量或 NumPy 数组 (`x_new`) 的 `Tensor` 协议缓存。接着，向服务器发送请求，得到响应（需要用 PIP 安装 `grpcio` 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)
```

这段代码很简单：引入包之后，创建一个 gRPC 通信通道，主机是 `localhost`，端口是 8500，然后用这个通道创建 gRPC 服务，并发送请求，超时时间是 10 秒（因为是同步的，收到响应前是阻塞的）。在这个例子中，通道是不安全的（没有加密和认证），但 gRPC 和 TensorFlow Serving 也支持 SSL/TLS 安全通道。

然后，将 `PredictResponse` 协议缓存转换为张量：

```
output_name = model.output_names[0]
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```

如果运行这段代码，打印 `y_proba.numpy().round(2)`。会得到和之前完全相同的结果。

## 部署新模型版本

现在创建一个新版本模型，将 `SavedModel` 输出到路径 `my_mnist_model/0002`：

```
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)
```

每隔一段时间（可配置），TensorFlow Serving 会检查新的模型版本。如果找到新版本，会自动过渡：默认的，会用上一个模型回复挂起的请求，用新版本模型处理新请求。挂起请求都答复后，前一模型版本就不加载了。可以在 TensorFlow 日志中查看：

```
[...]
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}
```

这个方法提供了平滑的过渡，但会使用很多内存（尤其是 GPU 内存，这是最大的限制）。在这个例子中，可以配置 TF Serving，用前一模型版本处理所有挂起的请求，再加载使用新模型版本。这样配置可以防止在同一时刻加载，但会中断服务一小段时间。

可以看到，TF Serving 使部署新模型变得很简单。另外，如果发现版本 2 效果不如预期，只要删除路径 `my_mnist_model/0002` directory 就能滚回到版本 1。

**提示：**TF Serving 的另一个功能是自动批次化，要使用的话，可以在启动时使用选项 `--enable_batching`。当 TF Serving 在短时间内收到多个请求时（延迟是可配置的），可以自动做批次化，然后再使用模型。这样能利用 GPU 提升性能。模型返回预测之后，TF Serving 会将每个预测返回给正确的客户端。通过提高批次延迟（见选项 `--batching_parameters_file`），可以获得更高的吞吐量。

如果每秒想做尽量多的查询，可以将 TF Serving 部署在多个服务器上，并对查询做负载均衡（见图 19-2）。这需要将 TF Serving 容器部署在多个服务器上。一种方法是使用 Kubernetes，这是一个开源工具，用于在多个服务器上做容器编排。如果你不想购买、维护、升级所有机器，可以使用云平台比如亚马逊 AWS、Microsoft Azure、Google Cloud Platform、IBM 云、阿里云、Oracle 云，或其它 Platform-as-a-Service (PaaS)。管理所有虚拟机、做容器编排（就算有 Kubernetes 的帮助），处理 TF Serving 配置、微调和监控，也是件很耗时的工作。幸好，一些服务提供商可以帮你完成所有工作。本章我们会使用 Google Cloud AI Platform，因为它是唯一带有 TPU 的平台，支持 TensorFlow 2，还有其它 AI 服务（比如，AutoML、Vision API、Natural Language API），也是我最熟悉的。也存在其它服务提供商，比如 Amazon AWS SageMaker 和 Microsoft AI Platform，它们也支持 TensorFlow 模型。

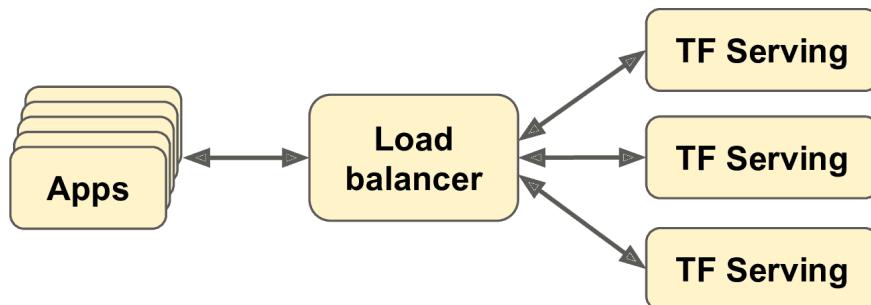


图 19-2 用负载均衡提升 TF Serving

现在，在云上部署 MNIST 模型。

## 在 GCP AI 上创建预测服务

在部署模型之前，有一些设置要做：

1. 登录 Google 账户，到 [Google Cloud Platform \(GCP\) 控制台](#)（见图 19-3）。如果没有 Google 账户，需要创建一个。

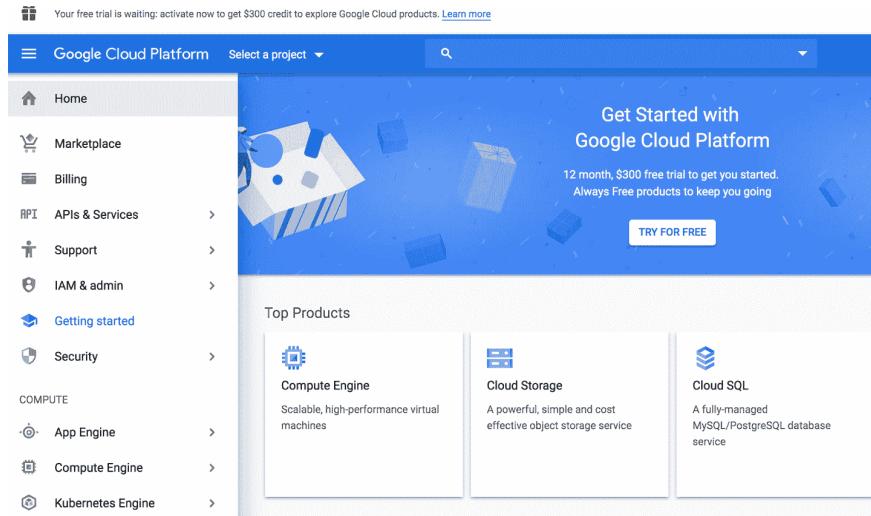


图 19-3 Google Cloud Platform 控制台

1. 如果是第一次使用 GCP，需要阅读、同意条款。写作本书时，新用户可以免费试用，包括价值 300 美元的 GCP 点数，可以使用 12 个月。本章只需一点点 GCP 点数就够。选择试用之后，需要创建支付信息，需要输入信用卡账号：这只是为了验证（避免人们薅羊毛），不必支付。根据需求，激活升级账户。
2. 如果不能用试用账户，就得掏钱了 T\_T。
3. GCP 中的每个资源都属于一个项目。包括所有的虚拟机，存储的文件，和运行的训练任务。创建账户时，GCP 会自动给你创建一个项目，名字是 `My First Project`。可以在项目设置改名。在导航栏选择 `IAM & admin → Settings`，改名，然后保存。项目有一个唯一 ID 和数字。创建项目时，可以选择项目 ID，选好 ID 后后面就不能修改了。项目数字是自动生成的，不能修改。如果你想创建一个新项目，点击 `New Project`，输入项目 ID。

**警告：**不用时一定注意关掉所有服务，否则跑几天或几个月，可能花费巨大。

1. 有了 GCP 账户和支付信息之后，就可以使用服务了。首先需要的 Google Cloud Storage (GCS)：用来存储 SavedModels，训练数据，等等。在导航栏，选择 `Storage → Browser`。所有的文件会存入一个或多个 bucket 中。点击 `Create Bucket`，选择 bucket 名（可能需要先激活 Storage API）。GCS 对 bucket 使用了单一全局的命名空间，所以像 `machine-learning` 这样的名字，可能用不了。确保 bucket 名符合 DNS 命名规则，因为 bucket 名会用到 DNS 记录中。另外，bucket 名是公开的，不要放私人信息。通常用域名或公司名作为前缀，保证唯一性，或使用随机数字作为名字。选择存放 bucket 的地方，其它选项用默认就行。然后点击 `Create`。
2. 上传之前创建的 `my_mnist_model`（包括一个或多个版本）到 bucket 中。要这么做，在 GCS Browser，点击 `bucket`，拖动 `my_mnist_model` 文件夹到 `bucket` 中（见图 19-4）。另外，可以点击 `Upload folder`，选在要上传的 `my_mnist_model` 文件夹。默认时，SavedModel 最大是 250MB，可以请求更大的值。

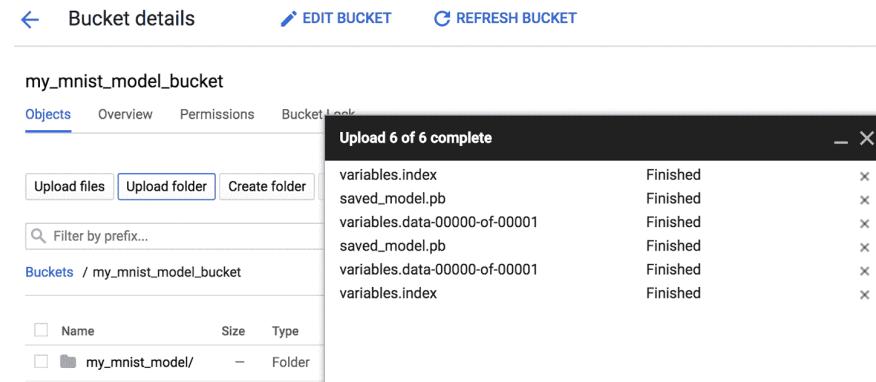


图 19-4 上传 SavedModel 到 Google Cloud Storage

- 配置 AI Platform（以前的名字是 ML Engine），让 AI Platform 知道要使用哪个模型和版本。在导航栏，下滚到 Artificial Intelligence，点击 AI Platform → Models。点击 Activate API（可能需要几分钟），然后点击 Create model。填写模型细节说明（见图 19-5），点击创建。

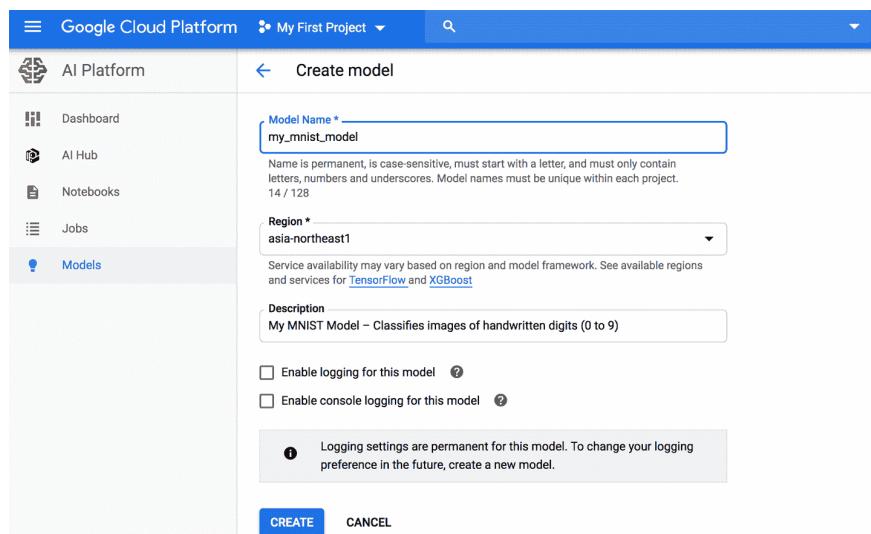


图 19-5 在 Google Cloud AI Platform 创建新模型

- AI Platform 有了模型，需要创建模型版本。在模型列表中，点击创建的模型，然后点击 Create version，填入版本细节说明（见图 19-6）：设置名字，说明，Python 版本（3.5 或以上），框架（TensorFlow），框架版本（2.0，或 1.13），ML 运行时版本（2.0，或 1.13），机器类型（选择 Single core CPU），模型的 GCS 路径（真实版本文件夹的完整路径，比如，gs://my-mnist-model-bucket/my\_mnist\_model/0002/），扩展（选择 automatic），TF Serving 容器的最小运行数（留空就成）。然后点击 Save。

[← Create version](#)

To create a new version of your model, make necessary adjustments to your saved model file before exporting and store your exported model in Cloud Storage. [Learn more](#)

Name	v0001
Name cannot be changed, is case sensitive, must start with a letter, and may only contain letters, numbers, and underscores. 5 / 128	
Description	Dense net with 2 layers (100, 10 units)
Python version	3.5
Select the Python version you used to train the model	
Framework	TensorFlow

图 19-6 在 Google Cloud AI Platform 上创建一个新模型版本

恭喜，这样就将第一个模型部署在云上了。因为选择的是自动扩展，当每秒查询数上升时，AI Platform 会启动更多 TF Serving 容器，并会对查询做负载均衡。如果 QPS 下降，就会关闭容器。所以花费直接和 QPS 关联（还和选择的机器类型和存储在 GCS 的数据量有关）。这个定价机制特别适合偶尔使用的用户，有使用波峰的服务，也适合初创企业。

笔记：如果不使用预测服务，AI Platform 会停止所有容器。这意味着，只用支付存储费用就成（每月每 GB 几美分）。当查询服务时，AI Platform 会启动 TF Serving 容器，启动需要几秒钟。如果延迟太长，可以将最小容器数设为 1。当然，这样花费会高。

现在查询预测服务。

## 使用预测服务

在底层，AI Platform 就是运行 TF Serving，所以原理上，如果知道要查询的 url，可以使用之前的代码。就是有一个问题：GCP 还负责加密和认证。加密是基于 SSL/TLS，认证是基于标记：每次请求必须向服务端发送秘密认证。所以在代码使用预测服务（或其它 GCP 服务）之前，必需要有标记。后面会讲如果获取标记，首先配置认证，使应用获得 GCP 的响应访问权限。有两种认证方法：

- 应用（即，客户端）可以用 Google 登录和密码信息做认证。使用密码，可以让应用获得 GCP 的同等权限。另外，不能将密码部署在应用中，否则会被盗。总之，不要选择这种方法，它只使用极少场合（例如，当应用需要访问用户的 GCP 账户）。
- 客户端代码可以用 service account 验证。这个账户代表一个应用，不是用户。权限十分有限。推荐这种方法。

因此，给应用创建一个服务账户：在导航栏，逐次 IAM & admin → Service accounts，点击 Create Service Account，填表（服务账户名、ID、描述），点击创建（见图 19-7）。然后，给这个账户一些访问权限。选

选择 ML Engine Developer 角色：这可以让服务账户做预测，没其它另外权限。或者，可以给服务账户添加用户访问权限（当 GCP 用户属于组织时很常用，可以让组织内的其它用户部署基于服务账户的应用，或者管理服务账户）；接着，点击 Create Key，输出私钥，选择 JSON，点击 Create。这样就能下载 JSON 格式的私钥了。

#### Create service account

- 1 Service account details — 2 Grant this service account access to project (optional) —
- 3 Grant users access to this service account (optional)

#### Service account details

Service account name  
my\_software

Display name for this service account

Service account ID  
my-software @onyx-smoke-242003.iam.gserviceaccount.com X C

Service account description  
This is my software, which relies on the predictions from my model

Describe what this service account will do

**CREATE** CANCEL

图 19-7 在 Google IAM 中创建一个新的服务账户

现在写一个小脚本来查询预测服务。Google 提供了几个库，用于简化服务访问：

#### Google API Client Library

- 基于 *OAuth 2.0* 和 REST。可以使用所有 GCP 服务，包括 AI Platform。可以用 PIP 安装：库名叫做 `google-api-python-client`。

#### Google Cloud Client Libraries

- 稍高级的库：每个负责一个特别的服务，比如 GCS、Google BigQuery、Google Cloud Natural Language、Google Cloud Vision。所有这些库都可以用 PIP 安装（比如，GCS 客户端库是 `google-cloud-storage`）。如果有可用的客户端库，最好不用 Google API 客户端，因为前者性能更好。

在写作本书的时候，AI Platform 还没有客户端库，所以我们使用 Google API 客户端库。这需要使用服务账户的私钥；设定 `GOOGLE_APPLICATION_CREDENTIALS` 环境参数就成，可以在启动脚本之前，或在如下的脚本中：

```
import os
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "my_service_account_key.json"
```

笔记：如果将应用部署到 Google Cloud Engine (GCE) 的虚拟机上，或 Google Cloud Kubernetes Engine 的容器中，或 Google Cloud App Engine 的网页应用上，或者 Google Cloud Functions 的微服务，如果没有设置 `GOOGLE_APPLICATION_CREDENTIALS` 环境参数，会使用默认的服务账户（比如，如果在 GCE 上运行应用，就用默认 GCE 服务账户）。

然后，必须创建一个包装了预测服务访问的资源对象：

```
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()
```

可以将 `/versions/0001`（或其它版本号），追加到 `model_path`，指定想要查询的版本：这么做可以用来 A/B 测试，或在推广前在小范围用户做试验。然后，写一个小函数，使用资源对象调用预测服务，获取预测结果：

```
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"]])
```

这个函数接收包含图片的 NumPy 数组，然后准备成字典，客户端库再将其转换为 JSON 格式。然后准备预测请求，并执行；如果响应有错误，就抛出异常；没有错误的话，就提取出每个实例的预测结果，绑定成 NumPy 数组。如下：

```
>>> Y_probas = predict(X_new)
>>> np.round(Y_probas, 2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. , 0. , 0.01, 0.01],
       [0. , 0.96, 0.01, 0. , 0. , 0. , 0. , 0. , 0.01, 0.01, 0. ]])
```

现在，就在云上部署好预测服务了，可以根据 QPS 自动扩展，可以从任何地方安全访问。另外，如果不使用的话，就基本不产生费用：只要每月对每个 GB 支付几美分。可以用 [Google Stackdriver](#) 获得详细日志。

如果将模型部署到移动 app，或嵌入式设备，该怎么做呢？

## 将模型嵌入到移动或嵌入式设备

如果需要将模型部署到移动或嵌入式设备上，大模型的下载时间太长，占用内存和 CPU 太多，这会是 app 响应太慢，设备发热，消耗电量。要避免这种情况，要使用对移动设备友好、轻量、高效的模型，但又不牺牲太多准确度。[TFLite](#) 库提供了一些部署到移动设备和嵌入式设备的 app 的工具，有三个主要目标：

- 减小模型大小，缩短下载时间，降低占用内存。
- 降低每次预测的计算量，减少延迟、电量消耗和发热。
- 针对设备具体限制调整模型。

要降低模型大小，[TFLite](#) 的模型转换器可以将 [SavedModel](#) 转换为基于 [FlatBuffers](#) 的轻量格式。这是一种高效的跨平台序列化库（有点类似协议缓存），最初是 Google 开发用于游戏的。FlatBuffers 可以直接加载进内存，无需预处理：这样可以减少加载时间和内存占用。一旦模型加载到了移动或嵌入设备上，[TFLite](#) 解释器会执行它并做预测。下面的代码将 [SavedModel](#) 转换成了 FlatBuffer，并存为了 `.tflite` 文件：

```
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)
```

提示：还可以使用 `from_keras_model()` 将 `tf.keras` 模型直接转变为 `FlatBuffer`。

转换器还优化了模型，做了压缩，降低了延迟。删减了所有预测用不到的运算（比如训练运算），并优化了可能的计算；例如， $3 \times a + 4 \times a + 5 \times a$  被压缩为  $(3 + 4 + 5) \times a$ 。还将可能的运算融合。例如，批归一化作为加法和乘法融合到了前一层。要想知道 TFLite 能优化到什么程度，下载一个预训练 TFLite 模型，解压缩，然后打开 [Netron 可视化工具](#)，然后上传 .pb 文件，查看原始模型。这是一个庞大复杂的图。接着，打开优化过的 .tflite 模型，并查看。

另一种减小模型的（不是使用更小的神经网络架构）方法是使用更小的位宽（bit-width）：例如，如果使用半浮点（16 位），而不是常规浮点（32 位），模型大小就能减小到一半，准确率会下降一点。另外，训练会更快，GPU 内存使用只有一半。

TFLite 的转换器可以做的更好，可以将模型的权重量化变为小数点固定的 8 位整数。相比为 32 位浮点数，可以将模型大小减为四分之一。最简单的方法是后训练量化：在训练之后做量化，使用对称量化方法。找到最大绝对权重值， $m$ ，然后将浮点范围  $-m$  到  $+m$  固定到固定浮点（整数）范围 -127 到 127。例如（见图 19-8），如果权重范围是 -1.5 到 +0.8，则字节 -127、0.0、+127 对应的是 -1.5、0、+1.5。使用对称量化时，0.0 总是映射到 0（另外，字节值 +68 到 +127 不会使用，因为超过了最大对应的浮点数 +0.8）。

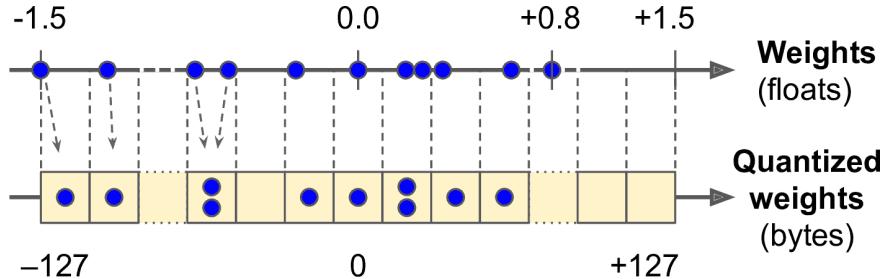


图 19-8 从 32 位浮点数到 8 位整数，使用对称量化

要使用后训练量化，只要在调用 `convert()` 前，将 `OPTIMIZE_FOR_SIZE` 添加到转换器优化的列表中：

```
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
```

这种方法可以极大地减小模型，下载和存储更快。但是，运行时量化过的权重会转换为浮点数（复原的浮点数与原始的不同，但偏差不大）。为了避免总是重新计算，缓存复原的浮点数，所以并没有减少内存使用。计算速度没有降低。

降低延迟和能量消耗的最高效的方法也是量化激活函数，让计算只用整数进行，没有浮点数运算。就算使用相同的位宽（例如，32 位整数，而不是 32 位浮点数），整数使用更少的 CPU 循环，耗能更少，热量更低。如果你还降低了位宽（例如，降到 8 位整数），速度提升会更多。另外，一些神经网络加速设备（比如边缘

TPU)，只能处理整数，因此全量化权重和激活函数是必须的。后训练处理就成；需要校准步骤找到激活的最大绝对值，所以需要给 TFLite 提供一个训练样本，模型就能处理数据，并测量量化需要的激活数据（这一步很快）。

量化最主要的问题是准确率的损失：等同于给权重和激活添加了噪音。如果准确率下降太多，则需要使用伪量化。这意味着，给模型添加假量化运算，使模型忽略训练中的量化噪音；最终的权重会对量化更鲁棒。另外，校准步骤可以在训练中自动进行，可以简化整个过程。

解释过了 TFLite 的核心概念，但要真正给移动 app 或嵌入式程序写代码需要另外一本书。幸好，可以看这本书《[TinyML: Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers](#)》，作者是 Pete Warden，他是 TFLite 团队 leader，另一位作者是 Daniel Situnayake。

浏览器中的 TensorFlow 如果想在网站中使用模型，让用户直接在浏览器中使用，该怎么做呢？使用场景很多，如下：

- 用户连接是间断或缓慢的，所以在客户端一侧直接运行模型，可以让网站更可靠。
- 如果想最快的获得响应（比如，在线游戏）。在客户端做查询肯定能降低延迟，使网站响应更快。
- 当网站服务是基于一些用户隐私数据时，在客户端做预测可以使用户数据不出用户机器，可以保护隐私。

对于所有这些情况，可以将模型输出为特殊格式，用 [TensorFlow.js js 库](#) 来加载。这个库可以用模型直接在用户的浏览器运行。TensorFlow.js 项目包括工具 `tensorflowjs_converter`，它可以将 SavedModel 或 Keras 模型文件转换为 TensorFlow.js Layers 格式：这是一个路径包含了一组二进制格式的共享权重文件，和文件 `model.json`，它描述了模型架构和权重文件的链接。这个格式经过优化，可以快速在网页上下载。用户可以用 TensorFlow.js 库下载模型并做预测。下面的代码片段是个例子：

```
import * as tf from '@tensorflow/tfjs';
const model = await tf.loadLayersModel('https://example.com/tfjs/model.j
const image = tf.fromPixels(webcamElement);
const prediction = model.predict(image);
```

TensorFlow.js 也是需要一本书来讲解。可以参考《[Practical Deep Learning for Cloud, Mobile, and Edge](#)》

接下来，来学习使用 GPU 加速计算。

## 使用 GPU 加速计算

第 11 章，我们讨论了几种可以提高训练速度的方法：更好的权重初始化、批归一化、优化器，等等。但即使用了这些方法，在单机上用单 CPU 训练庞大的神经网络，仍需要几天甚至几周。

本节，我们会使用 GPU 加速训练，还会学习如何将计算分布在多台设备上，包括 CPU 和多 GPU 设备（见图 19-9）。本章后面还会讨论在多台服务器做分布式计算。

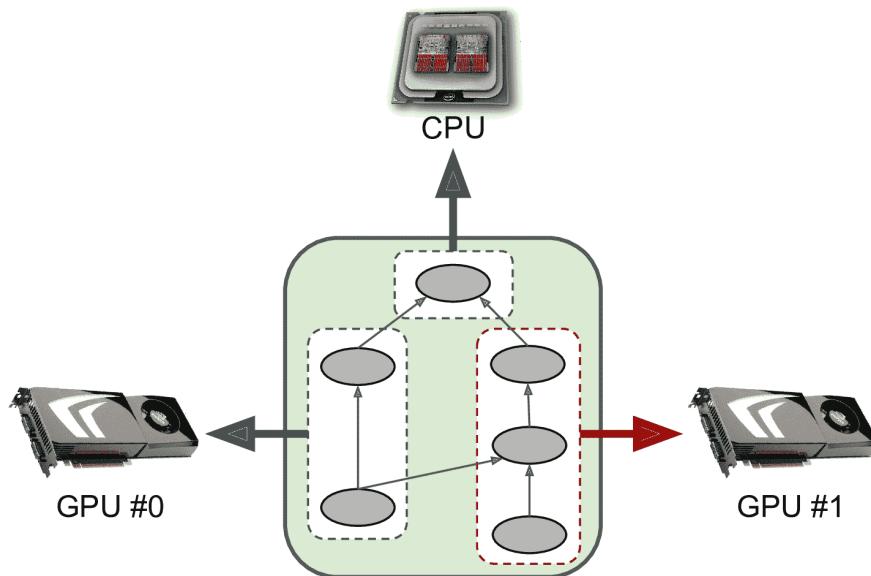


图 19-9 在多台设备上并行执行 TensorFlow 计算图

有了 GPU，可以将几天几周的训练，减少到几分钟或几小时。这样不仅能节省大量时间，还可以试验更多模型，用新数据重新训练模型。

**提示：**给电脑加上一块 GPU 显卡，通常可以提升性能。事实上，对于大多数情况，这样就足够了：根本不需要多台机器。例如，因为网络通信延迟，单台机器加 GPU 比多台机器加八块 GPU 同样快。相似的，使用一块强大的 GPU 通常比极快性能一般的 GPU 要强。

首先，就是弄一块 GPU。有两种方法：要么自己买一块 GPU，或者使用装有 GPU 的云虚拟机。我们使用第一种方法。

## 买 GPU

如果想买一块 GPU 显卡，最好花点时间研究下。Tim Dettmers 写了一篇[博客](#)帮你选择，并且他经常更新：建议仔细读读。写作本书时，TensorFlow 只支持 [Nvidia 显卡，且 CUDA 3.5+](#)（也支持 Google TPU），后面可能会支持更多厂家。另外，尽管 TCP 现在只在 GCP 上可用，以后可能会开售 TPU 卡。总之，查阅 TensorFlow 文档查看支持什么设备。

如果买了 Nvidia 显卡，需要安装驱动和库。包括 CUDA 库，可以让开发者使用支持 CUDA 的 GPU 做各种运算（不仅是图形加速），还有 CUDA 深度神经网络库（cuDNN），一个 GPU 加速库。cuDNN 提供了常见 DNN 计算的优化实现，比如激活层、归一化、前向和反向卷积、池化。它是 Nvidia 的深度学习 SDK 的一部分（要创建 Nvidia 开发者账户才能下载）。TensorFlow 使用 CUDA 和 cuDNN 控制 GPU 加速计算（见图 19-10）。

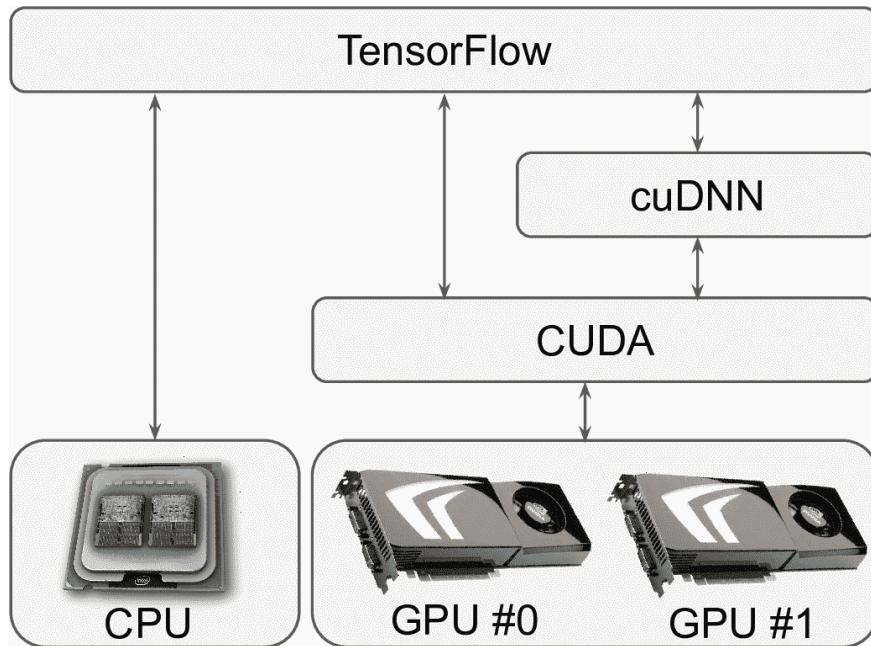


图 19-10 TensorFlow 使用 CUDA 和 cuDNN 控制 GPU，加速 DNN

安装好 GPU 和需要的库之后，可以使用 `nvidia-smi` 命令检测 CUDA 是否正确安装好，和每块卡的运行：

```
$ 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
+
```

写作本书时，你还需要安装 GPU 版本的 TensorFlow（即，`tensorflow-gpu` 库）；但是，趋势是将 CPU 版本和 GPU 版本合二为一，所以记得查看文档。因为安装每个库又长又容易出错，TensorFlow 还提供了一个 Docker 镜像，里面都装好了。但是为了让 Docker 容器能访问 GPU，还需要在主机上安装 Nvidia 驱动。

要检测 TensorFlow 是否连接 GPU，如下检测：

```
>>> 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()` 检测是否有可用的 GPU。函数 `gpu_device_name()` 给了第一个 GPU 名字：默认时，运算就运行在这块 GPU 上。函数 `list_physical_devices()` 返回了可用 GPU 设备的列表（这个例子中只有一

个)。

现在，如果你不想花费时间和钱在 GPU 上，就使用云上的 GPU VM。

## 使用带有 GPU 的虚拟机

所有主流的云平台都提供 GPU 虚拟机，一些预先配置了驱动和库（包括 TensorFlow）。Google Cloud Platform 使用了各种 GPU 额度：没有 Google 认证，不能创建 GPU 虚拟机。默认时，GPU 额度是 0，所以使用不了 GPU 虚拟机。因此，第一件事是请求更高的额度。在 GCP 控制台，在导航栏 IAM & admin → Quotas。点击 Metric。点击 None，解锁所有地点，然后搜索 GPU，选择 GPU（所有区域），查看对应的额度。如果额度是 0（或额度不足），则查看旁边的框，点击 Edit quotas。填入需求的信息，点击 Submit request。可能需要几个小时（活几天），额度请求才能被处理。默认时，每个区域每种 GPU 类型有 GPU 的额度。可以请求提高这些额度：点击 Metric，选择 None，解锁所有指标，搜索 GPU，选择想要的 GPU 类型（比如，NVIDIA P4 GPUs）。然后点击 Location，点击 None 解锁所有指标，点击想要的地点；选择相邻的框，点击 Edit quotas，发出请求。

GPU 额度请求通过后，就可以使用 Google Cloud AI Platform 的深度学习虚拟机镜像创建带有 GPU 的虚拟机了：到[这里](#)，点击 View Console，然后点击 Launch on Compute Engine，填写虚拟机配置表。注意一些地区没有全类型的 GPU，一些地区则没有 GPU（改变地区查看）。框架一定要选 TensorFlow 2.0，并要勾选 Install NVIDIA GPU driver automatically on first startup。最好勾选 Enable access to JupyterLab via URL instead of SSH：这可以在 GPU VM 上运行 Jupyter 笔记本。创建好 VM 之后，下滑导航栏到 Artificial Intelligence，点击 AI Platform → Notebooks。笔记本实例出现在列表中（可能需要几分钟，点击 Refresh 刷新），点击链接 Open JupyterLab。这样就能再 VM 上打开 JupyterLab，并连接浏览器了。你可以在 VM 上创建笔记本，运行任意代码，并享受 GPU 加速。

如果你想快速测试或与同事分享笔记本，最好使用 Colaboratory。

## Colaboratory

使用 GPU VM 最简单便宜的方法是使用 Colaboratory（或 Colab）。它是免费的，在[这个页面上](#)创建 Python 3 笔记本就成：这会在 Google Drive 上创建一个 Jupyter 笔记本（或者打开 GitHub、Google Drive 上的笔记本，或上传自己的笔记本）。Colab 的用户界面和 Jupyter 笔记本很像，除了还能像普通 Google 文档一样分享，还有一些其它细微差别（比如，通过代码加特殊注释，你可以创建的小工具）。

当你打开 Colab 笔记本，它是在一个免费的 Google VM 上运行，被称为 Colab Runtime。Runtime 默认是只有 CPU 的，但可以到 Runtime → Change runtime type，在 Hardware accelerator 下拉栏选取 GPU，然后点击保存。事实上，你还可以选取 TPU（没错，可以免费试用 TPU）。

如果用同一个 Runtime 类型运行多个 Colab 笔记本（见图 19-11），笔记本会使用相同的 Colab Runtime。如果一个笔记本写入了文件，其它笔记本就能读取这个文件。如果运行黑客的文件，可能读取隐私数据。密码也会泄露给黑客。另外，如果

你在 Colab Runtime 安装一个库，其它笔记本也会有这个库。缺点是库的版本必须相同。

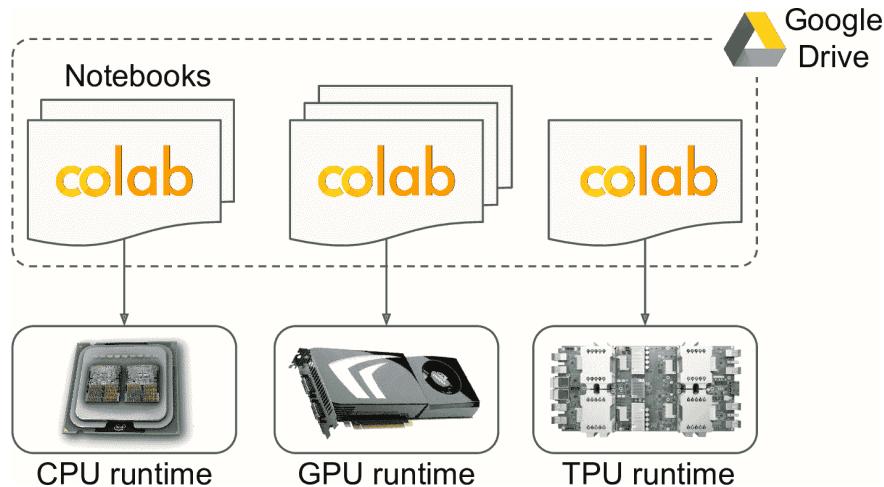


图 19-11 Colab Runtime 和笔记本

Colab 也有一些限制：就像 FAQ 写到，Colaboratory 的目的是交互使用，长时间背景的计算，尤其是在 GPU 上的，会被停掉。不要用 Colab 做加密货币挖矿。如果一定时间没有用 (~30 分钟)，网页界面就会自动断开连接。当你重新连接 Colab Runtime，可能就重置了，所以一定记着下载重要数据。即使从来没有断开连接，Colab Runtime 会自动在 12 个小时后断开连接，因为它不是用来做长时间运行的。尽管有这些限制，它仍是一个绝好的测试工具，可以快速获取结果，和同事协作。

## 管理 GPU 内存

TensorFlow 默认会在第一次计算时，使用可用 GPU 的所有内存。这么做是为了限制 GPU 内存碎片化。如果启动第二个 TensorFlow 程序（或任意需要 GPU 的程序），就会很快消耗掉所有内存。这种情况很少见，因为大部分时候是只跑一个 TensorFlow 程序：训练脚本，TF Serving 节点，或 Jupyter 笔记本。如果因为某种原因（比如，用同一台机器训练两个不同的模型）要跑多个程序，需要根据进程平分 GPU 内存。

如果机器上有多块 GPU，解决方法是分配给每个进程。要这么做，可以设定 CUDA\_VISIBLE\_DEVICES 环境变量，让每个进程只看到对应的 GPU。还要设置 CUDA\_DEVICE\_ORDER 环境变量为 PCI\_BUS\_ID，保证每个 ID 对应到相同的 GPU 卡。你可以启动两个程序，给每个程序分配一个 GPU，在两个独立的终端执行下面的命令：

```
$ 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
```

程序 1 能看到 GPU 卡 0 和 1，`/gpu:0` 和 `/gpu:1`。程序 2 只能看到 GPU 卡 2 和 3，`/gpu:1` 和 `/gpu:0`（注意顺序）。一切工作正常（见图 19-12）。当然，还可以用 Python 定义这些环境变量，`os.environ["CUDA_DEVICE_ORDER"]` 和 `os.environ["CUDA_VISIBLE_DEVICES"]`，只要使用 TensorFlow 前这么做就成。

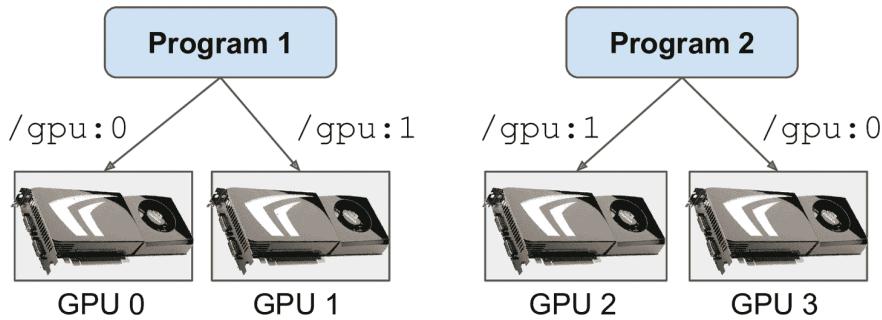


图 19-12 每个程序有两个 GPU

另一个方法是告诉 TensorFlow 使用具体量的 GPU 内存。这必须在引入 TensorFlow 之后这么做。例如，要让 TensorFlow 只使用每个 GPU 的 2G 内存，你必须创建虚拟 GPU 设备（也被称为逻辑 GPU 设备）每个物理 GPU 设备的内存限制为 2G（即，2048MB）：

```
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（假设有 4 个 GPU，每个最少 4GB）两个程序就可以并行运行了，每个都使用这四个 GPU（见图 19-13）。

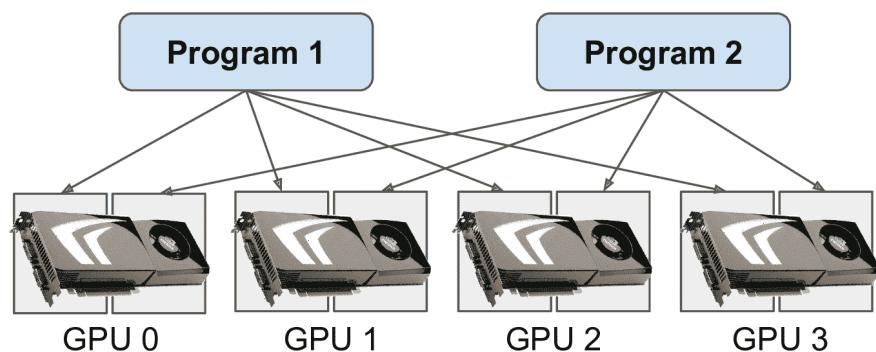


图 19-13 每个程序都可以使用 4 个 GPU，每个 GPU 使用 2GB

如果两个程序都运行时使用 nvidia-smi 命令，可以看到每个进程用了 2GB 的 GPU 内存：

```
$ 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
[...]
```

另一种方法是让 TensorFlow 只在需要内存时再使用（必须在引入 TensorFlow 后这么做）：

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_memory_growth(gpu, True)
```

另一种这么做的方法是设置环境变量 `TF_FORCE_GPU_ALLOW_GROWTH` 为 `true`。这么设置后，TensorFlow 不会释放获取的内存（避免内存碎片化），直到程序结束。这种方法无法保证确定的行为（比如，一个程序内存超标会导致另一个程序崩溃），所以在生产中，最好使用前面的方法。但是，有时这个方法是有用的：例如，当用机器运行多个 Jupyter 笔记本，其中一些使用 TensorFlow。这就是为什么在 Colab Runtime 中将环境变量 `TF_FORCE_GPU_ALLOW_GROWTH` 设为 `true`。

最后，在某些情况下，你可能想将 GPU 分为两个或多个虚拟 GPU —— 例如，如果你想测试一个分发算法。下面的代码将第一个 GPU 分成了两个虚拟 GPU，每个有 2GB（必须引入 TensorFlow 之后就这么做）：

```
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 GPU 被称为 `/gpu:0` 和 `/gpu:1`，可以像真正独立的 GPU 一样做运算和变量。下面来看 TensorFlow 如何确定安置变量和执行运算。

## 在设备上安置运算和变量

TensorFlow 白皮书介绍了一种友好的动态安置器算法，可以自动在多个可用设备上部署运算，可以测量计算时间，输入输出张量的大小，每个设备的可用内存，传入传出设备的通信延迟，用户提示。但在实际中，这个算法不怎么高效，所以 TensorFlow 团队放弃了动态安置器。

但是，`tf.keras` 和 `tf.data` 通常可以很好地安置运算和变量（例如，在 GPU 上做计算，CPU 上做预处理）。如果想要更多的控制，还可以手动在每个设备上安置运算和变量：

- 将预处理运算放到 CPU 上，将神经网络运算放到 GPU 上。
- GPU 的通信带宽通常不高，所以要避免 GPU 的不必要的数据传输。
- 给机器添加更多 CPU 内存通常简单又便宜，但 GPU 内存通常是焊接上去的：是昂贵且有限的，所以如果变量在训练中用不到，一定要放到 CPU 上（例如，数据集通常属于 CPU）。

默认下，所有变量和运算会安置在第一块 GPU 上（`/gpu:0`），除了没有 GPU 核的变量和运算：这些要放到 CPU 上（`/cpu:0`）。张量或变量的属性 `device` 告诉了它所在的设备：

```
>>> 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'
```

现在，可以放心地忽略前缀 `/job:localhost/replica:0/task:0`（它可以在使用 TensorFlow 集群时，在其它机器上安置运算；本章后面会讨论工作、复制和任务）。可以看到，第一个变量放到 GPU 0 上，这是默认设备。但是，第二个变量放到 CPU 上：这是因为整数变量（或整数张量运算）没有 GPU 核。

如果想把运算放到另一台非默认设备上，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'
```

笔记：CPU 总是被当做单独的设备（/cpu:0），即使你的电脑有多个 CPU 核。如果有多线程核，任意安置在 CPU 上的运算都可以并行运行。

如果在不存在设备或没有核的设备安置运算和变量，就会抛出异常。但是，在某些情况下，你可能只想用 CPU；例如，如果程序可以在 CPU 和 GPU 上运行，可以让 TensorFlow 在只有 CPU 的机器上忽略 `tf.device("/gpu:0")`。要这么做，在引入 TensorFlow 后，可以调用 `tf.config.set_soft_device_placement(True)`：安置请求失败时，TensorFlow 会返回默认的安置规则（即，如果有 GPU 和，默认就是 GPU 0，否则就是 CPU 0）。

TensorFlow 是如何在多台设备上执行这些运算的呢？

## 在多台设备上并行执行

第 12 章介绍过，使用 TF Functions 的好处之一是并行运算。当 TensorFlow 运行 TF 函数时，它先分析计算图，找到需要计算的运算，统计需要的依赖。

TensorFlow 接着将每个零依赖的运算（即，每个源运算）添加到运行设备的计算队列（见图 19-14）。计算好一个运算后，每个运算的依赖计数器就被删掉。当运算的依赖计数器为零时，就被推进设备的计算队列。TensorFlow 评估完所有需要的节点后，就返回输出。

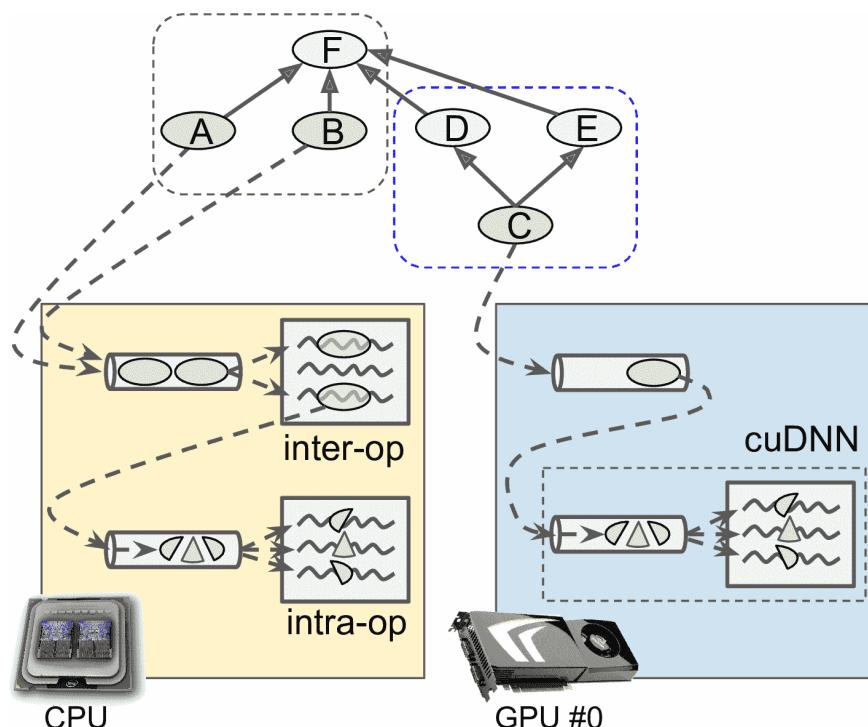


图 19-14 TensorFlow 计算图的并行执行

CPU 评估队列的运算被发送给称为 `inter-op` 的线程池。如果 CPU 有多个核，这些运算能高效并行计算。一些运算有多线程 CPU 核：这些核被分成多个子运算，放到另一个计算队列中，发到第二个被称为 `intra-op` 的线程池（多核 CPU 核共享）。总之，多个运算和自运算可以用不同的 CPU 核并行计算。

对于 GPU，事情简单一些。GPU 计算队列中的运算是顺序计算的。但是，大多数运算有多线程 GPU 核，使用 TensorFlow 依赖的库实现，比如 CUDA 和 cuDNN。这些实现有其自己的线程池，通常会用尽可能多的 GPU 线程（这就是为什么不需要 `inter-op` 线程池：每个运算已经使用 GPU 线程了）。

例如，见图 19-14，运算 A、B、C 是源运算，所以可以立即执行。运算 A 和 B 在 CPU 上，所以发到 CPU 计算队列，然后发到 `inter-op` 线程池，然后立即并行执行。运算 A 有多线程核：计算分成三个部分，在 `intra-op` 线程池内并行执行。运算 C 进入 GPU 0 的计算队列，在这个例子中，它的 GPU 核使用 cuDNN，它管理自己的 `intra-op` 线程池，在多个 GPU 线程计算。假设 C 最先完成。D 和 E 的依赖计数器下降为 0，两个运算都推到 GPU 0 的计算队列，顺序执行。C 只计算一次，即使 D 和 E 依赖它。假设 B 第二个结束。F 的依赖计数器从 4 降到 3，因为不是 0，所以霉运运行。当 A、D、E 都完成，F 的依赖计数器降到 0，被推到 CPU 的计算队列并计算。最后，TensorFlow 返回输出。

TensorFlow 的另一个奇妙的地方是当 TF 函数修改静态资源时，比如变量：它能确保执行顺序匹配代码顺序，即使不存在明确的依赖。例如，如果 TF 函数包含 `v.assign_add(1)`，后面是 `v.assign(v * 2)`，TensorFlow 会保证是按照这个顺序执行。

**提示：**通过调用 `tf.config.threading.set_inter_op_parallelism_threads()`，可以控制 `inter-op` 线程池的线程数。要设置 `intra-op` 的线程数，使用 `tf.config.threading.set_intra_op_parallelism_threads()`。如果不只想让 TensorFlow 占用所有的 CPU 核，或是只想单线程，就可以这么设置。

有了上面这些知识，就可以利用 GPU 在任何设备上做任何运算了。下面是可以做的事：

- 在独自的 GPU 上，并行训练几个模型：给每个模型写一个训练脚本，并行训练，设置 `CUDA_DEVICE_ORDER` 和 `CUDA_VISIBLE_DEVICES`，让每个脚本只看到一个 GPU。这么做很适合超参数调节，因为可以用不同的超参数并行训练。如果一台电脑有两个 GPU，单 GPU 可以一小时训练一个模型，两个 GPU 就可以训练两个模型。
- 在单 GPU 上训练模型，在 CPU 上并行做预处理，用数据集的 `prefetch()` 方法，给 GPU 提前准备批次数据。
- 如果模型接收两张图片作为输入，用两个 CNN 做处理，将不同的 CNN 放到不同的 GPU 上会更快。
- 创建高效的集成学习：将不同训练好的模型放到不同的 GPU 上，使预测更快，得到最后的预测结果。

如果想用多个 GPU 训练一个模型该怎么做呢？

## 在多台设备上训练模型

有两种方法可以利用多台设备训练单一模型：模型并行，将模型分成多台设备上的子部分；和数据并行，模型复制在多台设备上，每个模型用数据的一部分训练。下面来看这两种方法。

### 模型并行

前面我们都是在单一设备上训练单一神经网络。如果想在多台设备上训练一个神经网络，该怎么做呢？这需要将模型分成独立的部分，在不同的设备上运行。但是，模型并行有点麻烦，且取决于神经网络的架构。对于全连接网络，这种方法就没有什么提升（见图 19-15）。直观上，一种容易的分割的方法是将模型的每一层放到不同的设备上，但是这样行不通，因为每层都要等待前一层的输出，才能计算。所以或许可以垂直分割——例如，每层的左边放在一台设备上，右边放到另一台设备上。这样好了一点，两个部分能并行工作了，但是每层还需要另一半的输出，所以设备间的交叉通信量很大（见虚线）。这就抵消了并行计算的好处，因为通信太慢（尤其是 GPU 在不同机器上）。

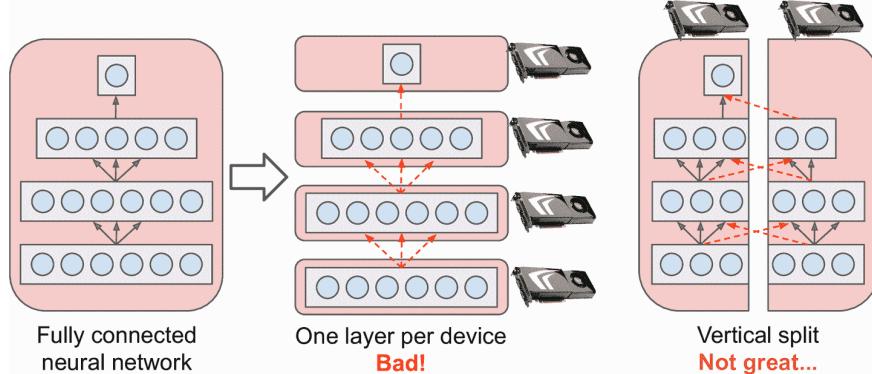


图 19-15 分割全连接神经网络

一些神经网络架构，比如卷积神经网络，包括浅层的部分连接层，更容易分割在不同设备上（见图 19-16）。

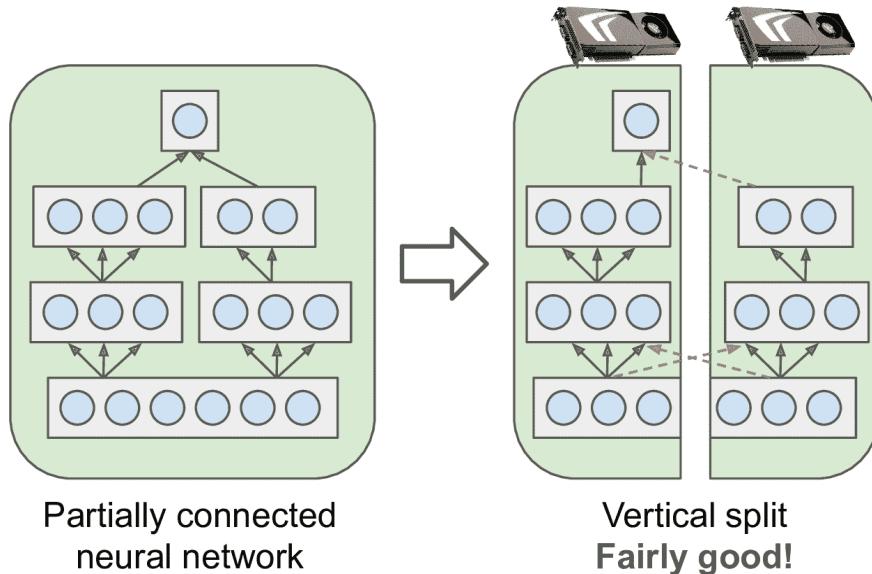


图 19-16 分割部分连接神经网络

深度循环神经网络更容易分割在多个 GPU 上。如果水平分割，将每层放到不同设备上，输入要处理的序列，在第一个时间步，只有一台设备是激活的（计算序列的第一个值），在第二步，两个设备激活（第二层处理第一层的输出，同时，第一层处理第二个值），随着信号传播到输出层，所有设备就同时激活了（图 19-17）。这么做，仍然有设备间通信，但因为每个神经元相对复杂，并行运行多个神经元的好处（原理上）超过了通信损失。但是，在实际中，将一摞 LSTM 运行在一个 GPU 上会更快。

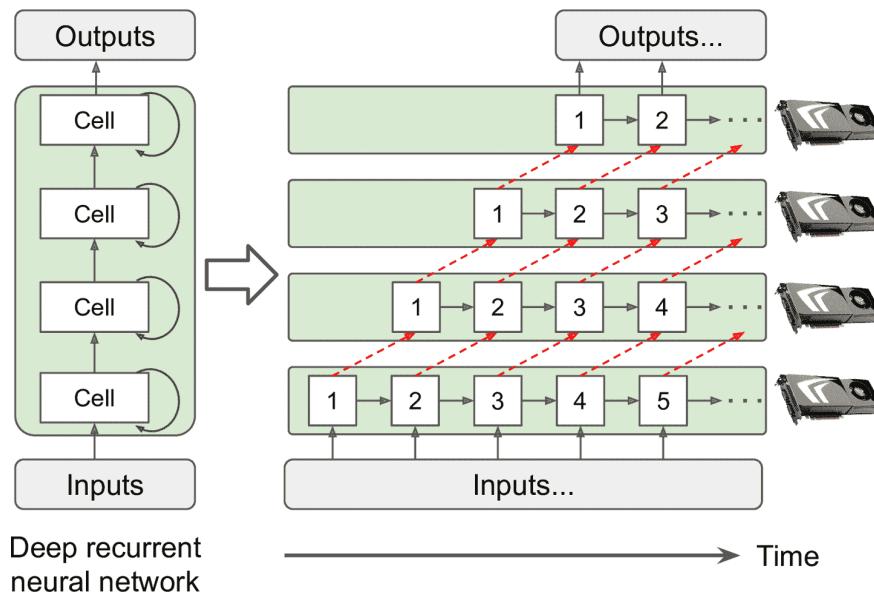


图 19-17 分割深度循环网络

总之，模型并行可以提高计算，训练一些类型的神经网络，但不是所有的，还需要特殊处理和调节，比如保证通信尽量在计算量大的机器内。下面来看更为简单高效的数据并行。

## 数据并行

另一种并行训练神经网络的方法，是将神经网络复制到每个设备上，同时训练每个复制，使用不同的训练批次。每个模型复制的计算的梯度被平均，结果用来更新模型参数。这种方法叫做数据并行。这种方法有许多变种，我们看看其中一些重要的。

### 使用镜像策略做数据并行

可能最简单的方法是所有 GPU 上的模型参数完全镜像，参数更新也一样。这么做，所有模型复制是完全一样的。这被称为镜像策略，很高效，尤其是使用一台机器时（见图 19-18）。

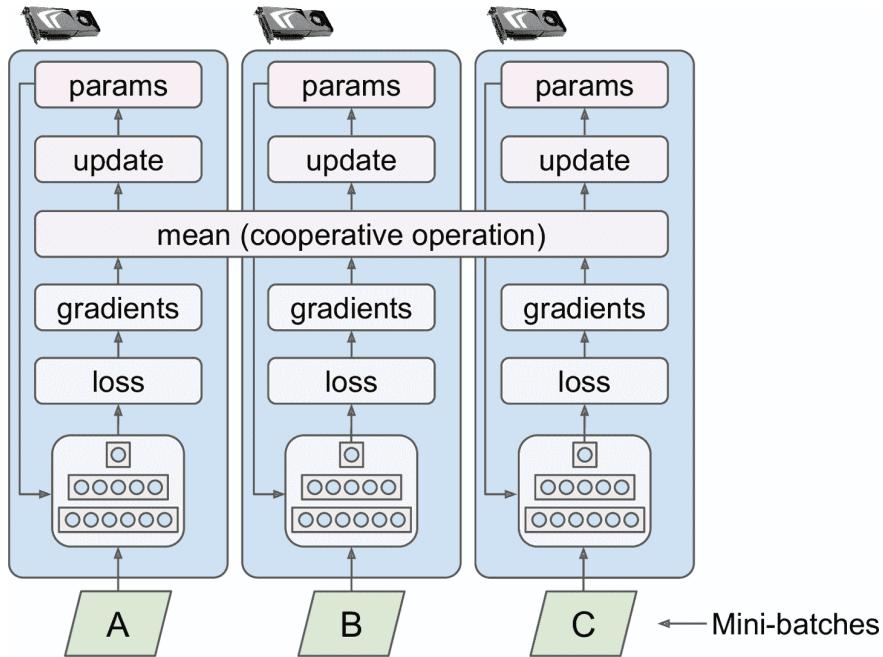


图 19-18 用镜像策略做数据并行

这种方法的麻烦之处是如何高效计算所有 GPU 的平均梯度，并将梯度分不到所有 GPU 上。这可以使用 AllReduce 算法，这是一种用多个节点齐心协力做 reduce 运算（比如，计算平均值，总和，最大值）的算法，还能让所有节点获得相同的最终结果。幸好，这个算法是现成的。

### 集中参数数据并行

另一种方法是将模型参数存储在做计算的 GPU（称为工作器）的外部，例如放在 CPU 上（见图 19-19）。在分布式环境中，可以将所有参数放到一个或多个只有 CPU 的服务器上（称为参数服务器），它的唯一作用是存储和更新参数。

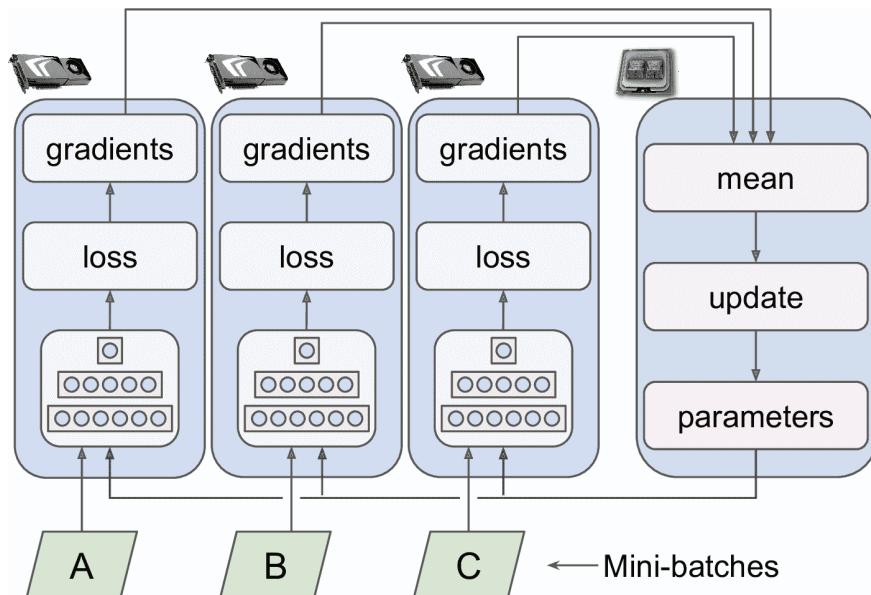


图 19-19 集中参数数据并行

镜像策略数据并行只能使用同步参数更新，而集中数据并行可以使用同步和异步更新两种方法。看看这两种方法的优点和缺点。

### 同步更新

同步更新中，累加器必须等待所有梯度都可用后，才计算平均梯度，再将其传给优化器，更新模型参数。当模型复制计算完梯度后，它必须等待参数更新，才能处理下一个批次。缺点是一些设备可能比一些设备慢，所以其它设备必须等待。另外，参数要同时复制到每台设备上（应用梯度之后），可能会饱和参数服务器的带宽。

**提示：**要降低每步的等待时间，可以忽略速度慢的模型复制的梯度（大概~10%）。例如，可以运行 20 个模型复制，只累加最快的 18 个，最慢的 2 个忽略。参数更新好后，前 18 个复制就能立即工作，不用等待 2 个最慢的。这样的设置被描述为 18 个复制加 2 个闲置复制。

### 异步更新

异步更新中，每当复制计算完了梯度，它就立即用其更新模型参数。没有累加过程（去掉了图 19-19 中的平均步骤），没有同步。模型复制彼此独立工作。因为无需等待，这种方法每分钟可以运行更多训练步。另外，尽管参数仍然需要复制到每台设备上，都是每台设备在不同时间进行的，带宽饱和风险降低了。

异步更新的数据并行是不错的方法，因为简单易行，没有同步延迟，对带宽的更佳利用。当模型复制根据一些参数值完成了梯度计算，这些参数会被其它复制更新几次（如果有  $N$  个复制，平均时  $N-1$  次），且不能保证计算好的梯度指向正确的方向（见图 19-20）。如果梯度过期，被称为陈旧梯度：它们会减慢收敛，引入噪音和抖动（学习曲线可能包含暂时的震动），或者会使训练算法发散。

### Cost

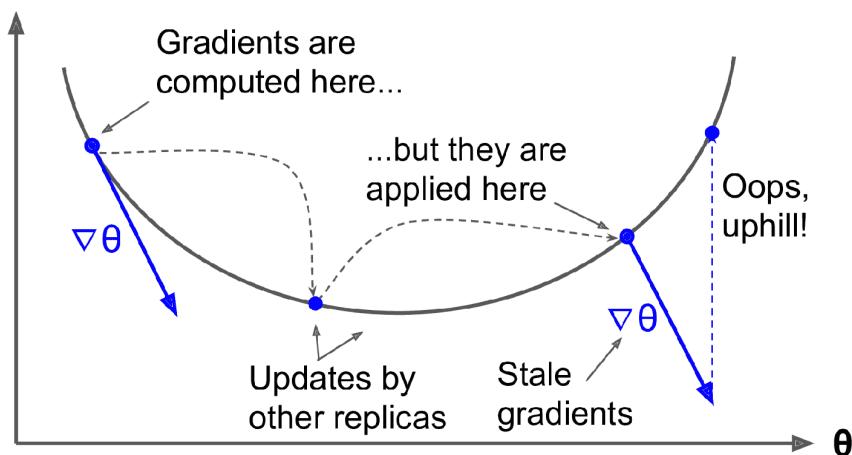


图 19-20 使用异步更新时会导致陈旧梯度

有几种方法可以减少陈旧梯度的坏处：

- 降低学习率。
- 丢弃陈旧梯度或使其变小。
- 调整批次大小。
- 只用一个复制进行前几个周期（被称为热身阶段）。陈旧梯度在训练初始阶段的破坏最大，当梯度很大且没有落入损失函数的山谷时，不同的复制会将参数推向不同方向。

Google Brain 团队在 2016 年发表了一篇论文，测量了几种方法，发现用闲置复制的同步更新比异步更新更加高效，收敛更快，模型效果更好。但是，这仍是一个活跃的研究领域，所以不要排除异步更新。

### 带宽饱和

无论使用同步还是异步更新，集中式参数都需要模型复制和参数模型在每个训练步开始阶段的通信，以及在训练步的后期和梯度在其它方向的通信。相似的，在使用镜像策略时，每个 GPU 生成的梯度需要和其它 GPU 分享。想好，总是存在临界点，添加额外的 GPU 不能提高性能，因为 GPU 内存数据通信的坏处抵消了计算负载的降低。超过这点，添加更多 GPU 反而使带宽更糟，会减慢训练。

**提示：**对于一些相对小、用大训练数据训练得到的模型，最好用单机大内存带宽单 GPU 训练。

带宽饱和对于大紧密模型更加严重，因为有许多参数和梯度要传输。对于小模型和大的系数模型，不那么严重（但没怎么利用并行计算），大多数参数是 0，可以高效计算。Jeff Dean, Google Brain 的发起者和领导，指明用 50 个 GPU 分布计算紧密模型，可以加速 25-40 倍；用 500 个 GPU 训练系数模型，可以加速 300 倍。可以看到，稀疏模型扩展更好。下面是一些具体例子：

- 神经机器翻译：8 个 GPU，加速 6 倍
- Inception/ImageNet：50 个 GPU，加速 32 倍
- RankBrain：500 个 GPU，加速 300 倍

紧密模型使用几十块 GPU，稀疏模型使用几百块 GPU，就达到了带宽瓶颈。许多研究都在研究这个问题（使用对等架构，而不是集中式架构，做模型压缩，优化通信时间和内容，等等），接下来几年，神经网络并行计算会取得很多成果。

同时，为了解决饱和问题，最好使用一些强大的 GPU，而不是大量一般的 GPU，最好将 GPU 集中在有内网的服务器中。还可以将浮点数精度从 32 位（`tf.float32`）降到 16 位（`tf.bfloat16`）。这可以减少一般的数据传输量，通常不会影响收敛和性能。最后，如果使用集中参数，可以将参数切片到多台参数服务器上：增加参数服务器可以降低网络负载，降低贷款饱和的风险。

下面就用多个 GPU 训练模型。

## 使用 Distribution Strategies API 做规模训练

许多模型都可以用单一 GPU 或 CPU 来训练。但如果训练太慢，可以将其分布到同一台机器上的多个 GPU 上。如果还是太慢，可以换成更强大的 GPU，或添加更多的 GPU。如果模型要做重计算（比如大矩阵乘法），强大的 GPU 算的更快，你还可以尝试 Google Cloud AI Platform 的 TPU，它运行这种模型通常更快。如果加不了 GPU，也使不了 TPU（例如，TPU 没有提升，或你想使用自己的硬件架构），则你可以尝试在多台服务器上训练，每台都有多个 GPU（如果这还不成，最后一种方法是添加并行模型，但需要更多尝试）。本节，我们会学习如何规模化训练模型，从单机多 GPU 开始（或 TPU），然后是多机多 GPU。

幸好，TensorFlow 有一个非常简单的 API 做这项工作：Distribution Strategies API。要用多个 GPU 训练 Keras 模型（先用单机），用镜像策略的数据并行，创建一个对象 `MirroredStrategy`，调用它的 `scope()` 方法，获取分布上下文，在上下文中包装模型的创建和编译。然后正常调用模型的 `fit()` 方法：

```

distribution = tf.distribute.MirroredStrategy()

with distribution.scope():
    mirrored_model = tf.keras.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)

```

在底层，`tf.keras` 是分布式的，所以在这个 `MirroredStrategy` 上下文中，它知道要复制所有变量和运算到可用的 GPU 上。`fit()` 方法，可以自动对所有模型复制分割训练批次，所以批次大小要可以被模型复制的数量整除。就是这样。比用一个 GPU，这么训练会快很多，而且代码变动很少。

训练好模型后，就可以做预测了：调用 `predict()` 方法，就能自动在模型复制上分割批次，并行做预测（批次大小要能被模型复制的数量整除）。如果调用模型的 `save()` 方法，会像常规模型那样保存。所以加载时，在单设备上（默认是 GPU 0，如果没有 GPU，就是 CPU），就和常规模型一样。如果想加载模型，并在可用设备上运行，必须在分布上下文中调用 `keras.models.load_model()`：

```

with distribution.scope():
    mirrored_model = keras.models.load_model("my_mnist_model.h5")

```

如果只想使用 GPU 设备的一部分，可以将列表传给 `MirroredStrategy` 的构造器：

```

distribution = tf.distribute.MirroredStrategy(["/gpu:0", "/gpu:1"])

```

默认时，`MirroredStrategy` 类使用 NVIDIA Collective Communications 库 (NCCL) 做 AllReduce 平均值运算，但可以设置 `tf.distribute.HierarchicalCopyAllReduce` 类的实例，或 `tf.distribute.ReductionToOneDevice` 类的实例的 `cross_device_ops` 参数，换其它的库。默认的 NCCL 是基于类 `tf.distribute.NcclAllReduce`，它通常很快，但一来 GPU 的数量和类型，所以也可以试试其它选项。

如果想用集中参数的数据并行，将 `MirroredStrategy` 替换为 `CentralStorageStrategy`：

```

distribution = tf.distribute.experimental.CentralStorageStrategy()

```

你还可以设置 `compute_devices`，指定作为工作器的设备（默认会使用所有的 GPU），还可以通过设置 `parameter_device`，指定存储参数的设备（默认使用 CPU，或 GPU，如果只有一个 GPU 的话）。

下面看看如何用 TensorFlow 集群训练模型。

## 用 TensorFlow 集群训练模型

TensorFlow 集群是一组并行运行的 TensorFlow 进程，通常是在不同机器上，彼此通信完成工作——例如，训练或执行神经网络。集群中的每个 TF 进程被称为任务 (task)，或 TF 服务器。它有 IP 地址，端口和类型（也被称为角色 (role) 或工作 (job)）。类型可以是 "worker"、"chief"、"ps"（参数服务器 (parameter server)）、"evaluator"：

- 每个工作器执行计算，通常是在有一个或多个 GPU 的机器上。

- `chief` 也做计算，也做其它工作，比如写 TensorBoard 日志或存储检查点。集群中只有一个 `chief` 。如果没有指定 `chief` ，第一个工作器就是 `chief` 。
- 参数服务器只保留变量值的轨迹，通常是在只有 CPU 的机器上。这个类型的任务只使用 `ParameterServerStrategy` 。
- 评估器只做评估。

要启动 TensorFlow 集群，必须先指定。要定义每个任务的 IP 地址，TCP 端口，类型。例如，下面的集群配置定义了集群有三种任务（两个工作器一个参数服务器，见图 19-21） 。集群配置是一个字典，每个工作一个键，值是任务地址（`IP:port` ）列表：

```
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
}
```

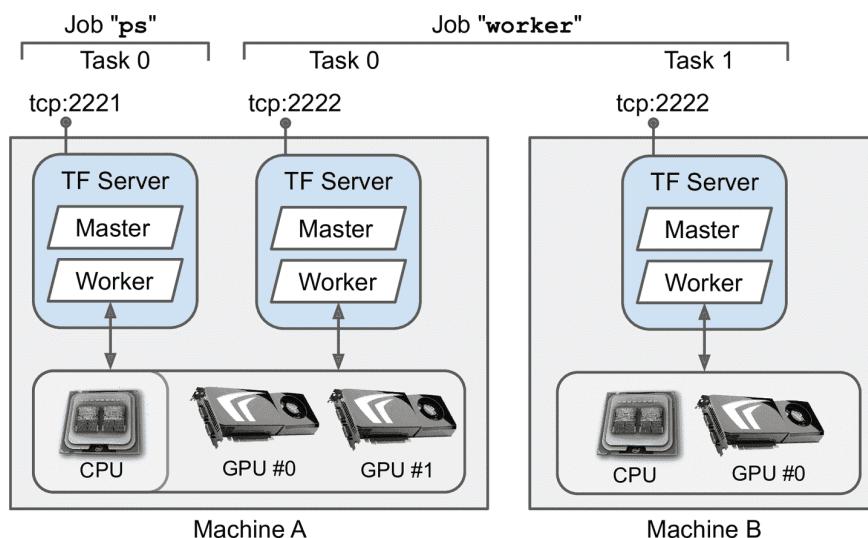


图 19-21 TensorFlow 集群

通常，每台机器只有一个任务，但这个例子说明，如果愿意，可以在一台机器上部署多个任务（如果有相同的 GPU，要确保 GPU 内存分配好）。

**警告：**默认，集群中的每个任务都可能与其它任务通信，所以要配置好防火墙确保这些机器端口的通信（如果每台机器用相同的端口，就简单一些）。

启动任务时，必须将集群配置给它，还要告诉它类型和索引（例如，工作器 0）。配置最简单的方法（集群配置和当前任务的类型和索引）是在启动 TensorFlow 前，设置环境变量 `TF_CONFIG` 。这是一个 JSON 编码的字典，包含集群配置（在键 `"cluster"` 下）、类型、任务索引（在键 `"task"` 下）。例如。下面的环境变量 `TF_CONFIG` 使用了刚才定义的集群，启动的任务是第一个工作器：

```
import os
import json

os.environ["TF_CONFIG"] = json.dumps({
    "cluster": cluster_spec,
    "task": {"type": "worker", "index": 0}
})
```

提示：通常要在 Python 外面定义环境变量 `TF_CONFIG`，代码不用包含当前任务的类型和索引（这样可以让所有工作器使用相同的代码）。

现在用集群训练一个模型。先用镜像策略。首先，给每个任务设定环境参数 `TF_CONFIG`。因为没有参数服务器（去除集群配置中的 `ps` 键），所以通常每台机器只有一个工作器。还要保证每个任务的索引不同。最后，在每个工作器上运行下面的训练代码：

```
distribution = tf.distribute.experimental.MultiWorkerMirroredStrategy()

with distribution.scope():
    mirrored_model = tf.keras.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)
```

这就是前面用的代码，只是这次我们使用的是 `MultiWorkerMirroredStrategy`（未来版本中，`MirroredStrategy` 可能既处理单机又处理多机）。当在第一个工作器上运行脚本时，它会阻塞所有 `AllReduce` 步骤，最后一个工作器启动后，训练就开始了。可以看到工作器以相同的速度前进（因为每步使用的同步）。

你可以从两个 `AllReduce` 实现选择做分布策略：基于 gRPC 的 `AllReduce` 算法用于网络通信，和 NCCL 实现。最佳算法取决于工作器的数量、GPU 的数量和类型和网络。默认，TensorFlow 会选择最佳算法，但是如果想强制使用某种算法，将 `CollectiveCommunication.RING` 或 `CollectiveCommunication.NCCL`（出自 `tf.distribute.experimental`）传给策略构造器。

如果想用带有参数服务器的异步数据并行，可以将策略变为 `ParameterServerStrategy`，添加一个或多个参数服务器，给每个任务配置 `TF_CONFIG`。尽管工作器是异步的，每个工作器的复制是同步工作的。

最后，如果你能用 Google Cloud 的 TPU，可以如下创建 `TPUStrategy`：

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()
tf.tpu.experimental.initialize_tpu_system(resolver)
tpu_strategy = tf.distribute.experimental.TPUStrategy(resolver)
```

提示：如果是研究员，可以免费试用 TPU，见[这里](#)。

现在就可以在多机多 GPU 训练模型了。如果想训练一个大模型，需要多个 GPU 多台服务器，要么买机器，要么买云虚拟机。云服务更便宜，

## 在 Google Cloud AI Platform 上训练大任务

如果你想用 Google AI Platform，可以用相同的代码部署训练任务，平台会管理 GPU VM。

要启动任务，你需要命令行工具 `gcloud`，它属于 [Google Cloud SDK](#)。可以在自己的机器上安装 SDK，或在 GCP 上使用 Google Cloud Shell。这是可以在浏览器中使用的终端；运行在免费的 Linux VM（Debian）上，SDK 已经安装配置好了。Cloud Shell 可以在 GCP 上任何地方使用：只要点击页面右上的图标 `Activate Cloud Shell`（见图 19-22）。

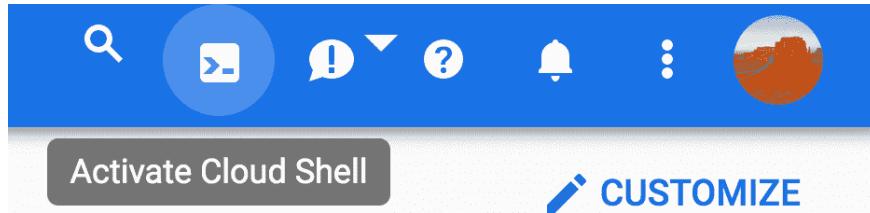


图 19-22 启动 Google Cloud Shell

如果想在自己机器上安装 SDK，需要运行 `gcloud init` 启动：需要登录 GCP 准许权限，选择想要的 GCP 项目，还有想运行的地区。`gcloud` 命令可以使用 GCP 所有功能。不用每次访问网页接口，可以写脚本开启或停止虚拟机、部署模型或做任意 GCP 动作。

运行训练任务之前，你需要写训练代码，和之前的分布设置一样（例如，使用 `ParameterServerStrategy`）。AI 平台会为每个 VM 设置 `TF_CONFIG`。做好之后，就可以在 TF 集群部署运行了，命令行如下：

```
$ 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
```

浏览这些选项。命令行启动名为 `my_job_20190531_164700` 的训练任务，地区是 `asia-southeast1`，级别是 `PREMIUM_1`：对应 20 个工作器和 11 个参数服务器（查看[其它等级](#)）。所有 VM 基于 AI Platform 的 2.0 运行时（VM 配置包括 TensorFlow 2.0 和其它包）和 Python 3.5。训练代码位于字典 `/my_project/src/trainer`，命令 `gcloud` 会自动绑定 PIP 包，并上传到 GCS 的 `gs://my-staging-bucket`。然后，AI Platform 会启动几个 VM，部署这些包，运行 `trainer.task` 模块。最后，参数 `--job-dir` 和其它参数（即，分隔符 `--` 后面的参数）会传给训练程序：主任务会使用参数 `--job-dir` 在 GCS 上保存模型，在这个例子中，是在 `gs://my-mnist-model-bucket/trained_model`。就是这样。在 GCP 控制台中，你可以打开导航栏，下滑到 Artificial Intelligence，打开 `AI Platform → Jobs`。可以看到在运行的任务，如果点击，可以看到图展示了每个任务的 CPU、GPU 和 RAM。点击 `View Logs`，可以使用 Stackdriver 查看详细日志。

**笔记：**如果将训练数据放到 GCS 上，可以创建 `tf.data.TextLineDataset` 或 `tf.data.TFRecordDataset` 来访问：用 GCS 路径作为文件名（例如，`gs://my-data-bucket/my_data_001.csv`）。这些数据集依赖包 `tf.io.gfile` 访问文件：支持本地文件和 GCS 文件（要保证服务账号可以使用 GCS）。

如果想探索几个超参数的值，可以用参数指定超参数值，执行多个任务。但是，不过想探索许多超参数，最好使用 AI Platform 的超参数调节服务。

## 在 AI Platform 上做黑盒超参数调节

AI Platform 提供了强大的贝叶斯优化超参数调节服务，称为 [Google Vizier](#)。要使用，创建任务时要传入 YAML 配置文件（`--config tuning.yaml`）。例如，可能如下：

```
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
```

它告诉 AI Platform，我们的目的是最大化指标 "accuracy"，任务会做最多 10 次试验（每次试验都从零开始训练），最多并行运行 2 个试验。我们想调节两个超参数：`n_layers`（10 到 100 间的整数），和 `momentum`（0.1 和 1.0 之间的浮点数）。参数 `scaleType` 指明了先验：`UNIT_LINEAR_SCALE` 是扁平先验（即，没有先验偏好），`UNIT_LOG_SCALE` 的先验是最优值靠近最大值（其它可能的先验是 `UNIT_REVERSE_LOG_SCALE`，最佳值靠近最小值）。

`n_layers` 和 `momentum` 参数会作为命令行参数传给训练代码。问题是训练代码如何将指标传回给 AI Platform，以便决定下一个试验使用什么超参数？AI Platform 会监督输出目录（通过 `--job-dir` 指定）的每个包含指标 "accuracy" 概括的事件文件（或是其它 `hyperparameterMetricTag` 指定的名字），读取这些值。训练代码使用 `TensorBoard()` 调回，就可以开始了。

任务完成后，每次试验中使用的超参数值和结果准确率会显示在任务的输出中（在 [AI Platform → Jobs page](#)）。

笔记：AI Platform 还可以用于在大量数据上执行模型：每个工作器从 GCS 读取部分数据，做预测，并保存在 GCS 上。

现在就可以用各种分布策略规模化创建先进的神经网络架构了，可以用自己的机器，也可以用云 —— 还可以用高效贝叶斯优化微调超参数。

## 练习

1. SavedModel 包含什么？如何检查内容？
2. 什么时候使用 TF Serving？它有什么特点？可以用什么工具部署 TF Serving？
3. 如何在多个 TF Serving 实例上部署模型？
4. 为什么使用 gRPC API 而不是 REST API，查询 TF Serving 模型？
5. 在移动和嵌入设备上运行，TFLite 减小模型的大小有什么方法？
6. 什么是伪量化训练，有什么用？
7. 什么是模型并行和数据并行？为什么推荐后者？

8. 在多台服务器上训练模型时，可以使用什么分布策略？如何进行选择？
9. 训练模型（或任意模型），部署到 TF Serving 或 Google Cloud AI Platform 上。写客户端代码，用 REST API 或 gRPC API 做查询。更新模型，部署新版本。客户端现在查询新版本。回滚到第一个版本。
10. 用一台机器多个 GPU、`MirroredStrategy` 策略，训练模型（如果没有 GPU，可以使用带有 GPU 的 Colaboratory，创建两个虚拟 GPU）。再用 `CentralStorageStrategy` 训练一次，比较训练时间。
11. 在 Google Cloud AI Platform 训练一个小模型，使用黑盒超参数调节。

参考答案见附录 A。