深度学习系列

Packt

Python Deep Learning Cookbook:
Over 75 practical recipes on neural network modeling, reinforcement learning, and transfer learning using Python

Python 深度学习实战

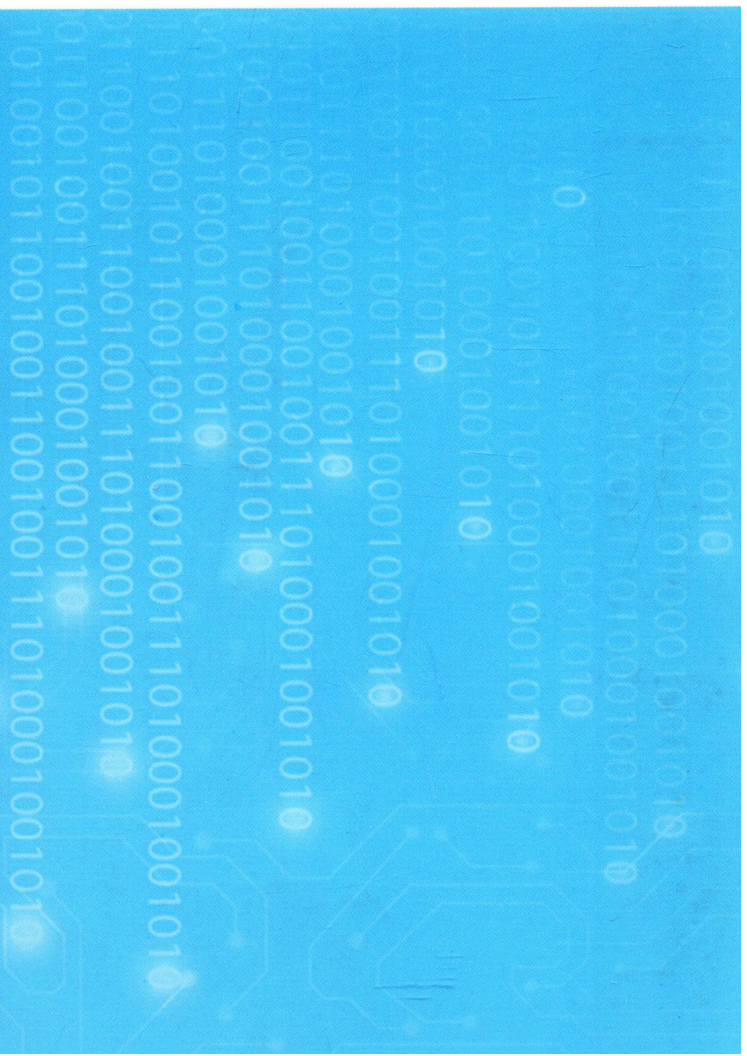
75个有关神经网络建模、强化学习与迁移学习的解决方案

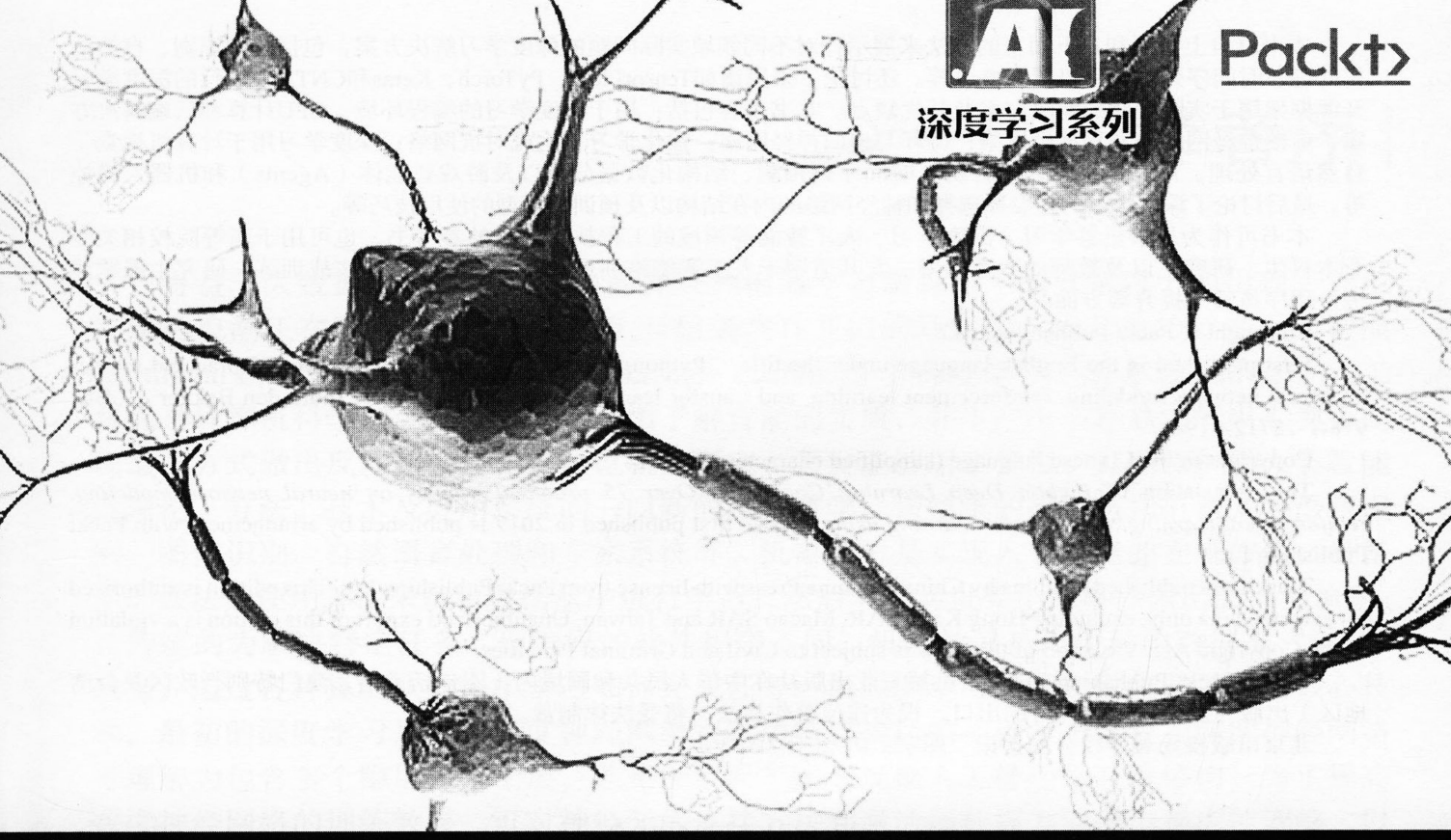
[荷]英德拉·丹·巴克（Indra den Bakker） 著
程国建 周冠武 译

机械工业出版社
CHINA MACHINE PRESS

作者简介

Indra den Bakker 是一位经验丰富的深度学习工程师和培训师。他是23insights平台的创始人，这是NVIDIA所属孵化项目计划的一部分，这是一个机器学习构建解决方案的初创型计划，可以改变世界上重要的行业。在开放课程平台Udacity，他指导了在深度学习和相关领域攻读微学位（Nanodegree）的学生，他还负责审查学生的实习项目。Indra拥有计算智能背景，并在创建23insights平台之前作为IPG Mediabrands的品牌代理以及Screen6的数据科学家若干年。





深度学习系列

Packt

Python Deep Learning Cookbook:
Over 75 practical recipes on neural network modeling, reinforcement learning, and transfer learning using Python

Python 深度学习实战

75个有关神经网络建模、强化学习与迁移学习的解决方案

[荷]英德拉·丹·巴克（Indra den Bakker） 著
程国建 周冠武 译

机械工业出版社
CHINA MACHINE PRESS

本书以自上而下和自下而上的方法来展示针对不同领域实际问题的深度学习解决方案，包括图像识别、自然语言处理、时间序列预测和机器人操纵等。还讨论了采用诸如TensorFlow、PyTorch、Keras和CNTK等流行的深度学习开源框架用于实际问题的解决方案及其优缺点。本书内容包括：用于深度学习的编程环境、GPU计算和云端解决方案；前馈神经网络与卷积神经网络；循环与递归神经网络；强化学习与生成对抗网络；深度学习用于计算机视觉、自然语言处理、语音识别、视频分析、时间序列预测、结构化数据分析以及游戏智能体（Agents）和机器人操控等。最后讨论了深度学习的超参数选择和神经网络的内在结构以及预训练模型的使用技巧等。

本书可作为从事机器学习、深度学习、人工智能等领域的工程技术人员的参考书，也可用于高等院校相关专业本科生、研究生以及教师的参考用书。尤其适用于人工智能培训班、大学生创新创业实战训练、研究生课题演练、程序员实力提升等方面。

图书在版编目（CIP）数据

Python 深度学习实战 : 75 个有关神经网络建模、强化学习与迁移学习的解决方案 / (荷) 英德拉・丹・巴克 (Indra den Bakker) 著 ; 程国建 , 周冠武译 . —北京：机械工业出版社，2018.5
（深度学习系列）

书名原文：Python Deep Learning Cookbook: Over 75 practical recipes on neural network modeling, reinforcement learning, and transfer learning using Python

ISBN 978-7-111-59872-5

Ⅰ . ① P…　Ⅱ . ①英… ②程… ③周…　Ⅲ . ①软件工具 – 程序设计　Ⅳ . ① TP311.561

中国版本图书馆 CIP 数据核字（2018）第 091240 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）
策划编辑：顾　谦　　责任编辑：顾　谦
责任校对：陈　越　　责任印制：孙　炜
北京中兴印刷有限公司印刷
2018 年 6 月第 1 版第 1 次印刷
184mm × 240mm ・ 16.75 印张 ・ 372 千字
0 001—4 000册
标准书号：ISBN 978-7-111-59872-5
定价：79.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务　　网络服务
服务咨询热线：010-88361066　机 工 官 网 ：www.cmpbook.com
读者购书热线：010-68326294　机 工 官 博 ：weibo.com/cmp1952
　　010-88379203　金　书　网 ：www.golden-book.com
封面无防伪标均为盗版　　教育服务网 ：www.cmpedu.com

译者序

得益于大数据的涌现、运算力的提升和机器学习新算法（深度学习）的出现，人工智能的浪潮正在席卷全球，诸多词汇开始萦绕在我们的耳边，首当其冲的就是人工智能（Artificial Intelligence）、机器学习（Machine Learning）和深度学习（Deep Learning）。人工智能是计算机科学的一个分支，它企图了解智能的实质，并生产出一种新的能以人类智能相似的方式做出反应的智能机器，也就是研究、开发能够用于模拟、延伸和扩展人类智能的理论、方法、技术及应用系统的一门新的技术科学。该领域的研究包括机器人、语言识别、图像识别、自然语言处理和专家系统等。机器学习是实现人工智能的主流方法，最基本的做法是使用算法来解析数据从中进行学习，然后对真实世界中的事件做出决策和预测。与传统的为解决特定任务、硬编码的软件程序不同，机器学习是用大量的数据来“训练”模型，通过各种算法从数据中学习如何完成任务。而深度学习又是实现机器学习的核心技术，最初的深度学习是利用深度神经网络来解决特征表达的一种学习过程。深度神经网络可理解为包含多个隐层（数十层，甚至上百层）的大规模人工神经元互连结构。为了提高深度神经网络的训练效果，可对神经元的连接方法和激活函数等方面做出相应的调整，以使模式识别或预测结果达到最优。深度学习摧枯拉朽般地实现了各种目标，使得似乎所有的机器辅助功能都变为可能，诸如无人驾驶汽车、个性化医疗保健、人脸识别、自然语言处理、网上购物推荐等都是深度学习的典型应用场景。

本书以自上而下和自下而上的方法来展示针对不同领域实际问题的深度学习解决方案，包括图像识别、自然语言处理、时间序列预测和机器人操纵等。还讨论了采用诸如TensorFlow、PyTorch、Keras 和 CNTK 等流行的深度学习开源框架用于实际问题的解决方案及其优缺点。本书内容包括：用于深度学习的编程环境、GPU 计算和云端解决方案；前馈神经网络与卷积神经网络；循环与递归神经网络；强化学习与生成对抗网络；深度学习用于计算机视觉、自然语言处理、语音识别、视频分析、时间序列预测、结构化数据分析以及游戏智能体（Agents）和机器人操控等。最后讨论了深度学习的超参数选择和神经网络的内在结构以及预训练模型的使用技巧等。本书将带读者进入深度学习的实战场景，通过使用诸如 TensorFlow 与 Keras 等 Python 深度学习流行框架而进行自然语言处理、图像识别、时间序列预测等实战演练。本书适用于人工智能培训班、大学生创新创业实战训练、研究生课题演练、程序员实力提升等方面。本书的出版得益于机械工业出版社的推荐以及周冠武博士和研究生们付出的辛劳，在此一并致谢。

程国建谨识

2018 年 3 月

原书前言

深度学习正在为广泛的行业带来革命性的变化。对于许多应用来说，深度学习通过做出更快和更准确的预测，证明其已经超越人类的预测。本书提供了自上而下和自下而上的方法来展示深度学习对不同领域现实问题的解决方案。这些应用程序包括计算机视觉、自然语言处理、时间序列预测和机器人。

本书主要内容

第 1 章　编程环境、GPU 计算、云解决方案和深度学习框架　主要包括与环境和 GPU 计算相关的信息和方案。对于在不同平台上设置环境有问题的读者来说，这是一个必读内容。

第 2 章　前馈神经网络　提供了与前馈神经网络有关的一系列方法，并作为其他章节的基础。本章的重点是为不同网络拓扑常见的实现问题提供解决方案。

第 3 章　卷积神经网络　着重介绍卷积神经网络及其在计算机视觉中的应用。它提供了有关 CNN 中使用的技术和优化方法。

第 4 章　递归神经网络　提供了一系列与递归神经网络相关的方法，包括 LSTM（长短期记忆）网络和 GRU（门控递归神经元）。本章的重点是为循环神经网络的常见的实现问题提供解决方案。

第 5 章　强化学习　涵盖强化学习神经网络的各种方法。本章介绍了在单智能体环境中深度强化学习的概念。

第 6 章　生成对抗网络　提供了与无监督学习问题相关的一系列方法，这包括图像生成和超分辨率的生成对抗网络。

第 7 章　计算机视觉　包含图像编码相关的数据处理方法，如视频帧。将提供使用 Python 处理图像数据的经典技术，以及用于图像检测、分类和分割的最佳解决方案。

第 8 章　自然语言处理　涵盖与文本数据处理相关的方法，包括与文本特征表示和处理相关的方法，包括文字嵌入和文本数据存储。

第 9 章　语音识别和视频分析　涵盖与流数据处理相关的方法，这包括音频、视频和帧序列。

第 10 章　时间序列和结构化数据　提供与数字运算相关的方法，这包括序列和时间序列。

第 11 章　游戏智能体和机器人　专注于最先进的深度学习研究应用，这包括与多智能体环境（模拟）和自主车辆中游戏智能体相关的方法。

第 12 章　超参数选择、调优和神经网络学习　阐述了神经网络学习过程涉及的方方面面。本章的总体目标是提供非常简洁和具体的技巧来提升网络性能。

第 13 章　网络内部构造　介绍了一个神经网络的内部构造，这包括张量分解、权重初始化、拓扑存储、瓶颈特征和相应的嵌入。

第 14 章　预训练模型　涵盖了流行的深度学习模型，如 VGG-16 和 Inception V4。

学习本书所需的准备工作

本书专注于 AI（人工智能）的 Python 实现，而不是 Python 本身。本书使用 Python 3 来构建各种应用程序，专注于如何以最好的方式利用各种 Python 库来构建真实世界的应用程序。本着这样的精神，尽力使所有的代码尽可能无误易懂，这将使读者能够轻松理解代码，并在不同的场景中使用它。

本书读者对象

本书面向那些希望使用深度学习算法来创建真实 Python 应用程序的机器学习专业人士。对机器学习概念和 Python 库（如 NumPy、SciPy 和 Scikit-learn）有深入的理解。此外，还需要具备线性代数和微积分的基本知识。

约定惯例

在本书中，读者将看到许多区分不同类型信息的文本样式。下面是这些样式的一些例子，以及对其含义的解释。

文本、数据库表名、文件夹名、文件名、文件扩展名、路径名、虚拟 URL、用户输入和 Twitter 句柄中的代码字如下所示："为提供虚拟数据集，将使用 numpy 和以下代码"。

设置如下一段代码：

```
import numpy as np
x_input = np.array([[1,2,3,4,5]])
y_input = np.array([[10]])
```

任意命令行的输入或输出如下：

```
curl -O
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/
x86_64/cuda-repo-ubuntu1604_8.0.61-1_amd64.deb
```

新术语和关键词以黑体显示。

在屏幕上看到的单词，例如菜单或对话框，就会像这样出现在文本中：

警告或重要关注。

提示和技巧图标。

读者反馈

欢迎读者反馈意见。让作者了解读者对本书的看法，喜欢什么或不喜欢什么。读者反馈对于作者开发真正让读者受益的主题非常重要。

若要给作者反馈意见，只需发送邮件到 feedback@packtpub.com，并在邮件标题中注明书名。

如果有读者擅长的主题或有兴趣参与撰写或出版的书籍，请查看 www.packtpub.com/authors 上的作者指南。

用户支持

既然读者购买了 Packt 出版社出版的书籍，那么出版社将会帮助读者获得最大收益。

示例代码下载

读者可以在 http://www.packtpub.com 上根据账户下载本书的示例代码。如果想要购买本书电子版，可以访问 http://www.packtpub.com/support 并注册，将直接通过电子邮件发送给读者。

下载代码文件步骤如下：

1）通过邮件地址和密码在网站上登录或注册。

2）鼠标指向顶部的 SUPPORT 选项。

3）单击 Code Downloads & Errata。

4）在 Search 框中输入书名。

5）选择想要下载代码文件的书籍。

6）在下拉菜单中选择购买本书的方式。

7）单击 Code Download。

下载完成后，请用以下软件最新版本来解压文件夹：

- WinRAR / 7-Zip for Windows。
- Zipeg / iZip / UnRarX for Mac。
- 7-Zip / PeaZip for Linux。

本书的代码包还托管在 GitHub 上，https://github.com/PacktPublishing/Hands-On-Deep-Learning-with-TensorFlow。另外在 https://github.com/PacktPublishing/ 上的大量图书和视频目录中还有其他代码包。请查阅！

勘误

尽管已尽力确保内容准确，但仍然难免会有错误。如果读者在书中发现了错误、文本或代码错误，如果能及时告知，将不胜感激。这样会帮助其他读者，并有助于在本书的后续版本中进行完善。如果读者发现任何错误，请访问 http://www.packtpub.com/submit-errata 告示。首先选择书名，点击勘误提交表单链接，然后输入详细的勘误内容。一旦通过验证，

将会接受读者的提交并将勘误表上传网站，或在该标题的勘误部分下添加到现有的勘误表中。

若要查看已提交的勘误表，请访问 https://www.packtpub.com/books/content/support，并在搜索栏中输入书名。相关信息将会显示在 Errata 部分中。

版权保护

在互联网上受版权保护的资料，涉及的盗版问题是一个存在于所有媒体的严重问题。Packt 出版社非常重视保护版权和许可。如果读者在网上发现任何非法复制的作品，请立即提供地址和网址，以便追踪索赔。请通过 copyright@packtpub.com 联系我们，并提供疑似盗版材料的链接。非常感谢您在保护作者和为您提供宝贵内容方面的帮助。

问题

如果读者对本书有任何问题，请通过 questions@packtpub.com 联系我们，我们将竭尽全力为读者解决。

www.PacktPub.com
电子书、折扣优惠等

下载本书相关的文件资料，请访问 www.PacktPub.com。

您是否知道 Packt 出版社为每本出版发行的书籍都提供了电子书版本，其中包括 PDF 和 ePub 文件？您可以通过 www.PacktPub.com 升级电子书版本，作为纸质版用户，还可以享受电子书的折扣。有关更多详细信息，请通过 customercare@packtpub.com 与我们联系。

在 www.PacktPub.com，读者还可以阅读免费技术文章，订阅一系列免费时事通信，并获得 Packt 出版社纸质书和电子书的独家折扣和优惠。

使用 Mapt 可获得最需要的软件技能。Mapt 可让读者充分访问所有 Packt 出版社的图书和视频课程，以及行业领先的工具，帮助读者规划个人发展并推动读者的事业发展。

为什么订阅？

- 可以在 Packt 出版社发行的每本书中全面搜索。
- 复制、粘贴、打印和标注内容。
- 可通过 web 浏览器访问。

本书审稿人

Radovan Kavicky 是 GapData 研究院的首席数据科学家和总裁（https://www.gapdata.org），该研究院总部设在斯洛伐克的布拉迪斯拉发市，致力于利用数据的力量和经济学智慧为公共利益服务。Radovan Kavicky 是一位具有教育和学术背景的宏观经济学家、专业咨询师和分析师（在公共和私营部门为客户提供咨询服务方面具有超过 8 年的经验），他具

有强大的数学和分析能力，能够提供顶级研究和分析工作。从 MATLAB、SAS 和 Stata 开始，目前转向 Python、R 和 Tableau。他是斯洛伐克经济协会的成员，也是开放数据、开放预算举措和开放政府合作伙伴的传播者。他是 PyData Bratislava、R<-Slovakia 和 SK/CZ Tableau 用户组（skczTUG）的创始人。他曾在 TechSummit（Bratislava 2017）和 PyData（Berlin 2017）峰会上发表演讲，并且是全球 Tableau #DataLeader 网络（2017）的成员。读者可以在 Twitter 上 @radovankavicky、@GapDataInst 或 @PyDataBA 关注他。其完整档案和经历可参阅网页 https://www.linkedin.com/in/radovankavicky/ 和 https://github.com/radovankavicky。

客户反馈意见

感谢读者购买这本 Packt 出版社出版的书籍。在 Packt 出版社，质量是编辑过程的核心所在。为了帮助我们改进工作，请在本书的亚马逊网页 http://www.amazon.com/dp/178712519X 上留下诚恳的评论。

如果读者想加入我们的特约审稿团队，可以发送电子邮件给我们：customerreviews@packtpub.com。 我们通过免费电子书和视频授予我们的特约审稿人以换取宝贵的反馈意见，以此帮助我们改进书籍的质量！

目　录

第 1 章
编程环境、GPU 计算、云解决方案和深度学习框架

本章重点介绍构建深度学习框架用到的一些流行技术方案。首先，提供在本地机器和云端搭建稳定而灵活的环境的解决方案，然后再详细讨论所有流行的 Python 深度学习框架：

- 搭建一个深度学习环境；
- 在亚马逊网络服务（Amazon Web Services，AWS）上启动实例；
- 在 Google 云平台（GCP）上启动实例；
- 安装 CUDA 和 cuDNN；
- 安装 Anaconda 和库文件；
- 连接服务器上的 Jupyter Notebooks；
- 用 TensorFlow 构建最先进的即用模型；
- 直观地用 Keras 建立网络；
- 使用 PyTorch 的 RNN（循环神经网络）动态计算图；
- 用 CNTK 实现高性能模型；
- 使用 MXNet 构建高效的模型；
- 使用简单、高效的 Gluon 编码定义网络。

1.1 简介

深度学习的最新进展在一定程度上可以归因于计算能力的进步。计算能力的提高，更具体地说，就是使用 GPU（图形加速器）来处理数据，它推动了神经网络从浅层到更深层次的飞跃。在本章中，将向您展示如何为本书中使用的不同深度学习框架搭建稳定的环境，为后面的内容奠定基础。本书包含许多开源的深度学习框架，供研究人员和业内人士使用。每个框架都有其自身的优点，其中大多数是由一些大型科技公司提供的支持。

仔细按照第 1 章中的步骤，读者应该能够使用本地或基于云的 CPU（中央处理器）和 GPU（图形处理器）来充分利用本书中的方法。本书中，使用了 Jupyter Notebooks 来执行所有的代码块。这些 Notebooks 以每个代码块的方式提供交互式反馈，使其非常值得品读。

本方法中的下载链接适用于支持 NVIDIA GPU 的 Ubuntu 机器或服务器。如果需要，请对应地更改链接和文件名。读者可以自由使用任何其他环境、包管理器（例如 Docker 容器）

或版本（如果需要）。但是，这些操作可能需要额外的步骤。

1.2 搭建一个深度学习环境

在开始训练深度学习模型之前，需要搭建深度学习环境。尽管可以在 CPU 上运行深度学习模型，但运行更深和更复杂的模型时，必须使用 GPU，它实现的速度可明显提高。

如何去做…

1）首先，需要检查是否有权访问本地计算机上支持 CUDA 的 NVIDIA GPU，可通过网页 https: //developer. nvidia.com/cuda-gpus 查看概述。

2）如果 GPU 在该页面上列出，则可以继续安装 CUDA 和 cuDNN（如果尚未完成）。按照“安装 CUDA 和 cuDNN”一节中的步骤进行操作。

3）如果无法在本地计算机上访问 NVIDIA GPU，可以使用云解决方案。按照“启动云解决方案”一节中的步骤进行操作。

1.3 在 AWS 上启动实例

AWS 是最流行的云解决方案。如果无法访问本地 GPU，或者更喜欢使用服务器，则可以在 AWS 上设置 EC2 实例。在这个方法中，提供了启动一个支持 GPU 服务器的步骤。

做好准备

在继续讨论这个方法之前，假设读者已经在亚马逊 AWS 上拥有一个账户，并且熟悉其平台和附加的成本。

如何去做…

1）确保想要工作的区域可以访问 P2 或 G3 实例。 这些实例分别包括 NVIDIA K80 GPU 和 NVIDIA Tesla M60 GPU。K80 GPU 比 M60 GPU 更快，并且比起 M60 GPU 的 8GB 内存，K80 GPU 的 12GB 内存更大。

虽然 NVIDIA K80 和 M60 GPU 是运行深度学习模型的强大 GPU，但这些 GPU 却不是最先进的技术。其他更快的 GPU 已经由 NVIDIA 公司推出，需要一段时间才能加入到云解决方案中。然而，这些云计算的一大优势在于，可以直接扩展机器中的 GPU 数量，例如，亚马逊公司的 p2.16xlarge 实例有 16 个 GPU。

2）启动 AWS 实例时有两个选项：选项 1，从头开始构建一切；选项 2，可以使用来自 AWS 市场的预配置的**亚马逊机器映像（AMI）**。如果选择选项 2，将不得不支付额外费用。有关示例，请参阅 https：//aws.amazon.com/marketplace/pp/B06VSPXKDX 上的 AMI。

3）亚马逊公司提供最新详细的步骤来启动深度学习 AMI：https：//aws.amazon.com/blogs/ai/get-started-with-deep- learning-using-the-aws-deep-learning-ami/。

4）如果要从头开始构建服务器，请启动 P2 或 G3 实例，然后按照安装 CUDA 和 cuDNN 以及安装 Anaconda 和 Libraries 方法下的步骤进行操作。

5）一定要确保在完成后停止正在运行的实例，以避免不必要的成本。

一个节省成本的好方法是使用 AWS Spot 实例。这使读者可以对备用亚马逊 EC2 计算容量进行标定。

1.4 在 GCP 上启动实例

另一个流行的云提供商是 Google 公司，其 GCP 越来越受欢迎，并且拥有更大的优势，其中包括更新的 GPU 类型 NVIDIA P100 以及 16 GB 的 GPU 内存。在这个方案中，提供了启动一个支持 GPU 的计算步骤。

做好准备

在开始这个方案之前，应该明了 GCP 及其租用成本。

如何去做…

1）在首次使用 GPU 启动计算实例之前，需要请求增加 GPU 配额。详情参阅 https：//console.cloud.google.com/projectselector/iam-admin/quotas。

2）首先，选择要使用的项目，并相应地应用**测度**和**区域**滤波器。GPU 实例应该显示如图 1.1 所示。

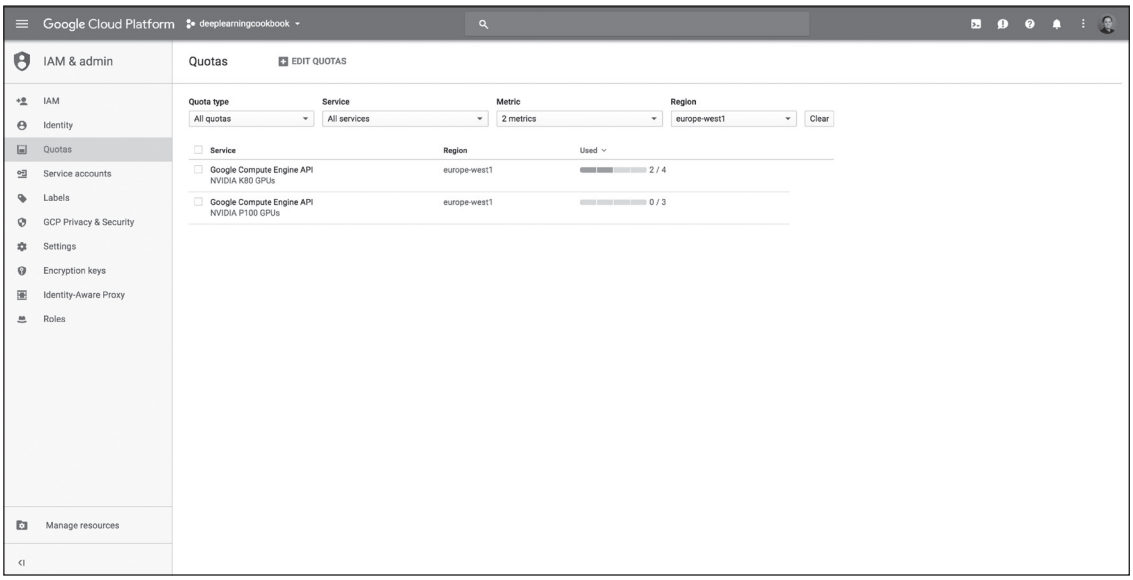


图 1.1　用于增加 GPU 配额的 GCP 仪表盘

3）选择要更改的配额，单击 EDIT QUOTAS，然后按照步骤操作。

4）当指标增加时，将收到电子邮件确认。

5）之后，可以创建一个支持 GPU 的机器。

6）启动机器时，如果想使用 Jupyter Notebook，请确保勾选允许 HTTP 通信及其 HTTP 通信框。

1.5 安装 CUDA 和 cuDNN

如果想利用 NVIDIA GPU 进行深度学习，这一部分是必不可少的。CUDA 工具包专为 GPU 加速应用程序设计，编译器针对数学运算进行了优化。此外，cuDNN 库（CUDA 深度神经网络库）是一个加速深度学习例程的库，例如 GPU 上的卷积、池化和激活。

做好准备

在开始使用此方案之前，请确保已经在 https：//developer.nvidia.com/cudnn 注册了 NVIDIA 公司的加速计算开发程序（Accelerated Computing Developer Program）。只有注册后，才能访问安装 cuDNN 库所需的文件。

如何去做…

1）首先在终端中用以下命令下载 NVIDIA（如果需要，请对应地调整下载链接。请确保现在使用的是 CUDA 8 而不是 CUDA 9）：

```
curl -O
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/
x86_64/cuda-repo-ubuntu1604_8.0.61-1_amd64.deb
```

2）接下来，解压文件并更新软件包列表中的所有软件包。之后，删除下载文件：

```
sudo dpkg -i cuda-repo-ubuntu1604_8.0.61-1_amd64.deb
sudo apt-get update
rm cuda-repo-ubuntu1604_8.0.61-1_amd64.deb
```

3）现在，准备使用以下命令安装 CUDA：

```
  sudo apt-get install cuda-8-0
```

4）接下来，需要设置环境变量并将它们添加到 shell 脚本 .bashrc 中：

```
echo 'export CUDA_HOME=/usr/local/cuda' >> ~/.bashrc
echo 'export PATH=$PATH:$CUDA_HOME/bin' >> ~/.bashrc
echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CUDA_HOME/lib64' >>
~/.bashrc
```

5）确保使用以下命令重新加载 shell 脚本：

```
source ~/.bashrc
```

6）读者可以使用终端中的以下命令来检查 CUDA 8.0 驱动程序和工具包是否已正确安装：

```
nvcc --version
nvidia-smi
```

最后一个命令的输出应该看起来如图 1.2 所示。

```
indradenbakker@instance-4:~$ nvidia-smi
Thu Oct  5 13:34:51 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 384.81                 Driver Version: 384.81                    |
|-------------------------------+----------------------+----------------------+
| 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 P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   34C    P0    27W / 250W |      0MiB / 16276MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

图 1.2　显示连接 GPU 的 nvidia-smi 输出示例

7）在这里，正确连接一个 16 GB 的 NVIDIA P100 GPU 并加以使用。

8）现在准备安装 cuDNN。确保 NVIDIA cuDNN 文件在机器上可用，例如如果需要，从本地机器复制到服务器。对于 Google 云计算引擎（确保已经设置了 gcloud 并且项目和区域设置正确），可以使用以下命令（将 local-directory 和 instance-name 替换为自己的设置）：

```
gcloud compute scp local-directory/cudnn-8.0-linux-x64-v6.0.tgz
instance-name
```

9）首先解压文件，然后以 root 身份复制到正确的目录：

```
cd
tar xzvf cudnn-8.0-linux-x64-v6.0.tgz
sudo cp cuda/lib64/* /usr/local/cuda/lib64/
sudo cp cuda/include/cudnn.h /usr/local/cuda/include/
```

10）为清理空间，可以删除用于安装的文件，如下所示：

```
rm -rf ~/cuda
rm cudnn-8.0-linux-x64-v5.1.tgz
```

1.6 安装 Anaconda 和库文件

对于 Python 用户来说，最流行的环境管理员之一是 Anaconda。 使用 Anaconda 设置、切换和删除环境非常简单。因此，可以在同一台计算机上轻松运行 Python 2 和 Python 3，并根据需要在已安装的不同版本库之间进行切换。在本书中，只关注 Python 3，并且每个方案都可以在一个环境中运行：environment-python-deep-learning-cookbook。

如何去做…

1）可以直接在计算机上下载 Anaconda 的安装文件，如下所示（对应地调整 Anaconda 文件）：

```
curl -O
https://repo.continuum.io/archive/Anaconda3-4.3.1-Linux-x86_64.sh
```

2）接下来，运行 bash 脚本（若有必要，相应地调整文件名）：

```
bash Anaconda3-4.3.1-Linux-x86_64.sh
```

按照所有提示，当要求将 .bashrc 文件添加到 PATH 时（默认值是“no”），选择“是”。

3）之后，重新加载文件：

```
source ~/.bashrc
```

4）现在，来建立一个 Anaconda 环境。开始从 GitHub 存档复制文件并打开目录：

```
git clone
https://github.com/indradenbakker/Python-Deep-Learning-Cookbook-Kit
.git
cd Python-Deep-Learning-Cookbook-Kit
```

5）使用以下命令创建环境：

```
conda env create -f environment-deep-learning-cookbook.yml
```

6）这将创建一个名为 environment-deep-learning-cookbook 的环境，并安装包含在 .yml 文件中的所有库和依赖项。本书中使用的所有库都包含在内，例如 NumPy、OpenCV、Jupyter 和 Scikit-learn。

7）激活环境：

```
source activate environment-deep-learning-cookbook
```

8）现在已经准备好运行 Python 了。按照下面的步骤安装 Jupyter 和本书中使用的深度学习框架。

1.7 连接服务器上的 Jupyter Notebooks

正如简介中所提到的，Jupyter Notebooks 在过去的几年里已经获得了很多的关注。Notebooks 是运行代码块的直观工具。在“安装 Anaconda 和 Libraries”方案中创建 Anaconda 环境时，将 Jupyter 包含在函数库列表中进行安装。

如何去做…

1）如果尚未安装 Jupyter，则可以在服务器上激活的 Anaconda 环境中使用以下命令：

```
 conda install jupyter
```

2）接下来，回到本地机器上的终端。

3）一种选择是使用 SSH-tunnelling 访问运行在服务器上的 Jupyter Notebook。例如，使用 GCP 时：

```
gcloud compute ssh --ssh-flag="-L 8888:localhost:8888"  --zone
"europe-west1-b" "instance-name"
```

读者现在已经登录到服务器，并且本地计算机上的端口 8888 将转发到端口为 8888 的服务器。

4）在继续前行之前，确保激活正确的 Anaconda 环境（相应地调整环境变量名称）：

```
source activate environment-deep-learning-cookbook
```

5）可以为 Jupyter Notebooks 创建一个专用目录：

```
mkdir notebooks
cd notebooks
```

6）现在可以按如下所示启动 Jupyter 环境：

```
jupyter notebook
```

这将在服务器上启动 Jupyter Notebook。接下来，可以访问本地浏览器，并在启动 Notebook 后使用提供的链接访问 Notebook，例如 http：//localhost：8888/?token=1fa4e9aea99cd7be2b974557eee3d344ca3c992f5861834f。

1.8 用 TensorFlow 构建最先进的即用模型

TensorFlow 是目前最流行的框架之一。TensorFlow 框架由 Google 公司在内部创建、维护和使用。这个通用的开源框架可以用于数据流图的任何数值计算。使用 TensorFlow 的最大优点之一是可以使用相同的代码并将其部署在本地 CPU、云 GPU 或 Android 设备上。TensorFlow 也可以用来在多个 GPU 和 CPU 上运行深度学习模型。

如何去做…

1）首先，将展示如何从终端安装 TensorFlow（确保调整链接到对应的平台和现有 Python 适用的 TensorFlow 版本）：

```
pip install --ignore-installed --upgrade
https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow_gpu-
1.3.0-cp35-cp35m-linux_x86_64.whl
```

这将安装支持 GPU 的 TensorFlow 版本和正确的依赖关系。

2）现在可以将 TensorFlow 库导入 Python 环境中：

```
import tensorflow as tf
```

3）为了提供一个虚拟数据集，将使用 numpy 和下面的代码：

```
import numpy as np
x_input = np.array([[1,2,3,4,5]])
y_input = np.array([[10]])
```

4）定义 TensorFlow 模型时，不能将数据直接提供给模型。应该创建一个占位符，充当数据馈送的入口点：

```
x = tf.placeholder(tf.float32, [None, 5])
y = tf.placeholder(tf.float32, [None, 1])
```

5）之后，可以使用一些变量对占位符进行操作，例如：

```
W = tf.Variable(tf.zeros([5, 1]))
b = tf.Variable(tf.zeros([1]))
y_pred = tf.matmul(x, W)+b
```

6）接下来，定义一个损失函数如下：

```
loss = tf.reduce_sum(tf.pow((y-y_pred), 2))
```

7）需要指定优化器和想要最小化的变量：

```
train = tf.train.GradientDescentOptimizer(0.0001).minimize(loss)
```

8）在 TensorFlow 中，初始化所有变量是很重要的。 因此，创建一个名为 init 的变量：

```
init = tf.global_variables_initializer()
```

应该注意到这个命令还没有初始化变量，这可在运行会话时完成。

9）接下来，创建一个会话，并运行 10 个周期的训练：

```
sess = tf.Session()
sess.run(init)

for i in range(10):
    feed_dict = {x: x_input, y: y_input}
    sess.run(train, feed_dict=feed_dict)
```

10）如果也想节约成本，可以通过添加如下来完成：

```
sess = tf.Session()
sess.run(init)

for i in range(10):
    feed_dict = {x: x_input, y: y_input}
    _, loss_value = sess.run([train, loss], feed_dict=feed_dict)
    print(loss_value)
```

11）如果想要使用多个 GPU，应该明确地加以指定。例如，从 TensorFlow 文档中取出这部分代码：

```
c = []
for d in ['/gpu:0', '/gpu:1']:
    with tf.device(d):
        a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2,
3])
        b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3,
2])
c.append(tf.matmul(a, b))
with tf.device('/cpu:0'):
    sum = tf.add_n(c)
# 创建会话，设置log_device_placement为True
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
# 操作运行
print(sess.run(sum))
```

正如您所看到的，这给在计算中如何处理和使用哪种设备提供了很大的灵活性。

这只是 TensorFlow 如何工作的简要介绍。在具体实现网络时，模型实现的粒度级别为用户增加了很大的灵活性。但是，如果读者是神经网络的新用户，那可能是压倒性的问题。这就是为什么 Keras 框架（TensorFlow 上的一个封装）对于那些希望开始构建神经网络而不需要太多细节的人来说是一个很好的选择。因此，在本书中，前几章将主要关注 Keras，而后面的章将包括更多使用其他框架的方案，如 TensorFlow。

1.9 直观地用 Keras 建立网络

Keras 是深度学习工程师熟知并使用的深度学习框架。它提供了对 TensorFlow、CNTK 和 Theano 框架的封装。这个封装器可以通过堆叠不同类型的层来轻松创建深度学习模型。Keras 的优点在于代码的简单性和可读性。如果想在训练过程中使用多个 GPU，则需要像使用 TensorFlow 一样设置设备。

如何去做…

1）首先在本地 Anaconda 环境中安装 Keras，如下所示：

```
conda install -c conda-forge keras
```

应确保深度学习环境在执行此命令之前已被激活。

2）接下来，将 Keras 库导入到 Python 环境中：

```
from keras.models import Sequential
from keras.layers import Dense
```

该命令输出 Keras 所使用的后端。默认情况下，使用的是 TensorFlow 框架，如图 1.3 所示。

```
In [1]: from keras.models import Sequential
        from keras.layers import Dense

        Using TensorFlow backend.
```

图 1.3　Keras 输出显示使用的后端

3）为了提供一个虚拟数据集，将使用 numpy 和下面的代码：

```
import numpy as np
x_input = np.array([[1,2,3,4,5]])
y_input = np.array([[10]])
```

4）使用顺序模式时，在 Keras 中堆叠多个图层是很简单的。在这个例子中，使用一个具有 32 个神经元的隐层和一个神经元的输出层：

```
model = Sequential()
model.add(Dense(units=32, input_dim=x_input.shape[1]))
model.add(Dense(units=1))
```

5）接下来，需要对模型进行编译。在编译时，可以配置不同的设置，如损失函数、优化器和测度标准：

```
model.compile(loss='mse',
              optimizer='sgd',
              metrics=['accuracy'])
```

6）在 Keras 中，可以轻松输出显示模型的摘要。它还将显示所定义模型中的参数数量：

```
model.summary()
```

在图 1.4 中，可以看到所构建模型的摘要。

```
In [5]: model.summary()

        _________________________________________________________________
        Layer (type)                 Output Shape              Param #
        =================================================================
        dense_1 (Dense)              (None, 32)                192
        _________________________________________________________________
        dense_2 (Dense)              (None, 1)                 33
        =================================================================
        Total params: 225
        Trainable params: 225
        Non-trainable params: 0
        _________________________________________________________________
```

图 1.4　Keras 模型摘要示例

7）使用一个命令直接训练模型，同时将结果保存到一个名为 history 的变量中：

```
history = model.fit(x_input, y_input, epochs=10, batch_size=32)
```

8）对于测试，预测函数可以在训练后使用：

```
pred = model.predict(x_input, batch_size=128)
```

在对 Keras 的简短介绍中，已经证明了在几行代码中实现神经网络是很容易的。但是，不要把简单与能力混为一谈。比起上面提到的内容，Keras 框架提供了更多的东西，如果需要，可以把模型调整到一个粒度级别。

1.10　使用 PyTorch 的 RNN 动态计算图

PyTorch 是 Python 深度学习框架，近来获得了很多的关注。PyTorch 是使用 Lua 的 Torch 的 Python 实现。它由 Facebook 公司支持，并且由于 GPU 加速张量计算，其速度很快。使用 PyTorch 优于其他框架的一个巨大好处是，图形是动态创建的，而不是静态创建的。这意味着网络是动态的，可以调整网络，而不必重新开始。因此，对于每个示例而言，即时创建的图形可能会有所不同。PyTorch 支持多个 GPU，可以手动设置在哪个设备（CPU 或 GPU）上执行哪些运算。

如何去做…

1）首先，在 Anaconda 环境中安装 PyTorch，如下所示：

```
conda install pytorch torchvision cuda80 -c soumith
```

如果想在另一个平台上安装 PyTorch，可以查看一下 PyTorch 的网站以获取清晰的指导：http：//pytorch.org/。

2）将 PyTorch 导入到 Python 环境中：

```
import torch
```

3）虽然 Keras 为构建神经网络提供了更高层次的抽象，但是 PyTorch 也内置了这个特性。这意味着可以使用更高级别的构建块构建，甚至可以手动构建前向和后向过程。在本例中，将使用更高层次的抽象。首先，需要设定随机训练数据的大小：

```
batch_size = 32
input_shape = 5
output_shape = 10
```

4）为了启动 GPU，将使用如下张量：

```
torch.set_default_tensor_type('torch.cuda.FloatTensor')
```

这确保所有的计算将使用附加的 GPU。

5）可以用它来生成随机训练数据：

```
from torch.autograd import Variable
X = Variable(torch.randn(batch_size, input_shape))
y = Variable(torch.randn(batch_size, output_shape),
requires_grad=False)
```

6）使用一个简单的神经网络，其中一个有 32 个神经元的隐层和一个神经元的输出层：

```
model = torch.nn.Sequential(
          torch.nn.Linear(input_shape, 32),
          torch.nn.Linear(32, output_shape),
        ).cuda()
```

使用 .cuda() 扩展来确保模型在 GPU 上运行。

7）接下来，定义 MSE 损失函数：

```
loss_function = torch.nn.MSELoss()
```

8）现在准备开始使用下面的代码来训练模型，共迭代 10 次：

```
learning_rate = 0.001
for i in range(10):
    y_pred = model(x)
    loss = loss_function(y_pred, y)
    print(loss.data[0])
    # 零梯度
    model.zero_grad()
    loss.backward()

    # 更新权值
    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

PyTorch 框架为实现简单的神经网络和更复杂的深度学习模型提供了很大的自由度。在这个介绍中没有演示的是在 PyTorch 中使用动态图。这是一个非常强大的功能，本书将在其他内容中进行展示。

1.11 用 CNTK 实现高性能模型

微软公司不久前推出了开源深度学习框架：微软认知工具包，这个框架称为 CNTK。由于性能的原因，CNTK 是用 C ++ 语言编写的，并且有一个 Python API。CNTK 支持单 GPU 和多 GPU 的使用。

如何去做…

1）首先，用 pip 安装 CNTK，如下所示：

```
pip install
https://cntk.ai/PythonWheel/GPU/cntk-2.2-cp35-cp35m-linux_x86_64.wh
l
```

若有必要，调整磁盘文件 (参阅 https：//docs.microsoft.com/en-us/ cognitive-toolkit/Setup-Linux-Python?tabs=cntkpy22)。

2）安装完 CNTK 后，可以将它导入到 Python 环境中：

```
import cntk
```

3）创建一些可以用来训练的简单虚拟数据：

```
import numpy as np
x_input = np.array([[1,2,3,4,5]], np.float32)
y_input = np.array([[10]], np.float32)
```

4）接下来，需要为输入数据定义占位符：

```
X = cntk.input_variable(5, np.float32)
y = cntk.input_variable(1, np.float32)
```

5）使用 CNTK，可以直接堆叠多个层。在一个输出层上堆叠了一个带有 32 个输入的密集层，并带有 1 个输出神经元：

```
from cntk.layers import Dense, Sequential
model = Sequential([Dense(32),
                Dense(1)])(X)
```

6）接下来，定义损失函数：

```
loss = cntk.squared_error(model, y)
```

7）现在，准备用优化器来完成模型构建：

```
learning_rate = 0.001
trainer = cntk.Trainer(model, (loss),
cntk.adagrad(model.parameters, learning_rate))
```

8）最后，训练模型，如下所示：

```
for epoch in range(10):
        trainer.train_minibatch({X: x_input, y: y_input})
```

正如在此介绍中已经展示的那样，用适当的高级封装来建立 CNTK 模型是很简单的。但是，就像 TensorFlow 和 PyTorch 一样，可以选择更细化的级别来实现模型，这给了读者很大的自由度。

1.12 使用 MXNet 构建高效的模型

MXNet 深度学习框架允许使用 Python 构建高效的深度学习模型。除了 Python，它还允许使用 R、Scala 和 Julia 等流行语言构建模型。亚马逊和百度等公司都支持 Apache MXNet。MXNet 已被证明是快速的基准框架之一，它支持对单 GPU 和多 GPU 的使用。通过使用简单评估，MXNet 能够并行自动执行操作。此外，MXNet 框架使用符号界面，称为 Symbol，这简化了构建神经网络体系结构。

如何去做…

1）在支持 GPU 的 Ubuntu 上安装 MXNet，可以在终端中使用以下命令：

```
pip install mxnet-cu80==0.11.0
```

对于其他平台和非 GPU 支持，请查看 https：//mxnet. incubator.apache.org/get_started/install.html。

2）接下来，在 Python 环境中导入 mxnet：

```
import mxnet as mx
```

3）创建一些分配给 GPU 和 CPU 的简单虚拟数据：

```
import numpy as np
x_input = mx.nd.empty((1, 5), mx.gpu())
x_input[:] = np.array([[1,2,3,4,5]], np.float32)

y_input = mx.nd.empty((1, 5), mx.cpu())
y_input[:] = np.array([[10, 15, 20, 22.5, 25]], np.float32)
```

4）可以很容易地复制和调整数据。在可能的情况下，MXNet 将自动执行并行操作：

```
x_input
w_input = x_input
z_input = x_input.copyto(mx.cpu())
x_input += 1
w_input /= 2
z_input *= 2
```

5）输出显示如下：

```
print(x_input.asnumpy())
print(w_input.asnumpy())
print(z_input.asnumpy())
```

6）如果想将数据提供给模型，应该先创建一个迭代器：

```
batch_size = 1
train_iter = mx.io.NDArrayIter(x_input, y_input, batch_size,
shuffle=True, data_name='input', label_name='target')
```

7）接下来，可以为模型创建符号：

```
X = mx.sym.Variable('input')
Y = mx.symbol.Variable('target')
fc1 = mx.sym.FullyConnected(data=X, name='fc1', num_hidden = 5)
lin_reg = mx.sym.LinearRegressionOutput(data=fc1, label=Y,
name="lin_reg")
```

8）在开始训练之前，需要定义模型：

```
model = mx.mod.Module(
    symbol = lin_reg,
    data_names=['input'],
    label_names = ['target']
)
```

9）开始训练：

```
model.fit(train_iter,
            optimizer_params={'learning_rate':0.01, 'momentum': 0.9},
            num_epoch=100,
            batch_end_callback = mx.callback.Speedometer(batch_size, 2))
```

10）使用训练好的模型进行预测：

```
model.predict(train_iter).asnumpy()
```

这里简短地介绍了 MXNet 框架。在这个介绍中，演示了如何轻松地将变量和计算分配给 CPU 或 GPU 以及如何使用 Symbol 接口。但是，还有很多需要探索的内容，MXNet 是构建灵活、高效的深度学习模型的强大框架。

1.13 使用简单、高效的 Gluon 编码定义网络

Gluon 是使用广泛的深度学习框架的最新成员。Gluon 最近由 AWS 和微软公司推出，提供了简单、易于理解的代码，而不会损失性能。Gluon 已经包含在 MXNet 的最新版本中，将在未来版本的 CNTK（和其他框架）中提供。就像 Keras 一样，Gluon 是其他许多深度学习框架的一个封装。Keras 和 Gluon 的主要区别在于，Gluon 首先将重点放在命令框架上。

如何去做…

1）Gluon 包含在最新版本的 MXNet 中（按照使用 MXNet 构建高效模型的步骤来安装 MXNet）。

2）安装完成后，可以直接输入 gluon 如下：

```
from mxnet import gluon
```

3）接下来，创建一些虚拟数据。为此，需要将数据载入 MXNet 的 NDArray 或 Symbol 中：

```
import mxnet as mx
import numpy as np
x_input = mx.nd.empty((1, 5), mx.gpu())
x_input[:] = np.array([[1,2,3,4,5]], np.float32)

y_input = mx.nd.empty((1, 5), mx.gpu())
y_input[:] = np.array([[10, 15, 20, 22.5, 25]], np.float32)
```

4）使用 Gluon，通过叠加层建立神经网络非常简单：

```
net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.nn.Dense(16, activation="relu"))
    net.add(gluon.nn.Dense(len(y_input)))
```

5）接下来，初始化参数，将参数存储在 GPU 上，如下所示：

```
net.collect_params().initialize(mx.init.Normal(), ctx=mx.gpu())
```

6）用下面的代码设置损失函数和优化器：

```
softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), 'adam',
{'learning_rate': .1})
```

7）开始训练或建模：

```
n_epochs = 10

for e in range(n_epochs):
    for i in range(len(x_input)):
        input = x_input[i]
        target = y_input[i]
        with mx.autograd.record():
            output = net(input)
            loss = softmax_cross_entropy(output, target)
            loss.backward()
        trainer.step(input.shape[0])
```

这里简短演示了如何使用 Gluon 实现神经网络架构。Gluon 是一个功能强大的扩展应用，可以用简捷的代码来实现深度学习架构。同时，使用 Gluon 几乎没有性能损失。

第 2 章
前馈神经网络

在本章中，将实现**前馈神经网络（FNN）**并讨论深度学习的基石：

- 理解感知器；
- 实现一个单层神经网络；
- 构建一个多层神经网络；
- 开始使用激活函数；
- 关于隐层和隐层神经元的实验；
- 实现一个自动编码器；
- 调整损失函数；
- 测试不同的优化器；
- 使用正则化技术提高泛化能力；
- 添加 Dropout 以防止过拟合。

2.1 简介

本章的重点是为 FNN 和其他网络拓扑的常见实现问题提供解决方案。本章讨论的技术也适用于后续内容。

FNN 是信息只向一个方向移动而不循环的网络（正如在第 4 章 递归神经网络 中看到的）。FNN 主要用于监督学习，其中数据不是顺序或时间依赖的，例如用于一般分类和回归任务。首先介绍感知器，然后介绍如何使用 NumPy 实现感知器。感知器展现了单一神经元的工作机制。接下来，通过增加神经元数量来增加复杂性，并引入单层和多层神经网络。大数量神经元和多层次网络，增加了结构的深度，也符合深度学习这个名字。

2.2 理解感知器

首先，需要了解神经网络的基础知识。神经网络由一层或多层神经元组成，以人脑中的生物神经元命名。通过实现感知器来演示单个神经元的工作机理。在感知器中，单个神经元执行所有的计算，之后扩大神经元的数量来创建深度神经网络，如图 2.1 所示。

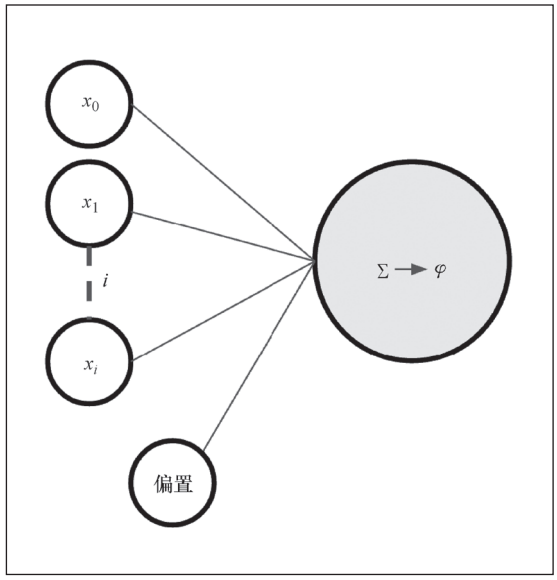


图 2.1　感知器图解

感知器可以有多个输入。在这些输入上，神经元执行计算并输出单个值，例如对两个类别进行分类的二进制值。该神经元执行的计算是输入和权重的简单矩阵乘法。将得到的值加起来并添加一个偏置，如下：

$$\sum_i w_i x_i + b$$

这些计算可以很容易地扩展到高维输入。激活函数 $\varphi(x)$ 确定正向传递中感知器的最终输出：

$$\varphi(x) = \begin{cases} 1x < 0.5 \\ 0x \geqslant 0.5 \end{cases}$$

权重和偏置随机初始化。在每个**周期**（对训练数据迭代）之后，基于网络输出与期望输出之间的**误差值**乘以**学习率**来更新权重。因此，网络训练将按新权重施加到训练数据上（**反向传播阶段**），并且输出的精度会提高。感知器是在训练数据上优化的线性组合。作为一个激活函数，将使用一个单位阶跃函数：如果输出超过一定的阈值将被激活（因此，成为一个 0 与 1 的二进制分类器）。如果这些类是线性可分，感知器就能够以 100% 的精度对模式进行分类。在下一个方案中，将向读者展示如何用 NumPy 实现感知器。

如何去做…

1）导入函数库和数据集，如下：

```
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
# 使用Iris植物数据库
from sklearn.datasets import load_iris

SEED = 2017
```

2）首先，将导入的数据进行分组，如下所示：

```
# Iris数据库中的前两类线性可分(Iris-Setosa 和 Iris-Versicolour)
iris = load_iris()
idxs = np.where(iris.target<2)
X = iris.data[idxs]
y = iris.target[idxs]
```

3）用下面的代码片断绘制四个变量中两个变量的数据展示：

```
plt.scatter(X[Y==0][:,0],X[Y==0][:,2], color='green', label='Iris-
Setosa')
plt.scatter(X[Y==1][:,0],X[Y==1][:,2], color='red', label='Iris-
Versicolour')
plt.title('Iris Plants Database')
plt.xlabel('sepal length in cm')
plt.ylabel('sepal width in cm')
plt.legend()
plt.show()
```

在图 2.2 中，绘制了两个类的分布：

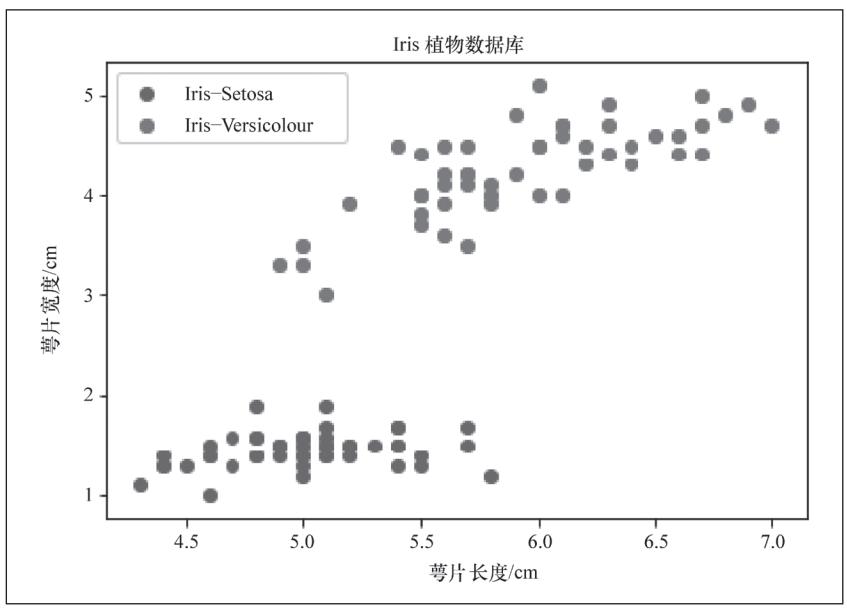


图 2.2　Iris 植物数据库（两类）

4）为了验证结果，将数据分成如下的训练集和测试集：

```
X_train, X_val, y_train, y_val = train_test_split(X, y,
test_size=0.2, random_state=SEED)
```

5）接下来，初始化感知器的权重和偏置：

```
weights = np.random.normal(size=X_train.shape[1])
bias = 1
```

6）在训练之前，需要定义超参数：

```
learning_rate = 0.1
n_epochs = 15
```

7）现在，开始用 for 循环来训练感知器：

```
del_w = np.zeros(weights.shape)
hist_loss = []
hist_accuracy = []

for i in range(n_epochs):
    # 使用阶跃函数,若输出 > 0.5，预测结果为 1，否则为 0
    output = np.where((X_train.dot(weights)+bias)>0.5, 1, 0)

    # 计算 MSE
    error = np.mean((y_train-output)**2)

    # 更新权值与偏置
    weights-= learning_rate * np.dot((output-y_train), X_train)
    bias += learning_rate * np.sum(np.dot((output-y_train),
X_train))

    # 计算 MSE
    loss = np.mean((output - y_train) ** 2)
    hist_loss.append(loss)

    # 确定验证精度
    output_val = np.where(X_val.dot(weights)>0.5, 1, 0)
    accuracy = np.mean(np.where(y_val==output_val, 1, 0))
    hist_accuracy.append(accuracy)
```

8）保存训练损失值和验证精度，以便可以绘图呈现：

```
fig = plt.figure(figsize=(8, 4))
a = fig.add_subplot(1,2,1)
imgplot = plt.plot(hist_loss)
plt.xlabel('epochs')
a.set_title('Training loss')

a=fig.add_subplot(1,2,2)
imgplot = plt.plot(hist_accuracy)
plt.xlabel('epochs')
a.set_title('Validation Accuracy')
plt.show()
```

在图 2.3 中，显示了由此产生的训练损失值和验证精度。

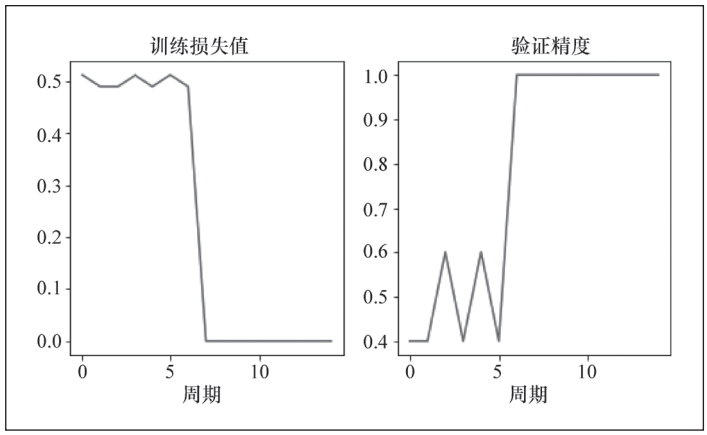


图 2.3　训练损失值和验证精度

2.3　实现一个单层神经网络

现在可以转向**神经网络**，从实现最简单的神经网络形式开始：单层神经网络。它与感知器的区别在于计算过程是由多个神经元完成的，因此构成一个网络。正如所期望的那样，增加更多的神经元会增加可解决问题的难度。这些神经元分别执行它们的计算并堆叠在一个层中，这个层称为**隐层**。所以，在该层堆叠的神经元称为**隐层神经元**。现在，只考虑单个隐层。输出层表现为感知器。这一次，作为输入，在隐层中的神经元为隐层神经元而不是输入变量，如图 2.4 所示。

在感知器的实现中，使用了单位阶跃函数来确定类别。在下一个方案中，将对隐层神经元和输出函数使用一个非线性激活函数 sigmoid。通过用非线性激活函数代替阶跃函数，网络也将能够发现非线性模式。稍后在“激活函数”一节中将进一步介绍。在后向传递中，使用 sigmoid 函数的导数来更新权重。

在下面的方案中，将用 NumPy 对两个非线性可分的类进行分类。

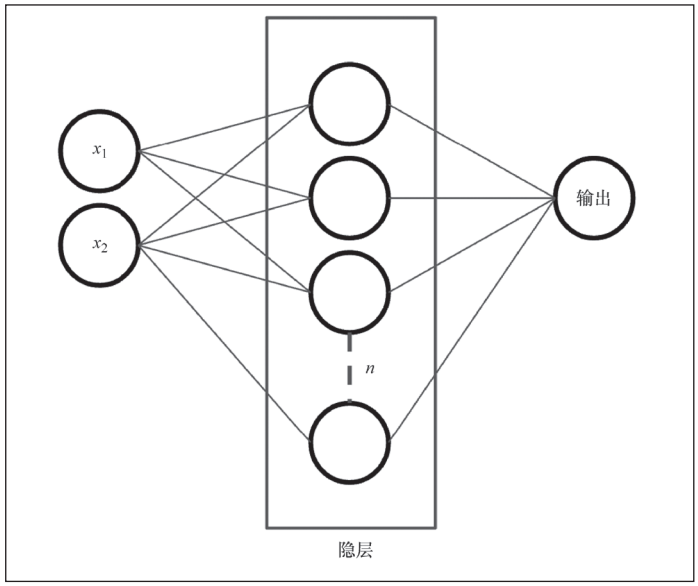


图 2.4　具有两个输入变量、n 个隐层神经元和一个输出神经元的单层神经网络

如何去做…

1）导入函数库和数据集：

```
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
# 使用 scikit-learn 中的 make_circles 函数
from sklearn.datasets import make_circles

SEED = 2017
```

2）首先，创建训练数据：

```
# 创建内圈及外圈
X, y = make_circles(n_samples=400, factor=.3, noise=.05,
random_state=2017)
outer = y == 0
inner = y == 1
```

3）绘制数据的分布来显示两个类：

```
plt.title("Two Circles")
plt.plot(X[outer, 0], X[outer, 1], "ro")
plt.plot(X[inner, 0], X[inner, 1], "bo")
plt.show()
```

非线性可分数据示例如图 2.5 所示。

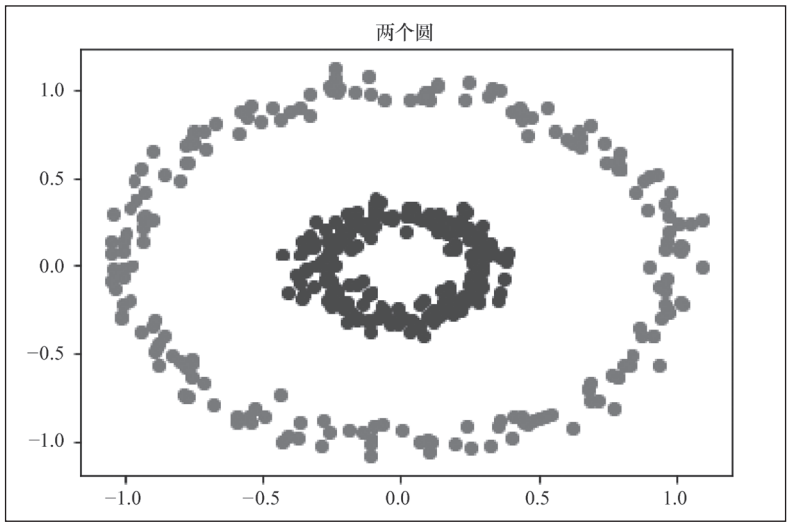


图 2.5　非线性可分数据示例

4）标准化数据，以确保两个圆的中心是（1,1）：

```
X = X+1
```

5）为了确定算法的性能，对数据进行分割：

```
X_train, X_val, y_train, y_val = train_test_split(X, y,
test_size=0.2, random_state=SEED)
```

6）线性激活函数在这种情况下不起作用，所以使用 sigmoid 函数：

```
def sigmoid(x):
 return 1 / (1 + np.exp(-x))
```

7）接下来，定义超参数：

```
n_hidden = 50 # 隐层神经元数目
n_epochs = 1000
learning_rate = 1
```

8）初始化权重和其他变量：

```
# 初始化权值
weights_hidden = np.random.normal(0.0, size=(X_train.shape[1],
n_hidden))
weights_output = np.random.normal(0.0, size=(n_hidden))

hist_loss = []
hist_accuracy = []
```

9）运行单层神经网络并输出统计信息：

```
for e in range(n_epochs):
 del_w_hidden = np.zeros(weights_hidden.shape)
 del_w_output = np.zeros(weights_output.shape)

 # 按批量1循环加载训练数据
 for x_, y_ in zip(X_train, y_train):
  # 前向计算
  hidden_input = np.dot(x_, weights_hidden)
  hidden_output = sigmoid(hidden_input)
  output = sigmoid(np.dot(hidden_output, weights_output))

  # 后向计算
  error = y_ - output
  output_error = error * output * (1 - output)
  hidden_error = np.dot(output_error, weights_output) *
hidden_output
                      * (1 - hidden_output)
  del_w_output += output_error * hidden_output
  del_w_hidden += hidden_error * x_[:, None]

 # 更新权值
 weights_hidden += learning_rate * del_w_hidden / X_train.shape[0]
 weights_output += learning_rate * del_w_output / X_train.shape[0]

 # 输出状态（验证损失的精度）
 if e % 100 == 0:
  hidden_output = sigmoid(np.dot(X_val, weights_hidden))
  out = sigmoid(np.dot(hidden_output, weights_output))
  loss = np.mean((out - y_val) ** 2)
  # 最终预测值基于阈值0.5
  predictions = out > 0.5
  accuracy = np.mean(predictions == y_val)
  print("Epoch: ", '{:>4}'.format(e),
        "; Validation loss: ", '{:>6}'.format(loss.round(4)),
        "; Validation accuracy: ",
'{:>6}'.format(accuracy.round(4)))
```

在图 2.6 中，显示了训练期间的输出结果：

```
迭代周期:      0 ;  验证损失值:   0.4194 ;  验证精度:    0.3
迭代周期:    100 ;  验证损失值:   0.2034 ;  验证精度:  0.8167
迭代周期:    200 ;  验证损失值:   0.1519 ;  验证精度:  0.8667
迭代周期:    300 ;  验证损失值:   0.1165 ;  验证精度:    0.9
迭代周期:    400 ;  验证损失值:   0.0926 ;  验证精度:   0.95
迭代周期:    500 ;  验证损失值:   0.0761 ;  验证精度:    1.0
迭代周期:    600 ;  验证损失值:   0.0643 ;  验证精度:    1.0
迭代周期:    700 ;  验证损失值:   0.0555 ;  验证精度:    1.0
迭代周期:    800 ;  验证损失值:   0.0488 ;  验证精度:    1.0
迭代周期:    900 ;  验证损失值:   0.0436 ;  验证精度:    1.0
```

图 2.6　训练期间统计信息

2.4 构建一个多层神经网络

在前面的方案中创建的实际上是 FNN 最简单的形式：信息只在一个方向流动的神经网络。对于下一个方案，将隐层的数量从一层扩展到多层。增加额外的层可以增加网络的性能，学习复杂的非线性模式。

如图 2.7 所示，增加附加层后，神经元连接（权重，也称为可训练参数）的数量会呈指数增长。在下一个方案中，将创建一个具有两个隐层的网络来预测葡萄酒的品质。这是一个回归任务，所以对输出层使用线性激活。对于隐层，使用 ReLU 激活函数。这个方案使用 Keras 框架来实现前馈网络。

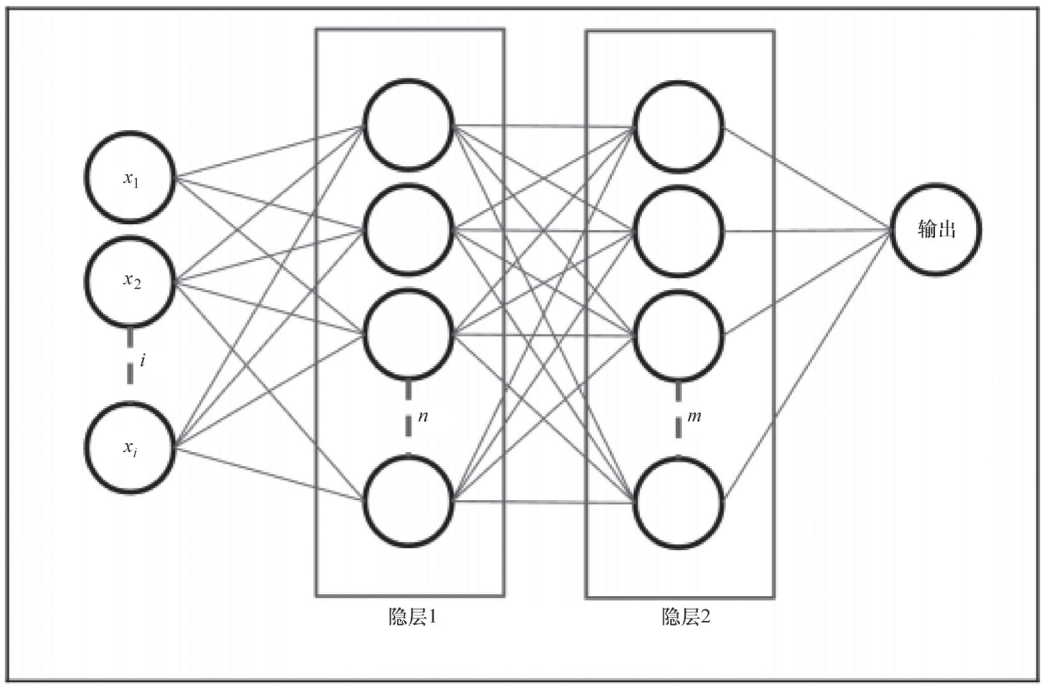


图 2.7　两层神经网络，具有 i 个输入变量、n 个隐层神经元和 m 个隐层神经元以及单个输出神经元

如何去做…

1）从导入函数库和数据集开始：

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

```
from keras.optimizers import Adam

from sklearn.preprocessing import StandardScaler

SEED = 2017
```

2）加载数据集：

```
data = pd.read_csv('Data/winequality-red.csv', sep=';')
y = data['quality']
X = data.drop(['quality'], axis=1)
```

3）拆分数据，进行网络训练和测试：

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=SEED)
```

4）输出显示平均品质和第一行训练集：

```
print('Average quality training set:
{:.4f}'.format(y_train.mean()))
X_train.head()
```

在图 2.8 中，可以看到一个网络训练的结果输出实例。

训练集平均质量：5.623143

	固定酸度	挥发酸度	柠檬酸	残糖	氯化物	游离二氧化硫	总二氧化硫	密度	pH值	硫酸盐	酒精
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4

图 2.8　训练数据

5）接下来的一个重要步骤是对输入数据进行标准化：

```
scaler = StandardScaler().fit(X_train)
X_train = pd.DataFrame(scaler.transform(X_train))
X_test = pd.DataFrame(scaler.transform(X_test))
```

6）确定基准预测：

```
# 对每个验证输入的训练数据预测其平均质量
print('MSE:', np.mean((y_test - ([y_train.mean()] *
y_test.shape[0])) ** 2).round(4))
## MSE: 0.594
```

7）现在，通过定义网络架构来构建神经网络模型：

```
model = Sequential()
# 第一隐层含100个神经元
model.add(Dense(200, input_dim=X_train.shape[1],
activation='relu'))
# 第二隐层含50个神经元
model.add(Dense(25, activation='relu'))
# 输出层
model.add(Dense(1, activation='linear'))
# 设置优化器
opt = Adam()
# 编译模型
model.compile(loss='mse', optimizer=opt, metrics=['accuracy'])
```

8）定义回调函数，以便使用早停技术并保存最佳模型：

```
callbacks = [
             EarlyStopping(monitor='val_acc', patience=20,
verbose=2),
ModelCheckpoint('checkpoints/multi_layer_best_model.h5',
monitor='val_acc', save_best_only=True, verbose=0)
            ]
```

9）运行批大小为 64 的模型 5000 个周期，验证集按 20%分割：

```
batch_size = 64
n_epochs = 5000
model.fit(X_train.values, y_train, batch_size=batch_size,
epochs=n_epochs, validation_split=0.2,
             verbose=2,
             callbacks=callbacks)
```

10）加载最佳权重后，可以在测试集上输出显示性能：

```
best_model = model
best_model.load_weights('checkpoints/multi_layer_best_model.h5')
best_model.compile(loss='mse', optimizer='adam',
```

```
metrics=['accuracy'])

# 评价测试集
score = best_model.evaluate(X_test.values, y_test, verbose=0)
print('Test accuracy: %.2f%%' % (score[1]*100))

## 测试精度：66.25%
## 基准数据库精度：62.4%
```

对于小数据集，建议重新训练全部训练集数据（无验证集），并上调与附加数据成比例的周期数。另一种选择是在进行预测时使用交叉验证和平均结果。

2.5 开始使用激活函数

如果只使用线性激活函数，神经网络将代表大量的线性组合。然而，神经网络的功用在于它们对复杂非线性行为的建模能力。在前面的方案中简单介绍了非线性激活函数 Sigmoid 和 ReLU，还有很多流行的非线性激活函数，如 ELU、Leaky ReLU、TanH 和 Maxout。

没有一条关于哪个激活最适合隐层神经元的规则。深度学习是一个相对较新的领域，大多数结果是通过反复试验和误差调整来获得，而不是数学证明。对于输出神经元，使用单个输出神经元和线性激活函数来执行回归任务。对于具有 *n* 类的分类任务，使用 *n* 个输出节点和一个 Softmax 激活函数。Softmax 函数迫使网络输出 0~1 之间的概率为互斥类，概率总和为 1。对于二元分类，也可以使用单个输出节点和 Sigmoid 激活函数输出概率值。

为隐层神经元选择正确的激活函数至关重要。在反向传播过程中，更新取决于激活函数的导数。对于深度神经网络，更新权重的梯度可以在前几层（也称为**梯度渐变**问题）中变为零，或者可以指数级增长（也称为**梯度爆炸**问题）。特别是当激活函数仅在小值时取导数（例如 Sigmoid 激活函数）或激活函数导数取值大于 1 时，上述情况尤为明显。

ReLU 等激活函数可以防止这种情况发生。当输出为正时，ReLU 的导数为 1，否则为 0。当使用 ReLU 激活函数时，会产生一个稀疏网络，其数量相对较少。在这种情况下，通过网络传递的损失似乎更有用。在一些情况下，ReLU 导致太多的神经元死亡。在这种情况下，应该尝试一个变种，比如 Leaky ReLU。在下一个方案中，用深度 FNN 对手写数字进行分类，以此来对比 Sigmoid 和 ReLU 激活函数之间的结果差异。

如何去做…

1）导入函数库和数据集：

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical
from keras.callbacks import Callback

from keras.datasets import mnist

SEED = 2017
```

2）加载 MNIST 数据集：

```
(X_train, y_train), (X_val, y_val) = mnist.load_data()
```

3）显示每个标签的示例并输出对显示每个标签的计数：

```
# 按标号绘制第一个图像
unique_labels = set(y_train)
plt.figure(figsize=(12, 12))

i = 1
for label in unique_labels:
 image = X_train[y_train.tolist().index(label)]
 plt.subplot(10, 10, i)
 plt.axis('off')
 plt.title("{0}: ({1})".format(label,
y_train.tolist().count(label)))

 i += 1
 _ = plt.imshow(image, cmap='gray')
plt.show()
```

获得如图 2.9 所示结果。

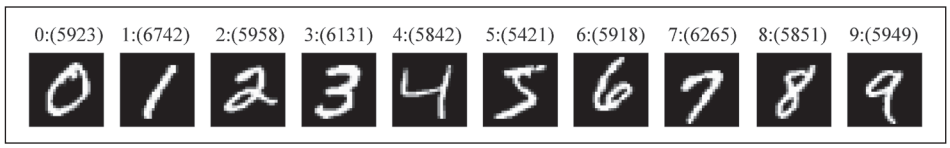


图 2.9　MNIST 数据集中的标签（和计数）示例

4）预处理数据：

```
# 数据标准化
X_train = X_train.astype('float32')/255.
X_val = X_val.astype('float32')/255.

# 对标号独热编码
y_train = np_utils.to_categorical(y_train, 10)
y_val = np_utils.to_categorical(y_val, 10)

# 拼合数据 - 将图像拉伸为一系列连续值
X_train = np.reshape(X_train, (60000, 784))
X_val = np.reshape(X_val, (10000, 784))
```

5）用 Sigmoid 激活函数定义模型：

```
model_sigmoid = Sequential()
model_sigmoid.add(Dense(700, input_dim=784, activation='sigmoid'))
model_sigmoid.add(Dense(700, activation='sigmoid'))
model_sigmoid.add(Dense(700, activation='sigmoid'))
model_sigmoid.add(Dense(700, activation='sigmoid'))
model_sigmoid.add(Dense(700, activation='sigmoid'))
model_sigmoid.add(Dense(350, activation='sigmoid'))
model_sigmoid.add(Dense(100, activation='sigmoid'))
model_sigmoid.add(Dense(10, activation='softmax'))

# 用SGD编译模型
model_sigmoid.compile(loss='categorical_crossentropy',
optimizer='sgd', metrics=['accuracy'])
```

6）使用 ReLU 激活函数定义模型：

```
model_relu = Sequential()
model_relu.add(Dense(700, input_dim=784, activation='relu'))
model_relu.add(Dense(700, activation='relu'))
model_relu.add(Dense(700, activation='relu'))
model_relu.add(Dense(700, activation='relu'))
model_relu.add(Dense(700, activation='relu'))
model_relu.add(Dense(350, activation='relu'))
model_relu.add(Dense(100, activation='relu'))
model_relu.add(Dense(10, activation='softmax'))

# 用SGD编译模型
model_relu.compile(loss='categorical_crossentropy',
optimizer='sgd', metrics=['accuracy'])
```

7）创建一个回调函数来存储每个批次的损失值：

```
class history_loss(cb.Callback):
 def on_train_begin(self, logs={}):
 self.losses = []

 def on_batch_end(self, batch, logs={}):
 batch_loss = logs.get('loss')
 self.losses.append(batch_loss)
```

8）运行模型：

```
n_epochs = 10
batch_size = 256
validation_split = 0.2

history_sigmoid = history_loss()
model_sigmoid.fit(X_train, y_train, epochs=n_epochs,
batch_size=batch_size,
 callbacks=[history_sigmoid],
 validation_split=validation_split, verbose=2)

history_relu = history_loss()
model_relu.fit(X_train, y_train, epochs=n_epochs,
batch_size=batch_size,
 callbacks=[history_relu],
 validation_split=validation_split, verbose=2)
```

9）绘制损失分布图例：

```
plt.plot(np.arange(len(history_sigmoid.losses)), sigmoid,
label='sigmoid')
plt.plot(np.arange(len(history_relu.losses)), relu, label='relu')
plt.title('Losses')
plt.xlabel('number of batches')
plt.ylabel('loss')
plt.legend(loc=1)
plt.show()
```

这段代码给出了如图 2.10 所示的结果。

10）提取每层模型的最大权重：

```
w_sigmoid = []
w_relu = []
for i in range(len(model_sigmoid.layers)):
    w_sigmoid.append(max(model_sigmoid.layers[i].get_weights()[1]))
    w_relu.append(max(model_relu.layers[i].get_weights()[1]))
```

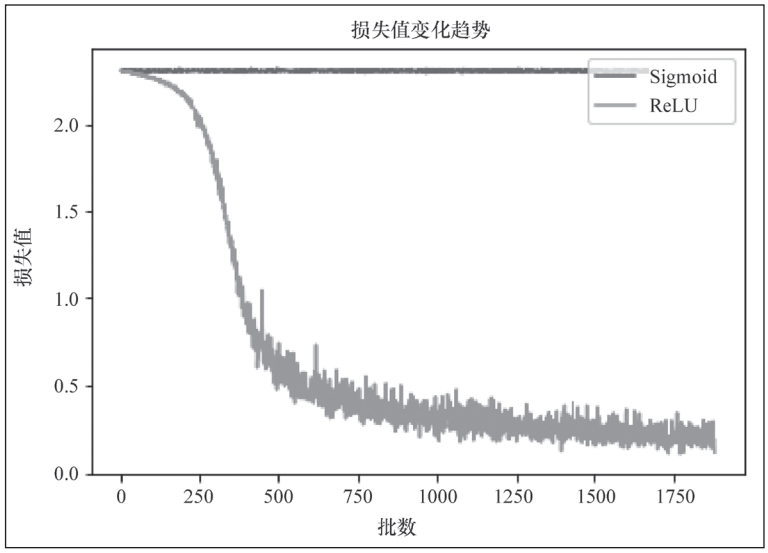


图 2.10　Sigmoid 和 ReLU 模型的损失

11）绘制两个模型的权重分布图例：

```
fig, ax = plt.subplots()

index = np.arange(len(model_sigmoid.layers))
bar_width = 0.35

plt.bar(index, w_sigmoid, bar_width, label='sigmoid',
color='b', alpha=0.4)
plt.bar(index + bar_width, w_relu, bar_width, label='relu',
color='r', alpha=0.4)
plt.title('Weights across layers')
plt.xlabel('layer number')
plt.ylabel('maximum weight')
plt.legend(loc=0)

plt.xticks(index + bar_width / 2, np.arange(8))
plt.show()
```

得到如图 2.11 所示结果。

在第 3 章　卷积神经网络　中，将向读者展示如何在 MNIST 数据集上获得超过 99%的精度。

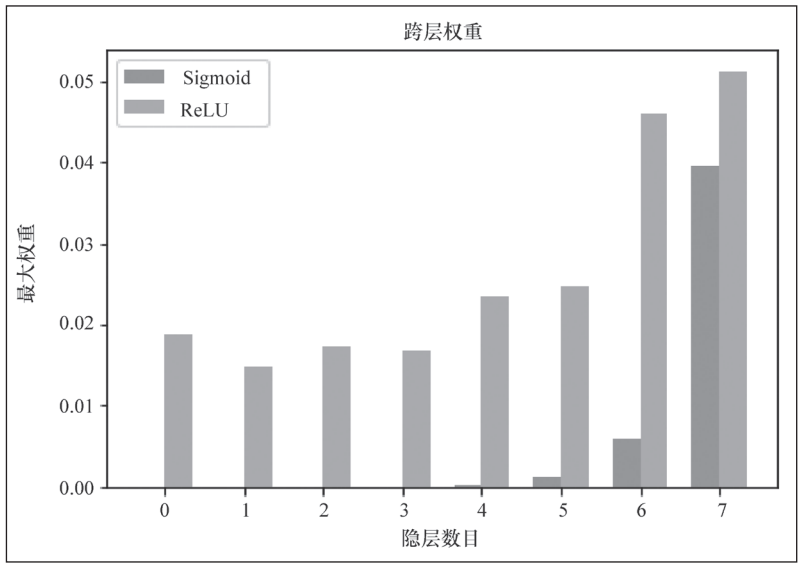


图 2.11　Sigmoid 和 ReLU 激活函数的各层之间的最大权重

2.6　关于隐层和隐层神经元的实验

一般神经网络中最常用的层是完全连接的层。在全连接层中，两个连续层中的神经元都是成对连接的。但是，同一隐层中的神经元之间不会共享任何连接。如前所述，层与层之间的连接权重也称为可训练参数。这些连接的权重由网络进行训练。连接越多，参数越多，建模越复杂。大部分最先进的模型都有超过 1 亿个参数。但是，具有多个层次和神经元的深度神经网络需要更多的时间来训练。而且，对于非常深的模型，推断预测要花费更长的时间（这在实时环境中可能存在问题）。下面将介绍其他常用网络特有的层类型。

选择正确数量的隐层和隐层神经元可能很重要。当使用的节点太少时，模型将无法拾取所有的信号，这会导致精度降低和预测性能变差（欠拟合）。使用太多的节点，模型将倾向于过度训练数据（见正规化技术，防止过拟合），泛化能力差。因此，要看验证数据的性能，才能找到适当的平衡。在下一个方案中，将展示一个过拟合的例子，并输出可训练参数的数量。

如果有大量可用的高维训练数据，则深度 FNN 表现良好。对于简单的分类或回归任务，通常单层神经网络效果最好。

如何去做…

1）导入函数库和数据集：

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD

SEED = 2017
```

2）加载数据集：

```
data = pd.read_csv('Data/winequality-red.csv', sep=';')
y = data['quality']
X = data.drop(['quality'], axis=1)
```

3）将数据集分解为训练数据和测试数据：

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=SEED)
```

4）标准化输入数据：

```
scaler = StandardScaler().fit(X_train)
X_train = pd.DataFrame(scaler.transform(X_train))
X_test = pd.DataFrame(scaler.transform(X_test))
```

5）定义模型和优化器并进行编译：

```
model = Sequential()
model.add(Dense(1024, input_dim=X_train.shape[1],
activation='relu'))
model.add(Dense(1024, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(512, activation='relu'))
# 输出层
model.add(Dense(1, activation='linear'))
# 设置优化器
opt = SGD()
# 编译模型
model.compile(loss='mse', optimizer=opt, metrics=['accuracy'])
```

6）设置超参数并训练模型：

```
n_epochs = 500
batch_size = 256

history = model.fit(X_train.values, y_train, batch_size=batch_size,
epochs=n_epochs, validation_split=0.2, verbose=0)
```

7）对测试集进行测试：

```
predictions = model.predict(X_test.values)
print('Test accuracy:
{:f>2}%'.format(np.round(np.sum([y_test==predictions.flatten().roun
d()])/y_test.shape[0]*100, 2)))
```

8）绘制训练精度及验证精度图例：

```
plt.plot(np.arange(len(history.history['acc'])),
history.history['acc'], label='training')
plt.plot(np.arange(len(history.history['val_acc'])),
history.history['val_acc'], label='validation')
plt.title('Accuracy')
plt.xlabel('epochs')
plt.ylabel('accuracy ')
plt.legend(loc=0)
plt.show()
```

获得图 2.12 所示结果。

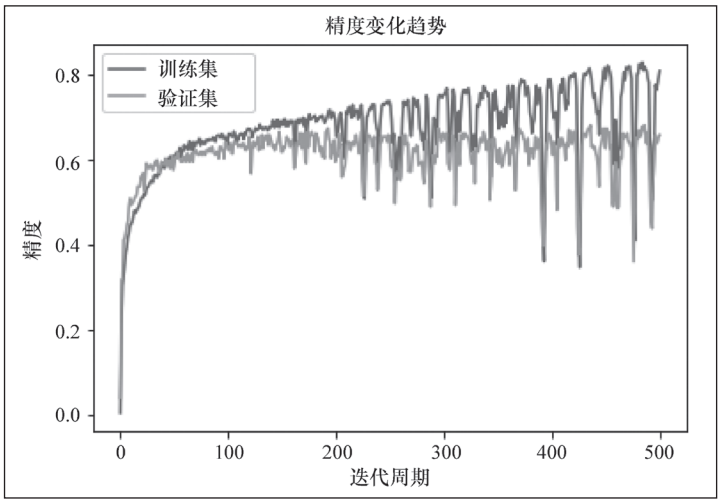


图 2.12　训练精度和验证精度

应该重点关注验证精度，并且在大约 450 个周期之后使用早停法来停止训练。这将产生最高的验证精度。在用正则化提高泛化能力添加 Dropout 措施以防止过拟合的内容中，介绍防止过拟合的技术。通过使用这些技术，可以创建更深的模型，而不会过拟合训练数据。

更多…

一般来说，对于 FNN，使用的隐层神经元数量逐层减少。这意味着在第一个隐层使用的隐层神经元最多，并且减少每个附加隐层的隐层神经元数量。通常隐层神经元的数量在每个步骤中除以 2。请记住，这是一个经验法则，隐层和隐层神经元的数量应根据验证结果，通过反复试验与减小误差来获得。

在第 13 章　网络内部构造　和第 14 章　预训练模型　中，将介绍隐层和隐层神经元数量的优化技巧。

2.7　实现一个自动编码器

对于自动编码器，使用不同的网络架构，如图 2.13 所示。在前几层中，减少了隐层神经元的数量。在中间，又开始增加隐层神经元的数量，直到隐层神经元的数量与输入变量的数量相同。中间的隐层可以看作输入的编码变体，其输出决定了编码变体的质量。

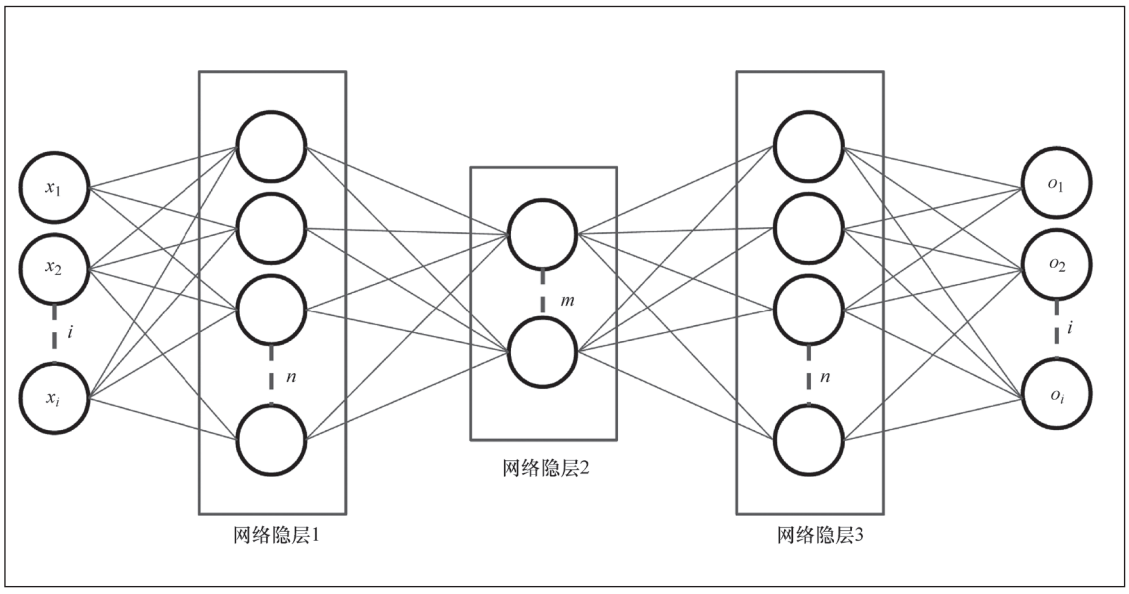


图 2.13　具有三个隐层的自动编码器网络，其中 $m < n$

在下一个方案中，将在 Keras 中实现一个自动编码器，将**街景门牌号（SVHN）**基准库从 32 × 32 图像解码为 32 个浮点数。可以通过解码成 32 × 32 图像并比较图像来确定编码器的质量。

如何去做…

1）用下面的代码导入必要的函数库：

```
import numpy as np
from matplotlib import pyplot as plt
import scipy.io

from keras.models import Sequential
from keras.layers.core import Dense
from keras.optimizers import Adam
```

2）加载数据集并提取所需要的数据：

```
mat = scipy.io.loadmat('./data/train_32x32.mat')
mat = mat['X']
b, h, d, n = mat.shape
```

3）预处理数据：

```
# 灰度值计算
img_gray = np.zeros(shape =(n, b * h))

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

for i in range(n):
    #转换灰度值
    img = rgb2gray(mat[:,:,:,i])
    img = img.reshape(1, 1024)
    img_gray[i,:] = img

# 标准化
X_train = img_gray/255.
```

4）接下来，定义自动编码器的网络架构：

```
img_size = X_train.shape[1]
model = Sequential()
model.add(Dense(256, input_dim=img_size, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))

model.add(Dense(64, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(img_size, activation='sigmoid'))

opt = Adam()
model.compile(loss='binary_crossentropy', optimizer=opt)
```

5）现在，开始训练自动编码器：

```
n_epochs = 100
batch_size = 512

model.fit(X_train, X_train, epochs=n_epochs, batch_size=batch_size,
shuffle=True, validation_split=0.2)
```

6）看看自动编码如何在训练集上执行：

```
pred = model.predict(X_train)
```

7）绘制一些原始图像及其解码版本，如图 2.14 所示。

```
n = 5
plt.figure(figsize=(15, 5))
for i in range(n):
    # 显示原图
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(mat[i].reshape(32, 32), cmap='gray')

    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(pred[i].reshape(32, 32), cmap='gray')
plt.show()
```

在第 3 章　卷积神经网络　中，将向读者展示如何为 SVHN 数据集实现一个卷积自动编码器。

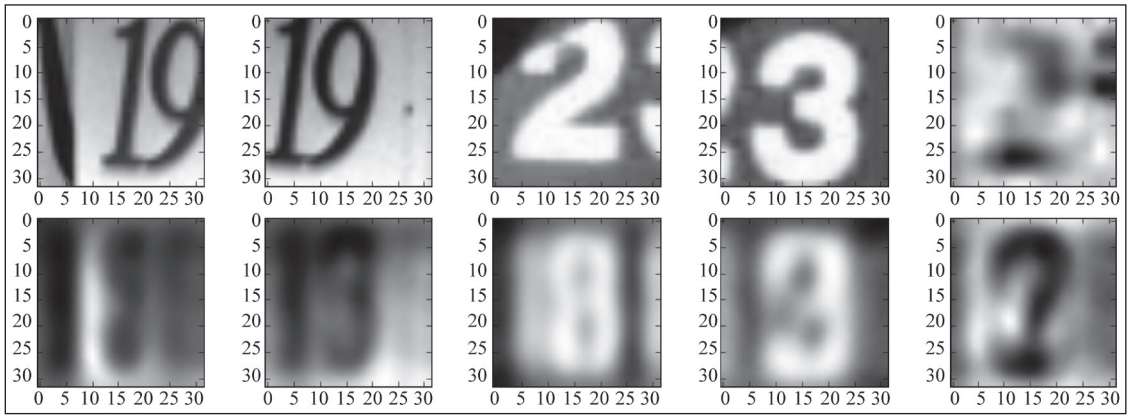


图 2.14　门牌号码自动编码网络输入与输出示例

2.8　调整损失函数

对监督学习问题，在训练一个神经网络时，其目标是最小化损失函数。损失函数（也称为误差函数、成本函数或优化函数）将预测结果与正向传播期间的基本事实进行比较。这个损失函数的输出用于优化反向传播期间的权重。因此，损失函数在训练网络中至关重要。通过设置正确的损失函数，使网络针对期望的预测进行优化。例如，对于不平衡的数据集，需要一个不同的损失函数。

在以前的方案中，使用**均方误差**（MSE）和分类交叉熵作为损失函数，以及其他常用的损失函数，另一种是选择创建自定义损失函数。自定义损失函数可以优化到所需的输出。当实施**生成对抗网络**（GAN）时，这将变得更加重要。在下面的方案中，将训练分为有调整权重和无调整权重的网络体系结构，按此来计算不平衡类别的损失函数。

如何去做…

1）用下面的代码导入函数库：

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn.metrics import confusion_matrix

from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
```

2）导入 MNIST 数据集并创建一个 9s 和 4s 的不平衡数据集：

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# 提取 9s 中的全部样本，提取 4s 中的100个样本

y_train_9 = y_train[y_train == 9]
y_train_4 = y_train[y_train == 4][:100]
X_train_9 = X_train[y_train == 9]
X_train_4 = X_train[y_train == 4][:100]
X_train = np.concatenate((X_train_9, X_train_4), axis=0)
y_train = np.concatenate((y_train_9, y_train_4), axis=0)

y_test_9 = y_test[y_test == 9]
y_test_4 = y_test[y_test == 4]
X_test_9 = X_test[y_test == 9]
X_test_4 = X_test[y_test == 4]
X_test = np.concatenate((X_test_9, X_test_4), axis=0)
y_test = np.concatenate((y_test_9, y_test_4), axis=0)
```

3）标准化和扁平化数据：

```
X_train = X_train.astype('float32')/255.
X_test = X_test.astype('float32')/255.
X_train = X_train.reshape(len(X_train), np.prod(X_train.shape[1:]))
X_test = X_test.reshape(len(X_test), np.prod(X_test.shape[1:]))
```

4）将目标转换为二进制分类问题并输出显示计数：

```
y_train_binary = y_train == 9
y_test_binary = y_test == 9
print(np.unique(y_train_binary, return_counts=True))
```

5）定义网络体系结构并编译：

```
model = Sequential()
model.add(Dense(512, input_dim=X_train.shape[1],
activation='relu'))
model.add(Dropout(0.75))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.75))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.75))
model.add(Dense(128, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

opt = Adam()
model.compile(loss='binary_crossentropy', optimizer=opt,
metrics=['binary_accuracy'])
```

6）创建一个回调函数来使用早停法：

```
callbacks = [EarlyStopping(monitor='val_loss', patience=5)]
```

7）定义每个类的损失权重：

```
class_weight_equal = {False : 1., True: 1}
class_weight_imbalanced = {False : 100, True: 1}
```

8）对两个类训练相同权重的模型：

```
n_epochs = 1000
batch_size = 512
validation_split = 0.01

model.fit(X_train, y_train_binary, epochs=n_epochs,
batch_size=batch_size, shuffle=True,
validation_split=validation_split, class_weight=class_weight_equal,
          callbacks=callbacks, verbose=0
)
```

9）在测试集上进行测试并输出混淆矩阵：

```
preds_equal = model.predict(X_test)
confusion_matrix(y_test_binary, np.round(preds_equal),
labels=[True, False])

#数组([[1009, 0],
# [ 982, 0]])
```

10）接下来，用不平衡的权重训练，并在测试集上进行测试：

```
model.compile(loss='binary_crossentropy', optimizer='rmsprop',
metrics=['binary_accuracy'])

model.fit(X_train, y_train_binary, epochs=n_epochs,
          batch_size=batch_size, shuffle=True,
validation_split=validation_split,
class_weight=class_weight_imbalanced,
          callbacks=callbacks, verbose=0
         )

preds_imbalanced = model.predict(X_test)
confusion_matrix(y_test_binary, np.round(preds_imbalanced),
labels=[True, False])

#数组([[1009, 3],
# [ 546, 436]])
```

2.9 测试不同的优化器

最流行和最有名的优化器是**随机梯度下降**（SGD）。这种技术也广泛应用于其他机器学习模型。SGD 是一种通过迭代发现最小值或最大值的方法。SGD 有许多流行的变种，其通过使用自适应学习率来加快收敛速度和减少调整。表 2.1 概述了深度学习中最常用的优化器。

表 2.1 深度学习中最常用的优化器

优化算法	超 参 数	说 明
SGD	Learning rate, decay	+ 学习速率直接影响性能（较小的学习速率避免了局部最小值） – 需要更多的手动调整 – 收敛慢
AdaGrad	Learning rate, epsilon, decay	+ 所有参数的自适应学习速率（非常适合稀疏数据） – 学习速率太小，停止学习
AdaDelta	Learning rate, rho, epsilon, decay	+ 开始收敛快 – 在最低点附近减速
Adam	Learning rate, beta 1, beta 2, epsilon, decay	+ 所有参数的自适应学习速率和动量
RMSprop	Learning rate, rho, epsilon, decay	+– 类似于 AdaDelta，但对于非凸面问题效果更好
Momentum	Learning rate, momentum, decay	+ 收敛更快 + 减少错误方向的移动 – 如果动量太大，则在最小值附近振荡
Nesterov 加速梯度法（Nesterov Accelerated Gradient, NAG）	Learning rate, momentum, decay	+ 与动量类似，但较早放缓

优化器的选择是任意的，其很大程度上取决于用户调整优化器的能力。没有最好的解决方案能解决所有问题。SGD 能使用户通过选择一个小的学习率来避免局部最优化，但是缺点是训练时间太长。在下面的方案中，将用不同的优化器来训练网络，并比较结果。

如何去做…

1）导入函数库：

```
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras.optimizers import SGD, Adadelta, Adam, RMSprop, Adagrad,
Nadam, Adamax
```

2）导入数据集并提取目标变量：

```
data = pd.read_csv('../Data/winequality-red.csv', sep=';')
y = data['quality']
X = data.drop(['quality'], axis=1)
```

3）拆分数据集进行训练、验证和测试：

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=2017)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=0.2, random_state=2017)
```

4）定义一个创建模型的函数：

```
def create_model(opt):
model = Sequential()
model.add(Dense(100, input_dim=X_train.shape[1],
activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(25, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='linear'))
return model
```

5）创建一个函数，定义在训练期间将使用的回调函数：

```
def create_callbacks(opt):
callbacks = [
EarlyStopping(monitor='val_acc', patience=200, verbose=2),
ModelCheckpoint('best_model_' + opt + '.h5', monitor='val_acc',
save_best_only=True, verbose=0)
]
return callbacks
```

6）创建一个想要尝试的优化器字典：

```
opts = dict({
'sgd': SGD(),
 'sgd-0001': SGD(lr=0.0001, decay=0.00001),
 'adam': Adam(),
 'adadelta': Adadelta(),
 'rmsprop': RMSprop(),
 'rmsprop-0001': RMSprop(lr=0.0001),
 'nadam': Nadam(),
 'adamax': Adamax()
})
```

7）训练网络并存储结果：

```
results = []
# 遍历优化器
for opt in opts:
 model = create_model(opt)
 callbacks = create_callbacks(opt)
 model.compile(loss='mse', optimizer=opts[opt],
metrics=['accuracy'])
 hist = model.fit(X_train.values, y_train, batch_size=128,
epochs=5000,
  validation_data=(X_val.values, y_val),
  verbose=0,
  callbacks=callbacks)
 best_epoch = np.argmax(hist.history['val_acc'])
 best_acc = hist.history['val_acc'][best_epoch]
 best_model = create_model(opt)
 # 加载具有最高验证精度的模型
 best_model.load_weights('best_model_' + opt + '.h5')
 best_model.compile(loss='mse', optimizer=opts[opt],
metrics=['accuracy'])
 score = best_model.evaluate(X_test.values, y_test, verbose=0)
 results.append([opt, best_epoch, best_acc, score[1]])
```

8）比较结果：

```
res = pd.DataFrame(results)
res.columns = ['optimizer', 'epochs', 'val_accuracy',
'test_accuracy']
res
```

得到如图 2.15 所示结果。

	优化器	迭代周期	验证集精度	测试集精度
0	sgd-0001	106	0.519531	0.53750
1	adam	112	0.589844	0.61250
2	sgd	0	0.000000	0.000000
3	adamax	213	0.578125	0.58750
4	rmsprop	600	0.597656	0.61250
5	adadelta	109	0.570312	0.58125
6	rmsprop-0001	38	0.539062	0.56250
7	nadam	84	0.597656	0.59375

图 2.15　对葡萄酒质量数据集使用不同优化器训练的结果

在第 12 章　超参数选择、调优和神经网络学习　中，将演示如何使用网格搜索来进行参数调优。网格搜索可以用来寻找合适的优化器（结合其他超参数）来调整神经网络学习过程。

2.10　使用正则化技术提高泛化能力

过度训练数据是机器学习最大的挑战之一。有许多机器学习算法能够通过记忆所有的情况来对训练数据进行训练。在这种情况下，该算法可能无法泛化并对新数据做出正确的预测。这对于深度学习来说是一个特别大的威胁，其中神经网络具有大量的可训练参数。因此，创建一个具有代表性的验证集是非常重要的。

在深度学习中，解决新问题的一般建议是先尽可能地过拟合训练数据。这可以确保模型能够对训练数据进行训练并且足够复杂。之后，应该尽可能地调整，以确保该模型能够泛化未经验证的数据（验证集）。

大多数用于防止过拟合的技术都可以置于正则化之下。正则化包括机器学习中的所有技术，明确地减少测试（因泛化）误差，有时以较高的训练误差为代价。这样的技术可以是对参数空间加以限制的形式。假设一个具有较小权重的模型比一个具有较大权重的模型更简单。在下面的方案中，将应用 **L1 正则化**来防止模型过拟合。

如何去做…

1）首先，导入所需的函数库：

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras import regularizers
```

2）导入数据并提取特征：

```
data = pd.read_csv('Data/bike-sharing/hour.csv')
# 特征工程
ohe_features = ['season', 'weathersit', 'mnth', 'hr', 'weekday']
for feature in ohe_features:
    dummies = pd.get_dummies(data[feature], prefix=feature,
drop_first=False)
    data = pd.concat([data, dummies], axis=1)
```

```
drop_features = ['instant', 'dteday', 'season', 'weathersit',
                 'weekday', 'atemp', 'mnth', 'workingday', 'hr',
'casual', 'registered']
data = data.drop(drop_features, axis=1)
```

3）标准化数值数据：

```
norm_features = ['cnt', 'temp', 'hum', 'windspeed']
scaled_features = {}
for feature in norm_features:
    mean, std = data[feature].mean(), data[feature].std()
    scaled_features[feature] = [mean, std]
    data.loc[:, feature] = (data[feature] - mean)/std
```

4）分割数据集进行训练、验证和测试：

```
# 保存最后月份数据用于测试
test_data = data[-31*24:]
data = data[:-31*24]
# 提取目标域
target_fields = ['cnt']
features, targets = data.drop(target_fields, axis=1),
data[target_fields]
test_features, test_targets = test_data.drop(target_fields,
axis=1), test_data[target_fields]
# 创建验证集(基于最后)
X_train, y_train = features[:-30*24], targets[:-30*24]
X_val, y_val = features[-30*24:], targets[-30*24:]
```

5）定义网络体系结构：

```
model = Sequential()
model.add(Dense(250, input_dim=X_train.shape[1],
activation='relu'))
model.add(Dense(150, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(25, activation='relu'))
model.add(Dense(1, activation='linear'))

# 编译模型
model.compile(loss='mse', optimizer='sgd', metrics=['mse'])
```

6）在训练数据上训练网络体系结构并使用验证集：

```
n_epochs = 4000
batch_size = 1024

history = model.fit(X_train.values, y_train['cnt'],
                    validation_data=(X_val.values, y_val['cnt']),
                    batch_size=batch_size, epochs=n_epochs, verbose=0
                   )
```

7）绘制训练过程和验证损失图例，如图 2-16 所示。

```
plt.plot(np.arange(len(history.history['loss'])),
history.history['loss'], label='training')
plt.plot(np.arange(len(history.history['val_loss'])),
history.history['val_loss'], label='validation')
plt.title('Overfit on Bike Sharing dataset')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend(loc=0)
plt.show()
```

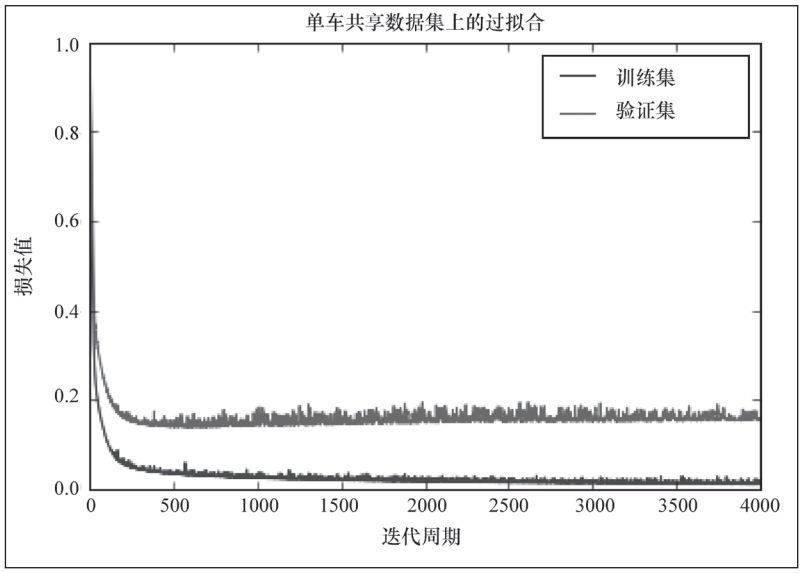


图 2.16　训练数据模型的过拟合（训练损失在 1000 个周期后略有上升趋势）

8）绘制最小损失图例，以及在多少个周期后达到了这个最小值：

```
print('Minimum loss: ', min(history.history['val_loss']),
          '\nAfter ', np.argmin(history.history['val_loss']), '
epochs')

# 最小损失值 : 0.140975862741
# 730次迭代周期
```

9）用 L2 正则化定义网络体系结构：

```
model_reg = Sequential()
model_reg.add(Dense(250, input_dim=X_train.shape[1],
activation='relu',
            kernel_regularizer=regularizers.l2(0.005)))
model_reg.add(Dense(150, activation='relu'))
model_reg.add(Dense(50, activation='relu'))
model_reg.add(Dense(25, activation='relu',
            kernel_regularizer=regularizers.l2(0.005)))
model_reg.add(Dense(1, activation='linear'))

# 编译模型
model_reg.compile(loss='mse', optimizer='sgd', metrics=['mse'])
```

10）训练调整后的网络：

```
hist_reg = model_reg.fit(X_train.values, y_train['cnt'],
validation_data=(X_val.values, y_val['cnt']),
                 batch_size=1024, nb_epoch=4000, verbose=0
                )
```

11）绘制出结果：

```
plt.plot(np.arange(len(history_reg.history['loss'])),
history_reg.history['loss'], label='training')
plt.plot(np.arange(len(history_reg.history['val_loss'])),
history_reg.history['val_loss'], label='validation')
plt.title('Use regularisation for Bike Sharing dataset')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend(loc=0)
plt.show()
```

得到如图 2.17 所示结果。

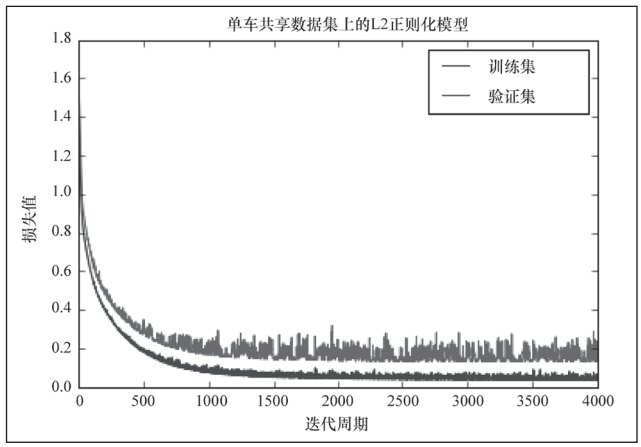


图 2.17　L2 正则化模型防止过拟合

12）输出显示正则化模型的统计数据：

```
print('Minimum loss: ', min(history_reg.history['val_loss']),
      '\nAfter ', np.argmin(history_reg.history['val_loss']), '
epochs')

# 最小损失值： 0.13514482975
# 3647次迭代周期
```

2.11 添加 Dropout 以防止过拟合

另一个流行的正则化方法是 Dropout 技术。在学习阶段通过随机删除神经元之间的连接，Dropout 方法迫使神经网络学习多个独立的表达。例如，当使用 0.5 的 Dropout 时，必须先对网络训练两个迭代，此后再训练新的连接权重。因此，一个有 Dropout 的网络可看作一个网络的集合。

在下面的方案中，将通过添加 Dropout 方法来改进一个明显过度训练数据的模型。

如何去做…

1）导入函数库如下：

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline


import numpy as np
from matplotlib import pyplot as plt
```

2）加载数据集并提取特征：

```
data = pd.read_csv('../Data/Bike-Sharing-Dataset/hour.csv')
# 特征工程
ohe_features = ['season', 'weathersit', 'mnth', 'hr', 'weekday']
for feature in ohe_features:
 dummies = pd.get_dummies(data[feature], prefix=feature,
drop_first=False)
 data = pd.concat([data, dummies], axis=1)
```

```
drop_features = ['instant', 'dteday', 'season', 'weathersit',
 'weekday', 'atemp', 'mnth', 'workingday', 'hr', 'casual',
'registered']
data = data.drop(drop_features, axis=1)
```

3）标准化特征：

```
norm_features = ['cnt', 'temp', 'hum', 'windspeed']
scaled_features = {}
for feature in norm_features:
 mean, std = data[feature].mean(), data[feature].std()
 scaled_features[feature] = [mean, std]
 data.loc[:, feature] = (data[feature] - mean)/std
```

4）拆分数据集以进行训练、验证和测试：

```
# 保存最后一个月的数据用于测试
test_data = data[-31*24:]
data = data[:-31*24]

# 提取目标域
target_fields = ['cnt']
features, targets = data.drop(target_fields, axis=1),
data[target_fields]
test_features, test_targets = test_data.drop(target_fields,
axis=1), test_data[target_fields]

# 创建验证集（基于最后）
X_train, y_train = features[:-30*24], targets[:-30*24]
X_val, y_val = features[-30*24:], targets[-30*24:]
```

5）定义模型：

```
model = Sequential()
model.add(Dense(250, input_dim=X_train.shape[1],
activation='relu'))
model.add(Dense(150, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(25, activation='relu'))
model.add(Dense(1, activation='linear'))

# 编译模型
model.compile(loss='mse', optimizer='sgd', metrics=['mse'])
```

6）设置超参数并训练模型：

```
n_epochs = 1000
batch_size = 1024

history = model.fit(X_train.values, y_train['cnt'],
    validation_data=(X_val.values, y_val['cnt']),
    batch_size=batch_size, epochs=n_epochs, verbose=0
    )
```

7）绘制训练过程和测试损失图例：

```
plt.plot(np.arange(len(history.history['loss'])),
history.history['loss'], label='training')
plt.plot(np.arange(len(history.history['val_loss'])),
history.history['val_loss'], label='validation')
plt.title('Overfit on Bike Sharing dataset')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend(loc=0)
plt.show()
```

得到的结果如图 2.18 所示。

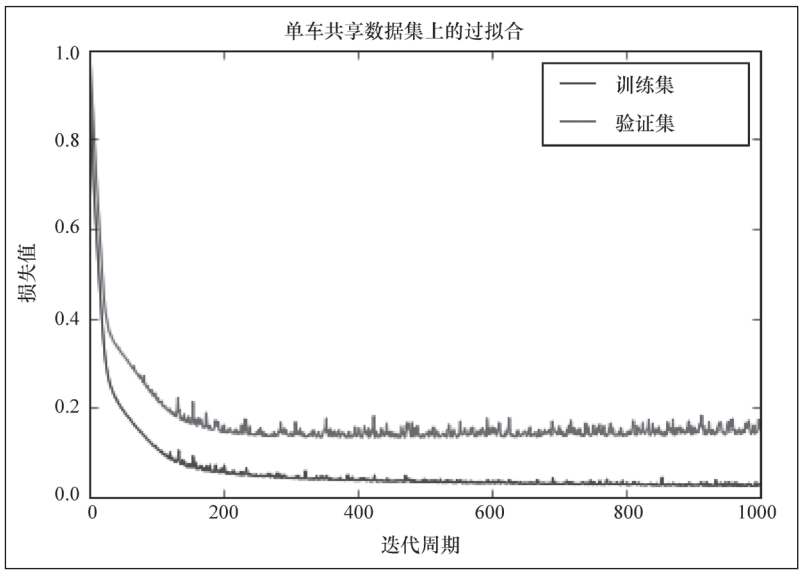


图 2.18　模型过拟合训练数据（训练损失值在 450 个周期后略有增加）

8）输出最小损失：

```
print('Minimum loss: ', min(hist.history['val_loss']),
 '\nAfter ', np.argmin(hist.history['val_loss']), ' epochs')

#最小损失值： 0.132234960794
#426 次迭代周期
```

9）在网络体系结构中添加 Dropout 以防止过拟合：

```
model_drop = Sequential()
model_drop.add(Dense(250, input_dim=X_train.shape[1],
activation='relu'))
model_drop.add(Dropout(0.20))
model_drop.add(Dense(150, activation='relu'))
model_drop.add(Dropout(0.20))
model_drop.add(Dense(50, activation='relu'))
model_drop.add(Dropout(0.20))
model_drop.add(Dense(25, activation='relu'))
model_drop.add(Dropout(0.20))
model_drop.add(Dense(1, activation='linear'))

# 编译模型
model_drop.compile(loss='mse', optimizer='sgd', metrics=['mse'])
```

10）训练新模型：

```
history_drop = model_drop.fit(X_train.values, y_train['cnt'],
    validation_data=(X_val.values, y_val['cnt']),
    batch_size=batch_size, epochs=n_epochs, verbose=0
    )
```

11）绘制结果，如图 2.19 所示。

```
plt.plot(np.arange(len(history_drop.history['loss'])),
history_drop.history['loss'], label='training')
plt.plot(np.arange(len(history_drop.history['val_loss'])),
history_drop.history['val_loss'], label='validation')
plt.title('Use dropout for Bike Sharing dataset')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend(loc=0)
plt.show()
```

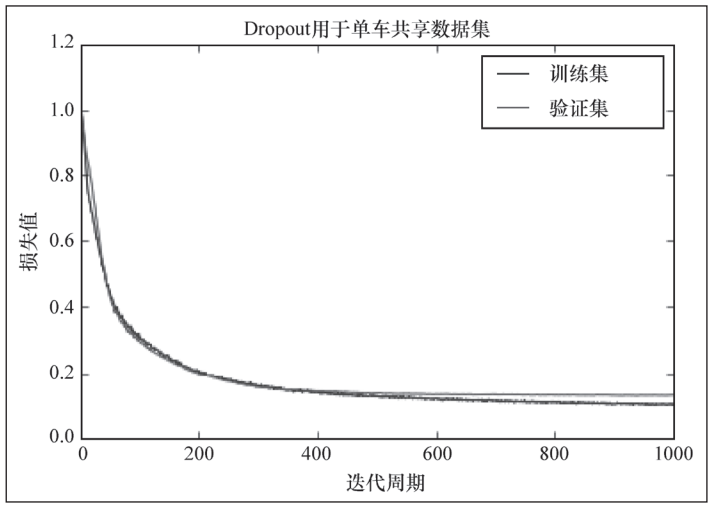


图 2.19　模型使用 Dropout 方法以防止过拟合

12）最后，输出最终统计结果数据：

```
print('Minimum loss: ', min(history_drop.history['val_loss']),
 '\nAfter ', np.argmin(history_drop.history['val_loss']), '
epochs')

# 最小损失值： 0.126063346863
# 998次迭代周期
```

第 3 章
卷积神经网络

在本章中，将重点讨论**卷积神经网络**（CNN），涵盖以下主题：

- 开始使用滤波器和参数共享；
- 应用层合并技术；
- 使用批量标准化进行优化；
- 理解填充和步长；
- 试验不同类型的初始化；
- 实现卷积自动编码器；
- 将一维 CNN 应用于文本。

3.1 简介

本章重点介绍 CNN 及其构建模块。在本章中，将提供有关 CNN 中使用的技术和优化方案。CNN，也称为 ConvNet，是一种特定类型的 FNN，其中网络具有一个或多个卷积层。卷积层可以用完全连接的层补充。如果网络只包含卷积层，网络结构命名为**完全卷积网络**（FCN）。

卷积网络和深度学习在计算机视觉中是密不可分的。但是，CNN 也可以用于其他应用，比如各种 NLP 问题，将在本章中介绍。

3.2 开始使用滤波器和参数共享

下面介绍卷积网络中最重要的部分：卷积层。在卷积层中，有对输入数据进行卷积的块（如滑动窗口）。该技术为每个块共享参数，以便可以在整个输入数据中检测块内的特征。块的大小称为内核大小或滤波器大小。综上所述，卷积层提取了整个特征集合内的局部特征。在图 3.1 中用图像作为输入数据来对此加以说明。

输入数据可以包含多个不同的特征，所以这个技巧可以应用多次。这被称为滤波器数量或滤波器深度。简单地说，如果在图像数据上使用具有五个滤波器的卷积层，则卷积块尝试学习图像中的五个不同特征，例如鼻子、眼睛、嘴巴、耳朵和睫毛等的面部特征。虽然这些特征从来没有明确规定，但是网络本身试图学习有价值的特征。

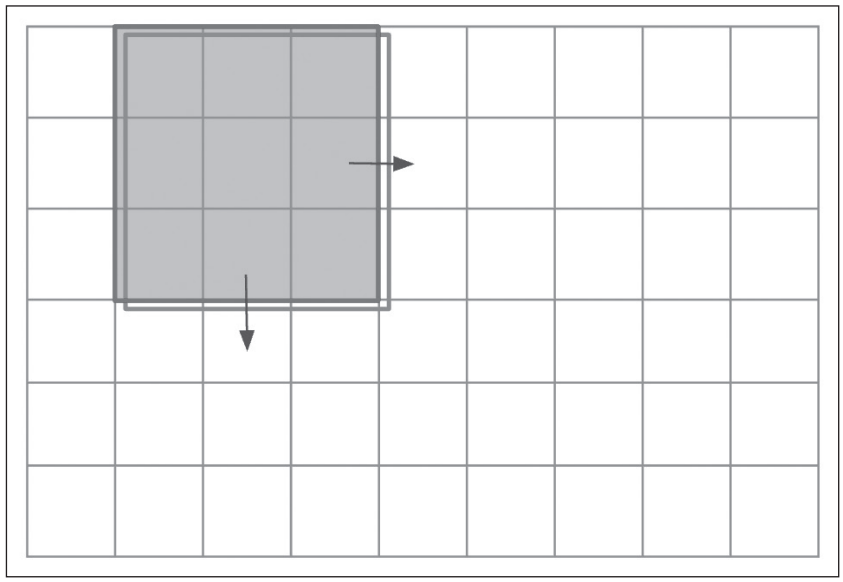


图 3.1　在 9 × 6 × 1 输入图像上的一个核心大小为 3 × 3 和滤波深度为 2（红色和绿色）的卷积块示例

对于 CNN，建议使用越来越多的滤波器，例如每个卷积层的滤波器数量的逐层翻倍：32、64 和 128。

在下面的方案中，将实现一个 CNN 来对第 2 章中使用的 MNIST 图像进行分类。

如何去做…

1）导入所有必要的函数库：

```
import numpy as np
from matplotlib import pyplot as plt

from keras.utils.np_utils import to_categorical
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Flatten
from keras.layers import Conv2D
from keras.optimizers import Adam
from keras.datasets import mnist
```

2）加载 MNIST 数据集：

```
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

3）重整训练数据来表示单通道图像输入：

```
img_rows, img_cols = X_train[0].shape[0], X_train[0].shape[1]
X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)
```

4）标准化输入数据：

```
X_train = X_train.astype('float32')/255.
X_test = X_test.astype('float32')/255.
```

5）对标签进行独热编码（one-hot encode）[⊖]：

```
n_classes = len(set(y_train))
y_train = to_categorical(y_train, n_classes)
y_test = to_categorical(y_test, n_classes)
```

6）定义 CNN 体系结构：

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
input_shape=input_shape))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(n_classes, activation='softmax'))

opt = Adam()
model.compile(loss=losses.categorical_crossentropy,
              optimizer=opt,
metrics=['accuracy'])
```

7）设置网络超参数和回调函数：

```
batch_size = 128
n_epochs = 11

callbacks = [EarlyStopping(monitor='val_acc', patience=5)]
```

⊖ 直观来说就是有多少个状态就有多少比特码，而且只有一个比特码为 1，其他全为 0。——译者注

8）训练模型：

```
model.fit(X_train, y_train, batch_size=batch_size, epochs=n_epochs,
verbose=1, validation_split=0.2, callbacks=callbacks)
```

9）在测试集上显示结果：

```
score = model.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
# 测试损失值：0.0288668684929
# 测试精度：0.9927

# 模型预测
preds = model.predict(X_test)

n_examples = 10
plt.figure(figsize=(15, 15))
for i in range(n_examples):
    ax = plt.subplot(2, n_examples, i + 1)
    plt.imshow(X_test[i, :, :, 0], cmap='gray')
    plt.title("Label: {}\nPredicted:
    {}".format(np.argmax(y_test[i]), np.argmax(preds[i])))
    plt.axis('off')
plt.show()
```

输出如图 3.2 所示。

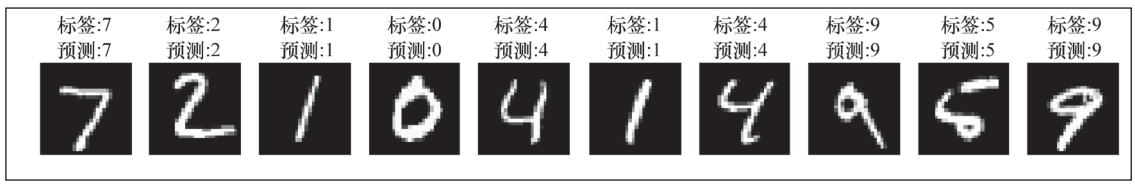


图 3.2　真实标签、预测标签和输入图像的 10 个例子

10）绘制错误分类的图像及其标签：

```
plt.    figure(figsize=(15, 15))

j=1
for i in range(len(y_test)):
    if(j==10):
        break
    label = np.argmax(y_test[i])
    pred = np.argmax(preds[i])
    if label != pred:
```

```
            ax = plt.subplot(2, n_examples, j + 1)
            plt.imshow(X_test[i, :, :, 0], cmap='gray')
            plt.title("Label: {}\nPredicted: {}".format(label, pred))
            plt.axis('off')
            j+=1
    plt.show()
```

正如在输出中看到的那样，一些错误分类的例子其实很难预测。例如，实际标签为 8，而预测为 9 的第七个示例，如图 3.3 所示。

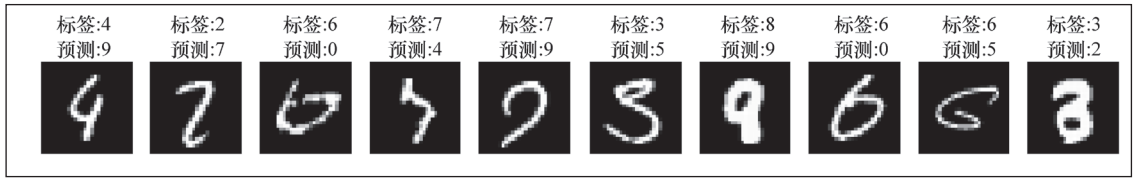


图 3.3　图像错误分类的 10 个例子

这个模型的结果已经相当不错了。在深度有限的情况下（三个卷积层和一个全连接层），大约经过 11 个周期，在测试集上获得了 99.30%的精度。该模型拥有 16 230 794 个可训练参数，并在 NVIDIA Tesla K80 GPU 上以 45s 左右的时间运行一个周期。据报道，一份最新的研究论文已经获得了 0.23 的错误率（其对应 99.77%的精度）。此模型使用了 35 个深度神经网络的集合，包括图像增强。通过稍后讨论的一些附加技术，可以进一步提高模型测试得分。

3.3　应用层合并技术

一种流行的 CNN 优化技术是层合并。层合并是一种用智能方式来减少可训练参数的方法。两个最常用的层合并技术是平均池化和最大池化。首先，对于指定的块大小，对输入进行平均并提取。对于后者，提取块中的最大值。这些层合并提供了平移不变性。换句话说，一个特征的确切位置是不太相关的。而且，通过减少可训练参数的数量，限制了网络的复杂性，以防止过拟合出现。另一个好处是它将大大减少训练和推理时间。

在下一个方案中，将在上一个方案中实现的 CNN 中添加最大层合并，同时增加卷积层中的滤波器数量。

如何去做…

1）导入所有必要的函数库：

```
import numpy as np

from keras.utils import np_utils
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Flatten
```

```
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import Adam

from keras.callbacks import EarlyStopping
```

2）加载 MNIST 数据集：

```
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

3）重整训练数据来表示灰度图像输入：

```
img_rows, img_cols = X_train[0].shape[0], X_train[0].shape[1]
X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)
```

4）标准化输入数据：

```
X_train = X_train.astype('float32')/255.
X_test = X_test.astype('float32')/255.
```

5）对标签进行独热编码：

```
n_classes = len(set(y_train))
y_train = np_utils.to_categorical(y_train, n_classes)
y_test = np_utils.to_categorical(y_test, n_classes)
```

6）定义 CNN 体系结构并输出网络体系结构：

```
model = Sequential()

model.add(Conv2D(64, kernel_size=(3, 3), activation='relu',
input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, kernel_size=(3, 3), activation='relu',
padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(256, kernel_size=(3, 3), activation='relu',
padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(n_classes, activation='softmax'))

model.compile(loss=losses.categorical_crossentropy,
optimizer='adam', metrics=['accuracy'])
model.summary()
```

7）设置网络超参数并定义回调函数：

```
batch_size = 128
n_epochs = 200

callbacks = [EarlyStopping(monitor='val_acc', patience=5)]
```

8）训练模型：

```
model.fit(X_train, y_train, batch_size=batch_size, epochs=n_epochs,
verbose=1, validation_split=0.2, callbacks=callbacks)
```

9）在测试集上显示结果：

```
score = model.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

# 测试损失值： 0.0197916838032
# 测试精度： 0.9955
```

在每个卷积层之后通过应用最大合并层，能够将错误率降低到 0.45（与之前的 0.70 的错误率相比，下降了 36%）。而且，通过应用最大层技术，可训练参数的数量明显减少到 665 994。可以得知，一个周期只需要大约 11s。

3.4 使用批量标准化进行优化

CNN 的另一个众所周知的优化技术是批量归一化。该技术在将其馈送到下一层之前对当前批次的输入进行标准化。因此，每个批次应在均值为 0、标准差为 1 附近激活，并且可以避免内部协变量的漂移。这使得每批数据的输入分布对网络的影响较小，因此该模型能够更好地泛化和更快地训练。

在下面的方案中，将向您展示如何将批量标准化应用于具有十个类的图像数据集（CIFAR-10）。首先，训练没有批量标准化的网络体系结构来演示性能的差异。

如何去做…

1）导入所有必要的函数库：

```
import numpy as np
from matplotlib import pyplot as plt

from keras.utils import np_utils
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.callbacks import EarlyStopping
from keras.layers import Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization
```

2）加载 CIFAR10 数据集：

```
from keras.datasets import cifar10
(X_train, y_train), (X_val, y_val) = cifar10.load_data()
```

3）标准化输入数据：

```
X_train = X_train.astype('float32')/255.
X_val = X_val.astype('float32')/255.
```

4）对标签进行独热编码：

```
n_classes = 10
y_train = np_utils.to_categorical(y_train, n_classes)
y_val = np_utils.to_categorical(y_val, n_classes)
```

5）定义 CNN 体系结构并输出网络体系结构：

```
input_shape = X_train[0].shape

model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), input_shape=input_shape,
padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(32, kernel_size=(3, 3), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

model.add(Dropout(0.25))

model.add(Conv2D(64, kernel_size=(3, 3), padding='same'))
model.add(Activation('relu'))
```

```
model.add(Conv2D(64, kernel_size=(3, 3), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

model.add(Dropout(0.25))

model.add(Conv2D(128, kernel_size=(3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(128, kernel_size=(3, 3), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dense(n_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
model.summary()
```

6）定义一个回调函数来防止过度配合：

```
callbacks = [EarlyStopping(monitor='val_acc', patience=5,
verbose=1)]
```

7）设置网络超参数：

```
batch_size = 256
n_epochs = 300
```

8）接下来，训练第一个模型：

```
history = model.fit(X_train, y_train, batch_size=batch_size,
epochs=n_epochs, verbose=1, validation_data=(X_val, y_val),
callbacks=callbacks)
```

9）现在，把批量标准化添加到网络体系结构中：

```
model_bn = Sequential()

model_bn.add(Conv2D(32, kernel_size=(3, 3),
input_shape=input_shape, padding='same'))
model_bn.add(Activation('relu'))
model_bn.add(BatchNormalization())
```

```
model_bn.add(Conv2D(32, kernel_size=(3, 3), padding='same'))
model_bn.add(Activation('relu'))
model_bn.add(BatchNormalization())
model_bn.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

model_bn.add(Dropout(0.25))

model_bn.add(Conv2D(64, kernel_size=(3, 3), padding='same'))
model_bn.add(Activation('relu'))
model_bn.add(BatchNormalization())
model_bn.add(Conv2D(64, kernel_size=(3, 3), padding='same'))
model_bn.add(Activation('relu'))
model_bn.add(BatchNormalization())
model_bn.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

model_bn.add(Dropout(0.25))

model_bn.add(Conv2D(128, kernel_size=(3, 3), padding='same'))
model_bn.add(Activation('relu'))
model_bn.add(BatchNormalization())
model_bn.add(Conv2D(128, kernel_size=(3, 3), padding='same'))
model_bn.add(Activation('relu'))
model_bn.add(BatchNormalization())
model_bn.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

model_bn.add(Dropout(0.25))

model_bn.add(Flatten())
model_bn.add(Dense(512, activation='relu'))
model_bn.add(BatchNormalization())
model_bn.add(Dropout(0.5))
model_bn.add(Dense(128, activation='relu'))
model_bn.add(Dense(n_classes, activation='softmax'))

model_bn.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
model_bn.summary()
```

10）现在准备对批量归一化的模型进行训练：

```
history_bn = model_bn.fit(X_train, y_train,
        batch_size=batch_size,
        epochs=n_epochs,
        verbose=1,
        validation_data=(X_val, y_val), callbacks=callbacks)
```

11）绘制两个模型的验证精度来比较性能：

```
val_acc_bn = history_bn.history['val_acc']
val_acc = history.history['val_acc']
plt.plot(range(len(val_acc)), val_acc, label='CNN model')
plt.plot(range(len(val_acc_bn)), val_acc_bn, label='CNN model with
BN')
plt.title('Validation accuracy on Cifar10 dataset')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.legend()
plt.show()
```

两个模型的结果如图 3.4 所示。

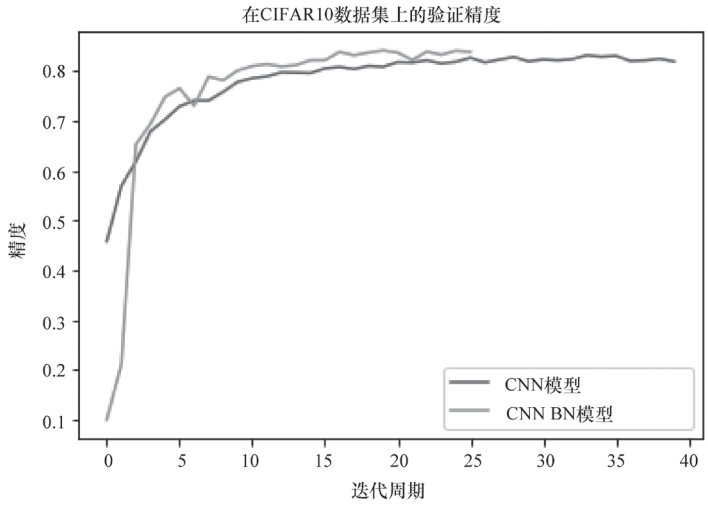


图 3.4　对比具有和不具有批量归一化方法模型的验证精度

正如所看到的，具有批量标准化的模型在多个周期之后在验证精度上领先，并且领先于没有批量标准化的模型，最高验证精度为 84.16%，而没有批量标准化的模型的验证精度为 83.19%。批量归一化的模型也收敛得很快（26 个周期对比 40 个周期）。然而，每个周期 25s，批量标准化的模型比没有批量标准化的模型（17s）要少一个周期。

3.5　理解填充和步长

到目前为止，已经使用网络的默认步长唯一。这表示模型将对每个轴上一个输入进行卷积（步长为 1）。但是，当数据集在像素级别上包含较少的粒度信息时，可以尝试用更大的值作为步长。通过增加步长，卷积层在每个轴上跳过更多的输入变量，因此减少了可训练参数的数量。这可以加速收敛，而不会有太大的性能损失。

另一个可以调整的参数是填充。填充定义了如何处理输入数据（例如图像）的边界。如果没有添加填充，则只包含边界像素（在图像的情况下）。因此，如果期望边界包含有价值的信息，可以尝试向数据添加填充。这增加了可以在数据上进行卷积时使用的虚拟数据

的边界。使用填充的好处是数据的维度在每个卷积层上保持相同，这意味着可以将更多的卷积层叠加在一起。在图 3.5 中，可以看到带有零填充的步长为 1 的示例，以及带有填充的步长为 2 的示例。

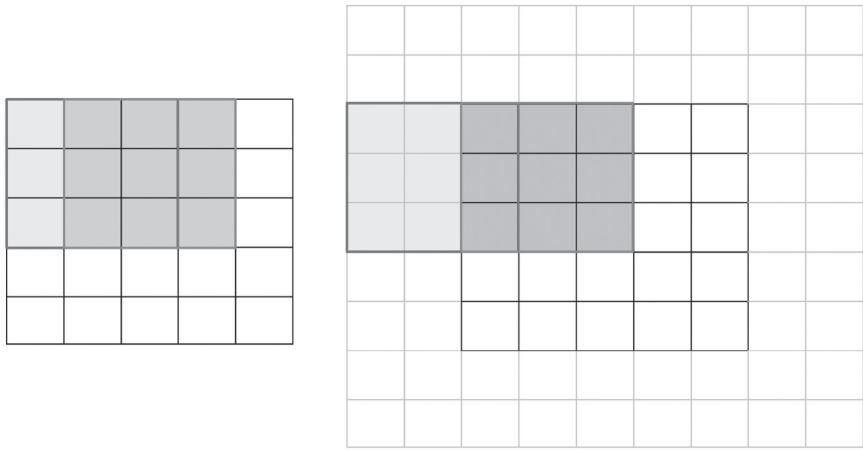


图 3.5　左图：5 × 5 的输入图像，带有步长为 1 和零填充；右图：5 × 5 的输入图像，步长为 2，填充相同

对于填充和步长选择值没有一般的规则。这在很大程度上取决于数据的大小和复杂性，以及所使用的潜在预处理技术。接下来，将尝试不同的步长和填充设置，并比较模型的结果。使用的数据集包含猫和狗的图像，要执行的任务是对动物进行分类。

如何去做…

1）导入所有必要的函数库，如下所示：

```
import glob
import numpy as np
import cv2
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split

from keras.utils import np_utils
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten,
Lambda
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
from keras.layers import Conv2D, MaxPooling2D

SEED = 2017
```

2）加载文件名并输出训练集大小：

```
# 指定数据目录、提取两类全部模式文件名
DATA_DIR = 'Data/PetImages/'
cats = glob.glob(DATA_DIR + "Cat/*.jpg")
dogs = glob.glob(DATA_DIR + "Dog/*.jpg")

print('#Cats: {}, #Dogs: {}'.format(len(cats), len(dogs)))
# #Cats: 12500, #Dogs: 12500
```

3）为了更好地理解数据集，绘制每个类中的三个例子，如图 3.6 所示。

```
n_examples = 3
plt.figure(figsize=(15, 15))
i = 1
for _ in range(n_examples):
    image_cat = cats[np.random.randint(len(cats))]
    img_cat = cv2.imread(image_cat)
    img_cat = cv2.cvtColor(img_cat, cv2.COLOR_BGR2RGB)
    plt.subplot(3, 2, i)
    _ = plt.imshow(img_cat)
    i += 1
    image_dog = dogs[np.random.randint(len(dogs))]
    img_dog = cv2.imread(image_dog)
    img_dog = cv2.cvtColor(img_dog, cv2.COLOR_BGR2RGB)
    plt.subplot(3, 2, i)
    i += 1
    _ = plt.imshow(img_dog)
plt.show()
```

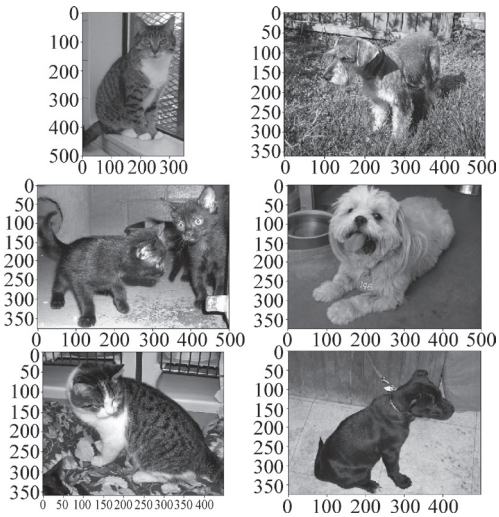


图 3.6　具有猫和狗标签的示例图像

4）将数据集划分成一个训练集和一个测试集，如下所示：

```
dogs_train, dogs_val, cats_train, cats_val = train_test_split(dogs,
cats, test_size=0.2, random_state=SEED)
```

5）训练集比较大，使用批生成器，这样就不必在内存中加载所有图像：

```
def batchgen(cats, dogs, batch_size, img_size=50):
    # 创建空 numpy 数组
    batch_images = np.zeros((batch_size, img_size, img_size, 3))
    batch_label = np.zeros(batch_size)

    # 创建批量样本生成器
    while 1:
        n = 0
        while n < batch_size:
            # 随机挑选一张狗或猫图像
            if np.random.randint(2) == 1:
                i = np.random.randint(len(dogs))
                img = cv2.imread(dogs[i])
                if img is None:
                    break
                img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
                # 图像具有不同的维度，需要全部归整到100×100
                img = cv2.resize(img, (img_size, img_size),
                 interpolation = cv2.INTER_AREA)
                y = 1

            else:
                img = cv2.imread(cats[i])
                if img is None:
                    break
                img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
                img = cv2.resize(img, (img_size, img_size),
                interpolation = cv2.INTER_AREA)
                y = 0
            batch_images[n] = img
            batch_label[n] = y
            n+=1
        yield batch_images, batch_label
```

6）接下来，定义一个函数，为步长和填充创建一个给定参数的模型：

```
def create_model(stride=1, padding='same', img_size=100):
    # 定义结构
    model = Sequential()
    model.add(Lambda(lambda x: (x / 255.) - 0.5,
    input_shape=(img_size, img_size, 3)))
    model.add(Conv2D(32, (3, 3), activation='relu',
    padding=padding, strides=stride))
    model.add(Conv2D(32, (3, 3), activation='relu',
    padding=padding, strides=stride))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.5))
    model.add(Conv2D(64, (3, 3), activation='relu',
    padding=padding, strides=stride))
    model.add(Conv2D(64, (3, 3), activation='relu',
    padding=padding, strides=stride))
    model.add(Dropout(0.5))

    model.add(Flatten())
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))
    opt = Adam(0.001)
        model.compile(loss='binary_crossentropy', optimizer=opt,
        metrics=['binary_accuracy'])
    return model
```

7）现在为每个设置定义一个模型，并且提取可训练参数的数量：

```
img_size = 100

models = []
for stride in [1, 2]:
    for padding in ['same', 'valid']:
        model = create_model(stride, padding, img_size)
        pars = model.count_params()
        models.append(dict({'setting': '{}_{}'.format(stride,
                    padding),
                'model': model,
                'parameters': pars
                }))
```

8）要输出模型的方案配置，可按下列步骤：

```
models[0]['model'].summary()
```

9）要使用早停技术，定义一个回调函数如下：

```
callbacks = [EarlyStopping(monitor='val_binary_accuracy',
patience=5)]
```

10）下一步，训练模型并存储结果：

```
batch_size = 512
n_epochs = 500
validation_steps = round((len(dogs_val)+len(cats_val))/batch_size)
steps_per_epoch =
round((len(dogs_train)+len(cats_train))/batch_size)

train_generator = batchgen(dogs_train, cats_train, batch_size,
img_size)
val_generator = batchgen(dogs_val, cats_val, batch_size, img_size)

history = []
for i in range(len(models)):
    print(models[i])
    history.append(
        models[i]['model'].
        fit_generator(train_generator,
        steps_per_epoch=steps_per_epoch, epochs=n_epochs,
        validation_data=val_generator,
        validation_steps=validation_steps,
        callbacks=callbacks
                              )
    )
```

11）可视化结果：

```
for i in range(len(models)):
    plt.plot(range(len(history[i].history['val_binary_accuracy'])),
history[i].history['val_binary_accuracy'],
label=models[i]['setting'])
    print('Max accuracy model {}: {}'.format(models[i]['setting'],
max(history[i].history['val_binary_accuracy'])))
plt.title('Accuracy on the validation set')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.legend()
plt.show()
```

通过使用步长 2，可训练参数的数量明显减少（当使用填充值 10 305 697~102 561 时）。对于这个数据集，它显示出当使用步长值大于 1 时没有性能损失。这是符合预期的，因为使用（调整后）$100 \times 100 \times 3$ 图像作为输入。跳过每个方向的像素不应该太多地影响性能。填充和步长不同设置的性能比较如图 3.7 所示。

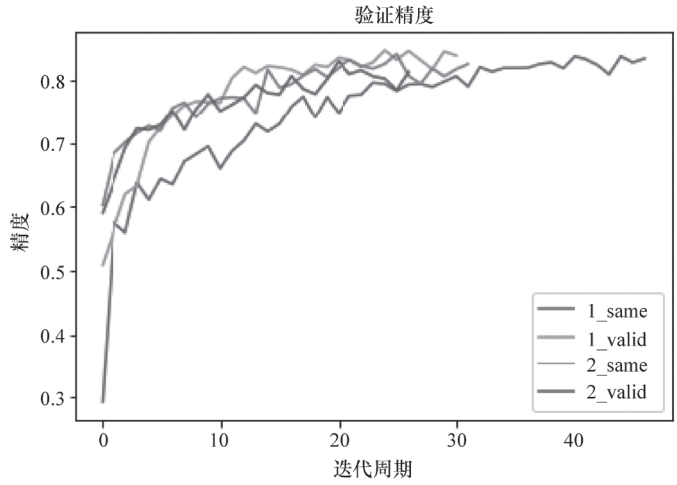


图 3.7　填充和步长不同设置的性能比较

3.6　试验不同类型的初始化

对 CNN 来说，权重和偏置的初始化可能是非常重要的。对于非常深的神经网络，一些初始化技术可能导致由最终层中梯度值的大小引起的梯度下降。在下面的方案中，将向读者展示如何针对一些知名网络使用不同的初始化，并显示性能的差异。通过选择正确的初始化，可以加速网络的收敛。在下面的方案中，首先用流行的高斯噪声初始化网络的权重和偏置，平均值等于零，标准偏置为 0.01。之后，使用 Xavier 初始化，还有常态和均匀分布方法，以及一些其他流行的初始化分布。

如何去做…

1）导入所有必要的函数库，如下：

```
import glob
import numpy as np
import cv2
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split

from keras.utils import np_utils
from keras import utils, losses, optimizers
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten,
Lambda
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras.layers import Conv2D, MaxPooling2D

SEED = 2017
```

2）开始加载文件名并输出训练集大小：

```
# 指定数据目录并提取两类全部模式文件名
DATA_DIR = 'Data/PetImages/'
cats = glob.glob(DATA_DIR + "Cat/*.jpg")
dogs = glob.glob(DATA_DIR + "Dog/*.jpg")

print('#Cats: {}, #Dogs: {}'.format(len(cats), len(dogs)))
# #Cats: 12500, #Dogs: 12500
```

3）接下来，将数据集划分成一个训练集和测试集：

```
dogs_train, dogs_val, cats_train, cats_val =
train_test_split(dogs, cats, test_size=0.2, random_state=SEED)
```

4）训练集比较大，使用批生成器，这样就不必在内存中加载所有图像：

```
def batchgen(cats, dogs, batch_size, img_size=50):
    # 创建空numpy数组
    batch_images = np.zeros((batch_size, img_size, img_size, 3))
    batch_label = np.zeros(batch_size)

    # 自定义批生成器
    while 1:
        n = 0
        while n < batch_size:
            # 随机挑选一张狗或猫图像
            if np.random.randint(2) == 1:
                i = np.random.randint(len(dogs))
                img = cv2.imread(dogs[i])
                if img is None:
                    break
                img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
                # 图像具有不同的维度，需要全部归整到100×100
                img = cv2.resize(img, (img_size, img_size),
                interpolation = cv2.INTER_AREA)
                y = 1

            else:
                i = np.random.randint(len(cats))
                img = cv2.imread(cats[i])
                if img is None:
                    break
                img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
                img = cv2.resize(img, (img_size, img_size),
```

```
                interpolation = cv2.INTER_AREA)
                y = 0
            batch_images[n] = img
            batch_label[n] = y
            n+=1
        yield batch_images, batch_label
```

5）接下来，定义一个函数来创建一个模型，给定步长和填充参数值：

```
def create_model(init_type='xavier', img_size=100):
    # 定义结构
    model = Sequential()
    model.add(Lambda(lambda x: (x / 255.) - 0.5,
    input_shape=(img_size, img_size, 3)))
    model.add(Conv2D(32, (3, 3), activation='relu',
    init=init_type))
    model.add(Conv2D(32, (3, 3), activation='relu',
    init=init_type))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.5))
    model.add(Conv2D(64, (3, 3), activation='relu',
    init=init_type))
    model.add(Conv2D(64, (3, 3), activation='relu',
    init=init_type))

    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))

    sgd = optimizers.Adam()
    model.compile(loss='binary_crossentropy', optimizer=sgd,
        metrics=['binary_accuracy'])

    return model
```

6）现在，为每个初始化类型定义一个模型：

```
models = []
for init_type in ['random_uniform', 'glorot_normal',
'glorot_uniform', 'lecun_uniform', 'he_uniform']:
    model = create_model(init_type, img_size=50)
    models.append(dict({'setting': '{}'.format(init_type),
            'model': model
            }))
```

7）尽早停止训练，以防止出现对训练数据的过拟合：

```
callbacks = [EarlyStopping(monitor='val_binary_accuracy',
 patience=3)]
```

8）下一步，训练模型并保存结果：

```
batch_size = 512
n_epochs = 500
steps_per_epoch = 100
validation_steps = round((len(dogs_val)+len(cats_val))/batch_size)

train_generator = batchgen(dogs_train, cats_train, batch_size)
val_generator = batchgen(dogs_val, cats_val, batch_size)

history = []
for i in range(len(models)):
    print(models[i])
    history.append(
        models[i]['model'].
        fit_generator(train_generator,
        steps_per_epoch=steps_per_epoch, epochs=n_epochs,
        validation_data=val_generator,
        validation_steps=validation_steps, callbacks=callbacks)
    )
```

9）可视化结果（见图 3.8）：

```
for i in range(len(models)):
    plt.plot(range(len(history[i].history['val_binary_accuracy'])),
history[i].history['val_binary_accuracy'],
label=models[i]['setting'])
    print('Max accuracy model {}: {}'.format(models[i]['setting'],
max(history[i].history['val_binary_accuracy'])))
plt.title('Accuracy on the validation set')
                    plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.legend()
plt.show()
```

正如读者所看到的，权重的初始化类型会对结果产生影响。测试所有不同的初始化方法可能需要大量的计算，所以往往不理想。对于 CNN 来说，每个框架的默认设置往往做得很好。对于二维卷积层，Glorot 均匀分布权重（也称为 Xavier 均匀初始化）通常用作默认值。

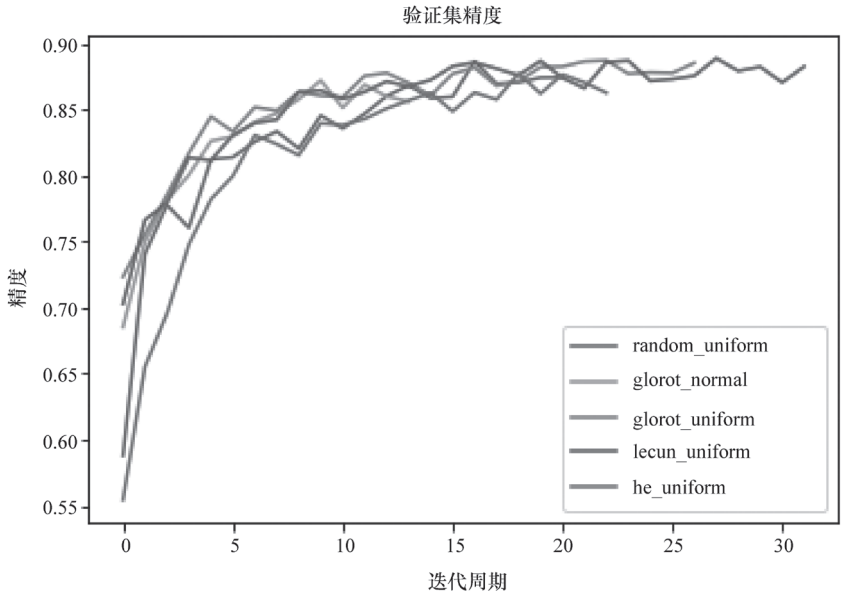


图 3.8　不同权重初始化设置的性能比较

3.7　实现卷积自动编码器

在前面如何为街景门牌号码数据集实现一个自动编码器，取得了不错的成效，但是输出肯定会需要改善。在下面的方案中，将展示一个卷积自动编码器如何产生更好的输出。

如何去做…

1）从导入函数库开始，如下所示：

```
import numpy as np
import scipy.io

from matplotlib import pyplot as plt
from keras.utils import np_utils
from keras.models import Sequential, Input, Model
from keras.layers.core import Dense, Dropout, Activation, Reshape,
Flatten, Lambda
from keras.layers import Conv2D, MaxPooling2D, UpSampling2D
from keras.callbacks import EarlyStopping
```

2）接下来，加载数据集并提取将在这个方案中使用的数据：

```
mat = scipy.io.loadmat('Data/train_32x32.mat')
mat = mat['X']
b, h, d, n = mat.shape
```

3）在将数据提供给网络之前，对数据进行预处理：

```
#转换全部 RGB 图像到灰度值
img_gray = np.zeros(shape =(n, b, h, 1))

def rgb2gray(rgb):
 return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

for i in range(n):
 #转换到灰度值
 img = rgb2gray(mat[:,:,:,i])
 img = img.reshape(1, 32, 32, 1)
 img_gray[i,:] = img

# 标准化输入
img_gray = img_gray/255.
```

4）现在为卷积自动编码器定义网络架构：

```
img_size = Input(shape=(b, h, 1))

x = Conv2D(16, (3, 3), activation='relu', padding='same')(img_size)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid',
padding='same')(x)

autoencoder = Model(img_size, decoded)
autoencoder.compile(optimizer='rmsprop',
loss='binary_crossentropy')#, metrics=['binary_accuracy'])

# 输出网络概要
autoencoder.summary()
```

5）接下来，定义早停技术的回调函数：

```
callbacks = EarlyStopping(monitor='val_loss', patience=5)
```

6）定义超参数并开始训练网络：

```
n_epochs = 1000
batch_size = 128

autoencoder.fit(
 img_gray, img_gray,
 epochs=n_epochs,
 batch_size=batch_size,
 shuffle=True, validation_split=0.2
    callbacks=callbacks
 )
```

7）存储解码图像如下：

```
pred = autoencoder.predict(img_gray)
```

8）现在，输出一些原始图像和对应的解码图像：

```
n = 5
plt.figure(figsize=(15, 5))
for i in range(n):
 # 显示原值
 ax = plt.subplot(2, n, i + 1)
 plt.imshow(mat[i].reshape(32, 32), cmap='gray')
ax = plt.subplot(2, n, i + 1 + n)
 plt.imshow(pred[i].reshape(32, 32), cmap='gray')
plt.show()
```

训练好的模型输出图像如图 3.9 所示。

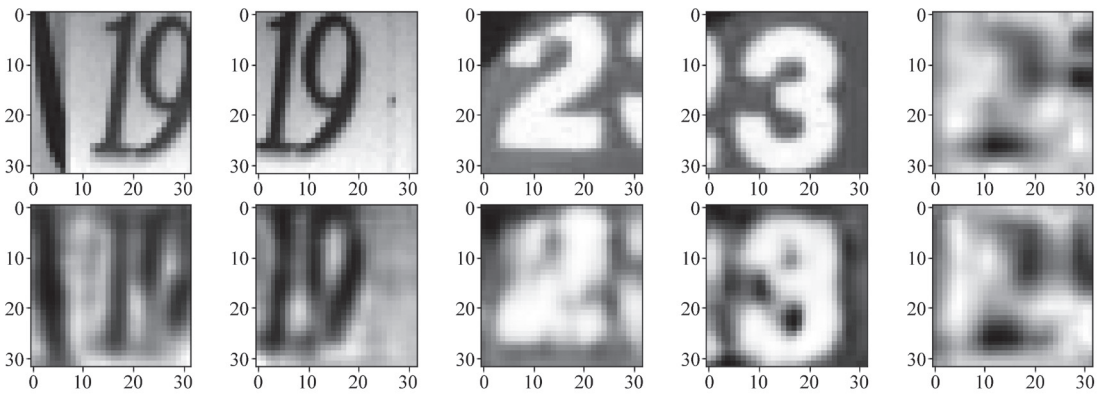


图 3.9　使用卷积自动编码器创建的原始和解码图像的示例

与第 2 章的解码图像相比，输出效果明显提高。在第 6 章　生成对抗网络　中，将介绍一种不同类型的解码 - 编码器网络结构。

3.8　将一维 CNN 应用于文本

到目前为止，已经将 CNN 应用于图像数据。如简介所述，CNN 也可以应用于其他类型的输入数据。在下面的方案中，将展示如何将 CNN 应用于文本数据。更具体地说，将使用 CNN 结构对文本进行分类。与二维图像不同，文本具有一维输入数据。因此，将在下一个方案中使用一维卷积图层。Keras 框架使得预处理输入数据变得非常简单。

如何去做…

1）开始导入函数库，如下所示：

```
import numpy as np

from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalMaxPooling1D
from keras.callbacks import EarlyStopping
from keras.datasets import imdb
```

2）使用 Keras 的 IMDB 数据集，用下面的代码加载数据：

```
n_words = 1000
(X_train, y_train), (X_test, y_test) =
imdb.load_data(num_words=n_words)
print('Train seq: {}'.format(len(X_train)))
print('Test seq: {}'.format(len(X_train)))
```

3）输出显示一个训练数据和测试数据的示例：

```
print('Train example: \n{}'.format(X_train[0]))
print('\nTest example: \n{}'.format(X_test[0]))

# 注意：数据已经预处理（字词映射到矢量）
```

4）通过填充序列，为网络准备输入：

```
# 序列填充长度 max_len
max_len = 200
X_train = sequence.pad_sequences(X_train, maxlen=max_len)
X_test = sequence.pad_sequences(X_test, maxlen=max_len)
```

5）现在准备好定义网络架构：

```
# 定义网络架构并编译
model = Sequential()
model.add(Embedding(n_words, 50, input_length=max_len))
model.add(Dropout(0.5))
model.add(Conv1D(128, 3, padding='valid', activation='relu',
strides=1,))
model.add(GlobalMaxPooling1D())
model.add(Dense(250, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
model.summary()
```

6）定义一个回调函数来防止过拟合训练数据：

```
callbacks = [EarlyStopping(monitor='val_acc', patience=3)]
```

7）定义超参数并开始训练网络：

```
batch_size = 64
n_epochs = 100

model.fit(X_train, y_train, batch_size=batch_size, epochs=n_epochs,
validation_split=0.2, callbacks=callbacks)
```

8）最后，检查训练好的网络在测试集上的性能：

```
print('\nAccuracy on test set: {}'.format(model.evaluate(X_test,
y_test)[1]))

# 测试集精度： 0.873
```

这里使用的简单模型已经能够准确地对文本所表达的情感进行分类，在测试集上的精度为 87.3%。在第 4 章中，将向读者展示如何通过将 CNN 与 RNN 相结合来进一步提高文本分类器的性能。

第 4 章
递归神经网络

本章重点介绍**递归神经网络**（RNN），包括以下内容：

- 实现一个简单的 RNN；
- 添加长短期记忆（LSTM）；
- 使用门控递归神经元（GRU）；
- 实现双向 RNN；
- 字符级文本生成。

4.1 简介

在本章中，将介绍一种新型的神经网络。在此之前，只考虑信息在其中单向流动的神经网络。接下来，将介绍 RNN。在这些网络中，对于序列中的每个元素，数据的处理方式是相同的，并且输出取决于先前的计算结果。这种结构在许多应用程序中已被证明是非常强大的，如**自然语言处理**（NLP）和时间序列预测。本章将介绍重要的构建模块，它彻底改变了如何处理神经网络中的时间序列或其他形式序列数据的方式。

一个简单 RNN 神经元，如图 4.1 所示。

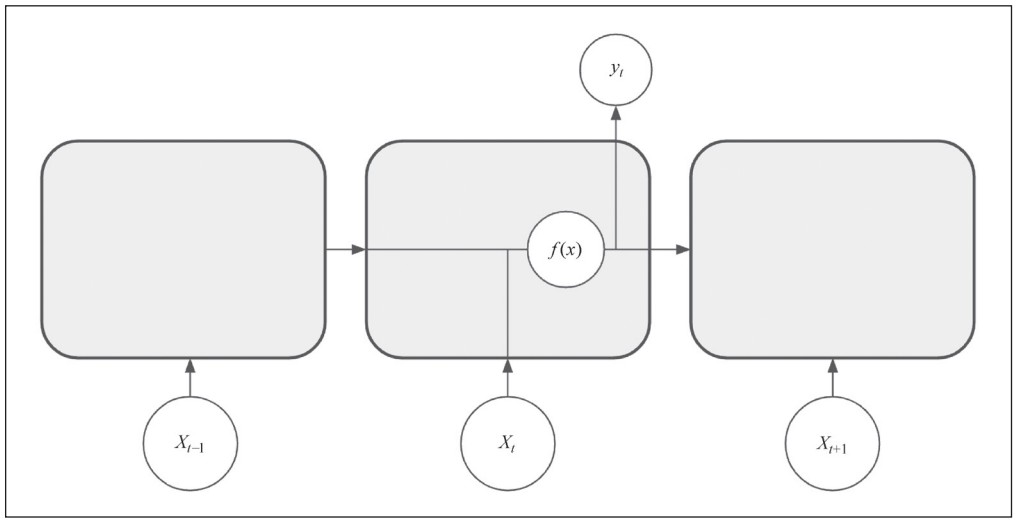


图 4.1　在 RNN 神经元中信息流动的示例

正如在图 4.1 中所看到的，RNN 的输出不仅依赖于当前的输入 X_t，还依赖于过去的输入（X_{t-1}）。本质上，这给网络提供了一种记忆机理。

有多种类型的 RNN，其中输入和输出维度可能不同。RNN 可以有一个输入变量和多个输出变量（1 到 n，n>1），多个输入变量和一个输出变量（n 到 1，n>1），或多个输入和输出变量（n 到 m，其中 n，m>1）。在本书中，将涵盖所有类型的变量。

4.2 实现一个简单的 RNN

首先实现一种简单的 RNN 形式，以便理解 RNN 的基本思想。在本例中，将为 RNN 提供四个二进制变量。这些变量表示了某一天的天气类型。例如，[1，0，0，0] 表示晴天，[1，0，1，0] 表示晴天和多风。目标值是表示当天降雨量百分比的双精度值。对于这个问题来说，某一天的降雨量也取决于前一天的降雨量数值。这使得这个问题非常适合于 4 到 1 的 RNN 模型。

如何去做…

1）在这个基本示例中，将使用 NumPy 实现简单的 RNN：

```
import numpy as np
```

2）从创建将要使用的虚拟数据集开始：

```
X = []
X.append([1,0,0,0])
X.append([0,1,0,0])
X.append([0,0,1,0])
X.append([0,0,0,1])
X.append([0,0,0,1])
X.append([1,0,0,0])
X.append([0,1,0,0])
X.append([0,0,1,0])
X.append([0,0,0,1])

y = [0.20, 0.30, 0.40, 0.50, 0.05, 0.10, 0.20,
0.30, 0.40]
```

3）对于回归问题，将使用 Sigmoid 激活函数：

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_der(x):
    return 1.0 - x**2
```

4）接下来，初始化网络层和权重：

```
layers = []
# 4 个输入变量、16 个隐层神经元和 1 个输出变量
n_units = (4, 16, 1)
n_layers = len(n_units)

layers.append(np.ones(n_units[0]+1+n_units[1]))
for i in range(1, n_layers):
    layers.append(np.ones(n_units[i]))

weights = []
for i in range(n_layers-1):
    weights.append(np.zeros((layers[i].size,
    layers[i+1].size)))

weights_delta = [0,]*len(weights)
```

5）现在已经准备好定义正向传递的函数：

```
def forwards(data):
    layers[0][:n_units[0]] = data
    layers[0][n_units[0]:-1] = layers[1]

    # 前向数据传播
    for i in range(1, n_layers):
        layers[i][...] = sigmoid(np.dot(layers[i-1],
        weights[i-1]))

    return layers[-1]
```

6）在反向传递中，将确定误差函数并更新权重：

```
def backwards(target, learning_rate=0.1, momentum=0.1):
    deltas = []
    error = target - layers[-1]
    delta = error * sigmoid_der(layers[-1])
    deltas.append(delta)

    # 确定隐层误差
    for i in range(n_layers-2, 0, -1):
        delta = np.dot(deltas[0], weights[i].T) *
sigmoid_der(layers[i])
        deltas.insert(0, delta)
```

```
        # 更新权重
        for i in range(len(weights)):
            layer = np.atleast_2d(layers[i])
            delta = np.atleast_2d(deltas[i])
            weights_delta_temp = np.dot(layer.T, delta)
            weights[i] += learning_rate*weights_delta_temp +
            momentum*weights_delta[i]
            weights_delta[i] = weights_delta_temp

        return (error**2).sum()
```

7）现在可以训练所建立的简单 RNN 了：

```
n_epochs = 10000
for i in range(n_epochs):
    loss = 0
    for j in range(len(X)):
        forwards(X[j])
        backwards(y[j])
        loss += (y[j]-forwards(X[j]))**2
    if i%1000 == 0: print('epoch {} - loss:
{:04.4f}'.format(i, loss[0]))
```

8）最后，检查一下结果：

```
for i in range(len(X)):
    pred = forwards(X[i])
    loss = (y[i]-pred)**2
    print('X: {}; y: {:04.2f}; pred:
    {:04.2f}'.format(X[i], y[i], pred[0]))
```

以上内容的目的是让读者熟悉 RNN 和内部结构。此示例已经表明，预测值不仅依赖于当前输入数据，而且还依赖于序列中的先前值。

4.3 添加 LSTM

简单 RNN 的一个限制是，它只考虑当前输入变量周围的直接输入变量。在许多应用中，特别是语言，人们需要更大程度地理解句子的上下文。这就是为什么 LSTM（长短期记忆）在将深度学习应用于文本等非结构化数据类型方面发挥了重要作用。一个 LSTM 神经元有一个输入门、一个遗忘门和一个输出门，如图 4.2 所示。

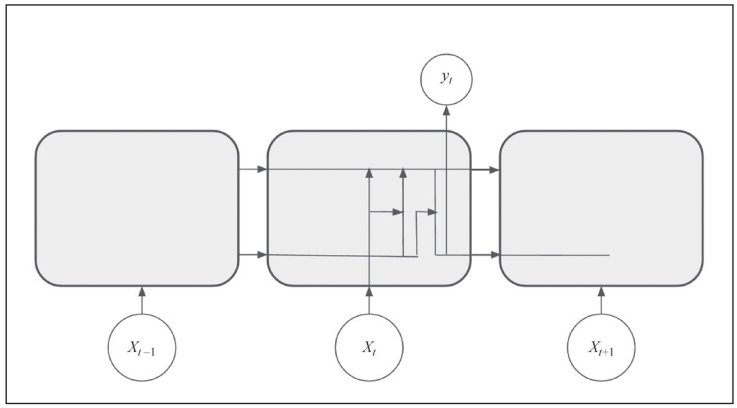


图 4.2　在 LSTM 神经元中信息流动的示例

下面将使用 Keras 框架对来自 IMDB 数据集的评论进行分类。

如何去做…

1）从导入函数库开始，如下所示：

```
import numpy as np

from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import LSTM

from keras.datasets import imdb
```

2）使用来自 Keras 的 IMDB 数据集，用以下代码加载数据：

```
n_words = 1000
(X_train, y_train), (X_test, y_test) =
imdb.load_data(num_words=n_words)
print('Train seq: {}'.format(len(X_train)))
print('Test seq: {}'.format(len(X_train)))
```

3）呈现一个训练和测试数据示例：

```
print('Train example: \n{}'.format(X_train[0]))
print('\nTest example: \n{}'.format(X_test[0]))

# 注意：数据已经预处理
# (字词映射到矢量 )
```

4）通过填充序列，为网络准备输入数据：

```
# 填充序列长度 max_len
max_len = 200
X_train = sequence.pad_sequences(X_train, maxlen=max_len)
X_test = sequence.pad_sequences(X_test, maxlen=max_len)
```

5）现在已经准备好定义网络架构：

```
# 定义网络结构并编译
model = Sequential()
model.add(Embedding(n_words, 50, input_length=max_len))
model.add(Dropout(0.2))
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(250, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
optimizer='adam', metrics=['accuracy'])
model.summary()
```

6）定义超参数并开始训练网络：

```
batch_size = 64
n_epochs = 10

model.fit(X_train, y_train, batch_size=batch_size,
epochs=n_epochs)
```

7）最后，可以在测试集上检查训练后网络的性能：

```
print('\nAccuracy on test set: {}'.
format(model.evaluate(X_test, y_test)[1])

# 测试集精度：0.82884
```

对文本分类，模型的测试精度低于一维 CNN 网络的测试精度。在第 8 章　自然语言处理　中，将展示如何将 CNN 和 RNN 结合起来创建一个更强大的文本分类网络。

4.4　使用 GRU

在 RNN 中经常使用的另一种类型的神经元类型是 GRU，这些神经元实际上比 LSTM 神经元简单，因为它们只有两个门：更新和复位。更新门决定内存，复位门将内存与当前输入相结合。数据流如图 4.3 所示。

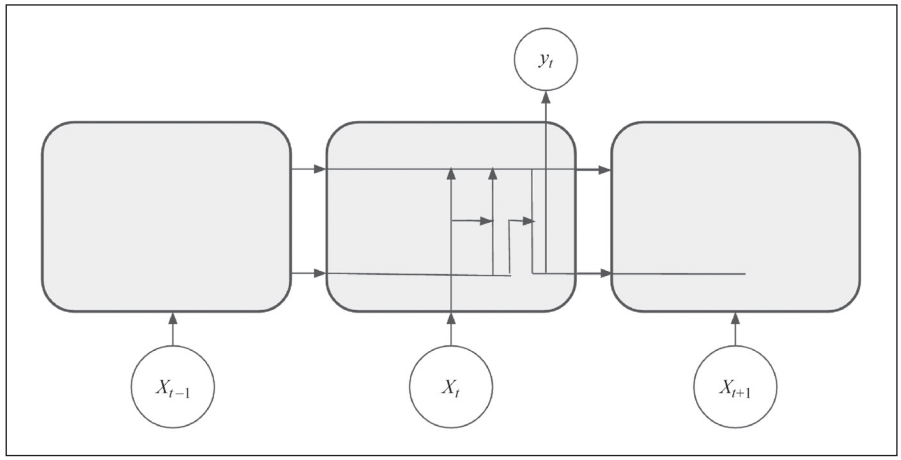


图 4.3　在 GRU 中信息流动的示例

下面将展示如何将 GRU 合并到 RNN 架构中，以便将其与 Keras 用于文本分类。

如何去做…

1）从导入函数库开始，如下：

```
import numpy as np
import pandas as pd

from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import GRU
from keras.callbacks import EarlyStopping

from keras.datasets import imdbimport numpy as np
import pandas as pd

from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import GRU
from keras.callbacks import EarlyStopping

from keras.datasets import imdb
```

2）使用 IMDB 数据集来进行文本情感的分类，用以下代码加载数据：

```
n_words = 1000
(X_train, y_train), (X_test, y_test) =
imdb.load_data(num_words=n_words)
print('Train seq: {}'.format(len(X_train)))
print('Test seq: {}'.format(len(X_train)))
```

3）通过填充序列，为网络准备输入数据：

```
# 填充序列长度 max_len
max_len = 200
X_train = sequence.pad_sequences(X_train, maxlen=max_len)
X_test = sequence.pad_sequences(X_test, maxlen=max_len)
```

4）已经准备好定义网络架构：

```
# 定义网络架构并编译
model = Sequential()
model.add(Embedding(n_words, 50, input_length=max_len))
model.add(Dropout(0.2))
model.add(GRU(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(250, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
optimizer='adam', metrics=['accuracy'])
model.summary()
```

5）使用早停方法来防止过拟合：

```
callbacks = [EarlyStopping(monitor='val_acc',
patience=3)]
```

6）定义超参数并开始训练网络：

```
batch_size = 512
n_epochs = 100

model.fit(X_train, y_train, batch_size=batch_size,
epochs=n_epochs, validation_split=0.2, callbacks=callbacks)
```

7）最后，可以在测试集上检查训练网络的性能：

```
print('\nAccuracy on test set: {}'.
format(model.evaluate(X_test, y_test)[1])

# 测试集精度：0.83004
```

4.5 实现双向 RNN

到目前为止，只考虑了网络中单向流动的信息。然而，有时在两个方向的网络中运行数据是有益的，称这种网络为双向 RNN。在下面的示例中，将实现与前面所实现功能相同的 LSTM 网络，但本次将使用双向 RNN 来分类情感。

如何去做…

1）从导入函数库开始，如下：

```
import numpy as np

from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Dropout,
Activation, Embedding, LSTM, Bidirectional
from keras.callbacks import EarlyStopping
from keras.datasets import imdb
```

2）使用来自 Keras 的 IMDB 数据集，用以下代码加载数据：

```
n_words = 1000
(X_train, y_train), (X_test, y_test) =
imdb.load_data(num_words=n_words)
print('Train seq: {}'.format(len(X_train)))
print('Test seq: {}'.format(len(X_train)))
```

3）呈现一个训练和测试数据的输出示例：

```
print('Train example: \n{}'.format(X_train[0]))
print('\nTest example: \n{}'.format(X_test[0]))

# 注意：数据已经预处理（字词映射到矢量）
```

4）通过填充序列，为网络准备输入数据：

```
# 填充序列长度 max_len
max_len = 200
X_train = sequence.pad_sequences
(X_train, maxlen=max_len)
X_test = sequence.pad_sequences
(X_test, maxlen=max_len)
```

5）现在已经准备好定义网络架构：

```
# 定义网络架构并编译
model = Sequential()
model.add(Embedding(n_words, 50, input_length=max_len))
model.add(Dropout(0.2))
model.add(Bidirectional(LSTM(100, dropout=0.2,
recurrent_dropout=0.2)))
model.add(Dense(250, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
optimizer='adam', metrics=['accuracy'])
model.summary()
```

6）为了防止过拟合，将使用早停方法：

```
callbacks = [EarlyStopping(monitor='val_acc', patience=3)]
```

7）定义超参数并开始训练网络：

```
batch_size = 1024
n_epochs = 100

model.fit(X_train, y_train, batch_size=batch_size, epochs=n_epochs,
validation_split=0.2, callbacks=callbacks)
```

8）最后，可以在测试集上检查训练网络的性能：

```
print('Accuracy on test set: {}'.format(model.evaluate(X_test,
y_test, batch_size=batch_size)[1]))

# 测试集精度：0.8391600004386902
```

正如所看到的，网络可以在两个方向上从解析数据中检索数据，这使得测试精度可些微提高到 83.91%。

4.6 字符级文本生成

RNN 不仅能够对文本进行解析和分类，还可以用来生成文本。在最简单的形式中，文本是以字符级别生成的。更具体地说，文本是逐个字符生成的字符。在生成文本之前，需要训练一个完整句子的解码器。通过在解码器中包含一个 GRU 层，模型不仅依赖于先前的输入，而且根据周围的上下文尝试预测下一个字符。在下面的方案中，将演示如何使用 PyTorch 实现一个字符级的文本生成器。

如何去做…

1）从导入函数库开始，如下：

```
import unidecode
import string
import random
import math

import torch
import torch.nn as nn
from torch.autograd import Variable
```

2）作为输入和输出，可以使用任意字符：

```
all_characters = string.printable
input_size = len(all_characters)
output_size = input_size
print(input_size)
```

3）将使用奥巴马演讲数据集：

```
filename = 'Data/obama.txt'
data = unidecode.unidecode(open(filename).read())
len_data = len(data)
```

4）在继续之前，需要定义超参数：

```
n_steps = 2000
batch_size = 512
hidden_size = 100
n_layers = 2
learning_rate = 0.01
len_text = 200
print_every = 50
```

5）定义一个函数，它将字符转换为张量：

```
def char_to_tensor(string):
    tensor = torch.zeros(len(string)).long()
    for c in range(len(string)):
        try:
            tensor[c] = all_characters.
            index(string[c])
        except:
            continue
    return tensor
```

6）接下来，定义一个批处理生成器：

```
def batch_gen(length_text, batch_size):
    X = torch.LongTensor(batch_size, length_text)
    y = torch.LongTensor(batch_size, length_text)
    for i in range(batch_size):
        start_index = random.randint
        (0, len_data - length_text)
        end_index = start_index + length_text + 1
        text = data[start_index:end_index]
        X[i] = char_to_tensor(text[:-1])
        y[i] = char_to_tensor(text[1:])
    X = Variable(X)
    y = Variable(y)
    X = X.cuda()
    y = y.cuda()
    return X, y
```

7）现在已经准备好定义网络架构：

```
class create_model(nn.Module):
    def __init__(self, input_size, hidden_size,
    output_size, n_layers=1):
        super(create_model, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers

        self.encoder = nn.Embedding(input_size, hidden_size)
        self.rnn = nn.GRU(hidden_size, hidden_size, n_layers)
        self.decoder = nn.Linear(hidden_size, output_size)

    def forward(self, input, hidden):
        batch_size = input.size(0)
        encoded = self.encoder(input)
        output, hidden = self.rnn(encoded.view(1,
        batch_size, -1), hidden)
        output = self.decoder(output.view(batch_size, -1))
        return output, hidden
```

```
    def init_hidden(self, batch_size):
        return Variable(torch.zeros
       (self.n_layers, batch_size, self.hidden_size))
```

8）继续创建模型并定义优化器和损失函数，如下：

```
decoder_model = create_model(
    input_size,
    hidden_size,
    output_size,
    n_layers=n_layers,
)

opt = torch.optim.Adam(decoder_model.parameters(),
lr=learning_rate)
loss = nn.CrossEntropyLoss()
decoder_model.cuda()
```

9）再创建一个函数用于在训练过程中生成文本：

```
def generate_text(decoder, start='The', predict_len=100):
    hidden = decoder.init_hidden(1).cuda()
    prime_input =
Variable(char_to_tensor(start).unsqueeze(0)).cuda()
    predicted = start

    for p in range(len(start) - 1):
        _, hidden = decoder(prime_input[:, p], hidden)
    x = prime_input[:,-1]
    for p in range(predict_len):
        output, hidden = decoder(x, hidden)
        output_dist = output.data.view(-1).div(0.8).exp()
        # 添加一些随机性
        top_i = torch.multinomial(output_dist, 1)[0]
        predicted_char = all_characters[top_i]
        predicted += predicted_char
        x = Variable(char_to_tensor(predicted_char).
       unsqueeze(0)).cuda()
    return predicted
```

10）最后，开始网络训练：

```
loss_avg = 0
for i in range(n_steps):
    X, y = batch_gen(len_text, batch_size)
    hidden = decoder_model.init_hidden(batch_size).cuda()
    decoder_model.zero_grad()
    loss_total = 0

    for c in range(len_text):
        output, hidden = decoder_model(X[:,c], hidden)
        loss_total += loss(output.view(batch_size, -1),
        y[:,c])

    loss_total.backward()
    opt.step()
    loss_value = loss_total.data[0] / len_text
    loss_avg += loss_value

    if i % print_every == 0:
        print('Epoch {}: loss {}'.format(i, loss_avg))
        print(generate_text(decoder_model,
       'The', 100), '\n')
```

在本章中，介绍了 RNN 和一些在 RNN 中使用的最流行的技术，还展示了一些 RNN 的应用。在第 5 章中，将向读者展示如何将这些网络应用于更高级别的问题。

第 5 章
强化学习

在本章中，将重点讨论强化学习，涵盖以下主题：

- 实现策略梯度；
- 实现深度 Q 学习算法。

5.1 简介

到目前为止，已经讨论了监督学习和非监督学习技术。机器学习的第三个支柱是**强化学习**（RL）。在强化学习中，任务既不是监督的，也不是非监督的。具体地说，在 RL 中，智能体在接收观测数据时有一个最终目标，但在每一步都不会收到环境的反馈。相反，只有在一定数量的步骤之后，智能体才会得到积极或消极的奖励。这很有趣，因为有人可能会说，对于某些任务，这和人类学习的方式是一样的。是什么使得这类问题比正常的监督学习问题更加复杂呢？就是人们没有明确地指出，在前一个步骤中的哪一个动作引起了对期望值的奖励，这就是所谓的**信用分配问题**。

RL 是当今的一个热门话题，因为在这个领域已经取得了很大的进步。此外，这些算法所解决的问题也很有趣，而且在视觉上很有吸引力。例如，使用 RL，计算机可以自学游戏，而不需要显式地指定它应该执行的步骤。

强化学习是一项计算昂贵的任务。这意味着，在获得有价值的结果之前，通常需要长时间运行算法。

5.2 实现策略梯度

在强化学习中，不能直接在网络中反向传播误差，因为没有为每一步设置一个真值。现在只感知到外界反馈，这也就是为什么需要**策略梯度**把奖励传播回网络。确定最佳操作的规则称为**策略**。学习这些策略的网络称为**策略网络**，这可以是任何类型的网络，例如一个简单的双层 FNN 或 CNN。环境越复杂，就越能从复杂的网络中获益。在使用策略梯度时，将绘制策略网络的输出分布。因为奖励并不总是直接可用，所以认为这个行为是正确的。之后，将**折扣奖励**作为标量，并将其反向传播到网络权重。

在下面的内容中，将通过在 TensorFlow 中实施一个策略梯度，教给一个智能体在 OpenAI 中玩游戏 Pong。Pong 是一个很好玩的游戏，因为它很简单（两个动作，即向上或

向下，如果排除掉什么都不做选项），这很容易形象化，并且有一个直接的奖励（如果智能体赢了游戏 +1 分，如果智能体输掉了游戏 –1 分）。在这个版本的 Pong 游戏中，要玩到其中一个玩家赢了 21 盘（称为一**局**）。人们希望智能体以最高的分数赢得一场比赛：21：0。

准备

在开始实施方案之前，确保安装了 OpenAI Gym 环境。按照网站上的说明进行操作：https：//gym.openai.com/docs/。当在服务器上运行 Gym 时，需要连接一个虚拟显示器。

如何去做…

1）首先导入必要的函数库，如下：

```
import numpy as np
import gym
import tensorflow as tf
import matplotlib.pyplot as plt
```

2）建立一个 Pong 环境，并绘制出一个框架（见图 5.1）：

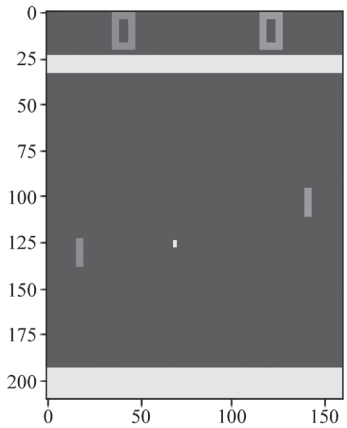


图 5.1　Pong 游戏的 OpenAI 框架

```
env = gym.make("Pong-v0") # environment info
observation = env.reset()

for i in range(22):
    # 20 帧后发球
    if i > 20:
        plt.imshow(observation)
        plt.show()
    # 得到下一个观察
    observation, _, _, _ = env.step(1)
```

3）在实现算法之前，需要一个函数预处理输入数据：

```
def preprocess_frame(frame):
    # 移去图像顶部和某些背景
    frame = frame[35:195, 10:150]
    # 图像帧灰度化并缩小1/2
    frame = frame[::2, ::2, 0]
    # 设置背景值为0
    frame[frame == 144] = 0
    frame[frame == 109] = 0
    # 设置球及拍数为1
    frame[frame != 0] = 1
    return frame.astype(np.float).ravel()
```

4）看看预处理数据是什么样子（见图 5.2）：

```
obs_preprocessed = preprocess_frame(observation).
reshape(80, 70)
plt.imshow(obs_preprocessed, cmap='gray')
plt.show()
```

5）为了从游戏中提取时间信息，使用了两个连续帧之间的差异（见图 5.3）：

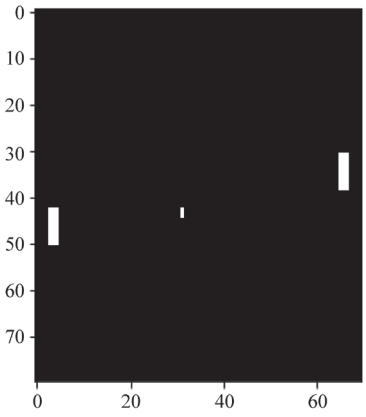


图 5.2　Pong 预处理框架（重塑到二维空间）

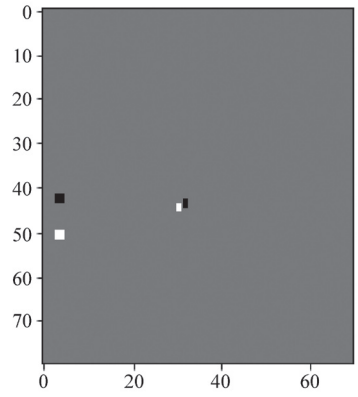


图 5.3　两个连续的 Pong 之间的区别（重塑到二维空间）

```
observation_next, _, _, _ = env.step(1)
diff = preprocess_frame(observation_next) -
preprocess_frame(observation)
plt.imshow(diff.reshape(80, 70), cmap='gray')
plt.show()
```

6）现在开始实施一个三层网络模型。从定义权重开始：

```
input_dim = 80*70
hidden_L1 = 400
hidden_L2 = 200
actions = [1, 2, 3]
n_actions = len(actions)
model = {}
with tf.variable_scope('L1',reuse=False):
    init_W1 = tf.truncated_normal_initializer
    (mean=0, stddev=1./np.sqrt(input_dim),
    dtype=tf.float32)
    model['W1'] = tf.get_variable("W1",
   [input_dim, hidden_L1], initializer=init_W1)

with tf.variable_scope('L2',reuse=False):
    init_W2 = tf.truncated_normal_initializer
    (mean=0, stddev=1./np.sqrt(hidden_L1),
    dtype=tf.float32)
    model['W2'] = tf.get_variable("W2",
   [hidden_L1, n_actions], initializer=init_W2)
```

7）接下来，为策略定义一个函数：

```
def policy_forward(x):
    x = tf.matmul(x, model['W1'])
    x = tf.nn.relu(x)
    x = tf.matmul(x, model['W2'])
    p = tf.nn.softmax(x)
    return p
```

8）对于此算法，需要定义折扣奖励的函数：

```
def discounted_rewards(reward, gamma):
    discounted_function = lambda a, v:
    a*gamma + v;
    reward_reverse = tf.scan(discounted_function,
    tf.reverse(reward, [True, False]))
    discounted_reward = tf.reverse(reward_reverse,
    [True, False])
    return discounted_reward
```

9）在继续之前，需要定义超参数：

```
learning_rate = 0.001
gamma = 0.99
batch_size = 10
```

10）必须通过定义占位符来单独设置反向更新：

```
# 定义 TensorFlow 占位符
episode_x = tf.placeholder(dtype=tf.float32,
shape=[None, input_dim])
episode_y = tf.placeholder(dtype=tf.float32,
shape=[None, n_actions])
episode_reward = tf.placeholder(dtype=tf.float32,
shape=[None, 1])

episode_discounted_reward = discounted_rewards
(episode_reward, gamma)
episode_mean, episode_variance=
tf.nn.moments(episode_discounted_reward,
[0], shift=None)

# 标准化折扣后的收益
episode_discounted_reward -= episode_mean
episode_discounted_reward /=
tf.sqrt(episode_variance + 1e-6)

# 优化器设定
tf_aprob = policy_forward(episode_x)
loss = tf.nn.l2_loss(episode_y - tf_aprob)
optimizer = tf.train.AdamOptimizer(learning_rate)
gradients = optimizer.compute_gradients(loss,
var_list=tf.trainable_variables(),
grad_loss=episode_discounted_reward)
train_op = optimizer.apply_gradients(gradients)

# 图形初始化
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()

# 训练模型存储设定
saver = tf.train.Saver(tf.global_variables())
save_path = 'checkpoints/pong_rl.ckpt'
```

11）现在，可以初始化值并运行算法：

```
obs_prev = None
xs, ys, rs, = [], [], []
reward_sum = 0
episode_number = 0
reward_window = None
reward_best = -22
history = []

observation = env.reset()
while True:
# if True: env.render()
# 如果希望看见智能体，去掉注释符即可
# 训练过程随玩
```

```
# 预处理观测值，加载不同的图像给网络
obs_cur = preprocess_frame(observation)
obs_diff = obs_cur - obs_prev if obs_prev
is not None else np.zeros(input_dim)
obs_prev = obs_cur

# 按策略采样一次动作
feed = {episode_x: np.reshape(obs_diff, (1,-1))}
aprob = sess.run(tf_aprob, feed) ; aprob = aprob[0,:]
action = np.random.choice(n_actions, p=aprob)
label = np.zeros_like(aprob) ; label[action] = 1

# 返回环境动作并提取下一个观测、回报以及状态
observation, reward, done, info =
env.step(action+1)
reward_sum += reward
# 记录游戏历史
xs.append(obs_diff) ; ys.append(label) ;
rs.append(reward)

if done:
    history.append(reward_sum)
    reward_window = -21 if reward_window is
    None else np.mean(history[-100:])
    # 用存储值更新权重
    # （更新策略）
    feed = {episode_x: np.vstack(xs), episode_y:
    np.vstack(ys), episode_reward: np.vstack(rs), }
    _ = sess.run(train_op, feed)
    print('episode {:2d}: reward: {:2.0f}'.
    format(episode_number, reward_sum))
    xs, ys, rs = [], [], []
    episode_number += 1
    observation = env.reset()
    reward_sum = 0
    # 10个场景后存储最佳模型
    if (episode_number % 10 == 0) & (reward_window >
    reward_best):
        saver.save(sess, save_path,
        global_step=episode_number)
        reward_best = reward_window
        print("Save best model {:2d}:
        {:2.5f} (reward window)"
        .format(episode_number, reward_window))
```

12）当算法运行 2500 周期后，可以停止训练并绘制结果（见图 5.4）：

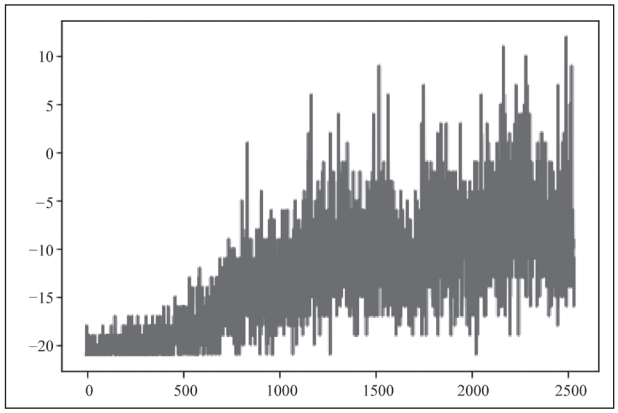


图 5.4　得分数与训练周期数

```
plt.plot(history)
plt.show()
```

13）最后，查看最终模型在 OpenAI 的 Gym 中是如何表现的（见图 5.5）：

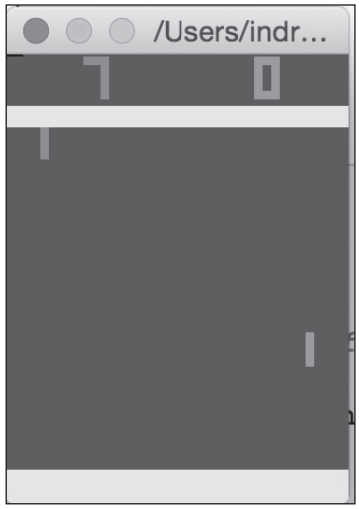


图 5.5　训练策略梯度智能体

```
observation = env.reset()
while True:
    if True: env.render()

    # 预处理观测值，将不同的图像加载给网络
    obs_cur = preprocess_frame(observation)
    obs_diff = obs_cur - obs_prev if
    obs_prev is not None else np.zeros(input_dim)
    obs_prev = obs_cur
```

```
# 按策略采样一次动作
feed = {tf_x: np.reshape(obs_diff, (1,-1))}
aprob = sess.run(tf_aprob, feed) ; aprob = aprob[0,:]
action = np.random.choice(n_actions, p=aprob)
label = np.zeros_like(aprob) ; label[action] = 1

# 返回环境动作并提取下一个观测、回报以及状态
observation, reward, done, info = env.step(action+1)
if done: observation = env.reset()
```

正如所看到的，在实现能够更稳定地赢得这一局之前，需要相当多的训练。如果希望查看智能体在训练期间的执行情况，则可以取消 while 循环中第一行代码的注释。这将在训练过程中呈现每次观察结果。

改进前面设置的选项之一是调整网络架构并微调超参数。一个更复杂的网络架构可能会有所帮助，但也会增加训练时间。策略梯度是当前最流行的 RL 方法，因为它们是端到端的，并且在正确调优时显示了更好的结果。

5.3 实现深度 *Q* 学习算法

另一种流行的强化学习方法是 *Q* 学习（*Q*-Learning）。在 *Q* 学习中，并不关注将观察映射到特定的行为，而是尝试将一些值赋给当前状态（观察）并根据该值采取操作。状态和行为可以看作**马尔科夫决策过程**，环境是随机的。在马尔科夫过程中，下一个状态只取决于当前状态和后面的操作。因此，假设所有以前的状态（和操作）都是无关的。

Q 在 *Q* 学习中代表品质；函数 *Q*（*s*,*a*）为状态 *s* 中的操作 *a* 提供品质打分。函数可以是任何类型的。在一个简单的表单中，它可以是一个查找表。然而，在一个更复杂的环境中，这是行不通的，这就是深度学习的机理所在。在接下来的内容中，将从 OpenAI 中实现一个深度 *Q* 学习算法来进行突破。

准备

在开始实施方案之前，确保安装了 OpenAI 的 Gym 环境。按照网站上的说明操作：https://gym.openai.com/docs/。当在服务器上运行 Gym 时，需要连接一个虚拟显示器。

如何去做…

1）从导入必要的函数库开始，如下：

```
import gym
import random
import numpy as np
import matplotlib.pyplot as plt
from collections import deque
```

```
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import Dense, Flatten
from keras.layers.convolutional import Conv2D
from keras import backend as K
```

2）首先，绘制游戏的一个输入图像示例（见图 5.6）：

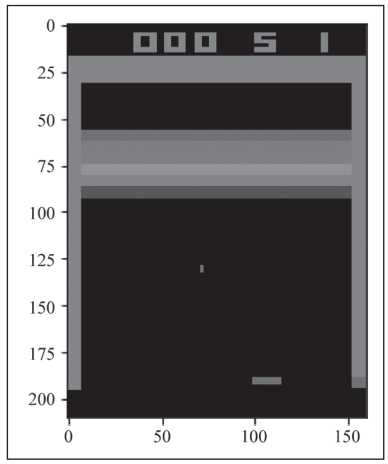


图 5.6　OpenAI 的 Breakout 示例输入图像

```
env = gym.make('BreakoutDeterministic-v4')
observation = env.reset()

for i in range(3):
    # 两帧图像后发球
    if i > 1:
        print(observation.shape)
        plt.imshow(observation)
        plt.show()
    # 获得下一次发球
    observation, _, _, _ = env.step(1)
```

3）现在，可以定义一个函数来预处理输入数据：

```
def preprocess_frame(frame):
    # 移除顶部帧以及一些背景
    frame = frame[35:195, 10:150]
    # 图像帧灰度化并缩小1/2
    frame = frame[::2, ::2, 0]
    # 设置背景为 0
    frame[frame == 144] = 0
    frame[frame == 109] = 0
    # 球和拍数设为 1
    frame[frame != 0] = 1
    return frame.astype(np.float).ravel()
```

4）输出前面的预处理图像，以了解算法将如何处理（见图 5.7）：

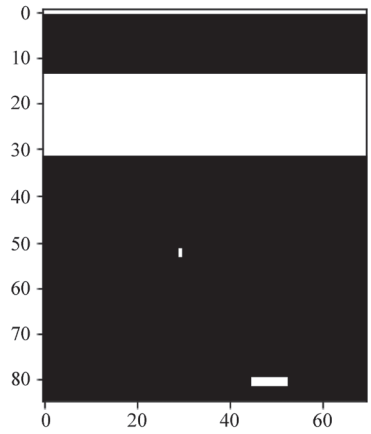


图 5.7　Breakout 预处理框架

```
obs_preprocessed = preprocess_frame(observation)
plt.imshow(obs_preprocessed, cmap='gray')
plt.show()
```

5）为实现深度 Q 学习，需要定义一个智能体来执行大部分任务：

```
class DQLAgent:
    def __init__(self, cols, rows, n_actions, batch_size=32):
        self.state_size = (cols, rows, 4)
        self.n_actions = n_actions
        self.epsilon = 1.
        self.epsilon_start, self.epsilon_end = 1.0, 0.1
        self.exploration_steps = 1000000.
        self.epsilon_decay_step = (self.epsilon_start -
self.epsilon_end) / self.exploration_steps
        self.batch_size = batch_size
        self.discount_factor = 0.99
        self.memory = deque(maxlen=400000)
        self.model = self.build_model()
        self.target_model = self.build_model()
        self.optimizer = self.optimizer()
        self.avg_q_max, self.avg_loss = 0, 0

    def optimizer(self):
        a = K.placeholder(shape=(None,), dtype='int32')
        y = K.placeholder(shape=(None,), dtype='float32')

        py_x = self.model.output
```

```
        a_one_hot = K.one_hot(a, self.n_actions)
        q_value = K.sum(py_x * a_one_hot, axis=1)
        error = K.abs(y - q_value)

        quadratic_part = K.clip(error, 0.0, 1.0)
        linear_part = error - quadratic_part
        loss = K.mean(0.5 *
        K.square(quadratic_part) + linear_part)

        opt = Adam(lr=0.00025, epsilon=0.01)
        updates = opt.get_updates
        (self.model.trainable_weights, [], loss)
        train = K.function([self.model.input, a, y],
        [loss], updates=updates)

        return train

    def build_model(self):
    model = Sequential()
    model.add(Conv2D(32, (8, 8), strides=(4, 4),
    activation='relu', input_shape=self.state_size))
    model.add(Conv2D(64, (4, 4), strides=(2, 2),
    activation='relu'))
    model.add(Conv2D(64, (3, 3), strides=(1, 1),
    activation='relu'))
    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dense(self.n_actions))
    model.summary()
    return model

def update_model(self):
    self.target_model.set_weights
    (self.model.get_weights())

def action(self, history):
    history = np.float32(history / 255.0)
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.n_actions)
    else:
        q_value = self.model.predict(history)
        return np.argmax(q_value[0])

def replay(self, history, action, reward,
           next_history, dead):
    self.memory.append((history, action,
    reward, next_history, dead))

def train(self):
    if len(self.memory) < self.batch_size:
        return
    if self.epsilon > self.epsilon_end:
        self.epsilon -= self.epsilon_decay_step
```

```
        mini_batch = random.sample(self.memory,
        self.batch_size)
        history = np.zeros((self.batch_size,
        self.state_size[0], self.state_size[1],
        self.state_size[2]))
        next_history = np.zeros((self.batch_size,
        self.state_size[0], self.state_size[1],
        self.state_size[2]))
        target = np.zeros((self.batch_size,))
        action, reward, dead = [], [], []

        for i in range(self.batch_size):
                history[i] = np.float32
                (mini_batch[i][0] / 255.)
                next_history[i] = np.float32
                (mini_batch[i][3] / 255.)
                action.append(mini_batch[i][1])
                reward.append(mini_batch[i][2])
                dead.append(mini_batch[i][4])

            target_value = self.target_model.
            predict(next_history)

            for i in range(self.batch_size):
                if dead[i]:
                    target[i] = reward[i]
                else:
                    target[i] = reward[i] +
                    self.discount_factor * \
                    np.amax(target_value[i])

            loss = self.optimizer([history, action, target])
            self.avg_loss += loss[0]
```

6）接下来，设置超参数和一般参数，并初始化智能体：

```
env = gym.make('BreakoutDeterministic-v4')

# 整体设定
n_warmup_steps = 50000
update_model_rate = 10000
cols, rows = 85, 70
n_states = 4

# 超参数
batch_size = 32

# 初始化
agent = DQLAgent(cols, rows, n_actions=3)
scores, episodes = [], []
n_steps = 0
```

7）现在准备开始训练模型：

```
while True:
    done = False
    dead = False
    step, score, start_life = 0, 0, 5
    observation = env.reset()

    state = preprocess_frame(observation,
    cols, rows)
    history = np.stack((state, state,
    state, state), axis=2)
    history = np.reshape([history],
    (1, cols, rows, n_states))

    while not done:
# env.render()
        n_steps += 1
        step += 1
        # 获取动作
        action = agent.action(history)
        observation, reward, done, info =
        env.step(action+1)
        # 提取下一状态
        state_next = preprocess_frame
        (observation, cols, rows)
        state_next = np.reshape([state_next],
        (1, cols, rows, 1))
        history_next = np.append(state_next,
        history[:, :, :, :3], axis=3)

        agent.avg_q_max += np.amax(agent.model
        .predict(history)[0])
        reward = np.clip(reward, -1., 1.)

        agent.replay(history, action, reward,
        history_next, dead)
        agent.train()
        if n_steps % update_model_rate == 0:
            agent.update_model()
        score += reward

        if dead:
            dead = False
        else:
            history = history_next

        if done:
            print('episode {:2d}; score:
            {:2.0f}; q {:2f}; loss {:2f}; steps {}'
                  .format(n_steps, score,
                  agent.avg_q_max / float(step),
                  agent.avg_loss / float(step), step))
```

```
            agent.avg_q_max, agent.avg_loss = 0, 0
    # 存储模型权重

    if n_steps % 1000 == 0:
        agent.model.save_weights("weights/breakout_dql.h5")
```

8）当算法得分足够高时，可以停止训练。

9）看看最终模型是如何执行的（见图 5.8）：

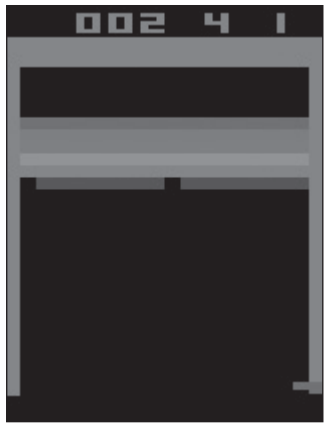


图 5.8　操作过程中训练深度 *Q* 学习智能体

```
env = gym.make('BreakoutDeterministic-v4')
agent = DQLAgent(cols, rows, n_action=3)

for i in range(5):
    observation = env.reset()

    state = pre_processing(observation,
    cols, rows)
    history = np.stack((state, state,
    state, state), axis=2)
    history = np.reshape([history], (1, cols,
    rows, n_states))

    while not done:
        env.render()
        action = agent.get_action(history)
        observe, reward, done, info =
        env.step(action+1)
```

与策略梯度一样，深度 *Q* 学习实现的网络体系结构可以设置得像人们喜欢的那样复杂。在更复杂的环境中，采纳更复杂的网络架构是获得良好结果的关键。强化学习是一个令人着迷的领域。在接下来的内容中，特别是在第 11 章　游戏智能体和机器人　中，将向读者展示如何将强化学习应用到其他游戏和问题上。此外，还将介绍一些先进的技术，以及用这些技术加速训练，例如采用并行化方法。

第 6 章
生成对抗网络

本章主要讨论**生成对抗性网络**（GAN），包含以下内容：

- 了解 GAN；
- 实现深度卷积 GAN（DCGAN）；
- 使用超分辨率 GAN（SRGAN）来提高图像分辨率。

6.1 简介

在本章中，将介绍 GAN。就像在自动编码器网络中一样，GAN 有一个生成器网络和一个鉴别器网络。然而，GAN 的机理完全不同。它代表着一个无监督的学习问题，在学习过程中两个网络相互竞争，同时相互合作。重要的是，生成器和鉴别器不能相互压倒对方。GAN 背后的思想是根据训练数据生成新的例子。应用程序的范围可以从生成新的手写 MNIST 字符图像到生成音乐。GAN 最近受到了很多关注，因为使用它们的结果令人着迷。

6.2 了解 GAN

了解 GAN 之后，再来开始实现 GAN。确定 GAN 的样本质量很困难，更低的损失价值并不总是代表更好的质量。通常，对于图像来说，确定质量的唯一方法是通过视觉检查生成的示例。可以确定生成的图像是否足够真实，或多或少像一个简单的图灵测试。在下面的内容中，将使用著名的 MNIST 数据集和 Keras 框架来介绍 GAN。

如何去做…

1）首先导入必要的函数库，如下所示：

```
import numpy as np
from keras.models import Sequential, Model
from keras.layers import Input, Dense, Activation, Flatten, Reshape
from keras.layers import Conv2D, Conv2DTranspose, UpSampling2D
from keras.layers import LeakyReLU, Dropout
from keras.layers import BatchNormalization
from keras.optimizers import Adam
from keras import initializers
```

```
from keras.datasets import mnist

import matplotlib.pyplot as plt
```

2）通过使用 Keras，可以轻松地导入两个数据集，代码如下：

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()

img_rows, img_cols = X_train.shape[1:]

X_train = X_train.reshape(-1, img_rows*img_cols,
1).astype(np.float32)/255.
```

3）对于 GAN，需要定义三个网络架构。先从 discriminator_model（鉴别器模型）开始：

```
 def discriminator_model(dropout=0.5):
    model = Sequential()
    model.add(Dense(1024, input_dim=784,
kernel_initializer=initializers.RandomNormal(stddev=0.02)))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(dropout))
    model.add(Dense(512))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(dropout))
    model.add(Dense(256))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(dropout))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=adam)
    return model
```

4）接下来，定义 generator_model（生成器模型）：

```
def generator_model():
    model = Sequential()
    model.add(Dense(256, input_dim=100,
kernel_initializer=initializers.RandomNormal(stddev=0.02)))
    model.add(LeakyReLU(0.2))
    model.add(Dense(512))
    model.add(LeakyReLU(0.2))
    model.add(Dense(1024))
    model.add(LeakyReLU(0.2))
    model.add(Dense(784, activation='tanh'))
    model.compile(loss='binary_crossentropy', optimizer=adam)
    return model
```

5）最后，可以创建网络及其组合：

```
discriminator = discriminator_model()
generator = generator_model()
discriminator.trainable = False
gan_input = Input(shape=(100,))
x = generator(gan_input)
gan_output = discriminator(x)
gan = Model(inputs=gan_input, outputs=gan_output)
gan.compile(loss='binary_crossentropy', optimizer='adam')
```

6）在训练 GAN 之前，需要定义一个函数来绘制生成的图像：

```
def plot_images(samples=16, step=0):
    images = generator.predict(noise)
    plt.figure(figsize=(5,5))
    for i in range(samples):
        plt.subplot(4, 4, i+1)
        image = images[i, :,]
        image = np.reshape(image, [img_rows, img_cols])
        plt.imshow(image, cmap='gray')
        plt.axis('off')
    plt.show()
```

7）现在可以开始训练了。在训练模型时，将有如下输出结果（见图 6.1）：

```
batch_size = 32
n_steps = 100000
plot_every = 1000

noise_input = np.random.uniform(-1.0, 1.0, size=[16, 100])
for step in range(n_steps):
    noise = np.random.normal(0, 1, size=[batch_size, 100])
    batch = X_train[np.random.randint(0, X_train.shape[0],
    size=batch_size)].reshape(batch_size, 784)

    gen_output = generator.predict(noise)
    X = np.concatenate([batch, gen_output])

    y_D = np.zeros(2*batch_size)
    y_D[:batch_size] = 0.9

    discriminator.trainable = True
    loss_D = discriminator.train_on_batch(X, y_D)

    noise = np.random.normal(0, 1, size=[batch_size, 100])
    y_G = np.ones(batch_size)
    discriminator.trainable = False
    loss_G = gan.train_on_batch(noise, y_G)
    if step % plot_every == 0:
        plot_images(samples=noise_input.shape[0], step=(step+1))
```

图 6.1　迭代 100000 步后生成图像的例子

6.3　实现 DCGAN

在前面的内容中，所建网络在几个迭代之后能够生成现实的例子。MNIST 数据集具有较低的平移不变性，因此网络更容易生成这些示例。在 GAN 早期，网络非常不稳定，小的变化可能会破坏输出。2016 年出现了 DCGAN。在 DCGAN 中，鉴别器和生成器都是全卷积型的，DCGAN 的输出已被证明是比较稳定的。在下面的内容中，将使用 Fashion-MNIST 数据集来增加数据集的复杂性，并演示如何在 PyTorch 中实现 DCGAN。

如何去做…

1）首先导入必要的函数库，如下所示：

```
import matplotlib.pyplot as plt
import itertools

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
from torch.utils.data.dataset import Dataset

import torchvision.datasets as dset
import torchvision.transforms as transforms
```

2）然后，在函数中定义鉴别器网络：

```
class discriminator(nn.Module):
    def __init__(self):
        super(discriminator, self).__init__()
        self.conv1 = nn.Conv2d(1, d, 4, 2, 1, bias=False)
        self.conv2 = nn.Conv2d(d, d*2, 4, 2, 1, bias=False)
        self.conv2_bn = nn.BatchNorm2d(*2)
        self.conv3 = nn.Conv2d(d*2, *4, 4, 2, 1, bias=False)
        self.conv3_bn = nn.BatchNorm2d(*4)
        self.conv4 = nn.Conv2d(d*4, d*8, 4, 2, 1, bias=False)
        self.conv4_bn = nn.BatchNorm2d(d*8)
        self.conv5 = nn.Conv2d(d*8, 3, 4, 1, 0, bias=False)

    def weight_init(self, mean, std):
        for m in self._modules:
            normal_init(self._modules[m], mean, std)

    def forward(self, input):
        x = F.leaky_relu(self.conv1(input), 0.2)
        x = F.leaky_relu(self.conv2_bn(self.conv2(x)), 0.2)
        x = F.leaky_relu(self.conv3_bn(self.conv3(x)), 0.2)
        x = F.leaky_relu(self.conv4_bn(self.conv4(x)), 0.2)
        x = F.sigmoid(self.conv5(x))
        return x
```

3）在前面的函数中，使用了 normal_init 函数，将在生成器中来使用它，将其定义如下：

```
def normal_init(m, mean, std):
    if isinstance(m, nn.ConvTranspose2d) or isinstance(m,
nn.Conv2d):
        m.weight.data.normal_(mean, std)
```

4）接下来，定义生成器：

```
class generator(nn.Module):
    def __init__(self):
        super(generator, self).__init__()
        self.deconv1 = nn.ConvTranspose2d(100, 1024, 4, 1, 0)
        self.deconv1_bn = nn.BatchNorm2d(1024)
        self.deconv2 = nn.ConvTranspose2d(1024, 512, 4, 2, 1)
        self.deconv2_bn = nn.BatchNorm2d(512)
        self.deconv3 = nn.ConvTranspose2d(512, 256, 4, 2, 1)
        self.deconv3_bn = nn.BatchNorm2d(256)
        self.deconv4 = nn.ConvTranspose2d(256, 128, 4, 2, 1)
        self.deconv4_bn = nn.BatchNorm2d(128)
        self.deconv5 = nn.ConvTranspose2d(128, 1, 4, 2, 1)
```

```
    def weight_init(self, mean, std):
        for m in self._modules:
            normal_init(self._modules[m], mean, std)

    def forward(self, input):
        x = F.relu(self.deconv1_bn(self.deconv1(input)))
        x = F.relu(self.deconv2_bn(self.deconv2(x)))
        x = F.relu(self.deconv3_bn(self.deconv3(x)))
        x = F.relu(self.deconv4_bn(self.deconv4(x)))
        x = F.tanh(self.deconv5(x))
        return x
```

5）为了在训练中绘制多个随机输出，创建一个函数来绘制生成的图像：

```
def plot_output(epoch):
    z_ = torch.randn((5*5, 100)).view(-1, 100, 1, 1)
    z_ = Variable(z_.cuda(), volatile=True)

    G.eval()
    test_images = G(z_)
    G.train()

    size_figure_grid = 5
    fig, ax = plt.subplots(size_figure_grid, size_figure_grid,
figsize=(5, 5))
    for i, j in itertools.product(range(size_figure_grid),
range(size_figure_grid)):
        ax[i, j].get_xaxis().set_visible(False)
        ax[i, j].get_yaxis().set_visible(False)
```

6）在接下来的步骤中，将设置训练中使用的超参数。超参数对 GAN 的结果有很大的影响。生成器和鉴别器不应该相互压倒对方。批处理大小是 GAN 的一个重要的超参数。深度学习的一个普遍规律是，在一定程度上，批量越大越好。然而，对于 GAN，较小的批处理大小有时也可以更好地工作。

```
n_epochs = 24
batch_size = 32
learning_rate = 0.0002
```

7）对于所建模型，将使用 Fashion-MNIST 数据集，并将其加载到以下代码中：

```
transform = transforms.Compose([transforms.Scale(64),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,),
(0.5,))])

train_dataset = fashion_mnist(root='./data',
                              train=True,
                              transform=transform,
                              download=True
                             )

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)
```

8）通过调用定义函数来创建两个网络，然后初始化权重：

```
G = generator(128)
D = discriminator(128)
G.weight_init(mean=0.0, std=0.02)
D.weight_init(mean=0.0, std=0.02)
```

9）接下来，需要确保使用 CUDA ：

```
G.cuda()
D.cuda()
```

10）对于 GAN，可以使用**二元交叉熵**（BCE）损失函数：

```
BCE_loss = nn.BCELoss()
```

11）对于这两个网络，需要使用如下配置来定义优化器：

```
beta_1 = 0.5
beta_2 = 0.999
G_optimizer = optim.Adam(G.parameters(), lr=learning_rate,
betas=(beta_1, beta_2))
D_optimizer = optim.Adam(D.parameters(), lr=learning_rate/2,
betas=(beta_1, beta_2))
```

12）现在，可以开始使用以下代码块来训练网络：

```
for epoch in range(n_epochs):
    D_losses = []
    G_losses = []
    for x_, _ in train_loader:
        D.zero_grad()

        mini_batch = x_.size()[0]

        y_real_ = torch.ones(mini_batch)
        y_fake_ = torch.zeros(mini_batch)

        x_ = Variable(x_.cuda())
        y_real_ = Variable(y_real_.cuda())
        y_fake_ = Variable(y_fake_.cuda())
        D_result = D(x_).squeeze()
        D_real_loss = BCE_loss(D_result, y_real_)

        z_ = torch.randn((mini_batch, 100)).view(-1, 100, 1, 1)
        z_ = Variable(z_.cuda())
        G_result = G(z_)

        D_result = D(G_result).squeeze()
        D_fake_loss = BCE_loss(D_result, y_fake_)
        D_fake_score = D_result.data.mean()
        D_train_loss = D_real_loss + D_fake_loss

        D_train_loss.backward()
        D_optimizer.step()
        D_losses.append(D_train_loss.data[0])
        G.zero_grad()

        z_ = torch.randn((mini_batch, 100)).view(-1, 100, 1, 1)
        z_ = Variable(z_.cuda())

        G_result = G(z_)
        D_result = D(G_result).squeeze()
        G_train_loss = BCE_loss(D_result, y_real_)
        G_train_loss.backward()
        G_optimizer.step()
        G_losses.append(G_train_loss.data[0])

    print('Epoch %d - loss_d: %.3f, loss_g: %.3f' % ((epoch + 1),
torch.mean(torch.FloatTensor(D_losses)),
torch.mean(torch.FloatTensor(G_losses))))
    # 绘制样本输出
    plot_output(epoch)
```

在图 6.2 中，可以看到运行了 24 个周期后的网络输出：

图 6.2　运行 24 个迭代周期后生成图像的示例

6.4　使用 SRGAN 来提高图像分辨率

在第 3 章　卷积神经网络　中，演示了如何使用 CNN 来自动编码图像以获得图像的压缩。在数字时代，更重要的是能够将图像的分辨率提高档次。例如，一个压缩版本的图像可以很容易地通过互联网共享。当图像到达接收端时，它的质量需要增加，也称为**超分辨成像**（SR）。在下面的内容中，将向读者展示如何通过使用 PyTorch 框架进行深度学习来训练网络模型以此来提高图像分辨率。

如何去做…

1）首先，需要导入所需的函数库：

```
import os
from os import listdir
from os.path import join
import numpy as np
import random
import matplotlib.pyplot as plt

import torchvision
from torchvision import transforms
import torchvision.datasets as datasets

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
```

2）在这个方案中，将使用 CelebA 人脸数据集。将在 PyTorch 中导入可以自定义功能的数据：

```
class data_from_dir(data.Dataset):
     def __init__(self, image_dir, transform=None):
        super(DatasetFromFolder, self).__init__()
        self.image_dir = image_dir
        self.image_filenames = [ x for x in listdir(image_dir) if
        is_image_file(x)]
        self.transform = transform
    def __getitem__(self, index):
        # 加载图像
        image = Image.open(join(self.image_dir,
        self.image_filenames[index])).convert('RGB')
        image = self.transform(image)
        return image

    def __len__(self):
        return len(self.image_filenames)
```

3）现在定义用于模型的超参数：

```
batch_size = 16
image_size = 125
image_channels = 3
n_conv_blocks = 2
up_sampling = 2
n_epochs = 100
learning_rate_G = 0.00001
learning_rate_D = 0.0000001
```

4）在继续之前，将展示一个示例图像和一个缩小版。将缩小的图像用作输入，将原始图像用作目标图像。选择将原始图像缩小为 1/2（见图 6.3）：

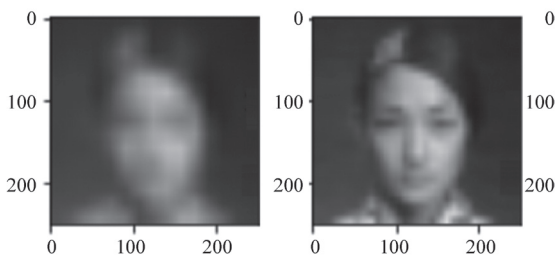


图 6.3　缩小图像和原始图像的示例（目标为 250 × 250 像素）

5）现在可以定义鉴别器网络。这个网络架构非常简单，并且基于众所周知的分类体系结构。鉴别器的任务是对真实的输出和生成器的输出进行分类：

```
class discriminator_model(nn.Module)
    def __init__(self):
        super(discriminator_model, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, 3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(64, 64, 3, stride=2, padding=1)
        self.conv2_bn = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, 3, stride=1, padding=1)
        self.conv3_bn = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(128, 128, 3, stride=2, padding=1)
        self.conv4_bn = nn.BatchNorm2d(128)
        self.conv5 = nn.Conv2d(128, 256, 3, stride=1, padding=1)
        self.conv5_bn = nn.BatchNorm2d(256)
        self.conv6 = nn.Conv2d(256, 256, 3, stride=2, padding=1)
        self.conv6_bn = nn.BatchNorm2d(256)
        self.conv7 = nn.Conv2d(256, 512, 3, stride=1, padding=1)
        self.conv7_bn = nn.BatchNorm2d(512)
        self.conv8 = nn.Conv2d(512, 512, 3, stride=2, padding=1)
        self.conv8_bn = nn.BatchNorm2d(512)
        self.fc1 = nn.Linear(2048, 1024)
        self.fc2 = nn.Linear(1024, 1)

    def forward(self, x):
        x = F.elu(self.conv1(x))
        x = F.elu(self.conv2_bn(self.conv2(x)))
        x = F.elu(self.conv3_bn(self.conv3(x)))
        x = F.elu(self.conv4_bn(self.conv4(x)))
        x = F.elu(self.conv5_bn(self.conv5(x)))
        x = F.elu(self.conv6_bn(self.conv6(x)))
        x = F.elu(self.conv7_bn(self.conv7(x)))
        x = F.elu(self.conv8_bn(self.conv8(x)))
        x = x.view(x.size(0), -1)
        x = F.elu(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        return x
```

6）在定义生成器体系结构之前，需要定义将重复使用的两个类。首先是一个卷积块，它由批量处理标准化、卷积层和 Dropout 组成：

```
class conv_block(nn.Module):
    def __init__(self, in_channels, k, layers, p=0.2):
        super(conv_block, self).__init__()
        self.layers = layers

        for i in range(layers):
            self.add_module('batchnorm' + str(i+1),
            nn.BatchNorm2d(in_channels))
            self.add_module('conv' + str(i+1),
            nn.Conv2d(in_channels, k, 3, stride=1,
            padding=1))
            self.add_module('drop' + str(i+1),
            nn.Dropout2d(p=p))
```

```
            in_channels += k

    def forward(self, x):
        for i in range(self.layers):
            y = self.__getattr__('batchnorm' + str(i+1))(x.clone())
            y = F.elu(y)
            y = self.__getattr__('conv' + str(i+1))(y)
            y = self.__getattr__('drop' + str(i+1))(y)
            x = torch.cat((x,y), dim=1)
        return x
```

7）另一个类用于向上采样：

```
class upsample_block(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(upsample_block, self).__init__()
        self.upsample1 = nn.Upsample(scale_factor=2,
        mode='nearest')
        self.conv1 = nn.Conv2d(in_channels, out_channels, 3,
       stride=1, padding=1)

    def forward(self, x):
        return F.elu(self.conv1(self.upsample1(x)))
```

8）其次是生成器模型：

```
class generator_model(nn.Module):
    def __init__(self, n_conv_blocks, n_upsample_blocks):
        super(generator_model, self).__init__()
        self.n_dense_blocks = n_blocks
        self.upsample = upsample

        self.conv1 = nn.Conv2d(3, 64, 9, stride=1, padding=1)

        inchannels = 64
        for i in range(self.n_conv_blocks):
            self.add_module('conv_block' + str(i+1),
            conv_block(inchannels, 12, 4))
            inchannels += 12*4

        self.conv2 = nn.Conv2d(inchannels, 64, 3,
        stride=1, padding=1)
        self.conv2_bn = nn.BatchNorm2d(64)

        in_channels = 64
        out_channels = 256
        for i in range(self.n_upsample_blocks):
            self.add_module('upsample_block' + str(i+1),
```

```
            upsample_block(in_channels, out_channels))
            in_channels = out_channels
            out_channels = int(out_channels/2)

        self.conv3 = nn.Conv2d(in_channels, 3, 9,
        stride=1, padding=1)

    def forward(self, x):
        x = self.conv1(x)

        for i in range(self.n_dense_blocks):
            x = self.__getattr__('conv_block' + str(i+1))(x)

        x = F.elu(self.conv2_bn(self.conv2(x)))

        for i in range(self.upsample):
            x = self.__getattr__('upsample_blcok' + str(i+1))(x)

        return self.conv3(x)
```

9）现在可以创建数据集了。首先，定义所需要的转换函数：

```
normalize = transforms.Normalize(mean = [0.485, 0.456, 0.406],
                                 std = [0.229, 0.224, 0.225])

scale = transforms.Compose([transforms.ToPILImage(),
                            transforms.Scale(image_size),
                            transforms.ToTensor(),
                            transforms.Normalize
                            (mean = [0.485, 0.456, 0.406],
                            std = [0.229, 0.224, 0.225])
                            ])

transform =
transforms.Compose([transforms.Scale(image_size*n_upsampling),
                                 transforms.ToTensor()])
```

10）把数据载入到 PyTorch 的 DataLoader 中：

```
dataset = data_from_dir('Data/CelebA/splits/train',
transform=transform)
dataloader = torch.utils.data.DataLoader(dataset,
batch_size=batch_size, shuffle=True)

netG = Generator(n_conv_blocks, n_upsampling)
netD = Discriminator()
```

11）作为损失函数，将使用 BCELoss 函数：

```
adversarial_loss = nn.BCELoss()
```

12）需要设置占位符并激活 CUDA 的使用：

```
target_real = Variable(torch.ones(batch_size, 1))
target_fake = Variable(torch.zeros(batch_size, 1))
target_real = target_real.cuda()
target_fake = target_fake.cuda()

inputs_G = torch.FloatTensor(batch_size, image_channels,
image_size, image_size)

netG.cuda()
netD.cuda()
feature_extractor =
FeatureExtractor(torchvision.models.vgg19(pretrained=True))
feature_extractor.cuda()

content_loss = nn.MSELoss()
adversarial_loss = nn.BCELoss()
content_loss.cuda()
adversarial_loss.cuda()
```

13）对于这两个网络，将使用 Adam 优化器：

```
opt_G = optim.Adam(netG.parameters(), lr=learning_rate_G)
opt_D = optim.Adam(netD.parameters(), lr=learning_rate_D)
```

14）最后，在训练网络之前，定义一个函数来绘制中间结果：

```
def plot_output(inputs_G, inputs_D_real, inputs_D_fake):
    image_size = (250, 250)
    transform = transforms.Compose([transforms.Normalize
                        (mean = [-2.118, -2.036, -1.804],
                         std = [4.367, 4.464, 4.444]),
                         transforms.ToPILImage(),
     transforms.Scale(image_size)])
    figure, (lr_plot, hr_plot, fake_plot) = plt.subplots(1,3)
    i = random.randint(0, inputs_G.size(0) -1)

    lr_image = transform(inputs_G[i])
    hr_image = transform(inputs_D_real[i])
    fake_hr_image = transform(inputs_D_fake[i])
```

```
    lr_image_ph = lr_plot.imshow(lr_image)
    hr_image_ph = hr_plot.imshow(hr_image)
    fake_hr_image_ph = fake_plot.imshow(fake_hr_image)

    figure.canvas.draw()
    plt.show()
```

15）现在可以开始使用下面的代码块来训练网络：

```
inputs_G = torch.FloatTensor(batch_size, 3, image_size, image_size)

for epoch in range(n_epochs):
    for i, inputs in enumerate(dataloader):

        for j in range(batch_size):
            inputs_G[j] = scale(inputs[j])
            inputs[j] = normalize(inputs[j])

        inputs_D_real = Variable(inputs.cuda())
        inputs_D_fake = net_G(Variable(inputs_G).cuda())
        net_D.zero_grad()

        outputs = net_D(inputs_D_real)
        D_real = outputs.data.mean()

        loss_D_real = adversarial_loss(outputs, target_real)
        loss_D_real.backward()

        outputs = net_D(inputs_D_fake.detach())
        D_fake = outputs.data.mean()

        loss_D_fake = adversarial_criterion(outputs, target_fake)
        loss_D_fake.backward()

        opt_D.step()

        net_G.zero_grad()
        real_features =
        Variable(feature_extractor(inputs_D_real).data)
        fake_features = feature_extractor(inputs_D_fake)

        loss_G_content = content_loss(fake_features,
        real_features)
        loss_G_adversarial =
        adversarial_loss(net_D(inputs_D_fake).detach(),
        target_real)

        loss_G_total = 0.005*lossG_content +
```

```
0.001*lossG_adversarial
        loss_G_total.backward()
        opt_G.step()

    plot_output(inputs_G, inputs_D_real.cpu().data,
inputsD_fake.cpu().data)
```

在图 6.4 中，可以看到网络运行 100 个周期后的输出。

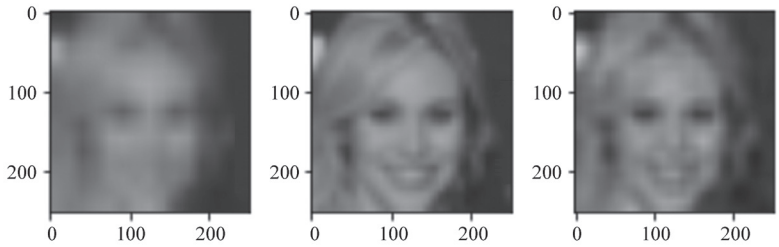


图 6.4　缩小的输入图像、原始图像和生成图像的示例

正如读者可能已经注意到的那样，生成的图像并不像原始图像那样清晰。训练一个生成器网络以提高图像的分辨率是一项繁重的计算任务。该方案中使用的数据集具有相对较低的平移不变性。然而，所建模型仍然需要进一步的优化以及更多周期的训练，以取得更好的结果。

第 7 章
计算机视觉

在本章中，将实现与图像编码数据处理（包括视频帧）有关的深度神经网络。以下是将要介绍的内容：

- 利用计算机视觉技术增广图像；
- 图像中的目标分类；
- 目标在图像中的本地化；
- 实时检测框架；
- 用 U-net 将图像分类；
- 语义分割与场景理解；
- 寻找人脸面部关键点；
- 人脸识别；
- 将样式转换为图像。

7.1 简介

本章的重点是深度学习在计算机视觉中的应用。深度学习，尤其是**卷积神经网络**（CNN），已经彻底改变了计算机视觉领域。大部分的基准测试都是由于引入更深的 CNN 而遭到放弃，有些则首次超过了人类水平的精度。在本章中，将展示计算机视觉中的各种应用。除了 CNN 之外，还将在这些方案中使用**递归神经网络**（RNN）。

7.2 利用计算机视觉技术增广图像

CNN 和计算机视觉在深度学习中不可分割。在深入研究深度学习在计算机视觉中的应用之前，将介绍基本的计算机视觉技术，读者可以在深度学习过程中应用这些技术，使构建的模型更加鲁棒。在训练过程中可以使用图像增广来增加不同示例样本的数量，并使所建模型对轻微变化更为鲁棒。而且，可以在测试过程中加以使用，即测试时间增广（TTA）。并非每种增强技术都适用于所有问题，例如将交通标志的左箭头翻转后的含义与原来的含义将会不同。本书将使用 OpenCV 实现图像增强功能。

如何去做…

1）先加载所需的函数库：

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import glob
```

2）接下来，加载一些将使用的样本图像，并绘制出来（见图 7.1）：

```
DATA_DIR = 'Data/augmentation/'
images = glob.glob(DATA_DIR + '*')

plt.figure(figsize=(10, 10))
i = 1
for image in images:
    img = cv2.imread(image)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.subplot(3, 3, i)
    plt.imshow(img)
    i += 1
plt.show()
```

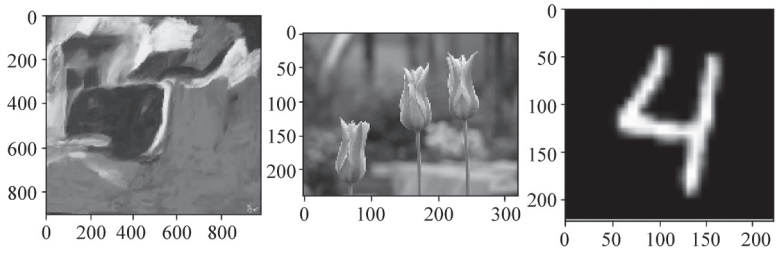


图 7.1　用于增广的示例图像

3）首先定义一个函数用来绘制图像增广的例子：

```
def plot_images(image, function, *args):
    plt.figure(figsize=(10, 10))
    n_examples = 3
    for i in range(n_examples):
        img = cv2.imread(image)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = function(img, *args)
        plt.subplot(3, 3, i+1)
        plt.imshow(img)
    plt.show()
```

4）定义一个函数来随机旋转图像并绘制一些例子（见图 7.2）：

```
 def rotate_image(image, rotate=20):
    width, height, _ = image.shape
    random_rotation = np.random.uniform(low=-rotate, high=rotate)
    M = cv2.getRotationMatrix2D((width/2, height/2),
random_rotation, 1)
    return(cv2.warpAffine(image, M, (width, height)))

plot_images(images[2], rotate_image, 40)
```

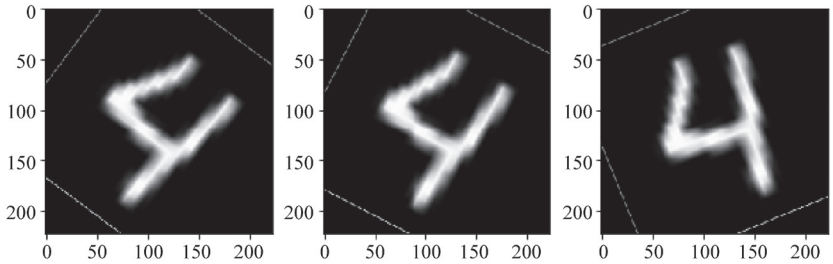


图 7.2　随机旋转的实例

5）接下来，定义一个函数来调整图像的亮度（见图 7.3）：

```
 def adjust_brightness(image, brightness=60):
    rand_brightness = np.random.uniform(low=-brightness,
high=brightness)
    return(cv2.add(image, rand_brightness))

plot_images(images[0], adjust_brightness, 85)
```

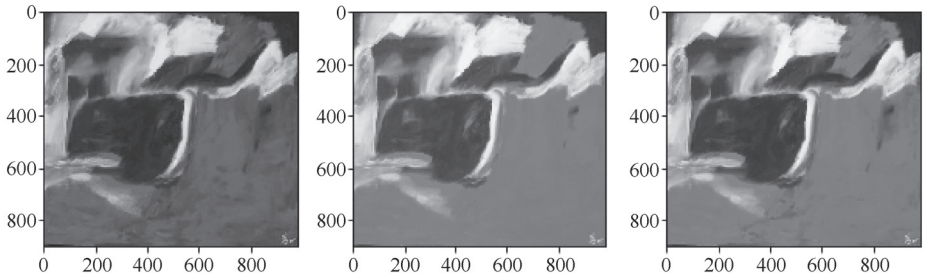


图 7.3　随机调整亮度的实例

6）以下函数将使用所提供的参数随机移动图像（见图 7.4）：

```
def random_shifts(image, shift_max_x=100, shift_max_y=100):
    width, height, _ = image.shape
    shift_x = np.random.randint(shift_max_x)
    shift_y = np.random.randint(shift_max_y)
    M = np.float32([[1, 0, shift_x],[0, 1, shift_y]])
    return (cv2.warpAffine(image, M, (height, width)))

plot_images(images[1], random_shifts, 100, 20)
```

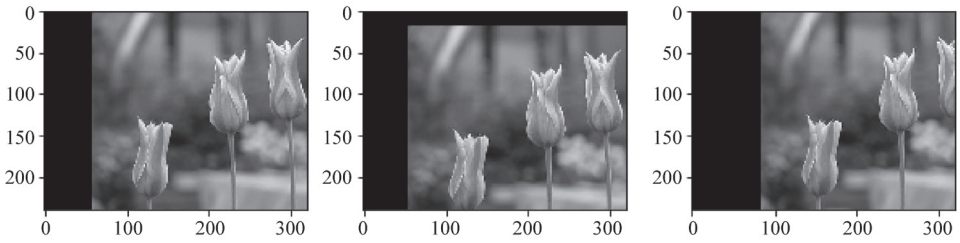


图 7.4　随机移动图像的实例

7）对于某些图像，放大或缩小是有益的（见图 7.5）：

```
def scale_image(image, scale_range=[0.6, 1.4]):
    width, height, _ = image.shape
    scale_x = np.random.uniform(low=scale_range[0],
high=scale_range[1])
    scale_y = np.random.uniform(low=scale_range[0],
high=scale_range[1])
    scale_matrix = np.array([[scale_x, 0., (1. - scale_x) * width /
2.],
                             [0., scale_y, (1. - scale_y) * height
/ 2.]],
                            dtype=np.float32)
    return(cv2.warpAffine(image, scale_matrix, (width, height),
flags=cv2.INTER_LINEAR,
                          borderMode=cv2.BORDER_REFLECT_101))

plot_images(images[2], scale_image, [0.7, 1.3])
```

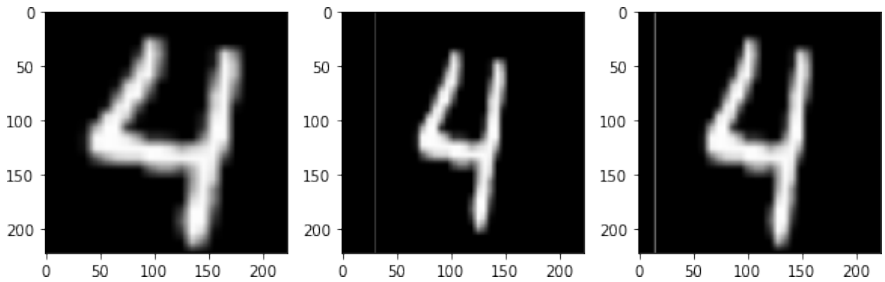


图 7.5　随机缩放图像的实例

8）最后的强广是为了随机翻转图像（见图 7.6）：

```
def random_flip(image, p_flip=0.5):
    rand = np.random.random()
    if rand < p_flip:
        image = cv2.flip(image, 1)
    return image

plot_images(images[2], random_flip)
```

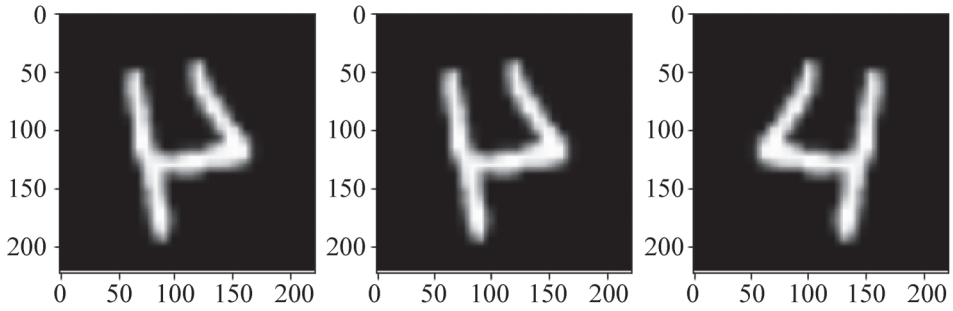


图 7.6　随机翻转图像的实例

9）随机地把所有的增广应用结合在一起，绘制 32 个例子的结果（见图 7.7）：

```
plt.figure(figsize=(15, 15))
image = images[1]
for i in range(32):
    img = cv2.imread(image)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = transform_image(img)
    plt.subplot(8, 8, i+1)
    plt.axis('off')
    plt.imshow(img, interpolation="nearest")
plt.show()
```

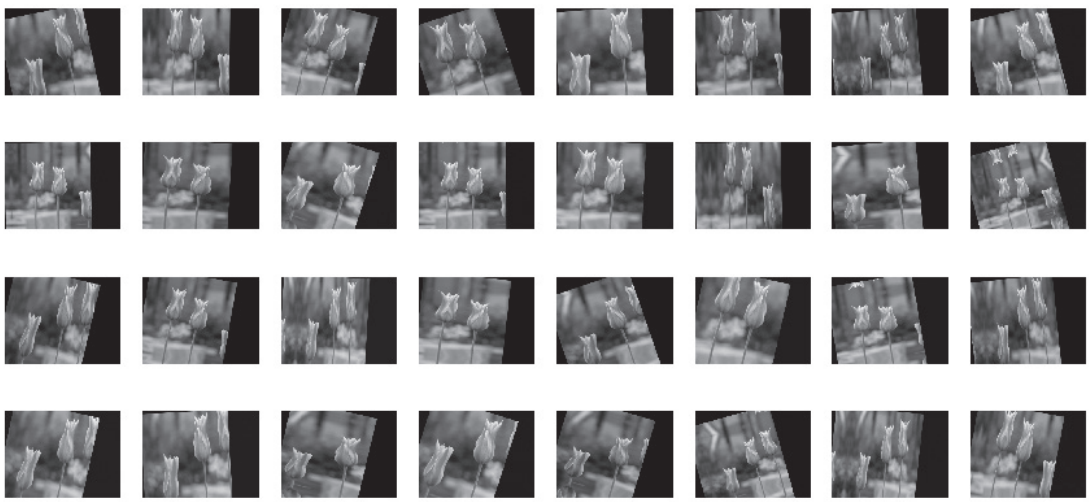


图 7.7　所有增广都随机应用 32 次

大多数深度学习框架都有自己的图像增广实现。例如，对于 PyTorch，请查看 torchvision.transforms。这些转换可以轻松添加诸如随机裁剪和翻转数据等增广功能。在 Keras 中，读者可以使用 ImageDataGenerator 进行随机图像增广。

7.3 图像中的目标分类

在本方案中，将向读者展示如何使用 CNN 对图像中的目标进行分类。将从头开始训练网络，对图像中的五种不同的花朵进行分类。图像有不同的大小。对于本方案，将使用 Keras。

如何去做…

1）创建一个新的 Python 文件并导入必要的函数库：

```
import numpy as np
import glob
import cv2
import matplotlib.pyplot as plt

from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

import keras
from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, Activation, Flatten,
Conv2D, MaxPooling2D, Lambda, Cropping2D
from keras.utils import np_utils
from keras import optimizers

SEED = 2017
```

2）接下来，加载数据集并提取标签：

```
# 指定数据目录并提取全部文件名
DATA_DIR = '../Data/'
images = glob.glob(DATA_DIR + "flower_photos/*/*.jpg")
# 从文件名提取标号
labels = [x.split('/')[3] for x in images]
```

3）查看这些数据（见图 7.8）：

```
unique_labels = set(labels)
plt.figure(figsize=(15, 15))
i = 1
for label in unique_labels:
    image = images[labels.index(label)]
    img = cv2.imread(image)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.subplot(5, 5, i)
    plt.title("{0} ({1})".format(label, labels.count(label)))
    i += 1
    _ = plt.imshow(img)
plt.show()
```

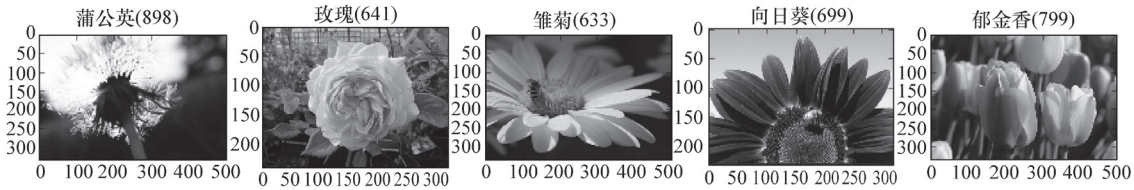


图 7.8　具有标签和计数的花朵数据集示例

4）将标签转换为二进制格式：

```
encoder = LabelBinarizer()
encoder.fit(labels)
y = encoder.transform(labels).astype(float)
```

5）将数据集分割为训练集和测试集：

```
X_train, X_val, y_train , y_val = train_test_split(images, y,
test_size=0.1, random_state=SEED)
```

6）定义网络架构：

```
# 定义结构
model = Sequential()
model.add(Lambda(lambda x: (x / 255.) - 0.5, input_shape=(100, 100,
3)))
model.add(Conv2D(16, (5, 5), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.5))
model.add(Conv2D(32, (5, 5), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.5))
```

```
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(5, activation='softmax'))

# 定义优化器并编译
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9,
nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd,
metrics=['accuracy'])
```

7）创建一个批量生成器，随机循环访问数据：

```
img_rows = img_cols = 100
img_channels = 3

def batchgen(x, y, batch_size, transform=False):
    # 创建空值numpy数组
    images = np.zeros((batch_size, img_rows, img_cols, img_channels))
    class_id = np.zeros((batch_size, len(y[0])))

    while 1:

for n in range(batch_size):
    i = np.random.randint(len(x))
    x_ = cv2.imread(x[i])
    x_ = cv2.cvtColor(x_, cv2.COLOR_BGR2RGB)
    # 图像大小不一，全部变换到100×100像素
    x_ = cv2.resize(x_, (100, 100))
    images[n] = x_
    class_id[n] = y[i]
    yield images, class_id
```

8）接下来，开始训练模型：

```
batch_size = 256
n_epochs = 20
s_epoch = 100
val_size = 0.2
val_steps = 20

train_generator = batchgen(X_train, y_train, batch_size, True)
val_generator = batchgen(X_valid, y_valid, batch_size, False)
```

```
history = model.fit_generator(train_generator,
                              steps_per_epoch=s_epoch,
                              nb_epoch=n_epochs,
                              validation_data=val_generator,
                              validation_steps = val_steps
                             )
```

9）绘制训练结果（见图 7.9）：

```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model_accuracy')
plt.ylabel('accuracy')
plt.xlabel('epochs')
plt.legend(['train', 'validation'], loc='lower right')
plt.show()
```

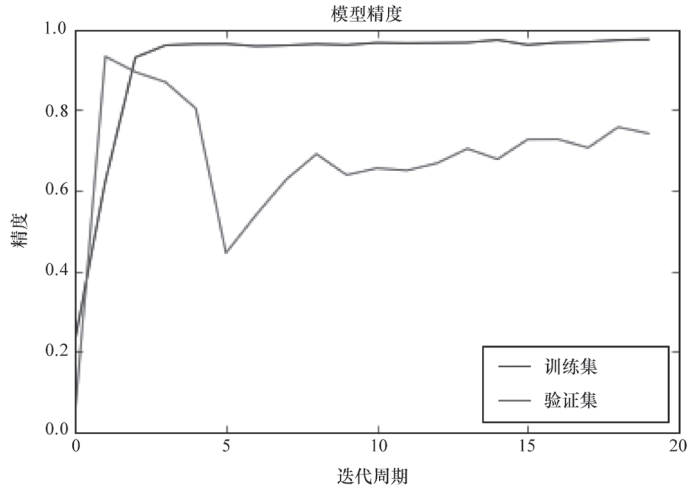


图 7.9　花朵数据集的训练结果

10）现在可以检查模型在看不见的测试集上的执行结果：

```
test_generator = batchgen(X_valid, y_valid, 1, False)
preds = model.predict_generator(test_generator, steps=len(X_valid))

y_valid_ = [np.argmax(x) for x in y_valid]
y_preds = [np.argmax(x) for x in preds]
accuracy_score(y_valid_, y_preds)
```

11）最后，绘制一些预测结果（见图 7.10）：

```
    n_predictions = 5
    plt.figure(figsize=(15, 15))
    for i in range(n_predictions):
        plt.subplot(n_predictions, n_predictions, i+1)
        plt.title("{0} ({1})".format(list(set(labels))[np.argmax(preds[i])],
                                     list(set(labels))[np.argmax(y_valid[i])]))
        img = cv2.imread(X_valid[i])
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.axis('off')
        plt.imshow(img)
        plt.tight_layout()
    plt.show()
```

图 7.10　五个测试图像的预测结果以及各个类别的真实标签

在第 14 章　预训练模型中，将向读者展示如何利用最先进的网络进行目标分类。

7.4　目标在图像中的本地化

现在可以对图像中的目标进行分类，下一步是对图像中的目标进行本地化和分类（检测）。在上一个方案使用的数据集中，花朵（物体）清晰可见，大部分在图像的中间，几乎覆盖了整个图像。但是，通常情况并非如此，希望检测图像中的一个或多个目标。在下面的内容中，将展示如何使用深度学习来检测图像中的目标。

将使用带有注释的货车数据集。图像由安装在货车前部的摄像头来拍摄，使用 TensorFlow 来实现目标检测器。

如何去做…

1）先导入函数库：

```
    import numpy as np
    import pandas as pd
    import glob
    import cv2
    import matplotlib.pyplot as plt
```

```
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, Activation, Flatten,
Conv2D, MaxPooling2D, Lambda, Cropping2D
from keras.utils import np_utils
from keras import backend as K
from keras.callbacks import EarlyStopping
from keras import optimizers
```

2）加载数据集并绘制 CSV 文件的第一行：

```
DATA_DIR = 'Data/object-detection-crowdai/'
labels = pd.read_csv(DATA_DIR + 'labels.csv',
usecols=[0,1,2,3,4,5])
# 仅定位货车
labels = labels[labels.Label == 'Truck']
# 仅使用有标注的货车图像
labels =
labels[~labels.Frame.isin(labels.Frame[labels.Frame.duplicated()].v
alues)]
labels.columns=['xmin', 'ymin', 'xmax', 'ymax', 'Frame', 'Label']
labels[30:50]
```

3）为了理解数据集，绘制一些示例图像及其边框（见图 7.11）：

```
image_list = ['1479498416965787036.jpg',
              '1479498541974858765.jpg']

plt.figure(figsize=(15,15))
i=1
for image in image_list:
    plt.subplot(len(image_list), len(image_list), i)
    img_info = labels[labels.Frame == image]
    img = cv2.imread(DATA_DIR + image)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    print(img.shape)
    cv2.rectangle(img, (img_info.xmin,
img_info.ymin),(img_info.xmax, img_info.ymax), (255, 0 , 255), 4)
    print(img_info)
    plt.imshow(img)
    i+=1
plt.show()
```

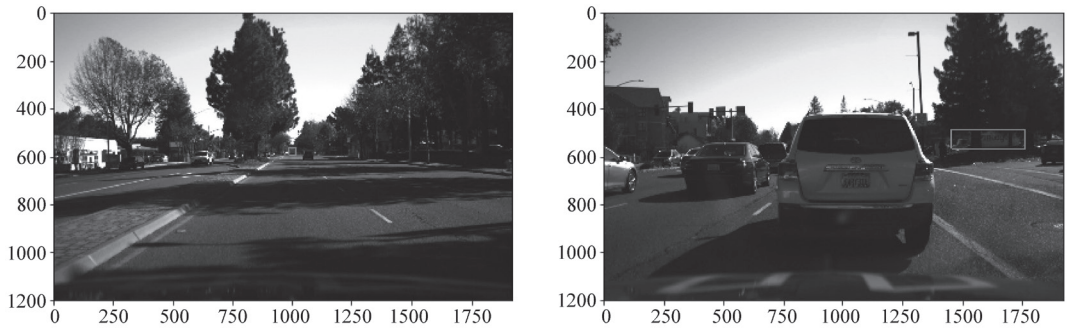


图 7.11　示例图像及其边框

4）接下来，输出关于数据集的一些统计数据：

```
X_train = labels.iloc[:1970]
X_val = labels.iloc[2000:]
print(X_train.shape)
print(X_val.shape)
```

5）此数据集中包含的图像是视频的帧，因此在分割数据进行训练和验证时（为了防止数据泄露），应该倍加小心。大多数图像有重叠（见图 7.12）：

```
image_list = ['1479502622732414408.jpg',
              '1479502623247392322.jpg',
              '1479502623755460204.jpg',
              '1479502623247392322.jpg',
              '1479502625253159017.jpg']
n_images = len(image_list)

plt.figure(figsize=(15,15))
for i in range(n_images):
    plt.subplot(n_images, n_images, i+1)
    plt.title("{0}".format(image_list[i]))
    img = cv2.imread(DATA_DIR + 'object-detection-crowdai/'
    + image_list[i])
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.axis('off')
    plt.imshow(img)
    plt.tight_layout()
plt.show()
```

图 7.12　数据集中的许多图像有重叠部分

6）因此，将基于时间对训练数据和验证数据进行分割：

```
X_train = labels.iloc[:1970] # 选择这个图像帧是因为汽车右转
makes a right turn
X_val = labels.iloc[2000:]
print(X_train.shape)
print(X_val.shape)
```

7）自定义函数用于测量 IOU[⊖]，如下所示：

```
def IOU_calc(y_true, y_pred, smooth=0.9):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return 2*(intersection + smooth) / (K.sum(y_true_f) +
K.sum(y_pred_f) + smooth)
```

8）现在，定义模型架构：

```
img_rows = 200
img_cols = 200
img_channels = 3

model = Sequential()
model.add(Lambda(lambda x: (x / 255.) - 0.5, input_shape=(img_rows,
img_cols, img_channels)))
model.add(Conv2D(16, (5, 5), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(32, (5, 5), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4, activation='sigmoid'))

# 定义优化器并编译
opt = optimizers.Adam(lr=1e-8)
model.compile(optimizer=opt, loss='mse', metrics=[IOU_calc])
model.summary()
```

9）为了确保训练数据适合内存容量，可以使用批量生成器：

⊖ IOU 是交并比的缩略语，即模型产生的目标窗口与原来标记窗口的交叠率，衡量目标检测的精度。——译者注

```
def batchgen(x, y, batch_size, transform=False):
    # 创建空值numpy数组
    images = np.zeros((batch_size, img_rows, img_cols,
img_channels))
    class_id = np.zeros((batch_size, 4))#len(y[0])))

    while 1:
        for n in range(batch_size):
            i = np.random.randint(len(x))
            x_ = x.Frame.iloc[i]
            x_ = cv2.imread(DATA_DIR + image)
            x_ = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            x_min = (x.iloc[i].xmin * (img_cols/1920)) / img_cols
            x_max = (x.iloc[i].xmax * (img_cols/1920)) / img_cols
            y_min = (x.iloc[i].ymin * (img_rows/1200)) / img_rows
            y_max = (x.iloc[i].ymax * (img_rows/1200)) / img_rows
            y_ = (x_min, y_min, x_max, y_max)
            x_ = cv2.resize(x_, (img_cols, img_rows))
            images[n] = x_
            class_id[n] = y_
            yield images, class_id
```

10）为了确保模型不会过拟合训练数据，将使用早停方法：

```
callbacks = [EarlyStopping(monitor='val_IOU_calc', patience=10,
verbose=0)]
```

11）现在可以训练所建模型，并将训练结果存储在历史记录中：

```
batch_size = 64
n_epochs = 1000
steps_per_epoch = 512
val_steps = len(X_val)

train_generator = batchgen(X_train, _, batch_size, True)
val_generator = batchgen(X_val, _, batch_size, False)

history = model.fit_generator(train_generator,
                              steps_per_epoch=steps_per_epoch,
                              epochs=n_epochs,
                              validation_data=val_generator,
                              validation_steps = val_steps,
                             callbacks=callbacks
                            )
```

7.5 实时检测框架

在一些应用中，之前实现的检测模型是不够的。对于这些应用来说，为了能够尽快启动，实时感知是重要的，这需要一个能够近实时地检测物体的模型。最流行的实时检测框架是 faster- RCNN、YOLO（**您仅需查看一次**）和 SSD（**单发多盒检测器**）。Faster-RCNN 在基准测试中表现出更高的精度，但是 YOLO 能够更快地推断出预测结果。

7.6 用 U-net 将图像分类

在之前的方案中，主要通过预测边界框来定位目标。但是在某些情况下，需要知道目标的确切位置，同时目标周围的边界框不够用，也称之为分割，即给目标加上掩模。为了预测物体的掩模，将使用流行的 U-net 模型结构。通过赢得多个图像分割比赛，U-net 模型已被证明是最先进的。 U-net 模型是一种特殊类型的编码器—解码器网络，具有跳转连接、卷积分块和卷积泛化的功能。

在下面的内容中，将向读者展示如何分割图像中的目标。具体来说，就是分割背景。为实现 U-net 网络架构，使用 Keras 框架。

如何去做…

1）首先导入所需函数库，如下所示：

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import glob

from keras.layers import Input, merge, Conv2D, MaxPooling2D,
UpSampling2D, Dropout, Cropping2D, merge
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras import backend as K
from keras.models import Model
```

2）然后，需要存储所有的训练文件名：

```
import os
filenames = []
for path, subdirs, files in os.walk('Data/1obj'):
    for name in files:
        if 'src_color' in path:
            filenames.append(os.path.join(path, name))
print('# Training images: {}'.format(len(filenames)))
```

3）绘制一些示例训练图像和它们的掩模（见图 7.13）：

```
n_examples = 3
for i in range(n_examples):
    plt.subplot(2, 2, 1)
    image = cv2.imread(filenames[i])
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    plt.imshow(image)

    plt.subplot(2, 2, 2)
    mask_file = filenames[i].replace('src_color', 'human_seg')
    mask = cv2.imread(glob.glob(mask_file[:-4]+'*')[0])
    ret, mask = cv2.threshold(mask, 0, 255, cv2.THRESH_BINARY_INV)
    mask = mask[:,:,0]
    plt.imshow((mask), cmap='gray')

    plt.show()
```

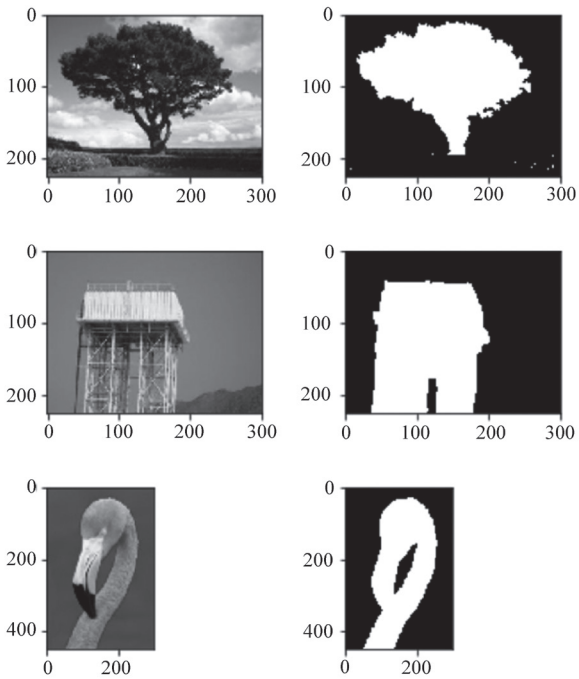


图 7.13　示例训练图像及其掩模

4）为了确定网络的性能和损失函数，将使用 dice 系数。需要实现这些功能，以便能够用 Keras 模型进行编译：

```
def dice_coef(y_true, y_pred, smooth=0.9):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return (2. * intersection + smooth) / (K.sum(y_true_f) +
K.sum(y_pred_f) + smooth)


def dice_coef_loss(y_true, y_pred):
    return -dice_coef(y_true, y_pred)
```

5）接下来，定义 U-net 模型架构：

```
img_rows = 240
img_cols = 240
img_channels = 3

inputs = Input((img_rows, img_cols, img_channels))

conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(inputs)
conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv1)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool1)
conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv2)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool2)
conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv3)
pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)


conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool3)
conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv4)
drop4 = Dropout(0.5)(conv4)
```

```
pool4 = MaxPooling2D(pool_size=(2, 2))(drop4)

conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool4)
conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv5)
drop5 = Dropout(0.5)(conv5)

up6 = Conv2D(512, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size =
(2,2))(drop5))
merge6 = merge([drop4,up6], mode = 'concat', concat_axis = 3)
conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge6)
conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv6)

up7 = Conv2D(256, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size =
(2,2))(conv6))
merge7 = merge([conv3,up7], mode = 'concat', concat_axis = 3)
conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge7)
conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv7)

up8 = Conv2D(128, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size =
(2,2))(conv7))
merge8 = merge([conv2,up8], mode = 'concat', concat_axis = 3)
conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge8)
conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv8)

up9 = Conv2D(64, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size =
(2,2))(conv8))
merge9 = merge([conv1,up9], mode = 'concat', concat_axis = 3)
conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge9)
conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv9)
conv9 = Conv2D(2, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv9)
conv10 = Conv2D(1, 1, activation = 'sigmoid')(conv9)

model = Model(input = inputs, output = conv10)
opt = Adam()
model.compile(optimizer = opt, loss=dice_coef_loss, metrics =
[dice_coef])
model.summary()
```

6）加载训练数据：

```
X = np.ndarray((len(filenames), img_rows, img_cols , img_channels),
dtype=np.uint8)
y = np.ndarray((len(filenames), img_rows, img_cols , 1),
dtype=np.uint8)
i=0
for image in filenames:
    img = cv2.imread(image)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = cv2.resize(img, (240,240))
    mask_file = image.replace('src_color', 'human_seg')
    label = cv2.imread(glob.glob(mask_file[:-4]+'*')[0], 0)
    ret, label = cv2.threshold(label, 0, 255,
cv2.THRESH_BINARY_INV)
    label = cv2.resize(img, (240, 240))
    label = label[:,:,0].reshape((240, 240, 1))
    img = np.array([img/255.])
    label = np.array([label])
    X[i] = img
    y[i] = label
    i+=1
```

7）最后，可以开始训练所建模型，如下所示：

```
n_epochs = 1
batch_size = 1
history = model.fit(X, y, batch_size=batch_size, epochs=10,
verbose=1, shuffle=True, validation_split=0.1)
```

7.7 语义分割与场景理解

在前面的内容中，着重于分割一个或两个特定的类。但是，在某些情况下，需要分割图像中的所有类以了解完整的场景。例如，对于自动驾驶汽车，重要的是汽车周围的所有物体都是分割的图像。在下面的内容中，出于性能方面的考虑，将分割一个类。但是，通过这个网络，可以直接扩展到多个类别。将使用的网络架构称为全卷积网络，因为在模型中只使用卷积层。将使用 VGG16 和 TensorFlow 框架的预训练网络权重。

如何去做…

1）首先，从加载函数库开始，如下所示：

```
import os
import glob
import tensorflow as tf
```

2）因为这里的任务比输出预测类稍微复杂一些，所以需要定义一个从不同网络层取值的函数：

```
def extract_layers(vgg_layer3_out, vgg_layer4_out,
vgg_layer7_out, n_classes):
    decode_layer1_preskip0 =
tf.layers.conv2d_transpose(vgg_layer7_out,
512, (2, 2), (2, 2), name='decode_layer1_preskip0')
    decode_layer1_preskip1 = tf.layers.conv2d(vgg_layer4_out,
512, (1, 1), (1, 1), name='decode_layer1_preskip1')
    decode_layer1_out = tf.add(decode_layer1_preskip0,
decode_layer1_preskip1, name='decode_layer1_out')
    decode_layer2_preskip0 = tf.layers.conv2d_transpose
(decode_layer1_out, 256, (2, 2), (2, 2),
name='decode_layer2_preskip0')
    decode_layer2_preskip1 = tf.layers.conv2d
(vgg_layer3_out, 256, (1, 1), (1, 1),
name='decode_layer2_preskip1')
    decode_layer2_out = tf.add(decode_layer2_preskip0,
decode_layer2_preskip1, name='decode_layer2_out')
    decode_layer3_out = tf.layers.conv2d_transpose
(decode_layer2_out, 128, (2, 2), (2, 2), name='decode_layer3_out')
    decode_layer4_out = tf.layers.conv2d_transpose
(decode_layer3_out, 64, (2, 2), (2, 2),
name='decode_layer4_out')
    decode_layer5_out = tf.layers.conv2d_transpose
(decode_layer4_out, n_classes, (2, 2), (2, 2), name='fcn_out')
    return decode_layer5_out
```

3）为了有效地使用内存，将只加载一个具有如下定义的批量生成器的函数图像：

```
def batch_generator(batch_size):
    image_paths = glob(os.path.join(data_path, 'image_2', '*.png'))
    label_paths = {
        re.sub(r'_(lane|road)_', '_', os.path.basename(path)): path
        for path in glob(os.path.join(data_path,
        'gt_image_2', '*_road_*.png'))}
    background_color = np.array([255, 0, 0])

    random.shuffle(image_paths)
    for batch_i in range(0, len(image_paths), batch_size):
        images = []
        gt_images = []
        for image_file in image_paths[batch_i:batch_i +
batch_size]:
            gt_image_file =
label_paths[os.path.basename(image_file)]
```

```
            image =
scipy.misc.imresize(scipy.misc.imread(image_file), image_shape)
            gt_image = scipy.misc.imresize(scipy.misc.imread
            (gt_image_file), image_shape)

            gt_bg = np.all(gt_image == background_color, axis=2)
            gt_bg = gt_bg.reshape(*gt_bg.shape, 1)
            gt_image = np.concatenate((gt_bg, np.invert(gt_bg)),
axis=2)

            images.append(image)
            gt_images.append(gt_image)

        yield np.array(images), np.array(gt_images)
```

4）设置超参数：

```
n_classes = 2
image_shape = (160, 576)
n_epochs = 23
batch_size = 16
```

5）现在，开始训练模型：

```
with tf.Session() as sess:
    tf.saved_model.loader.load(sess, path, path)
    vgg_image_input =
sess.graph.get_tensor_by_name('image_input:0')
    vgg_keep_prob = sess.graph.get_tensor_by_name('keep_prob:0')
    vgg_layer3_out = sess.graph.get_tensor_by_name('layer3_out:0')
    vgg_layer4_out = sess.graph.get_tensor_by_name('layer4_out:0')
    vgg_layer7_out = sess.graph.get_tensor_by_name('layer7_out:0')
    temp = set(tf.global_variables())
    out_layer = layers(vgg_layer3_out, vgg_layer4_out,
    vgg_layer7_out, num_classes)
    softmax = tf.nn.softmax(out_layer, name='softmax')
    logits = tf.reshape(out_layer, (-1, num_classes),
name='logits')
    labels = tf.reshape(correct_label, (-1, num_classes))
    cross_entropy_loss =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits
    (logits=logits, labels=labels))
    train_op =
tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy_loss)
```

```
    sess.run(tf.variables_initializer(set(tf.global_variables()) -
temp))
    for i in range(n_epochs):
        batches = batch_generator(batch_size)
        epoch_loss = 0
        epoch_size = 0
        for batch_input, batch_label in batches:
            _, loss = sess.run([train_op, cross_entropy_loss],
           feed_dict={input_image: batch_input,
correct_label: batch_label,
keep_prob: 0.5,
learning_rate: 1e-4})
            epoch_loss += loss * len(batch_input)
            epoch_size += len(batch_input)
        print("Loss at epoch {}: {}".format(i,
epoch_loss/epoch_size))
```

6）最后，输出预测的图像分割结果：

```
for image_file in glob(os.path.join(data_path, 'image_2',
'*.png')):
    image = scipy.misc.imresize(scipy.misc.imread(image_file),
image_shape)

    pred_softmax = sess.run(
        [tf.nn.softmax(logits)],
        {keep_prob: 1.0, image_pl: [image]})
    pred_softmax = pred_softmax[0][:, 1].reshape(image_shape[0],
image_shape[1])
    segmentation = (pred_softmax > 0.5).reshape(image_shape[0],
image_shape[1], 1)
    mask = np.dot(segmentation, np.array([[0, 255, 0, 127]]))
    mask = scipy.misc.toimage(mask, mode="RGBA")
    street_im = scipy.misc.toimage(image)
    street_im.paste(mask, box=None, mask=mask)
```

在图 7.14 中，可以看到模型的输出。

图 7.14　预测道路图像分割的示例

7.8 寻找人脸面部关键点

传统计算机视觉中最常用的应用之一是检测图像中的人脸，这为不同行业提供了许多解决方案。第一步是检测图像（或帧）中的**面部关键点**。这些面部关键点，也称为人脸面部标志，已被证明在人脸图像定位和面部朝向检测方面是独特和准确的。HOG + 线性 SVM 等传统的计算机视觉技术和机器学习技术仍然被广泛加以使用。在下面的内容中，将向读者展示如何使用深度学习来做到这一点。具体来说，将实施 CNN 检测人脸面部关键点。之后，将向读者展示如何使用这些关键点进行**头部姿态估计**、**脸部变形**以及使用 OpenCV 进行**跟踪**。

如何去做…

1）首先导入所有必要的函数库并设置种子，如下所示：

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

from sklearn.model_selection import train_test_split
import tensorflow as tf

SEED = 2017
```

2）接下来，加载数据集并输出一些数字：

```
DIR = 'Data/faces/'
training_file = pd.read_csv(DIR + 'training.csv')

cols = training_file.columns[:-1]
training_file['Image'] = training_file['Image'].apply(lambda Image:
np.fromstring(Image, sep=' '))
training_file = training_file.dropna()
```

3）在继续之前，需要重塑和标准化数据：

```
img_cols = 96
img_rows = 96
img_channels = 1
n_labels = 30 # (x, y)对成对15倍

X = np.vstack(training_file['Image'])
X = X.reshape(-1, img_cols, img_rows, 1)
y = training_file[cols].values

X = X / 255.
y = y / 96.

print(X.shape, y.shape)
```

4）把训练集随机分成一个训练集和一个验证集：

```
 X_train, X_val, y_train, y_val = train_test_split(X, y,
test_size=0.2, random_state=SEED)
```

5）绘制五个示例图像并标记人脸面部关键点（见图 7.15）：

```
plt.figure(figsize=(15, 15))

n_examples = 5
for i in range(n_examples):
    plt.subplot(n_examples, n_examples, i+1)
    rand = np.random.randint(len(X_train))
    img = X_train[rand].reshape(img_cols, img_rows)
    plt.imshow(img, cmap='gray')
    kp = y_train[rand]
    plt.scatter(kp[0::2] * img_cols, kp[1::2] * img_rows)

plt.show()
```

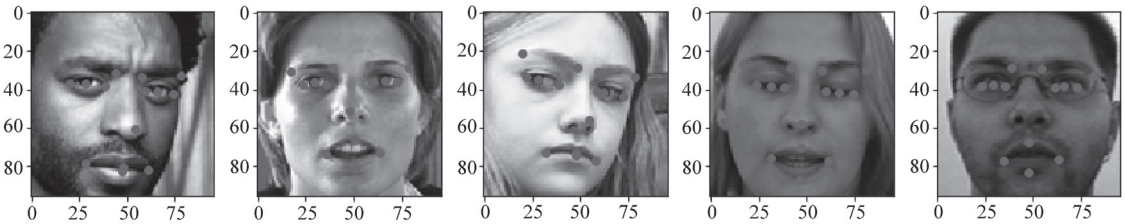


图 7.15　训练集和标签的随机示例

6）现在可以在一个函数中定义网络架构，以便可以重新用它来进行训练和测试：

```
 def model(data, dropout=1.0):
    conv = tf.nn.conv2d(data, conv1_weights, strides=[1, 1, 1, 1],
    padding='SAME')
    relu = tf.nn.relu(tf.nn.bias_add(conv, conv1_biases))
    pool = tf.nn.max_pool(relu, ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1], padding='SAME')

    conv = tf.nn.conv2d(pool, conv2_weights, strides=[1, 1, 1, 1],
    padding='SAME')
    relu = tf.nn.relu(tf.nn.bias_add(conv, conv2_biases))
    pool = tf.nn.max_pool(relu, ksize=[1, 2, 2, 1], strides=
    [1, 2, 2, 1], padding='SAME')

    pool_shape = pool.get_shape().as_list()
    reshape = tf.reshape(pool, [pool_shape[0],
    pool_shape[1] * pool_shape[2] * pool_shape[3]])

    hidden1 = tf.nn.relu(tf.matmul(reshape, fc1_weights) +
```

```
fc1_biases)
    hidden1 = tf.nn.dropout(hidden1, dropout, seed=SEED)

    hidden2 = tf.nn.relu(tf.matmul(hidden1, fc2_weights) +
fc2_biases)
    hidden2 = tf.nn.dropout(hidden2, dropout, seed=SEED)
    output = tf.matmul(hidden2, fc3_weights) + fc3_biases

    return output
```

7）定义超参数：

```
batch_size = 128
n_epochs = 500
learning_rate = 1e-4
print_every = 10
early_stopping_patience = 5
```

8）在构建模型之前，需要设置占位符：

```
inputs = tf.placeholder(tf.float32, shape=(batch_size,
img_cols, img_rows, img_channels))
targets = tf.placeholder(tf.float32, shape=(batch_size, n_labels))

evals = tf.placeholder(tf.float32, shape=(batch_size,
img_cols, img_rows, img_channels))

conv1_weights = tf.Variable(tf.truncated_normal([5, 5,
img_channels, 32], stddev=0.1,seed=SEED))
conv1_biases = tf.Variable(tf.zeros([32]))

conv2_weights = tf.Variable(tf.truncated_normal([5, 5, 32, 64],
stddev=0.1, seed=SEED))
conv2_biases = tf.Variable(tf.constant(0.1, shape=[64]))

fc1_weights = tf.Variable(tf.truncated_normal([img_cols //
4 * img_rows // 4 * 64, 512], stddev=0.1, seed=SEED))
fc1_biases = tf.Variable(tf.constant(0.1, shape=[512]))

fc2_weights = tf.Variable(tf.truncated_normal([512, 512],
stddev=0.1, seed=SEED))
fc2_biases = tf.Variable(tf.constant(0.1, shape=[512]))

fc3_weights = tf.Variable(tf.truncated_normal([512, n_labels],
stddev=0.1, seed=SEED))
fc3_biases = tf.Variable(tf.constant(0.1, shape=[n_labels]))
```

9）现在，可以初始化模型并开始训练，包括采用早停方法：

```
val_size = X_val.shape[0]

train_prediction = model(inputs, 0.5)
loss = tf.reduce_mean(tf.reduce_sum(tf.square(train_prediction -
targets), 1))
eval_prediction = model(evals)

train_step = tf.train.AdamOptimizer(learning_rate).minimize(loss)

init = tf.global_variables_initializer()
sess = tf.InteractiveSession()
sess.run(init)

history = []
patience_count = 0
for epoch in range(n_epochs):
    for step in range(int(len(X_train) / batch_size)):
        offset = step * batch_size
        batch_data = X_train[offset:(offset + batch_size), ...]
        batch_labels = y_train[offset:(offset + batch_size)]

        feed_dict = {inputs: batch_data, targets: batch_labels}
        _, loss_train = sess.run([train_step, loss],
feed_dict=feed_dict)
    predictions = np.ndarray(shape=(val_size, n_labels),
dtype=np.float32)
    for begin in range(0, val_size, batch_size):
        end = begin + batch_size
        if end <= val_size:
            predictions[begin:end, :] = sess.run(eval_prediction,
feed_dict={evals: X_val[begin:end, ...]})
        else:
            batch_predictions = sess.run(eval_prediction,
feed_dict={evals: X_val[-batch_size:, ...]})
            predictions[begin:, :] = batch_predictions[begin -
val_size:, :]
    loss_val = np.sum(np.power(predictions - y_val, 2)) /
(2 * predictions.shape[0])
    history.append(loss_val)
    if epoch % print_every == 0:
        print('Epoch {:04d}: train loss {:.8f};
validation loss {:.8f}'.format(epoch, loss_train, loss_val))
    if epoch > 0 and history[epoch-1] > history[epoch]:
        patience_count = 0
    else:
        patience_count += 1
    if patience_count > early_stopping_patience:
        break
```

10）为了验证，可以用预测的关键点和真值来绘制一些示例图像（见图 7.16）：

```
 plt.figure(figsize=(15, 15))

n_examples = 5
for i in range(n_examples):
    plt.subplot(n_examples, n_examples, i+1)
    rand = np.random.randint(len(X_val))
    img = X_val[rand].reshape(img_cols, img_rows)
    plt.imshow(img, cmap='gray')
    kp = y_val[rand]
    pred = predictions[rand]
    plt.scatter(kp[0::2] * img_cols, kp[1::2] * img_rows)
    plt.scatter(pred[0::2] * img_cols, pred[1::2] * img_rows)

plt.show()
```

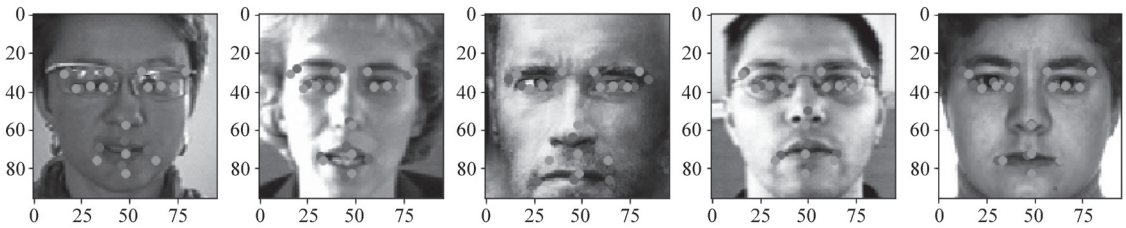


图 7.16　预测人脸和实际人脸面部关键点的示例

7.9　人脸识别

在之前的内容中，演示了如何使用神经网络来检测人脸面部关键点。在下面的内容中，将展示如何使用深度神经网络来识别人脸。通过从头开始对分类器进行训练，获得了很大的灵活性。

如何去做…

1）像往常一样，从导入函数库和设置种子开始：

```
import glob
import re
import matplotlib.pyplot as plt
import numpy as np
import cv2
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
from keras.models import Model
from keras.layers import Flatten, Dense, Input,
GlobalAveragePooling2D, GlobalMaxPooling2D, Activation
from keras.layers import Convolution2D, MaxPooling2D
from keras import optimizers
from keras import backend as K

seed = 2017
```

2）在下面的步骤中，加载数据并输出一些示例图像来了解数据（见图 7.17）：

```
 DATA_DIR = 'Data/lfw/'
images = glob.glob(DATA_DIR + '*/*.jpg')

plt.figure(figsize=(10, 10))

n_examples = 5
for i in range(5):
    rand = np.random.randint(len(images))
    image_name = re.search('Data/lfw\/(.+?)\/', images[rand],
re.IGNORECASE).group(1).replace('_', ' ')
    img = cv2.imread(images[rand])
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.subplot(n_examples, n_examples, i+1)
    plt.title(image_name)
    plt.imshow(img)
plt.show()
```

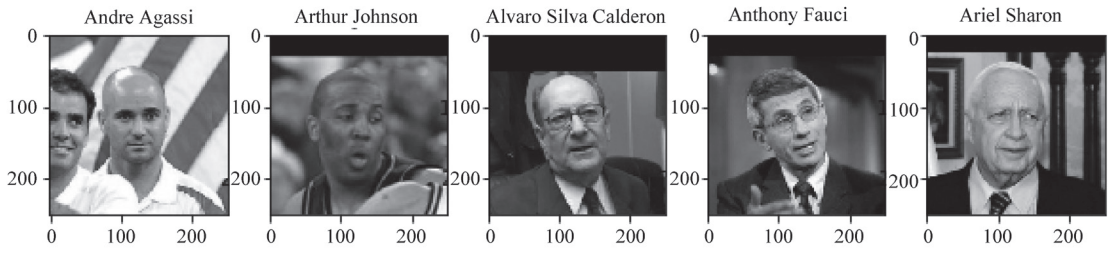


图 7.17　人脸示例图像

3）有些人（标签）有多个图像，绘制这些图像，看看它们是否相似（见图 7.18）：

```
 images_arnold = glob.glob(DATA_DIR +
'Arnold_Schwarzenegger/*.jpg')

plt.figure(figsize=(10, 10))

for i in range(n_examples):
    image_name = re.search('Data/lfw\/(.+?)\/', images_arnold[i],
re.IGNORECASE).group(1).replace('_', ' ')
    img = cv2.imread(images_arnold[i])
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.subplot(n_examples, n_examples, i+1)
# plt.title(image_name)
    plt.imshow(img)
plt.show()
```

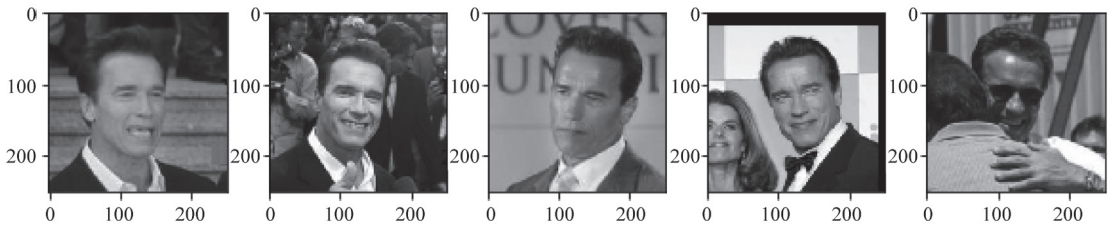


图 7.18　阿诺德 · 施瓦辛格的图像示例

可以看到在背景和脸部角度方面，图像是完全不同的。另外，在第五个例子中，脸部的一部分因为佩戴眼镜而被遮盖。

4）图像整齐地存储在每个类（person）的单独文件夹中，所以可以用一个简单的正则表达式来提取标签：

```
labels = np.asarray([re.search('Data/lfw\/(.+?)\/', image,
re.IGNORECASE).group(1) for image in np.asarray(images)])
```

5）输出关于数据集的一些统计数据：

```
print('Number of images: {}'.format(len(y)))
print('Number of unique labels: {}'.format(len(np.unique(labels))))
```

6）现在准备对标签进行预处理：

```
encoder = LabelBinarizer()
encoder.fit(labels)
y = encoder.transform(labels).astype(float)
```

7）为了验证模型，将使用 20%的分割来形成验证集：

```
 X_train, X_val, y_train , y_val = train_test_split(X, y,
test_size=0.2, random_state=seed)
```

8）接下来定义模型，将采用由著名 VGG16 网络所启发的网络架构：

```
 input_shape = (250, 250, 3)
img_input = Input(shape=input_shape)
inputs = img_input

# 块 1
x = Convolution2D(64, (3, 3), activation='relu', padding='same',
name='conv1_1')(img_input)
x = Convolution2D(64, (3, 3), activation='relu', padding='same',
name='conv1_2')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='pool1')(x)

# 块 2
x = Convolution2D(128, (3, 3), activation='relu', padding='same',
name='conv2_1')(x)
x = Convolution2D(128, (3, 3), activation='relu', padding='same',
name='conv2_2')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='pool2')(x)

# 块 3
x = Convolution2D(256, (3, 3), activation='relu', padding='same',
name='conv3_1')(x)
x = Convolution2D(256, (3, 3), activation='relu', padding='same',
name='conv3_2')(x)
x = Convolution2D(256, (3, 3), activation='relu', padding='same',
name='conv3_3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='pool3')(x)

# 块 4
x = Convolution2D(512, (3, 3), activation='relu', padding='same',
name='conv4_1')(x)
x = Convolution2D(512, (3, 3), activation='relu', padding='same',
name='conv4_2')(x)
x = Convolution2D(512, (3, 3), activation='relu', padding='same',
name='conv4_3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='pool4')(x)

# 块 5
x = Convolution2D(512, (3, 3), activation='relu', padding='same',
name='conv5_1')(x)
x = Convolution2D(512, (3, 3), activation='relu', padding='same',
name='conv5_2')(x)
x = Convolution2D(512, (3, 3), activation='relu', padding='same',
name='conv5_3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='pool5')(x)
```

```
x = Flatten(name='flatten')(x)
x = Dense(4096, name='fc6')(x)
x = Activation('relu', name='fc6/relu')(x)
x = Dense(4096, name='fc7')(x)
x = Activation('relu', name='fc7/relu')(x)
x = Dense(len(y[0]), name='fc8')(x)
x = Activation('relu', name='fc8/softmax')(x)

model = Model(inputs, x)

opt = optimizers.Adam()
model.compile(loss='categorical_crossentropy', optimizer=opt,
metrics=['accuracy'])
```

9）当前的训练集相当有限。因此，可使用图像增广技术来扩展数据集。将使用函数 random_shifts 随机翻转和随机缩放：

```
def random_shifts(image, shift_max_x=100, shift_max_y=100):
    width, height, _ = image.shape
    shift_x = np.random.randint(shift_max_x)
    shift_y = np.random.randint(shift_max_y)
    M = np.float32([[1, 0, shift_x],[0, 1, shift_y]])
    return (cv2.warpAffine(image, M, (height, width)))
    def random_flip(image, p_flip=0.5):
        rand = np.random.random()
        if rand < p_flip:
            image = cv2.flip(image, 1)
        return image

    def scale_image(image, scale_range=[0.6, 1.4]):
        width, height, _ = image.shape
        scale_x = np.random.uniform(low=scale_range[0],
        high=scale_range[1])
        scale_y = np.random.uniform(low=scale_range[0],
        high=scale_range[1])
        scale_matrix = np.array([[scale_x, 0., (1. - scale_x) *
        width / 2.], [0., scale_y, (1. - scale_y) * height / 2.]],
        dtype=np.float32)
        return(cv2.warpAffine(image, scale_matrix, (width, height),
        flags=cv2.INTER_LINEAR,
        borderMode=cv2.BORDER_REFLECT_101))
```

10）人们不想将所有图像加载到内存中，所以将实现一个使用图像增广功能的批生成器：

```
img_rows = img_cols = 250
img_channels = 3

def batchgen(x, y, batch_size, transform=False):
    # 创建空值numpy数组
    images = np.zeros((batch_size, img_rows, img_cols, img_channels))
    class_id = np.zeros((batch_size, len(y[0])))

    while 1:
        for n in range(batch_size):
            i = np.random.randint(len(x))
            x_ = cv2.imread(x[i])
            x_ = cv2.cvtColor(x_, cv2.COLOR_BGR2RGB)
            if transform:
                x_ = random_shifts(x_, 10, 10)
                x_ = random_flip(x_)
                x_ = scale_image(x_, [0.8, 1,2])
            images[n] = x_
            class_id[n] = y[i]
        yield images, class_id
```

11）对于模型，将定义以下超参数：

```
batch_size = 32
n_epochs = 1000
s_epoch =
val_steps = (len(X_val)/batch_size)
```

12）开始训练所建模型：

```
train_generator = batchgen(X_train, y_train, batch_size, True)
val_generator = batchgen(X_val, y_val, batch_size, False)

history = model.fit_generator(train_generator,
                              steps_per_epoch=s_epoch,
                              epochs=n_epochs,
                              validation_data=val_generator,
                              validation_steps = val_steps,
                              verbose=1
                             )
```

13）现在，看看所建模型如何执行并绘制一些结果（见图 7.19）：

```
 test_generator = batchgen(X_val, y_val, batch_size, False)
preds = model.predict_generator(test_generator, steps=1)

y_val_ = [np.argmax(x) for x in y_val]
y_preds = [np.argmax(x) for x in preds]

accuracy_score(y_val_, y_preds)
```

图 7.19　测试集上的预测图像示例

如果想添加新的人脸到模型，可以使用微调来调整已训练模型的权重。更具体地说，在对新例子进行训练时，将使用稍小的学习率。在第 14 章　预训练模型　中，将演示如何对此方案中的训练模型进行微调。

7.10　将样式转换为图像

在过去的几年中，由于深度学习将风格从一个图像转移到另一个图像已经有了巨大的性能提升。许多人尝试将某种风格，通常是从著名画家的风格转移到照片上。由此产生的图像往往是有趣的，因为它们显示了画家的风格和原始图像之间的混合。在下面的内容中，将向读者展示如何使用 VGG16 的预训练权重将一幅图像的样式转换为另一幅图像的样式。

如何去做…

1）从导入所需的函数库开始，如下所示：

```
from keras.preprocessing.image import load_img, img_to_array
from scipy.misc import imsave
import numpy as np
from scipy.optimize import fmin_l_bfgs_b
import time
import argparse

from keras.applications import vgg16
from keras import backend as K
```

2）接下来，加载将用于样式转换的两个图像并绘制它们（见图 7.20）：

```
base_image_path = 'Data/golden_gate.jpg'
style_reference_image_path = 'Data/starry_night.jpg'
result_prefix = 'result_'

width, height = load_img(base_image_path).size
img_rows = 400
img_cols = 600
img_channels = 3

plt.figure(figsize=(15, 15))
plt.subplot(2, 2, 1)
img = load_img(base_image_path)
```

```
plt.imshow(img)
plt.subplot(2, 2, 2)
img = load_img(style_reference_image_path)
plt.imshow(img)
plt.show()
```

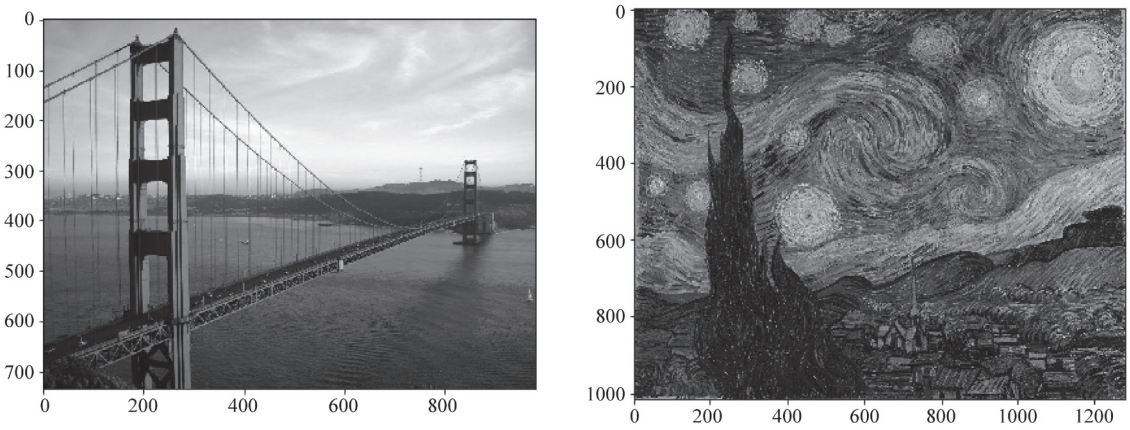


图 7.20　用于样式转换的输入图像（左图为原始图像；右图为样式参考图像）

3）定义两个可以预处理和处理图像的函数：

```
def preprocess_image(image_path):
    img = load_img(image_path, target_size=(img_rows, img_cols))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = trained_model.preprocess_input(img)
    return img

def deprocess_image(x):
    x = x.reshape((img_rows, img_cols, img_channels))
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    x = x[:, :, ::-1]
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

4）现在定义所需要的训练占位符：

```
base_image = K.variable(preprocess_image(base_image_path))
style_reference_image = K.variable(preprocess_image
(style_reference_image_path))
combination_image = K.placeholder((1, img_rows, img_cols, 3))
input_tensor = K.concatenate([base_image,
style_reference_image, combination_image], axis=0)
```

5）导入 VGG16 模型，并为模型的体系结构创建一个字典：

```
model = vgg16.VGG16(input_tensor=input_tensor, weights='imagenet',
include_top=False)
model_dict = dict([(layer.name, layer.output)
for layer in model.layers])
```

6）对于风格转移，需要定义三种不同的损失函数：

```
 def gram_matrix(x):
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram

def style_loss(style, combination):
    assert K.ndim(style) == 3
    assert K.ndim(combination) == 3
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_nrows * img_ncols
    return K.sum(K.square(S - C)) /
    (4. * (channels ** 2) * (size ** 2))

def content_loss(base, combination):
    return K.sum(K.square(combination - base))

def total_variation_loss(x):
    a = K.square(x[:, :img_nrows - 1, :img_ncols - 1, :] -
x[:, 1:, :img_ncols - 1, :])
    b = K.square(x[:, :img_nrows - 1, :img_ncols - 1, :] -
x[:, :img_nrows - 1, 1:, :])
    return K.sum(K.pow(a + b, 1.25))
```

7）定义超参数：

```
total_variation_weight = 1.0
style_weight = 1.0
content_weight = 0.25
iterations = 10
```

8）接下来，需要创建占位符函数来确定训练期间的损失：

```
loss = K.variable(0.)
layer_features = model_dict['block5_conv2']
base_image_features = layer_features[0, :, :, :]
combination_features = layer_features[2, :, :, :]
loss += content_weight * content_loss(base_image_features,
combination_features)

feature_layers = ['block1_conv1', 'block2_conv1',
'block3_conv1', 'block4_conv1', 'block5_conv1']
for layer_name in feature_layers:
    layer_features = model_dict[layer_name]
    style_reference_features = layer_features[1, :, :, :]
    combination_features = layer_features[2, :, :, :]
    sl = style_loss(style_reference_features, combination_features)
    loss += (style_weight / len(feature_layers)) * sl
loss += total_variation_weight *
total_variation_loss(combination_image)

grads = K.gradients(loss, combination_image)

outputs = [loss]
if isinstance(grads, (list, tuple)):
    outputs += grads
else:
    outputs.append(grads)

f_outputs = K.function([combination_image], outputs)
```

9）还需要定义函数来评估损失：

```
 def eval_loss_and_grads(x):
    x = x.reshape((1, img_nrows, img_ncols, 3))
    outs = f_outputs([x])
    loss_value = outs[0]
    if len(outs[1:]) == 1:
        grad_values = outs[1].flatten().astype('float64')
    else:
        grad_values =
np.array(outs[1:]).flatten().astype('float64')
    return loss_value, grad_values


class Evaluator(object):

    def __init__(self):
        self.loss_value = None
        self.grads_values = None
```

```
    def loss(self, x):
        assert self.loss_value is None
        loss_value, grad_values = eval_loss_and_grads(x)
        self.loss_value = loss_value
        self.grad_values = grad_values
        return self.loss_value

    def grads(self, x):
        assert self.loss_value is not None
        grad_values = np.copy(self.grad_values)
        self.loss_value = None
        self.grad_values = None
        return grad_values

evaluator = Evaluator()
```

10）开始样式转换并绘制每次迭代的结果（见图 7.21）：

```
 x = preprocess_image(base_image_path)

for i in range(iterations):
    print(i)
    x, min_val, info = fmin_l_bfgs_b(evaluator.loss, x.flatten(),
fprime=evaluator.grads, maxfun=20)

    print('Current loss value:', min_val)

    img = deprocess_image(x.copy())
    fname = result_prefix + '_at_iteration_%d.png' % i
    plt.imshow(img)
    plt.show()
```

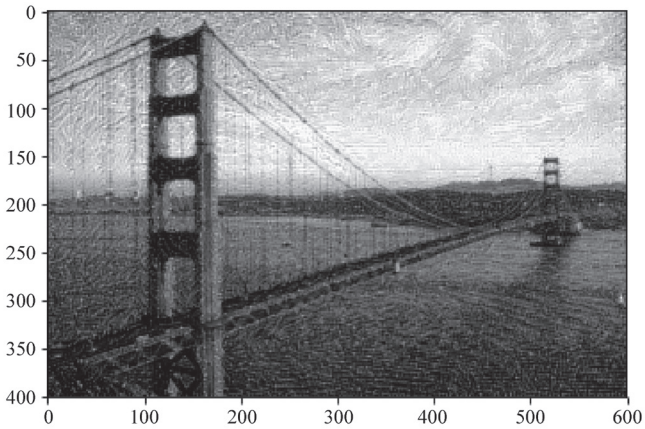


图 7.21　迭代 10 次后转换样式的示例

第 8 章
自然语言处理

本章包含与文本数据处理相关的内容，主要包括文本特征表示和处理、文字嵌入以及文本数据存储相关的方法：

- 情绪分析；
- 句子翻译；
- 文本摘要。

8.1 简介

深度学习另一个革命性的领域是**自然语言处理**（NLP）。这场革命主要由 RNN 的发表而引起，但 CNN 在处理文本时也被证明是有价值的。深度学习可被用于许多不同的 NLP 场景中，从分析 Twitter 供稿中的情感分析到文本摘要。本章将向读者展示如何应用深度学习来解决这些问题。

8.2 情绪分析

在这个时代，有越来越多的数据产生，尤其是每个人都可以在互联网上发表自己的观点，大规模自动分析这些帖子的价值对企业和政策来说是非常重要的。在第 4 章　递归神经网络　中，已经展示了如何应用 RNN 和 LSTM 神经元对短句进行分类，比如电影评论。在下面的内容中，将通过对 Twitter 消息的情感进行分类来逐渐进行复杂性分析。本章将通过预测二进制类和细粒度类来实现这一点。

如何去做…

1）首先导入所需函数库，如下所示：

```
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
import numpy as np
import random
import pickle
from collections import Counter

import tensorflow as tf
```

2）接下来，用 Nltk 包处理英文句子，并开始定义所需要的预处理函数：

```
lemmatizer = WordNetLemmatizer()
pos = '../Data/positive.txt'
neg = '../Data/negative.txt'
def create_lexicon(pos, neg):
    lexicon = []
    for fi in [pos, neg]:
        with open(fi, 'r') as f:
            contents = f.readlines()
            for l in contents[:10000000]:
                all_words = word_tokenize(l.lower())
                lexicon += list(all_words)
    lexicon = [lemmatizer.lemmatize(i) for i in lexicon]
    w_counts = Counter(lexicon)
    l2 =[]
    for w in w_counts:
        if 1000 > w_counts[w] > 50:
            l2.append(w)
    return l2
def sample_handling(sample,lexicon,classification):
    featureset = []
    with open(sample,'r') as f:
        contents = f.readlines()
        for l in contents[:10000000]:
            current_words = word_tokenize(l.lower())
            current_words = [lemmatizer.lemmatize(i) for i in
            current_words]
            features = np.zeros(len(lexicon))
            for word in current_words:
                if word.lower() in lexicon:
                    index_value = lexicon.index(word.lower())
                    features[index_value] += 1
            features = list(features)
            featureset.append([features,classification])
    return featureset
```

3）接下来，按如下步骤处理数据：

```
lexicon = create_lexicon(pos,neg)
features = []
features += sample_handling(pos, lexicon,[1,0])
features += sample_handling(neg, lexicon,[0,1])
random.shuffle(features)
features = np.array(features)
```

```
testing_size = int(0.1*len(features))

X_train = list(features[:,0][:-testing_size])
y_train = list(features[:,1][:-testing_size])
X_test = list(features[:,0][-testing_size:])
y_test = list(features[:,1][-testing_size:])
```

4）在实现模型之前，需要设置超参数：

```
n_epochs = 10
batch_size = 128
h1 = 500
h2 = 500
n_classes = 2
```

5）接下来，定义输入占位符，对权重和偏置参数进行初始化：

```
x_input = tf.placeholder('float')
y_input = tf.placeholder('float')

hidden_1 = {'weight':tf.Variable(tf.random_normal([len(X_train[0]),
h1])),
                  'bias':tf.Variable(tf.random_normal([h1]))}

hidden_2 = {'weight':tf.Variable(tf.random_normal([h1, h2])),
                  'bias':tf.Variable(tf.random_normal([h2]))}
output_layer = {'weight':tf.Variable(tf.random_normal([h2,
                n_classes])),
                'bias':tf.Variable(tf.random_normal([n_classes])),}
```

6）定义网络结构、损失和优化器，如下：

```
l1 = tf.add(tf.matmul(x_input, hidden_1['weight']),
hidden_1['bias'])
l1 = tf.nn.relu(l1)
l2 = tf.add(tf.matmul(l1, hidden_2'weight']), hidden_2['bias'])
l2 = tf.nn.relu(l2)
output = tf.matmul(l2, output_layer['weight']) +
output_layer['bias']

loss =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=outpu
t, labels=y_input))
opt = tf.train.AdamOptimizer().minimize(loss)
```

7）最后，开始训练模型：

```
with tf.Session() as sess:
 sess.run(tf.global_variables_initializer())

 for epoch in range(n_epochs):
 epoch_loss = 0
 i = 0
 while i < len(X_train):
 start = i
 end = i + batch_size
 batch_x = np.array(X_train[start:end])
 batch_y = np.array(y_train[start:end])

 _, batch_loss = sess.run([opt, loss], feed_dict={x_input: batch_x,
y_input: batch_y})
 epoch_loss += batch_loss
 i += batch_size

 print('Epoch {}: loss {}'.format(epoch, epoch_loss))
```

8.3 句子翻译

得益于几个大型搜索引擎，深度学习在文本中的另一个应用在性能和知名度方面得到了提高。翻译句子是一项艰巨的任务，因为每种语言都有自己的规则、例外和表达方式。人们往往遗忘的是，词语的翻译在很大程度上取决于语境。一些人认为，彻底解决语言问题可能是实现普通人工智能的一个巨大里程碑，因为智能和语言是相互关联的，语言中的细微差别对理解是至关重要的。

在下面的内容中，将展示如何使用序列到序列模型将英语翻译成法语，将使用 CNTK 框架。

如何去做…

1）开始加载函数库：

```
import numpy as np
import os

from cntk import Trainer, Axis
from cntk.io import MinibatchSource, CTFDeserializer, StreamDef,
StreamDefs, INFINITELY_REPEAT
from cntk.learners import momentum_sgd, fsadagrad,
momentum_as_time_constant_schedule, learning_rate_schedule,
UnitType
from cntk import input, cross_entropy_with_softmax,
classification_error, sequence,
                 element_select, alias, hardmax, placeholder,
```

```
    combine, parameter, times, plus
    from cntk.ops.functions import CloneMethod, load_model, Function
    from cntk.initializer import glorot_uniform
    from cntk.logging import log_number_of_parameters, ProgressPrinter
    from cntk.logging.graph import plot
    from cntk.layers import *
    from cntk.layers.sequence import *
    from cntk.layers.models.attention import *
    from cntk.layers.typing import *
```

2）首先，需要加载所使用的数据集，它包含英语和法语句子：

```
    data_eng = '../Data/translations/small_vocab_en'
    data_fr = '../Data/translations/small_vocab_fr'

    with open(data_eng, 'r', encoding='utf-8') as f:
        sentences_eng = f.read()
    with open(data_fr, 'r', encoding='utf-8') as f:
        sentences_fr = f.read()
```

3）输出关于数据集的一些统计数据：

```
    word_counts = [len(sentence.split()) for sentence in sentences_eng]
    print('Number of unique words in English: {}'.format(len({word: None for
    word in sentences_eng.lower().split()})))
    print('Number of sentences: {}'.format(len(sentences_eng)))
print('Average number of words in a sentence:
{}'.format(np.average(word_counts)))

n_examples = 5
for i in range(n_examples):
    print('\nExample {}'.format(i))
    print(sentences_eng.split('\n')[i])
    print(sentences_fr.split('\n')[i])
```

4）需要为两种语言创建查找表：一个用于词汇到整数的表列；另一个用于整数到词汇的表列。通过定义函数 create_lookup_tables 来执行此操作：

```
def create_lookup_tables(text):
    vocab = set(text.split())
    vocab_to_int = {'<S>': 0, '<E>': 1, '<UNK>': 2, '<PAD>': 3 }

    for i, v in enumerate(vocab, len(vocab_to_int)):
        vocab_to_int[v] = i

    int_to_vocab = {i: v for v, i in vocab_to_int.items()}

    return vocab_to_int, int_to_vocab

vocab_to_int_eng, int_to_vocab_eng =
create_lookup_tables(sentences_eng.lower())
vocab_to_int_fr, int_to_vocab_fr =
create_lookup_tables(sentences_fr.lower())
```

5）现在已经准备好了所有的查找表，可以开始转换输入（英语句子）和目标（法语句子）数据了：

```
def text_to_ids(source_text, target_text, source_vocab_to_int,
target_vocab_to_int):
   source_id_text = [[source_vocab_to_int[word] for word in
   sentence.split()] for sentence in source_text.split('\n')]
    target_id_text = [[target_vocab_to_int[word] for word in
    sentence.split()]+[target_vocab_to_int['<E>']] for sentence
    in target_text.split('\n')]
    return source_id_text, target_id_text

X, y = text_to_ids(sentences_eng.lower(), sentences_fr.lower(),
vocab_to_int_eng, vocab_to_int_fr)
```

6）在定义模型之前，需要定义超参数：

```
input_vocab_dim = 128
label_vocab_dim = 128
hidden_dim = 256
n_layers = 2
attention_dim = 128
attention_span = 12
embedding_dim = 200
n_epochs = 20
learning_rate = 0.001
batch_size = 64
```

7）对于模型要定义一个函数。在 create_model 函数中，包含 Embedding（嵌入）、Stabilizer（稳定器）、LSTM 层和 AttentionModel（关注模型）：

```
def create_model(n_layers):
    embed = Embedding(embedding_dim, name='embed')
    LastRecurrence = C.layers.Recurrence
    encode = C.layers.Sequential([
        embed,
        C.layers.Stabilizer(),
        C.layers.For(range(num_layers-1), lambda:
            C.layers.Recurrence(C.layers.LSTM(hidden_dim)))
        LastRecurrence(C.layers.LSTM(hidden_dim,        ""))),
        return_full_state=True),
        (C.layers.Label('encoded_h'), C.layers.Label('encoded_c')),
    ])
    with default_options(enable_self_stabilization=True):
        stab_in = Stabilizer()
        rec_blocks = [LSTM(hidden_dim) for i in range(n_layers)]
        stab_out = Stabilizer()
        out = Dense(label_vocab_dim, name='out')
        attention_model = AttentionModel(attention_dim, None, None,
        name='attention_model')

        @Function
        def decode(history, input):
            encoded_input = encode(input)
            r = history
            r = embed(r)
            r = stab_in(r)
            for i in range(n_layers):
                rec_block = rec_blocks[i]
                @Function
                def lstm_with_attention(dh, dc, x):
                    h_att =
attention_model(encoded_input.outputs[0], dh)
                    x = splice(x, h_att)
                    return rec_block(dh, dc, x)
                r = Recurrence(lstm_with_attention)(r)
            r = stab_out(r)
            r = out(r)
            r = Label('out')(r)
            return r

    return decode
```

8）定义损失函数：

```
def create_loss_function(model):
    @Function
    @Signature(input = InputSequence[Tensor[input_vocab_dim]],
labels = LabelSequence[Tensor[label_vocab_dim]])
    def loss (input, labels):
        postprocessed_labels = sequence.slice(labels, 1, 0)
        z = model(input, postprocessed_labels)
        ce = cross_entropy_with_softmax(z, postprocessed_labels)
        errs = classification_error (z, postprocessed_labels)
        return (ce, errs)
    return loss
```

9）最后，开始训练：

```
model = create_model(n_layers)
loss = create_loss_function(model)
learner = fsadagrad(model.parameters,
                    lr = learning_rate,
                    momentum =
momentum_as_time_constant_schedule(1100),
                    gradient_clipping_threshold_per_sample=2.3,
                    gradient_clipping_with_truncation=True)
trainer = Trainer(None, loss, learner)

total_samples = 0
n_batches = len(X)//batch_size

for epoch in range(n_epochs):
    for i in range(n_batches):
        batch_input = train_reader.next_minibatch(minibatch_size)
trainer.train_minibatch(batch_input[train_reader.streams.features],
batch_input[train_reader.streams.labels])
```

8.4 文本摘要

阅读理解（RC）是阅读文本、处理文本并理解其含义的能力。总结有两种类型：提取和抽象。**提取摘要**可以识别重要的文字，并将其余部分丢弃，从而缩短文章的篇幅。根据现实情况，由于文本是从不同的段落中摘取的，因此听起来可能会让人感到奇怪和不连贯。**抽象概括**要困难得多，它需要模型更深入地理解文本和语言。在下面的内容中，将使用 TensorFlow 框架实现文本摘要算法。

如何去做…

1）首先加载所有必要的函数库，如下：

```
import numpy as np
import tensorflow as tf
```

2）其次，加载文本数据：

```
article_filename =
'Data/summary/"Data/sumdata/train/train.article.txt'
title_filename = 'Data/summary/"Data/sumdata/train/train.title.txt'

with open(article_filename) as article_file:
 articles = article_file.readlines()
with open(title_filename) as title_file:
 titles = title_file.readlines()
```

3）为了使数据对模型具有可读性，需要定义一个函数，它为从整数到词汇表列创建一个查询表，反之亦然：

```
def create_lookup_tables(text):
    vocab = set(text.split())
    vocab_to_int = {'<S>': 0, '<E>': 1, '<UNK>': 2, '<PAD>': 3 }

    for i, v in enumerate(vocab, len(vocab_to_int)):
        vocab_to_int[v] = i

    int_to_vocab = {i: v for v, i in vocab_to_int.items()}

    return vocab_to_int, int_to_vocab

vocab_to_int_article, int_to_vocab_article =
create_lookup_tables(articles.lower())
vocab_to_int_title, int_to_vocab_title =
create_lookup_tables(titles.lower())
```

4）接下来，转换输入数据（文章）和目标数据（标题）：

```
def text_to_ids(source_text, target_text, source_vocab_to_int,
target_vocab_to_int):
source_id_text = [[source_vocab_to_int[word] for word in
sentence.split()] for sentence in source_text.split('\n')]
    target_id_text = [[target_vocab_to_int[word] for word in
sentence.split()]+[target_vocab_to_int['<E>']] for sentence in
target_text.split('\n')]
    return source_id_text, target_id_text

X, y = text_to_ids(articles.lower(), titles.lower(),
vocab_to_int_articles, vocab_to_int_titles)
```

5）在定义模型之前，需要设置超参数：

```
learning_rate = 0.001
hidden_units = 400
embedding_size = 200
n_layers = 1
dropout = 0.5
n_iters = 40
```

6）将使用一个双向模型，为此定义一个前向和一个后向的嵌入，并应用一个 Dropout 封装：

```
encoder_forward_cell = tf.contrib.rnn.GRUCell(state_size)
encoder_backward_cell = tf.contrib.rnn.GRUCell(state_size)
decoder_cell = tf.contrib.rnn.GRUCell(state_size)

encoder_forward_cell =
tf.contrib.rnn.DropoutWrapper(encoder_forward_cell,
output_keep_prob = (1-dropout))
encoder_backward_cell =
tf.contrib.rnn.DropoutWrapper(encoder_backward_cell,
output_keep_prob = (1-dropout))
decoder_cell = tf.contrib.rnn.DropoutWrapper(decoder_cell,
output_keep_prob = (1-dropout))
```

7）接下来，将所有元素叠加在一起：

```
with tf.variable_scope("seq2seq", dtype=dtype):
   with tf.variable_scope("encoder"):

        encoder_embedding = tf.get_variable("embedding",
[source_vocab_size, embedding_size],initializer=embedding_init)
        encoder_inputs_embedding =
tf.nn.embedding_lookup(encoder_embedding, self.encoder_inputs)
        encoder_outputs, encoder_states =
tf.nn.bidirectional_dynamic_rnn(encoder_forward_cell,
encoder_backward_cell, encoder_inputs_embedding,
sequence_length=self.encoder_len, dtype=dtype)

    with tf.variable_scope("init_state"):
        init_state = fc_layer(
            tf.concat(encoder_states, 1), state_size)
        # bidirectional_dynamic_rnn 网络现状奇特
        # 无需定义 batch_size(批量大小)
        self.init_state = init_state
        self.init_state.set_shape([self.batch_size, state_size])
        self.att_states = tf.concat(encoder_outputs, 2)
        self.att_states.set_shape([self.batch_size, None,
state_size*2])
```

```
        with tf.variable_scope("attention"):
            attention = tf.contrib.seq2seq.BahdanauAttention(
                state_size, self.att_states, self.encoder_len)
            decoder_cell = tf.contrib.seq2seq.DynamicAttentionWrapper(
                decoder_cell, attention, state_size * 2)
            wrapper_state =
    tf.contrib.seq2seq.DynamicAttentionWrapperState(
                self.init_state, self.prev_att)

        with tf.variable_scope("decoder") as scope:

            decoder_emb = tf.get_variable("embedding",
    [target_vocab_size, embedding_size],initializer=emb_init)

            decoder_cell =
    tf.contrib.rnn.OutputProjectionWrapper(decoder_cell,
    target_vocab_size)

            decoder_inputs_emb = tf.nn.embedding_lookup(decoder_emb,
    self.decoder_inputs)

            helper =
    tf.contrib.seq2seq.TrainingHelper(decoder_inputs_emb,
    self.decoder_len)
            decoder = tf.contrib.seq2seq.BasicDecoder(decoder_cell,
    helper, wrapper_state)

            outputs, final_state =
    tf.contrib.seq2seq.dynamic_decode(decoder)
            outputs_logits = outputs[0]
            self.outputs = outputs_logits

            weights = tf.sequence_mask(self.decoder_len,
    dtype=tf.float32)

            loss_t = tf.contrib.seq2seq.sequence_loss(outputs_logits,
    self.decoder_targets, weights, average_across_timesteps=False,
    average_across_batch=False)
            self.loss = tf.reduce_sum(loss_t) / self.batch_size

            params = tf.trainable_variables()
            opt = tf.train.AdadeltaOptimizer(self.learning_rate,
            epsilon=1e-6)
            gradients = tf.gradients(self.loss, params)
            clipped_gradients, norm = tf.clip_by_global_norm(gradients,
            max_gradient)
            self.updates = opt.apply_gradients(zip(clipped_gradients,
            params), global_step=self.global_step)
```

8）在开始训练之前，需要一个能够检索正确批次的函数：

```
def get_batches(int_text, batch_size, seq_length):
    n_batches = int(len(int_text) / (batch_size * seq_length))
    inputs = np.array(int_text[: n_batches * batch_size *
seq_length])
    outputs = np.array(int_text[1: n_batches * batch_size *
seq_length + 1])

    x = np.split(inputs.reshape(batch_size, -1), n_batches, 1)
    y = np.split(outputs.reshape(batch_size, -1), n_batches, 1)

    return np.array(list(zip(x, y)))
```

9）现在准备训练：

```
with tf.Session() as sess:
       model = create_model(sess, False)
       loss = 0.0
       current_step = sess.run(model.global_step)

       while current_step <= n_iters:
           rand = np.random.random_sample()
           bucket_id = min([i for i in range(len(train_buckets_scale))
                            if train_buckets_scale[i] > rand])

           encoder_inputs, decoder_inputs, encoder_len, decoder_len =
           model.get_batches(train_set, bucket_id)
step_loss, _ = model.step(sess, encoder_inputs,
decoder_inputs, encoder_len, decoder_len, False,
train_writer)
loss += step_loss * batch_size / np.sum(decoder_len)
current_step += 1
```

正如在本章中所展示的那样，可以用相对较小的数据集取得很好的结果。然而，理解语言是一项复杂的工作，为了理解语言中的所有细小的内容，人们应该以 TB 级的训练数据来训练他们的模型。对于诸如在某一地区从一种语言翻译到另一种语言的子任务（例如游客讲话），相对较小的训练集和堆叠的 RNN 模型可以很好地工作。

第 9 章
语音识别和视频分析

本章介绍与流数据处理有关的内容，包括来自 IoT 传感器的音频、视频、帧序列和数据。

- 从零开始实现语音识别流程；
- 使用语音识别技术辨别演讲人；
- 使用深度学习理解视频。

9.1 简介

在本章，将重点关注流式数据，而不是静态数据。具体而言，就是时间元素在数据分布（例如语音和视频）中起关键作用的数据。在处理这种数据时，输入数据的分布经常会随着时间的推移而迅速变化。例如，在传感器数据中，可以在高峰时段有一个小峰值，并且在运动游戏的视频中可以有很多变化。数据类型的另一个挑战是训练数据集的大小。用于视频分类的数据集通常大于 100 GB，这意味着计算能力至关重要。

9.2 从零开始实现语音识别流程

语音识别，也称其为**自动语音识别**（ASR）和**语音 - 文本**（STT / S2T），这有着悠久的历史。传统的人工智能方法已经在工业界使用了很长时间。然而，随着最近对深度学习的兴趣，语音识别在性能上得到了新的提升。世界上许多科技公司都对语音识别感兴趣，因为它可以用于不同的应用程序，例如谷歌**语音搜索**、苹果公司的 Siri 和亚马逊的 Alexa。

许多公司使用预训练的语音识别软件。但是，在下面的内容中，将演示如何从头开始实现和训练语音识别流程。这个新训练模型的精度将低于业内使用的模型。主要原因是训练数据的质量和数量对精度起着至关重要的作用。有趣的是，有大量的训练数据（数千小时的开源数据）可供使用。此外，许多（在线）视频有字幕，可用于训练。

这里将使用一个数据集，其中包含不同人的零到九个口头数字的发音。这些文件采用 .wav 格式，标签和演讲者包含在文件名中。为了实现语音识别模型，将使用 Keras 框架。

如何去做…

1）首先导入 Keras 和其他库，如下所示：

```
import glob
import numpy as np
import random
import librosa
from sklearn.model_selection import train_test_split

import keras
from keras.layers import LSTM, Dense, Dropout, Flatten
from keras.models import Sequential
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

2）首先，设置训练文件的位置，为了可再现性，还设置一个种子值：

```
SEED = 2017
DATA_DIR = 'Data/spoken_numbers_pcm/'
```

3）训练文件位于一个目录中，将使用批生成器来检索它们。首先将训练文件分成训练集和验证集：

```
files = glob.glob(DATA_DIR + "*.wav")
X_train, X_val = train_test_split(files, test_size=0.2,
random_state=SEED)

print('# Training examples: {}'.format(len(X_train)))
print('# Validation examples: {}'.format(len(X_val)))
```

4）在进行批处理器生成之前，需要定义一些常量：

```
n_features = 20
max_length = 80
n_classes = 10
```

5）确保使用的 .wav 文件全部标准化为相同的长度。

6）批处理生成器应该返回一批预处理的 wav 文件及其标签：

```
def batch_generator(data, batch_size=16):
    while 1:
        random.shuffle(data)
        X, y = [], []
        for i in range(batch_size):
            wav = data[i]
            wave, sr = librosa.load(wav, mono=True)
            label = one_hot_encode(int(wav.split('/')[-1][0]),
                    n_classes)
            y.append(label)
            mfcc = librosa.feature.mfcc(wave, sr)
            mfcc = np.pad(mfcc, ((0,0), (0, max_length-
            len(mfcc[0]))), mode='constant', constant_values=0)
            X.append(np.array(mfcc))
        yield np.array(X), np.array(y)
```

7）在批生成器中，使用了一个 dense_to_one_hot 函数生成 one_hot_encode 标签，这个函数可以定义如下：

```
def one_hot_encode(labels_dense, n_classes=10):
    return np.eye(n_classes)[labels_dense]
```

8）现在，提取一个训练集的例子来确定最终的形状：

```
 X_example, y_example = next(batch_generator(X_train,
batch_size=1))
 print('Shape of training example: {}'.format(X_example.shape))
 print('Shape of training example label:
{}'.format(y_example.shape))
```

9）接下来，在定义网络结构之前，先定义超参数：

```
learning_rate = 0.001
batch_size = 64
n_epochs = 50
dropout = 0.5

input_shape = X_example.shape[1:]
steps_per_epoch = 50
```

10）此处使用的网络结构非常简单，将在网络密集层的顶部堆叠 LSTM 层，如下所示：

```
model = Sequential() model.add(LSTM(256, return_sequences=True,
input_shape=input_shape, dropout=dropout))
model.add(Flatten()) model.add(Dense(128, activation='relu'))
model.add(Dropout(dropout))
model.add(Dense(n_classes, activation='softmax'))
```

11）作为损失函数，使用 softmax 交叉熵来编译模型：

```
opt = Adam(lr=learning_rate)
model.compile(loss='categorical_crossentropy', optimizer=opt,
metrics=['accuracy']) model.summary()
```

为了检查训练过程进度，将测量精度。

12）已经准备好开始训练，将把结果保存在历史记录中：

```
history = model.fit_generator(
  generator=batch_generator(X_train, batch_size),
  steps_per_epoch=steps_per_epoch,
  epochs=n_epochs,
  verbose=1,
  validation_data=batch_generator(X_val, 32),
  validation_steps=5
)
```

使用这个相对简单的模型获得的高验证精度显示了深度学习的能力。但是，必须指出，这个任务相对简单。随着标签数量的增加，获得良好结果所需的训练数据量也大大增加。而且，不同的口音、背景噪声和其他因素在现实世界的应用中也起着重要的作用。

9.3 使用语音识别技术辨别讲话人

除了语音识别外，还可以用声音片段做更多的事情。虽然语音识别的重点是将语音（口语）转换为数字数据，但也可以使用声音片段来识别说话人是谁，这也被称为语音识别。由于解剖学和行为模式的差异，每个人说话时都有不同的特点。演讲者本人验证和讲话者本人识别在这个数字时代得到了更多的关注。例如，家庭数字助理可以自动检测哪个人在说话。

在下面的内容中，将使用与之前的内容中相同的数据，在之前内容中，实现了一个语音识别流程，下面将对说数字的讲话人进行识别分类。

如何去做…

1）在此内容中，首先导入所需函数库：

```
import glob
import numpy as np
import random
import librosa
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer

import keras
from keras.layers import LSTM, Dense, Dropout, Flatten
from keras.models import Sequential
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

2）设置种子值和 wav 文件的位置：

```
SEED = 2017
DATA_DIR = 'Data/spoken_numbers_pcm/'
```

3）使用 Scikit-learn 的 train_test_split 函数将一个 wav 文件划分成一个训练集和一个验证集：

```
files = glob.glob(DATA_DIR + "*.wav")
X_train, X_val = train_test_split(files, test_size=0.2,
random_state=SEED)

print('# Training examples: {}'.format(len(X_train)))
print('# Validation examples: {}'.format(len(X_val)))
```

4）为了提取和呈现所有特定的标签，使用以下代码：

```
labels = []
for i in range(len(X_train)):
    label = X_train[i].split('/')[-1].split('_')[1]
    if label not in labels:
        labels.append(label)
print(labels)
```

5）现在可以定义 one_hot_encode 函数，如下：

```
label_binarizer = LabelBinarizer()
label_binarizer.fit(list(set(labels)))

def one_hot_encode(x): return label_binarizer.transform(x)
```

6）在将数据输入网络之前，需要进行一些预处理，使用以下设置：

```
n_features = 20
max_length = 80
n_classes = len(labels)
```

7）现在，可以定义批处理生成器。生成器包括所有预处理任务，例如读取 wav 文件并将其转换为可用的输入：

```
def batch_generator(data, batch_size=16):
    while 1:
        random.shuffle(data)
        X, y = [], []
        for i in range(batch_size):
            wav = data[i]
            wave, sr = librosa.load(wav, mono=True)
            label = wav.split('/')[-1].split('_')[1]
            y.append(one_hot_encode(label))
            mfcc = librosa.feature.mfcc(wave, sr)
            mfcc = np.pad(mfcc, ((0,0), (0, max_length-
            len(mfcc[0]))), mode='constant', constant_values=0)
            X.append(np.array(mfcc))
        yield np.array(X), np.array(y)
```

请注意，批量生成器与之前的方法相比有所差异。

8）在定义网络结构之前，定义超参数：

```
learning_rate = 0.001
batch_size = 64
n_epochs = 50
dropout = 0.5

input_shape = (n_features, max_length)
steps_per_epoch = 50
```

9）此处使用的网络结构非常简单，将在网络密集层的顶部堆叠 LSTM 层，如下所示：

```
 model = Sequential()
 model.add(LSTM(256, return_sequences=True,
input_shape=input_shape,
   dropout=dropout))
 model.add(Flatten())
 model.add(Dense(128, activation='relu'))
 model.add(Dropout(dropout))
 model.add(Dense(n_classes, activation='softmax'))
```

10）接下来，设置损失函数，编译模型，并输出模型的摘要：

```
opt = Adam(lr=learning_rate)
 model.compile(loss='categorical_crossentropy', optimizer=opt,
 metrics=['accuracy'])
 model.summary()
```

11）为了防止过拟合，将使用早停方法并自动保存具有最高验证精度的模型：

```
callbacks =
[ModelCheckpoint('checkpoints/voice_recognition_best_model_{epoch:0
2d}.hdf5', save_best_only=True),
            EarlyStopping(monitor='val_acc', patience=2)]
```

12）准备开始训练，并把结果存储在历史文件中：

```
 history = model.fit_generator(
   generator=batch_generator(X_train, batch_size),
   steps_per_epoch=steps_per_epoch,
   epochs=n_epochs,
   verbose=1,
   validation_data=batch_generator(X_val, 32),
   validation_steps=5,
   callbacks=callbacks
 )
```

在图 9.1 中，绘制了训练精度和验证精度随训练周期的变化情况。

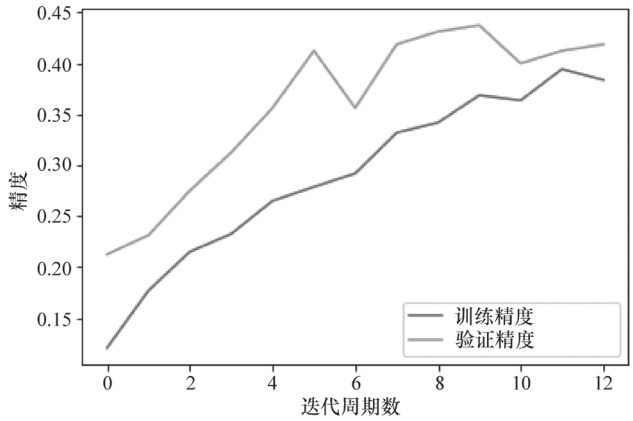


图 9.1　训练和验证精度

9.4　使用深度学习理解视频

在第 7 章　计算机视觉　中，展示了如何检测和分割单个图像中的目标。这些图像中的物体是固定的。但是，如果在输入中添加时间维度，则目标可以在特定场景内移动。了解在多个帧（视频）中发生的事情是一件非常困难的事情。在以下内容中，将要演示如何来处理视频，重点将是整合 CNN 和 RNN。CNN 用于提取单帧的特征，这些功能组合起来并用作 RNN 的输入，这也称为网络**堆叠**，即在一个模型的顶部建立（堆叠）第二个模型。

对于此内容，将使用包含 13321 个短视频的数据集。这些视频分布在共 101 个不同的类别中。由于这个任务的复杂性，本书不想从头开始训练模型。因此，将使用 Keras 中提供的已有预训练权重的 InceptionV3 模型。

如何去做…

1）首先导入所必要的函数库，如下所示：

```
from keras.applications.inception_v3 import InceptionV3
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layeSequential import Dense, GlobalAveragePooling2D
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

2）导入函数库之后，可以设置超参数：

```
batch_size = 32
n_epochs = 50
steps_per_epoch = 100
```

3）由于数据集的大小，使用批生成器很重要。在创建生成器之前，将定义 Keras 的 ImageDataGenerator 用于预处理目的：

```
train_datagen = ImageDataGenerator(
       rescale=1./255,
       shear_range=0.2,
       horizontal_flip=True,
       rotation_range=10.,
       width_shift_range=0.2,
        height_shift_range=0.2)

val_datagen = ImageDataGenerator(rescale=1./255)
```

4）为训练集选择应用小图像增广技术。为使模型鲁棒，可以决定应用更加主动的图像增广技术。

5）现在可以创建用于训练和验证的生成器，如下所示：

```
classes = np.loadtxt('Data/videos/classes.txt')
 train_generator = train_datagen.flow_from_directory(
 'Data/videos/train/',
 target_size=(299, 299),
 batch_size=batch_size,
 classes=classes,
 class_mode='categorical')
 validation_generator = val_datagen.flow_from_directory(
 'Data/videos/test/',
 target_size=(299, 299),
 batch_size=batch_size,
 classes=classes,
 class_mode='categorical')
```

6）首先加载 InceptionV3 模型：

```
inception_model = InceptionV3(include_top=False)
```

7）接下来，定义要添加到该模型中的网络层，如下所示：

```
x = inception_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(len(classes), activation='softmax')(x)
model = Model(inputs=inception_model.input, outputs=predictions)
```

8）但是，只需要训练最后两层，并保持其他权重固定：

```
for layer in model.layers[:-2]:
 layer.trainable = False
```

9）现在用 Adam 优化器编译模型并输出摘要：

```
model.compile(optimizer=Adam(), loss='categorical_crossentropy',
metrics=['accuracy'])
```

10）开始训练 CNN 模型：

```
model.fit_generator(
    train_generator,
    steps_per_epoch=steps_per_epoch,
    validation_data=validation_generator,
    validation_steps=10,
    epochs=n_epochs)
```

11）现在开始训练模型，然而希望提取特征，但不希望从最终层（分类）而是从最大化层获得信息。方法如下：

```
model.layers.pop()
model.layers.pop()
model.outputs = [model.layers[-1].output]
model.output_layers = [model.layers[-1]]
model.layers[-1].outbound_nodes = []
```

12）接下来，遍历视频帧以提取每个帧的这些特征并将它们保存到一个文件中：

```
for video in data:
    path = 'Data/videos/' + str(video)
    frames = get_frame(video)
    sequence = []
    for image in frames:
        features = model.predict(image)
        sequence.append(features)
    np.savetxt(path, sequence)
```

13）现在可以定义想要应用于这些特征的 LSTM 模型：

```
model_lstm = Sequential()
model_lstm.add(LSTM(1024, return_sequences=True,
input_shape=input_shape, dropout=dropout))
model_lstm.add(Flatten())
model_lstm.add(Dense(512, activation='relu'))
model_lstm.add(Dropout(dropout))
model_lstm.add(Dense(self.nb_classes, activation='softmax'))
```

14）确保用于第二层模型验证的数据拆分与第一层拆分相同（否则，将有数据遗漏）：

```
batch_size_lstm = 32
n_epochs_lstm = 40
```

15）最后，开始训练第二层模型：

```
model_lstm.fit(X_train,
            y_train,
            batch_size=batch_size_lstm,
            validation_data=(X_val, y_val),
            verbose=1,
            epochs=n_epochs_lstm)
```

已经证明最先进的模型在这些视频上能达到 90% 以上的精度。然而，本节内容中的主要目标是为那些对分类视频感兴趣的人提供一个垫脚石。

也可以使用浅层神经网络来代替使用 LSTM 作为第二层模型。其他选项包括混合视频中帧的结果或使用三维卷积网络。

第 10 章
时间序列和结构化数据

本章提供了与数字运算相关的内容，包括序列和时间序列、二进制编码和浮点数表示。

- 使用神经网络预测股票价格；
- 预测共享单车需求；
- 使用浅层神经网络进行二元分类。

10.1 简介

在前面的内容中，主要关注各类机器学习算法的对比，以及深度学习突出的应用领域。但是，对于更多结构化的数据集，深度学习也可以具有附加价值，特别是当数据具有时间关系时。在本章中，将主要侧重于应用深度学习来预测时间序列和趋势分析。

10.2 使用神经网络预测股票价格

当机器学习变得流行时，立即就引起对股票价格预测的很多关注。已经应用了许多不同的算法来预测股票价格，从更传统的算法，如从随机森林算法到近期的极端梯度上升算法。虽然后者在大多数情况下可能仍然优于深度学习方法，但它在使用神经网络方法中仍然很有价值。例如，可以用于网络集成或多层堆叠。在下面的内容中，将用 Keras 框架来预测股票价格。

如何去做…

1）从导入所需函数库开始，如下所示：

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

from keras.layers.core import Dense,
Activation, Dropout
from keras.layers.recurrent import LSTM
from keras.models import Sequential
```

2）加载数据并呈现第一行：

```
data = pd.read_csv('Data/stock-data-2000-2017.csv')
# 为方便起见，重排列名
data = data[['Open', 'High', 'Low', 'Volume',
'Close']]
data.head()
```

图 10.1 显示了上述代码的输出。

	开盘价	最高价	最低价	成交量	收盘价
0	3352.149902	3376.919922	3233.189941	1437770000	3355.560059
1	3326.889893	3383.399902	3229.010010	1874430000	3240.540039
2	3152.330078	3258.239990	3103.530029	2340450000	3168.489990
3	3241.260010	3249.110107	3071.250000	2128660000	3074.679932
4	3054.550049	3316.969971	3054.550049	2070750000	3316.770020

图 10.1　股票数据集的前五项

3）在继续之前，需要设置超参数：

```
sequence_length = 21 # 20 次预先输入
n_features = len(data.columns)
val_ratio = 0.1
n_epochs = 300
batch_size = 512
```

4）数据需要预先处理，然后才能输入到递归神经网络。每个输入应包含前 20 个（sequence_length - 1）输入的数据。这意味着，包含标签的一行数据的形状由特征数目（n_features）与序列长度（sequence_length）来决定：

```
data = data.as_matrix()
data_processed = []
for index in range(len(data) - sequence_length):
    data_processed.append(data[index :
    index + sequence_length])
data_processed = np.array(data_processed)
```

5）对时间序列数据进行建模时，应该非常小心地分割数据进行训练和验证。应该类似于训练数据和测试数据拆分。在这种情况下，应该将数据集按时间分成训练和验证两个部分：

```
val_split = round((1-val_ratio) *
data_processed.shape[0])
train = data_processed[: int(val_split), :]
val = data_processed[int(val_split) :, :]

print('Training data: {}'.format(train.shape))
print('Validation data: {}'.format(val.shape))
```

6）如图 10.1 所示，用作输入的值非常大。此外，其中一个变量 Volume 与其他变量相比具有更大的值，因此使用 Scikit-learn 的 MinMaxScaler 对数据进行标准化是非常重要的，如下所示：

```
train_samples, train_nx, train_ny = train.shape
val_samples, val_nx, val_ny = val.shape

train = train.reshape((train_samples, train_nx * train_ny))
val = val.reshape((val_samples, val_nx * val_ny))

preprocessor = MinMaxScaler().fit(train)
train = preprocessor.transform(train)
val = preprocessor.transform(val)

train = train.reshape((train_samples,
train_nx, train_ny))
val = val.reshape((val_samples,
val_nx, val_ny))
```

7）接下来，需要确保标签的正确提取。从 21 × 5 矩阵中，需要最后一个值作为标签。作为输入数据，将只使用前 20 行（使用第 21 行的输入变量就是抄袭）：

```
X_train = train[:, : -1]
y_train = train[:, -1][: ,-1]
X_val = val[:, : -1]
y_val = val[:, -1][ : ,-1]

X_train = np.reshape(X_train, (X_train.shape[0],
X_train.shape[1], n_features))
X_val = np.reshape(X_val, (X_val.shape[0],
X_val.shape[1], n_features))
```

8）现在可以定义模型结构。将两个 LSTM 神经元和一个输出层叠加在一起，其间有一个 Dropout，将使用 Adam 作为优化器，并使用 MSE 作为损失函数：

```
model = Sequential()
model.add(LSTM(input_shape=(X_train.shape[1:]), units = 128,
return_sequences=True))
model.add(Dropout(0.5))
model.add(LSTM(128, return_sequences=False))
model.add(Dropout(0.25))
model.add(Dense(units=1))
model.add(Activation("linear"))

model.compile(loss="mse", optimizer="adam")
```

9）开始训练模型，并将训练结果存储在历史记录中：

```
history = model.fit(
    X_train,
    y_train,
    batch_size=batch_size,
    epochs=n_epochs,
    verbose=2)
```

正如读者所看到的，在训练时没有使用验证分割。如果想在训练中使用验证集，应该按照时间划分训练集并使用固定的验证集。

10）对比之下，将在训练之后验证结果。另外将保存标签和预测之间的差异，如下所示：

```
preds_val = model.predict(X_val)
diff = []
for i in range(len(y_val)):
    pred = preds_val[i][0]
    diff.append(y_val[i] - pred)
```

11）由于已经标准化所有的输入变量和标签，预测结果也将加以标准化。这就是为什么需要使用逆变换来获得实际值：

```
real_min = preprocessor.data_min_[104]
real_max = preprocessor.data_max_[104]
print(preprocessor.data_min_[104])
print(preprocessor.data_max_[104])

preds_real = preds_val *
(real_max - real_min) + real_min
y_val_real = y_val *
(real_max - real_min) + real_min
```

12）最后，可以将实际的股票价格和预测价格按以下步骤绘制出来：

```
plt.plot(preds_real, label='Predictions')
plt.plot(y_val_real, label='Actual values')
plt.xlabel('test')
plt.legend(loc=0)
plt.show()
```

在图 10.2 中，绘制了实际股价和模型输出的预测值。正如读者所看到的那样，预测确实遵循了实际的价格，但是在大多数步骤中高估了价格。

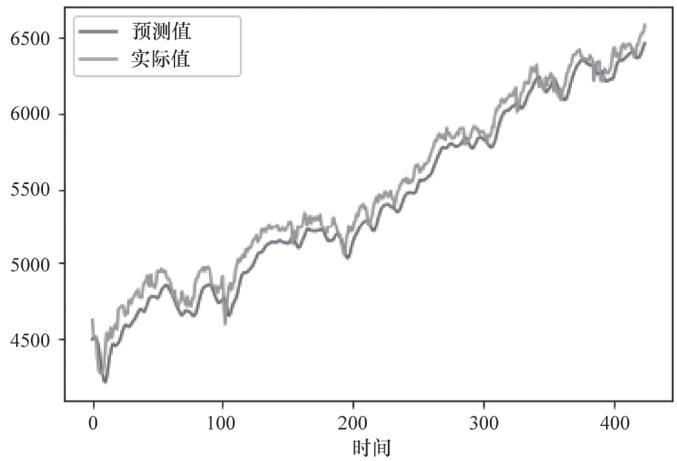


图 10.2　验证集的实际股票价格与预测值对比

10.3　预测共享单车需求

在以前的数据集中，其特征与标签强相关。然而，在某些时间序列中，特征有可能与标签相关性较小或与标签完全不相关。机器学习背后的主要思想是，算法本身试图找出哪些特征是有价值的，哪些不是。特别是在深度学习中，要保持有限的工程特征。在下面的内容中，将预测对单车共享租赁的需求。数据包括一些有趣的特征，如天气类型、假期、温度和季节。

如何去做…

1）首先导入所需函数库：

```
from sklearn import preprocessing
import pandas as pd
import numpy as np
from math import pi, sin, cos
from datetime import datetime
```

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
```

2）训练数据和测试数据分别存储在两个独立的 .csv 文件中：

```
train = pd.read_csv('Data/bike-sharing/train.csv')
test = pd.read_csv('Data/bike-sharing/test.csv')
data = pd.concat([train, test])
test_split = train.shape[0]
```

3）输出前五行数据进行查看：

```
data.head()
```

在图 10.3 中，可以看到 datetime 特征不能被神经网络正确解析。

	体感温度	偶然性	租车数	日期及时间	是否假日	湿度	登记预约人数	季节	温度	天气	风速	是否工作日
0	14.395	3.0	16.0	2011-01-01 00:00:00	0	81	13.0	1	9.84	1	0.0	0
0	13.635	8.0	40.0	2011-01-01 01:00:00	0	80	32.0	1	9.02	1	0.0	0
2	13.635	5.0	32.0	2011-01-01 02:00:00	0	80	27.0	1	9.02	1	0.0	0
3	14.395	3.0	13.0	2011-01-01 03:00:00	0	75	10.0	1	9.84	1	0.0	0
4	14.395	0.0	1.0	2011-01-01 04:00:00	0	75	1.0	1	9.84	1	0.0	0

图 10.3　训练数据前五项

4）按如下步骤预处理该变量：

```
data['hour'] = data.apply(lambda x:
datetime.strptime(x['datetime'],
'%Y-%m-%d %H:%M:%S').hour, axis=1)
data['weekday'] = data.apply(lambda x:
datetime.strptime(x['datetime'],
'%Y-%m-%d %H:%M:%S').weekday(), axis=1)
data['month'] = data.apply(lambda x:
datetime.strptime(x['datetime'],
'%Y-%m-%d %H:%M:%S').month, axis=1)
data['year'] = data.apply(lambda x:
datetime.strptime(x['datetime'],
'%Y-%m-%d %H:%M:%S').year, axis=1)
```

5）在一个周期结束之后，新的周期开始之前，尽管希望保持一定数量的特征工程，但可能更希望确保模型能够理解这一点。更具体地说，这些时间变量是循环的。可以用下面的代码添加这些信息：

```
data['hour_sin'] = data.apply(lambda x:
sin(x['hour'] / 24.0 * 2 * pi), axis=1)
data['hour_cos'] = data.apply(lambda x:
cos(x['hour'] / 24.0 * 2 * pi), axis=1)
data['weekday_sin'] = data.apply(lambda x:
sin(x['weekday'] / 7.0 * 2 * pi), axis=1)
data['weekday_cos'] = data.apply(lambda x:
cos(x['weekday'] / 7.0 * 2 * pi), axis=1)
data['month_sin'] = data.apply(lambda x:
sin(((x['month'] - 5) % 12) / 12.0 * 2 * pi), axis=1)
data['month_cos'] = data.apply(lambda x:
cos(((x['month'] - 5) % 12) / 12.0 * 2 * pi), axis=1)
data['season_sin'] = data.apply(lambda x:
sin(((x['season'] - 3) % 4) / 4.0 * 2 * pi), axis=1)
data['season_cos'] = data.apply(lambda x:
cos(((x['season'] - 3) % 4) / 4.0 * 2 * pi), axis=1)
```

6）预处理后，可以再次分割训练和测试数据：

```
X_train = data[:test_split].drop(['datetime',
'casual', 'registered', 'count'], inplace=False, axis=1)
X_test = data[test_split:].drop(['datetime',
'casual', 'registered', 'count'], inplace=False, axis=1)
y_train = data['count'][:test_split]
y_test = data['count'][test_split:]
```

7）在输入数据之前，应该对输入变量进行标准化：

```
scaler = preprocessing.StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

对这个问题的训练数据和测试数据的划分可以被认为是非正统的。训练集中的记录是每个月的前 19 天，测试集中的记录是从第 20 天到月底。如果想要正确地验证结果，也应该以相同的方式分割验证集。例如，使用 14~19 天作为验证集。

8）现在可以定义模型结构和优化器并编译模型：

```
model = Sequential()
model.add(Dense(200, input_dim=X_train.shape[1]))
model.add(Activation('relu'))
model.add(Dropout(0.1))
model.add(Dense(200))
model.add(Activation('relu'))
model.add(Dropout(0.1))
model.add(Dense(1))

opt = Adam()
model.compile(loss='mean_squared_logarithmic_error',
 optimizer=opt)
```

9）使用以下超参数：

```
n_epochs = 1000
batch_size = 128
```

10）使用早停方法开始训练：

```
callbacks = [EarlyStopping(monitor='val_loss',
patience=5)]
history = model.fit(X_train, y_train, shuffle=True,
epochs=n_epochs, batch_size=batch_size,
validation_split=0.1, verbose=1, callbacks=callbacks)
```

在本节内容中，使用了一个标准的前馈神经网络结构。这些数据的时间特性已被特征捕获，而不是由模型捕获。

10.4 使用浅层神经网络进行二元分类

在本书中，着重于为现实世界的问题提供随时可用的示例。对于一些相对简单的任务，简单的神经网络可以为问题提供足够好的解决方案。在下面的内容中，将演示如何在 Keras 中实现浅层神经网络进行二元分类。

如何去做…

1）从导入所需函数库开始，如下所示：

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder

from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import EarlyStopping
```

2）接下来，加载数据集：

```
dataframe = pandas.read_csv("Data/sonar.all-data",
header=None)
data = dataframe.values
```

3）从特征数据上分离标签：

```
X = data[:,0:60].astype(float)
y = data[:,60]
```

4）目前，标签是字符串。根据网络要求，需要将它们二进制化：

```
encoder = LabelEncoder()
encoder.fit(y)
y = encoder.transform(y)
```

5）定义一个具有单一隐层的简单网络：

```
model = Sequential()
model.add(Dense(32, input_dim=60, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
optimizer='adam', metrics=['accuracy'])
```

6）通过定义一个回调函数使用早停方法来防止过拟合：

```
callbacks = [EarlyStopping(monitor='val_acc', patience=20)]
```

7）接下来，定义超参数并开始训练：

```
n_epochs = 1000
batch_size = 2
model.fit(X, y, epochs=n_epochs, batch_size=batch_size,
validation_split=0.1, callbacks=callbacks)
```

在不到 100 个周期内，应该获得大于 90%的验证精度。

在本节内容中已经证明，实现深层网络并不总是可以获得良好的结果。在某些情况下，一个较小、较复杂的网络可以工作得很好，甚至很好。当数据结构合理、任务简单时，保持简单更具价值。

第 11 章
游戏智能体和机器人

本章重点介绍在复杂环境中应用最先进的深度学习应用程序，包括在复杂环境（模拟）中的游戏智能体以及自动驾驶相关的内容。

- 通过端到端学习来驾驶汽车；
- 通过深度强化学习来玩游戏；
- 用遗传算法（GA）优化超参数。

11.1 简介

正如本书所讨论的，深度学习对机器人和游戏智能体也有重大影响。其他机器学习算法已经显示出了如何在不太复杂的环境中击败人类玩家。然而，深度学习模式能够在更复杂的环境中实现超越人类的表现。驾驶汽车一直被认为是一项复杂的任务，只能由人类来完成，也可能在遥远的将来被机器人完成。人工智能和传感器技术的最新进展清楚地表明，自动驾驶不再是遥远的未来梦想。

11.2 通过端到端学习来驾驶汽车

深度学习令人着迷的地方是，不需要关注特征工程。理想的情况下，网络自己学习什么是重要的，什么不重要。一个很好的例子就是行为克隆：一个人演示一个特定的任务——这是训练数据，一个智能体（深度学习模型）试图复制那个行为，而没有指定步骤。对于只有一名智能体拟人的受保护环境中的自动驾驶车辆，智能体拟人应该学习在路上行驶。这里将演示如何实施一种教导智能体拟人在赛道上驾驶汽车的深度学习模式。

入门

对于这个方案，将使用 **Udacity 的自驾车模拟器**。这个模拟器是基于 Unity 的，安装这个模拟器的指令可以在下面的 GitHub 页面找到：https ：//github.com/udacity/self-driving-car-sim。

如何去做…

1）首先，导入所需函数库，如下所示：

```
import pandas as pd
import numpy as np
import cv2
import math

from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Lambda
from keras.layers import Input, ELU
from keras.optimizers import SGD, Adam, RMSprop
from keras.utils import np_utils
from keras.layers import Conv2D, MaxPooling2D, Flatten
from keras import initializers
from keras.callbacks import ModelCheckpoint
```

2）接下来，加载训练数据如下：

```
data_path = 'Data/SDC/training.csv'
data = pd.read_csv(data_path, header=None, skiprows=[0],
names=['center', 'left', 'right', 'steering', 'throttle', 'brake',
'speed'])
```

3）继续之前，需要定义一些图像参数：

```
 img_cols = 64
img_rows = 64
img_channels = 3
```

4）对于数据增广，定义以下函数：

```
 def augment_image(image):
    image = cv2.cvtColor(image,cv2.COLOR_RGB2HSV)
    random_bright = .25 + np.random.uniform()
    image[:,:,2] = image[:,:,2] * random_bright
    image = cv2.cvtColor(image,cv2.COLOR_HSV2RGB)
    return(image)
```

5）接下来，定义一个预处理输入图像的函数：

```
def preprocess_image(data):
   # 随机选择左、中、右图
   # 照相机图像
   rand = np.random.randint(3)
   if (rand == 0):
       path_file = data['left'][0].strip()
       shift_ang = .25
```

```
    if (rand == 1):
        path_file = data['center'][0].strip()
        shift_ang = 0.
    if (rand == 2):
        path_file = data['right'][0].strip()
        shift_ang = -.25
    y = data['steering'][0] + shift_ang

    # 读取图像
    image = cv2.imread(path_file)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # 裁剪图像
    shape = image.shape
    image = image[math.floor(shape[0]/4):
    shape[0]-20, 0:shape[1]]

    # 调整图像大小
    image = cv2.resize(image, (img_cols, img_rows),
    interpolation=cv2.INTER_AREA)

    # 增强图像
    image = augment_image(image)
    image = np.array(image)

    if np.random.choice([True, False]):
        image = cv2.flip(image,1)
        y = -y
    return(image, y)
```

6）由于总训练集相当大，所以使用批量生成器来实时加载图像：

```
def batch_gen(data, batch_size):
    # 创建空值 numpy 数组
    batch_images = np.zeros((batch_size, img_rows,
    img_cols, img_channels))
    batch_steering = np.zeros(batch_size)

    small_steering_threshold = 0.8

    # 自定义批样本生成器
    while 1:
        for n in range(batch_size):
            i = np.random.randint(len(data))
            data_sub = data.iloc[[i]].reset_index()
            # 按概率只保留具有小转向角度的训练数据
            keep = False
            while keep == False:
```

```
                x, y = preprocess_image(data_sub)
                pr_unif = np.random
                if abs(y) < .01:
                    next;
                if abs(y) < .10:
                    small_steering_rand = np.random.uniform()
                    if small_steering_rand >
                    small_steering_threshold:
                        keep = True
                else:
                    keep = True

            batch_images[n] = x
            batch_steering[n] = y
        yield batch_images, batch_steering
```

7）定义模型结构：

```
input_shape = (img_rows, img_cols, img_channels)
model = Sequential()
model.add(Lambda(lambda x: x/255.-0.5,
input_shape=input_shape))
model.add(Conv2D(3, (1, 1), padding='valid',
kernel_initializer='he_normal'))
model.add(ELU())
model.add(Conv2D(32, (3, 3), padding='valid',
kernel_initializer='he_normal'))
model.add(ELU())
model.add(Conv2D(32, (3, 3), padding='valid',
kernel_initializer='he_normal'))
model.add(ELU())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))
model.add(Conv2D(64, (3, 3), padding='valid',
kernel_initializer='he_normal'))
model.add(ELU())
model.add(Conv2D(64, (3, 3), padding='valid',
kernel_initializer='he_normal'))
model.add(ELU())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))
model.add(Conv2D(128, (3, 3), padding='valid',
kernel_initializer='he_normal'))
model.add(ELU())
model.add(Conv2D(128, (3, 3), padding='valid',
kernel_initializer='he_normal'))
model.add(ELU())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(512, kernel_initializer='he_normal',
```

```
activation='relu'))
model.add(ELU())
model.add(Dropout(0.5))
model.add(Dense(64, kernel_initializer='he_normal'))
model.add(ELU())
model.add(Dropout(0.5))
model.add(Dense(16, kernel_initializer='he_normal'))
model.add(ELU())
model.add(Dropout(0.5))
model.add(Dense(1, kernel_initializer='he_normal'))

opt = Adam(lr=1e-4, beta_1=0.9, beta_2=0.999,
epsilon=1e-08, decay=0.0)
model.compile(optimizer=opt, loss='mse')
model.summary()
```

8）为了存储训练模型，使用一个回调函数：

```
callbacks = [ModelCheckpoint('checkpoints/sdc.h5',
save_best_only=False)]
```

9）接下来，创建用于训练和验证的生成器：

```
train_generator = batchgen(data, batch_size)
val_generator = batchgen(data, batch_size)
```

10）在开始训练之前，需要设置一些超参数：

```
n_epochs = 8
batch_size = 64
steps_per_epoch = 500
validation_steps = 50
```

11）最后，开始训练：

```
history = model.fit_generator(train_generator,
steps_per_epoch=steps_per_epoch, epochs=n_epochs,
validation_data=val_generator,
validation_steps=validation_steps, callbacks=callbacks)
```

在本节内容中，训练了一个智能体，在没有指定任何规则的情况下在赛道上驾驶汽车。智能体复制训练数据的行为并学习驾驶，所以这也叫做端到端的学习。

11.3 通过深度强化学习来玩游戏

在下面的内容中，将演示如何利用深度学习模式来玩游戏。在这个例子中，展示了如何应用一个 Deep-Q 网络与 Keras 框架来取得进展。

如何去做…

1）从导入必要的函数库开始，如下所示：

```
import gym
import random
import numpy as np
import matplotlib.pyplot as plt
from collections import deque

from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import Dense, Flatten
from keras.layers.convolutional import Conv2D
from keras import backend as K
```

2）首先，绘制一个游戏示例输入图像（见图 11.1）：

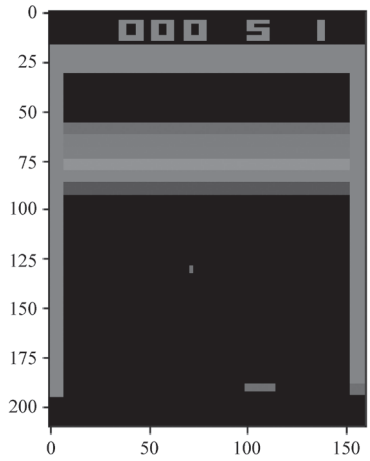


图 11.1 OpenAI 的 Breakout 输入图像示例

```
env = gym.make('BreakoutDeterministic-v4')
observation = env.reset()

for i in range(3):
    # 2帧图像后发球
    if i > 1:
        print(observation.shape)
        plt.imshow(observation)
```

```
        plt.show()
    # 获得下一次观测
    observation, _, _, _ = env.step(1)
```

3）现在，可以定义一个预处理输入数据的函数：

```
def preprocess_frame(frame):
    # 移去图像顶部及一些背景
    frame = frame[35:195, 10:150]
    # 转化为灰度图像并缩小1/2
    frame = frame[::2, ::2, 0]
    # 设背景值为0
    frame[frame == 144] = 0
    frame[frame == 109] = 0
    # 设球和拍数目为1
    frame[frame != 0] = 1
    return frame.astype(np.float).ravel()
```

4）输出前面的预处理图像，了解算法做了什么（见图 11.2）：

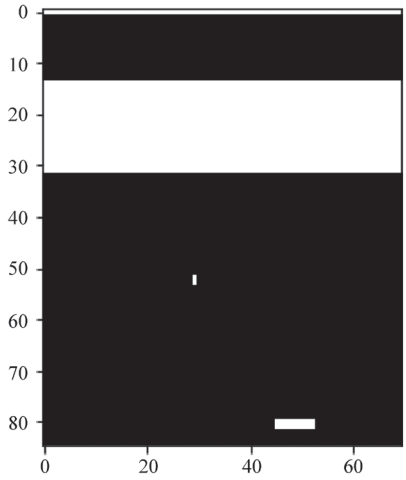


图 11.2　Breakout 的预处理帧

```
obs_preprocessed = preprocess_frame(observation)
plt.imshow(obs_preprocessed, cmap='gray')
plt.show()
```

5）对于 Q 学习的具体实现，需要定义一个执行大部分任务的智能体：

```
class DQLAgent:
    def __init__(self, cols, rows, n_actions,
    batch_size=32):
        self.state_size = (cols, rows, 4)
        self.n_actions = n_actions
        self.epsilon = 1.
        self.epsilon_start, self.epsilon_end = 1.0, 0.1
        self.exploration_steps = 1000000.
        self.epsilon_decay_step = (self.epsilon_start -
        self.epsilon_end) / self.exploration_steps
        self.batch_size = batch_size
        self.discount_factor = 0.99
        self.memory = deque(maxlen=400000)
        self.model = self.build_model()
        self.target_model = self.build_model()
        self.optimizer = self.optimizer()
        self.avg_q_max, self.avg_loss = 0, 0

    def optimizer(self):
        a = K.placeholder(shape=(None,), dtype='int32')
        y = K.placeholder(shape=(None,), dtype='float32')

        py_x = self.model.output

        a_one_hot = K.one_hot(a, self.n_actions)
        q_value = K.sum(py_x * a_one_hot, axis=1)
        error = K.abs(y - q_value)

        quadratic_part = K.clip(error, 0.0, 1.0)
        linear_part = error - quadratic_part
        loss = K.mean(0.5 *
        K.square(quadratic_part) + linear_part)

        opt = Adam(lr=0.00025, epsilon=0.01)
        updates = opt.get_updates
        (self.model.trainable_weights, [], loss)
        train = K.function([self.model.input, a, y],
        [loss], updates=updates)

        return train

    def build_model(self):
        model = Sequential()
        model.add(Conv2D(32, (8, 8), strides=(4, 4),
        activation='relu', input_shape=self.state_size))
```

```
        model.add(Conv2D(64, (4, 4), strides=(2, 2),
        activation='relu'))
        model.add(Conv2D(64, (3, 3), strides=(1, 1),
        activation='relu'))
        model.add(Flatten())
        model.add(Dense(512, activation='relu'))
        model.add(Dense(self.n_actions))
        model.summary()
        return model

    def update_model(self):
        self.target_model.set_weights
        (self.model.get_weights())

    def action(self, history):
        history = np.float32(history / 255.0)
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.n_actions)
        else:
            q_value = self.model.predict(history)
            return np.argmax(q_value[0])

    def replay(self, history, action, reward,
               next_history, dead):
        self.memory.append((history, action,
        reward, next_history, dead))

    def train(self):
        if len(self.memory) < self.batch_size:
            return
        if self.epsilon > self.epsilon_end:
            self.epsilon -= self.epsilon_decay_step

        mini_batch = random.sample(self.memory,
        self.batch_size)
        history = np.zeros((self.batch_size,
        self.state_size[0], self.state_size[1],
        self.state_size[2]))
        next_history = np.zeros((self.batch_size,
        self.state_size[0], self.state_size[1],
        self.state_size[2]))
        target = np.zeros((self.batch_size,))
        action, reward, dead = [], [], []

        for i in range(self.batch_size):
            history[i] = np.float32
            (mini_batch[i][0] / 255.)
            next_history[i] = np.float32
```

```
            (mini_batch[i][3] / 255.)
            action.append(mini_batch[i][1])
            reward.append(mini_batch[i][2])
            dead.append(mini_batch[i][4])

        target_value = self.target_model.
        predict(next_history)

        for i in range(self.batch_size):
            if dead[i]:
                target[i] = reward[i]
            else:
                target[i] = reward[i] +
                self.discount_factor * \
                np.amax(target_value[i])

        loss = self.optimizer([history, action,
        target])
        self.avg_loss += loss[0]
```

6）接下来，设置超参数和一些常规设置并初始化智能体：

```
env = gym.make('BreakoutDeterministic-v4')

# 整体参数设定
n_warmup_steps = 50000
update_model_rate = 10000
cols, rows = 85, 70
n_states = 4

# 超参数
batch_size = 32

# 初始化
agent = DQLAgent(cols, rows, n_actions=3)
scores, episodes = [], []
n_steps = 0
```

7）现在准备开始训练模型：

```
while True:
    done = False
    dead = False
    step, score, start_life = 0, 0, 5
    observation = env.reset()
```

```
        state = preprocess_frame(observation,
        cols, rows)
        history = np.stack((state, state,
        state, state), axis=2)
        history = np.reshape([history],
        (1, cols, rows, n_states))

        while not done:
    # env.render()
            n_steps += 1
            step += 1
            # 获取动作
            action = agent.action(history)
            observation, reward, done, info =
            env.step(action+1)
            # 提取下一个状态
            state_next = preprocess_frame
            (observation, cols, rows)
            state_next = np.reshape([state_next],
            (1, cols, rows, 1))
            history_next = np.append(state_next,
            history[:, :, :, :3], axis=3)

            agent.avg_q_max += np.amax(agent.model
            .predict(history)[0])
            reward = np.clip(reward, -1., 1.)

            agent.replay(history, action, reward,
            history_next, dead)
            agent.train()
            if n_steps % update_model_rate == 0:
                agent.update_model()
            score += reward

            if dead:
                dead = False
            else:
                history = history_next

            if done:
                print('episode {:2d}; score:
                {:2.0f}; q {:2f}; loss {:2f}; steps {}'
                      .format(n_steps, score,
                      agent.avg_q_max / float(step),
                      agent.avg_loss / float(step), step))

                agent.avg_q_max, agent.avg_loss = 0, 0
        # 存储模型权重
        if n_steps % 1000 == 0:
            agent.model.save_weights
            ("weights/breakout_dql.h5")
```

8）当要评估算法时，可以停止训练。

9）看看最终模型的表现（见图 11.3）：

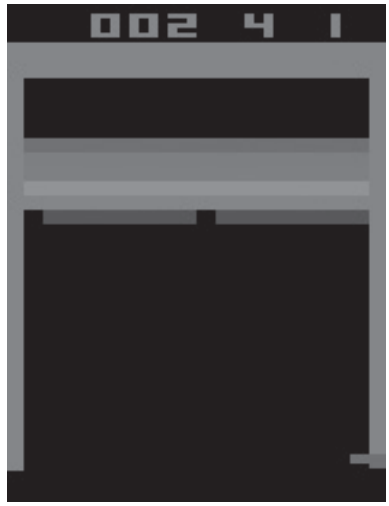


图 11.3　对深度 *Q* 学习智能体进行训练

```
env = gym.make('BreakoutDeterministic-v4')
agent = DQLAgent(cols, rows, n_action=3)

for i in range(5):
    observation = env.reset()

    state = pre_processing(observation,
    cols, rows)
    history = np.stack((state, state,
    state, state), axis=2)
    history = np.reshape([history], (1, cols,
    rows, n_states))

    while not done:
        env.render()
        action = agent.get_action(history)
        observe, reward, done, info =
        env.step(action+1)
```

11.4　用 GA 优化超参数

在以前的所有内容中，只考虑了静态网络结构。更具体地说，在训练网络或智能体时，网络没有改变。通常看到的是，网络结构和超参数会对结果产生很大的影响。但是人们通常不知道网络是否能够正常运行，所以需要对其进行彻底的测试。有不同的方法来优化这些超参数。在第 12 章　超参数选择、调优和神经网络学习　中，演示了如何应用网

格搜索（用蛮力）来寻找最优的超参数。但是，有时超参数空间是巨大的，使用蛮力将花费太多时间。

演化算法（EA）已被证明其功能强大。最令人印象深刻的结果之一是生命进化。进化中使用的**优化算法**已经被深入研究，其中之一是遗传算法。这个算法受到生命进化的启发，它使用进化、适应、交叉和变异等算子。这些算法背后的理论超出了本书的范围，但是本书希望简单介绍一下遗传算法来优化以下内容中的超参数。

如何去做…

1）首先导入所需函数库，如下所示：

```
from functools import reduce
from operator import add
import random

from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.utils.np_utils import to_categorical
from keras.callbacks import EarlyStopping,
ModelCheckpoint
```

2）导入函数库后，设置一些超参数：

```
n_classes = 10
batch_size = 128
n_epochs = 1000
```

3）接下来，加载和预处理训练数据：

```
(X_train, y_train), (X_val, y_val) = mnist.load_data()
X_train = X_train.reshape(60000, 28, 28, 1)
X_val = X_val.reshape(10000, 28, 28, 1)
X_train = X_train.astype('float32')
X_val = X_val.astype('float32')
X_val /= 255
X_val /= 255

y_train = to_categorical(y_train, n_classes)
y_val = to_categorical(y_val, n_classes)
```

4）再下来，定义一个创建和编译模型的函数。网络中的一些配置没有设定，但可以先设定为参数。其中包括丢失率、学习率以及最终全连接网络层中隐层神经元的数量：

```
def create_model(parameters, n_classes, input_shape):
    print(parameters)
    dropout = parameters['dropout']
    learning_rate = parameters['learning_rate']
    hidden_inputs = parameters['hidden_inputs']

    model = Sequential()
    model.add(Conv2D(32, (3, 3), padding='same',
input_shape=input_shape))
    model.add(Activation('relu'))
    model.add(Conv2D(32, (3, 3)))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(dropout))

    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(Activation('relu'))
    model.add(Conv2D(64, (3, 3)))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(dropout))

    model.add(Flatten())
    model.add(Dense(hidden_inputs))
    model.add(Activation('relu'))
    model.add(Dropout(dropout))
    model.add(Dense(n_classes))
    model.add(Activation('softmax'))
    opt = Adam(learning_rate)
    model.compile(loss='categorical_crossentropy',
    optimizer=opt, metrics=['accuracy'])

    return model
```

5）需要定义一个类 Network，可以使用它来创建一个具有随机参数的网络并训练网络，而且它应该能够检索网络的精度：

```
class Network():
    def __init__(self, parameter_space=None):
        self.accuracy = 0.
        self.parameter_space = parameter_space
        self.network_parameters = {}

    def set_random_parameters(self):
        for parameter in self.parameter_space:
            self.network_parameters[parameter] =
random.choice(self.parameter_space[parameter])
```

```
    def create_network(self, network):
        self.network_parameters = network

    def train(self):
        model = create_model(self.network_parameters,
        n_classes, input_shape)
        history = model.fit(X_train, y_train,
        batch_size=batch_size, epochs=n_epochs,
        verbose=0, validation_data=(X_val, y_val),
        callbacks=callbacks)
        self.accuracy = max(history.history['val_acc'])
```

6）接下来，将再定义一个 GA 种群类。通过所定义的 GA 类能够创建一个随机的种群并进化——包括繁殖与突变。此外，它应该能够检索种群中网络的一些统计数据：

```
class Genetic_Algorithm():
    def __init__(self, parameter_space, retain=0.3,
random_select=0.1, mutate_prob=0.25):
        self.mutate_prob = mutate_prob
        self.random_select = random_select
        self.retain = retain
        self.parameter_space = parameter_space

    def create_population(self, count):
        population = []

    for _ in range(0, count):
        network = Network(self.parameter_space)
        network.set_random_parameters()
        population.append(network)
    return population

def get_fitness(network):
    return network.accuracy

def get_grade(self, population):
    total = reduce(add, (self.fitness(network)
    for network in population))
    return float(total) / len(population)

def breed(self, mother, father):
    children = []
    for _ in range(2):
        child = {}
        for param in self.parameter_space:
            child[param] = random.choice(
                [mother.network[param],
                father.network[param]]
            )
```

```
            network = Network(self.nn_param_choices)
            network.create_set(child)
            if self.mutate_chance > random.random():
                network = self.mutate(network)
            children.append(network)
        return children

    def mutate(self, network):
        mutation = random.choice(list
        (self.parameter_space.keys()))
        network.network[mutation] =
        random.choice(self.parameter_space[mutation])
        return network

    def evolve(self, pop):
        graded = [(self.fitness(network),
        network) for network in pop]
        graded = [x[1] for x in sorted(graded,
        key=lambda x: x[0], reverse=True)]
        retain_length = int(len(graded)*self.retain)

        parents = graded[:retain_length]

        for individual in graded[retain_length:]:
            if self.random_select > random.random():
                    parents.append(individual)

            parents_length = len(parents)
            desired_length = len(pop) - parents_length
            children = []

            while len(children) < desired_length:

                male = random.randint(0,
                parents_length-1)
                female = random.randint(0,
                parents_length-1)

                if male != female:
                    male = parents[male]
                    female = parents[female]

                    children_new = self.breed(male,
                     female)

                    for child_new in children_new:
                        if len(children) < desired_length:
                            children.append(child_new)

            parents.extend(children)

            return parents
```

7）通过最后一个函数将检索一个种群的平均精度：

```
def get_population_accuracy(population):
    total_accuracy = 0
    for network in population:
        total_accuracy += network.get_accuracy

    return total_accuracy / len(population)
```

8）现在可以设置想要探索的其余超参数：

```
n_generations = 10
population_size = 20

parameter_space = {
    'dropout': [0.25, 0.5, 0.75],
    'hidden_inputs': [256, 512, 1024],
    'learning_rate': [0.1, 0.01, 0.001, 0.0001]
}
```

9）接下来，经由遗传算法来创建一个种群：

```
GA = Genetic_Algorithm(parameter_space)
population = GA.create_population(population_size)
```

10）开始训练遗传算法：

```
for i in range(n_generations):
    print('Generation {}'.format(i))

    for network in population:
        network.train()

    average_accuracy = get_population_accuracy(population)
    print('Average accuracy: {:.2f}'.
    format(average_accuracy))

    # 进化过程
    if i < n_generations - 1:
        s = GA.evolve(networks)
```

在本节内容中，专注于优化超参数，但是 GA 也可以用于进化完整的网络结构。例如，神经网络图层的数量和类型。

第 12 章
超参数选择、调优和神经网络学习

本章提供了关于神经网络学习过程中涉及的许多方面的一系列方案。方案的总体目标是提供非常简洁和具体的技巧来提升网络的性能。

- 用 TensorBoard 和 Keras 可视化训练过程；
- 使用批量和小批量工作；
- 使用网格搜索调整参数；
- 学习率和学习率调度；
- 比较优化器；
- 确定网络的深度；
- 添加 Dropout 以防止过拟合；
- 通过数据增广使模型更鲁棒；
- 利用 TTA 来提高精度。

12.1 简介

优化和调整深度学习体系结构可能具有挑战性。有时，超参数的微小变化会对训练产生很大的影响，导致性能下降。另一方面，过拟合训练数据是机器学习中众所周知的问题。在本章中，将演示如何应对这两者。另外，也将演示超参数调整的解决方案和步骤。

12.2 用 TensorBoard 和 Keras 可视化训练过程

如果可以可视化度量结果，分析训练结果（在训练期间或训练后）将会更方便。一个很好的工具是 TensorBoard，最初为 TensorFlow 开发，它也可以用于其他框架，如 Keras 和 PyTorch。TensorBoard 使人们能够跟踪损失、度量、权重、输出等。在下面的内容中，将向读者展示如何使用 Keras 的 TensorBoard，并利用它来交互式地显示训练数据。

如何去做…

1）首先，用 Python 导入所需的函数库，如下所示：

```
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping, TensorBoard,
ModelCheckpoint
from keras.utils import to_categorical
```

2）为本例加载 CIFAR10 数据集并预处理训练和验证数据：

```
(X_train, y_train), (X_val, y_val) = cifar10.load_data()

X_train = X_train.astype('float32')
X_train /=255.
X_val = X_val.astype('float32')
X_val /=255.

n_classes = 10
y_train = to_categorical(y_train, n_classes)
y_val = to_categorical(y_val, n_classes)
```

3）接下来，定义网络结构：

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
input_shape=X_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(n_classes))
model.add(Activation('softmax'))
```

4）使用默认设置的 Adam 优化器：

```
opt = Adam()
model.compile(loss='categorical_crossentropy', optimizer=opt,
metrics=['accuracy'])
```

5）接下来，使用 Keras 的 ImageDataGenerator 将数据增广添加到数据集中：

```
data_gen = ImageDataGenerator(
   rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    horizontal_flip=True,
    vertical_flip=False)

data_gen.fit(X_train)
```

6）在回调函数中，将按如下步骤添加 TensorBoard：

```
callbacks = [EarlyStopping(monitor='val_acc', patience=5,
verbose=2), ModelCheckpoint('checkpoints/weights.
{epoch:02d}.hdf5', save_best_only=True),
TensorBoard('~/notebooks/logs',
write_graph=True, write_grads=True, write_images=True,
embeddings_freq=0, embeddings_layer_names=None,
embeddings_metadata=None)]
```

7）接下来，登录一个新的终端窗口。如果登录到服务器，请确保开启一个到服务器的新连接，并对 SSH 通道使用不同的端口，例如使用 GCP（使用自己的命名替换实例区域和实例名称）：

```
gcloud compute ssh --ssh-flag="-L 6006:localhost:6006"  --zone
"instance-zone" "instance-name"
```

8）登录后，可以启动 TensorBoard 连接，如下所示：

```
tensorboard --logdir='~/notebooks/logs'
```

终端将输出可以连接 TensorBoard 的位置，例如 http：//instance-name：6006。在此实例中，因为已经启用了 SSH 通道，可以到本地浏览器并使用地址 http：//localhost：6006/ 来访问仪表板。TensorBoard 将不会显示任何数据。

9）现在回到 Python 环境，开始训练模型：

```
model.fit_generator(data_gen.flow(X_train, y_train,
batch_size=batch_size),
                    steps_per_epoch=X_train.shape[0] // batch_size,
                    epochs=n_epochs,
                    validation_data=(X_val, y_val),
                    callbacks=callbacks
                   )
```

开始训练模型后，应该看到数据进入 TensorBoard 仪表板。图 12.1 所示是 10 个周期后的输出。

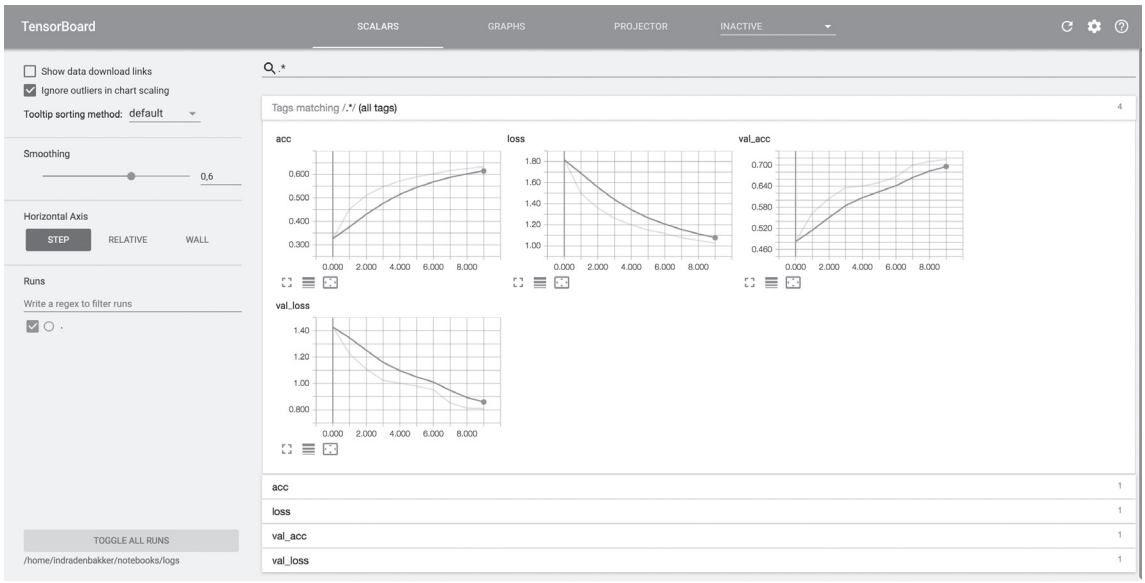


图 12.1　训练 10 个周期后的 TensorBoard 仪表板示例

TensorBoard 为监控训练数据提供了很大的灵活性。正如在前面的屏幕截图中所看到的，可以交互式地调整平滑参数。这只是与 TensorBoard 合作的许多优势之一。TensorBoard 的另一个重要特性是能够使用 T-SNE 或 PCA 可视化训练数据。切换到 PROJECTOR 选项卡，输出应该如图 12.2 所示。

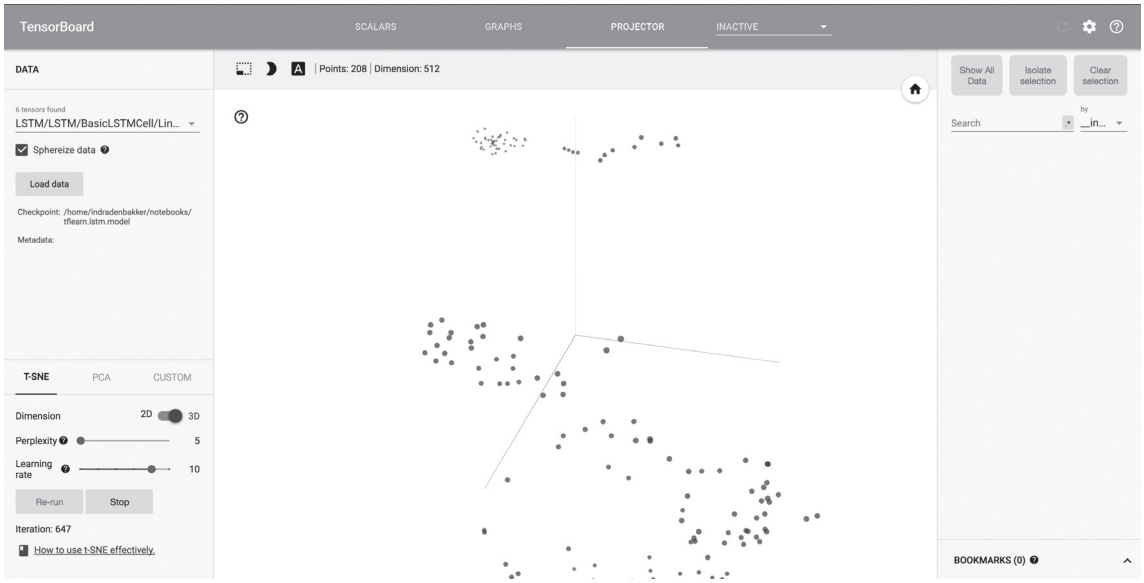


图 12.2　在训练数据上训练 T-SNE 的一个实例

12.3　使用批量和小批量工作

训练神经网络时，将训练数据加载入网络。训练数据的一次全面扫描称为一个周期。如果一步完成所有的训练数据，这称为批处理模式（批量大小等于训练集的大小）。然而在大多数情况下，将训练数据分成较小的子集，同时将数据提供给模型，就像其他机器学习算法一样，这称为小批量模式。有时候，不得不这样做，因为完整的训练集太大，不适合存储。如果仅考虑训练时间，那么批量越大越好（只要批次适合内存）。使用小批量还有其他优点：首先，它降低了训练过程的复杂性；其次，通过求和或平均梯度（减小方差）来降低噪声对模型的影响。在小批量模式下，优化器在效率和鲁棒性两者间采用某种平衡。

如果小批量的数量（也称为批量数量）太小，则该模型可以更快地收敛，但更能吸收噪声。如果批量太大，模型可能收敛慢一点，但是误差梯度的估计会更准确。对于深度学习，在处理大量数据时，使用大批量处理是适合的。这些批处理也非常适合在 GPU 上进行并行处理。在下面的内容中，将向读者展示调整批量大小时要采取的步骤及示例。

如何去做…

1）开始导入所用函数库，如下所示：

```
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping, TensorBoard,
ModelCheckpoint
```

2）使用 CIFAR10 数据集进行实验，加载数据并进行预处理：

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

X_train = X_train.astype('float32')/255.
X_test = X_test.astype('float32')/255.
n_classes = 10
y_train = keras.utils.to_categorical(y_train, n_classes)
y_test = keras.utils.to_categorical(y_test, n_classes)
```

3）作为模型结构，使用直接卷积神经网络，具有卷积块（两个卷积层）和输出层之前的完连接层：

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
input_shape=X_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(n_classes))
model.add(Activation('softmax'))
```

4）定义 Adam 优化器并编译该模型：

```
opt = Adam(lr=0.0001)
model.compile(loss='categorical_crossentropy', optimizer=opt,
metrics=['accuracy'])
```

5）因为想要使所建模型更加鲁棒，将使用一些数据增广来进行训练：

```
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    horizontal_flip=True,
    vertical_flip=False)

datagen.fit(x_train)
```

6）接下来，设置回调函数。通过使用早停方法来防止过拟合，提前用 ModelCheckpoint 自动保存最佳模型，并使用 TensorBoard 分析结果：

```
callbacks = [EarlyStopping(monitor='val_acc', patience=5,
verbose=2), ModelCheckpoint('checkpoints/weights.
{epoch:02d}-'+str(batch_size)+'.hdf5',
save_best_only=True),  TensorBoard()]
```

7）开始建立第一个模型，批量大小为 32 ：

```
batch_size = 32
n_epochs = 1000
history_32 = model.fit_generator(datagen.flow(X_train, y_train,
                                 batch_size=batch_size),
                     steps_per_epoch=X_train.shape[0] // batch_size,
                     epochs=epochs,
                     validation_data=(X_val, y_val),
                     callbacks=callbacks
                    )
```

8）接下来，重新编译该模型，以确保权重的初始化：

```
model.compile(loss='categorical_crossentropy', optimizer=opt,
metrics=['accuracy'])
```

9）现在可以开始以 256 的批量大小来训练所建模型：

```
batch_size = 256
history_32 = model.fit_generator(datagen.flow(X_train, y_train,
                                 batch_size=batch_size),
                    steps_per_epoch=X_train.shape[0] // batch_size,
                    epochs=epochs,
                    validation_data=(X_val, y_val),
                    callbacks=callbacks
                   )
```

这两个模型的验证精度与周期数量的变化情况如图 12.3 和图 12.4 所示（使用早停方法）。

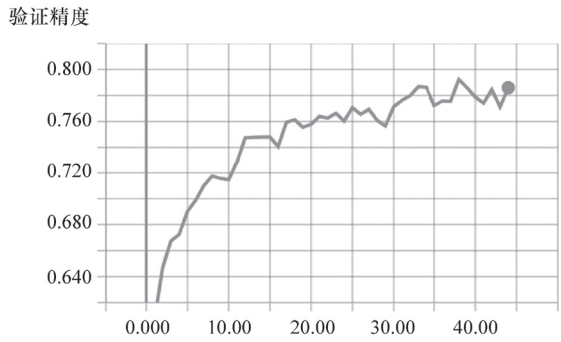


图 12.3　批量大小为 32 的模型验证精度

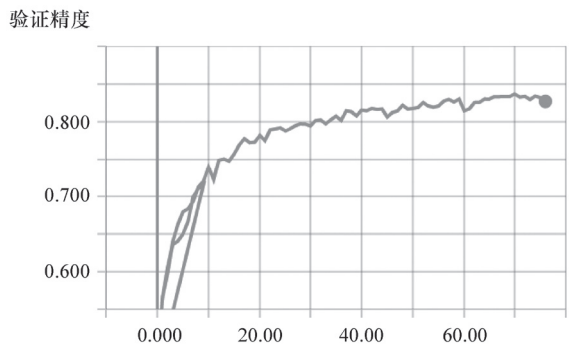


图 12.4　批量大小为 256 的模型验证精度

因此注意到两个模型的结果有很多不同之处。正如预期的那样，当使用更大批量的数据时，需要更多的周期（理论上，无论批量大小如何，收敛的步骤总数 / 更新数应该是相同的）。然而更有趣的是，批量大小为 256 的模型的验证精度略高一些，大约为 0.84%。而批量大小为 32 的模型在 79%左右。当用较小的批量训练所建模型时，模型可能会增加一点噪声。然而通过进一步微调模型（例如，使用早停方法和降低学习率），批量大小为 32 的模型应该能够达到相似的精度。

12.4 使用网格搜索调整参数

调整超参数是一项耗时且计算量巨大的任务。在本书中，只关注超参数的调优。大多数结果是用预先选定的值获得的。要选择正确的值，可以使用启发式或泛网格搜索。网格搜索是机器学习中常用的参数调整方法。

在下面的内容中，将演示如何在构建深度学习模型时应用网格搜索。为此要使用 Hyperopt 工具。

如何去做…

1）首先导入这个方案中要使用的函数库：

```
import sys
import numpy as np

from hyperopt import fmin, tpe, hp, STATUS_OK, Trials

from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
from keras.callbacks import EarlyStopping, TensorBoard,
ModelCheckpoint

from keras.datasets import cifar10
```

2）接下来，加载 cifar10 数据集并预处理数据：

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
validation_split = 0.1
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=validation_split, random_state=SEED)

X_train = X_train.astype('float32')
X_train /=255.
X_val = X_val.astype('float32')
X_val /=255.
X_test = X_test.astype('float32')
X_test /=255.

n_classes = 10
y_train = to_categorical(y_train, n_classes)
y_val = to_categorical(y_val, n_classes)
y_test = to_categorical(y_test, n_classes)
```

3）可以为不同的超参数设置搜索空间：

```
space = {
            'batch_size' : hp.choice('batch_size', [32, 64, 128,
            256]), 'n_epochs' : 1000,
        }
```

当使用 Hyperopt 时，将有很多的自由度及设置可进行测试，也可以调整网络结构。例如，可以动态设置图层的数量。

4）通过 TensorBoard 使用回调函数来进行训练过程的早停、设置检查点并可视化结果：

```
def get_callbacks(pars):
    callbacks =[EarlyStopping(monitor='val_acc', p    atience=5,
verbose=2),
ModelCheckpoint('checkpoints/{}.h5'.format(pars['batch_size']),
save_best_only=True),
             TensorBoard('~/notebooks/logs-gridsearch',
write_graph=True, write_grads=True, write_images=True,
embeddings_freq=0, embeddings_layer_names=None,
embeddings_metadata=None)]
    return callbacks
```

5）接下来，定义一个想要最小化的目标函数。这个函数包括网络结构、超参数、结果以及其他想要包含的内容。确保在成功运行的 return 语句中包含状态 STATUS_OK：

```
def f_nn(pars):
    print ('Parameters: ', pars)
    model = Sequential()
    model.add(Conv2D(32, (3, 3), padding='same',
input_shape=X_train.shape[1:]))
    model.add(Activation('relu'))
    model.add(Conv2D(32, (3, 3)))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(Activation('relu'))
    model.add(Conv2D(64, (3, 3)))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Flatten())
    model.add(Dense(512))
    model.add(Activation('relu'))
    model.add(Dropout(0.5))
    model.add(Dense(n_classes))
    model.add(Activation('softmax'))
    optimizer = Adam()
```

```
    model.compile(loss='categorical_crossentropy',
optimizer=optimizer, metrics=['accuracy'])
    history = model.fit(X_train, y_train, epochs=pars['n_epochs'],
batch_size=pars['batch_size'],
              validation_data=[X_val, y_val],
              verbose = 1, callbacks=get_callbacks(pars))
    best_epoch = np.argmax(history.history['val_acc'])
    best_val_acc = np.max(history.history['val_acc'])
    print('Epoch {} - val acc: {}'.format(best_epoch,
best_val_acc))
    sys.stdout.flush()
    return {'val_acc': best_val_acc, 'best_epoch': best_epoch,
'eval_time': time.time(), 'status': STATUS_OK}
```

6）现在可以初始化，并开始网格搜索，并呈现最终结果如下：

```
trials = Trials()
best = fmin(f_nn, space, algo=tpe.suggest, max_evals=50,
trials=trials)
print(best)
```

如果想通过多个 GPU 并行进行训练，则需要使用 MongoDB 创建处理异步更新的数据库。

12.5 学习率和学习率调度

使用较小的学习率有助于避免局部最优化，但是通常需要较长的时间才能收敛。什么可以帮助缩短训练时间？可以使用一个预热期。在这个时期，可以在前几个周期使用更大的学习率。经过一定的周期后，可以降低学习率。甚至有可能在每一步后都降低学习速度，但不推荐这样做，因为可能更会更好地使用不同的优化器（例如，如果想使用衰减，可以指定它作为超参数）。从理论上讲，当预热时段的学习速度太快时，可能就根本无法达到全局的最优状态。

在下面的内容中，将演示如何设置 Keras 自定义学习速率调度程序。

如何去做…

1）开始导入所用函数库，如下所示：

```
import math

from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import SGD
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
from keras.callbacks import EarlyStopping, TensorBoard,
ModelCheckpoint, LearningRateScheduler, Callback
from keras import backend as K
from keras.datasets import cifar10
```

2）接下来，加载和预处理数据：

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
validation_split = 0.1
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=validation_split, random_state=SEED)

X_train = X_train.astype('float32')
X_train /=255.
X_val = X_val.astype('float32')
X_val /=255.
X_test = X_test.astype('float32')
X_test /=255.

n_classes = 10
y_train = to_categorical(y_train, n_classes)
y_val = to_categorical(y_val, n_classes)
y_test = to_categorical(y_test, n_classes)
```

3）设置学习率，如下：

```
learning_rate_schedule = {0: '0.1', 10: '0.01', 25: '0.0025'}

class get_learning_rate(Callback):
    def on_epoch_end(self, epoch, logs={}):
        optimizer = self.model.optimizer
        if epoch in learning_rate_schedule:
            K.set_value(optimizer.lr,
learning_rate_schedule[epoch])
        lr = K.eval(optimizer.lr)
        print('\nlr: {:.4f}'.format(lr))
```

除了自定义的回调函数，Keras 还提供了一个方便的 LearningRateScheduler 和 ReduceLROnPlateau 回调函数。通过这些回调函数，如果受监控的损失函数或度量达到稳定状态，可以实现一个与周期相关的学习率方案或降低学习率。

4）将自定义函数添加到回调函数列表中：

```
callbacks =[EarlyStopping(monitor='val_acc', patience=5,
verbose=2),
            ModelCheckpoint('checkpoints/{epoch:02d}.h5',
            save_best_only=True),
            TensorBoard('~/notebooks/logs-lrscheduler',
            write_graph=True, write_grads=True,
            write_images=True, embeddings_freq=0,
            embeddings_layer_names=None,
            embeddings_metadata=None),
            get_learning_rate()
            ]
```

5）现在可以定义和编译所建模型：

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
input_shape=X_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(n_classes))
model.add(Activation('softmax'))
optimizer = SGD()
model.compile(loss='categorical_crossentropy',
optimizer=optimizer, metrics=['accuracy'])
```

6）最后，可以开始训练。读者会注意到学习率符合设定的进度：

```
n_epochs = 20
batch_size = 128

history = model.fit(X_train, y_train, epochs=n_epochs,
batch_size=batch_size,
          validation_data=[X_val, y_val],
          verbose = 1, callbacks=callbacks)
```

12.6 比较优化器

在第 2 章 前馈神经网络 中，简要地演示了使用不同的优化器。当然，只使用了一个大小为 1 的测试集。对于其他机器学习算法，深度学习中使用最广泛和最著名的优化器是**随机梯度下降**（SGD）。其他优化器是 SGD 的变种，试图通过增加启发式来加速收敛。另外，有些优化器调整的超参数较少。第 2 章 前馈神经网络 中的表格对深度学习中最常用的优化器进行了概述。

有人可能会争辩说，这种选择在很大程度上取决于用户调整优化器的能力。绝对没有理想的解决方案能适合所有问题，但是有些优化器要调整的参数很少，并且已经被证明优于其他默认设置的优化器。除了在第 2 章 前馈神经网络 中的测试之外，将在下面的内容中进行另一个测试来比较优化器。

如何去做…

1）首先加载函数库，如下：

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.wrappers.scikit_learn import KerasRegressor
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras.optimizers import SGD, Adadelta, Adam, RMSprop, Adagrad,
Nadam, Adamax
```

2）对于此测试，将创建训练集、验证集和测试集，并对所有数据集进行预处理，如下所示：

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
validation_split = 0.1
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=validation_split, random_state=SEED)

X_train = X_train.astype('float32')
X_train /=255.
X_val = X_val.astype('float32')
X_val /=255.
X_test = X_test.astype('float32')
X_test /=255.

n_classes = 10
y_train = to_categorical(y_train, n_classes)
y_val = to_categorical(y_val, n_classes)
y_test = to_categorical(y_test, n_classes)
```

3）接下来，定义一个创建模型的函数：

```
def create_model(opt):
 model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
 return model
```

4）另外，需要创建一个函数来定义在训练期间要使用的回调函数：

```
def create_callbacks(opt):
    callbacks = [EarlyStopping(monitor='val_acc', patience=5,
verbose=2),
             ModelCheckpoint('checkpoints/weights.{epoch:02d}-
'+opt+'.h5', save_best_only=False, verbose=True),
             TensorBoard()]
    return callbacks
```

5）创建一个人们想要尝试的优化器字典：

```
 opts = dict({
 'sgd': SGD(),
 'sgd-0001': SGD(lr=0.0001, decay=0.00001),
 'adam': Adam(),
 'adam': Adam(lr=0.0001),
 'adadelta': Adadelta(),
 'rmsprop': RMSprop(),
 'rmsprop-0001': RMSprop(lr=0.0001),
 'nadam': Nadam(),
 'adamax': Adamax()
})
```

也可以使用 Hyperopt 来运行不同的优化器，而不是实现自己的脚本。请参阅使用网格搜索调整参数相关内容。

6）训练网络并存储结果：

```
n_epochs = 1000
batch_size = 128

results = []
# 遍历优化器
for opt in opts:
    model = create_model(opt)
    callbacks = create_callbacks(opt)
    model.compile(loss='categorical_crossentropy',
optimizer=opts[opt], metrics=['accuracy'])
    hist = model.fit(X_train, y_train, batch_size=batch_size,
epochs=n_epochs,
    validation_data=(X_val, y_val),
    verbose=1,
    callbacks=callbacks)
    best_epoch = np.argmax(hist.history['val_acc'])
    best_acc = hist.history['val_acc'][best_epoch]
    best_model = create_model(opt)
    # 加载具有最高验证精度的模型权重
    best_model.load_weights('checkpoints/weights.{:02d}-
{}.h5'.format(best_epoch, opt))
    best_model.compile(loss='mse', optimizer=opts[opt],
metrics=['accuracy'])
    score = best_model.evaluate(X_test, y_test, verbose=0)
    results.append([opt, best_epoch, best_acc, score[1]])
```

7）比较结果：

```
res = pd.DataFrame(results)
res.columns = ['optimizer', 'epochs', 'val_accuracy', 'test_last',
'test_accuracy']
res
```

8）得到的结果如图 12.5 所示。

	优化器	迭代周期	验证精度	测试精度
0	sgd_0001	106	0.519531	0.53750
1	adam	112	0.589844	0.61250
2	sgd	0	0.000000	0.000000
3	adamax	213	0.578125	0.58750
4	rmsprop	600	0.597656	0.61250
5	adadelta	109	0.570312	0.58125
6	rmsprop–0001	38	0.539062	0.56250
7	nadam	84	0.597656	0.59375

图 12.5　使用不同优化器对 cifar10 数据集进行训练的结果

12.7　确定网络的深度

当从头开始建立深度学习模型时，事先很难确定应该堆叠多少（不同类型）层。一般来说，参看一个著名的深度学习模型是一个好主意，并将其作为进一步构建的基础。一般来说，尽可能多地尝试过拟合训练数据是很好的，这确保模型能够在输入数据上进行训练。之后，运用 dropout 等正则化技术来防止过拟合，提升泛化能力。

12.8　添加 Dropout 以防止过拟合

在第 2 章　前馈神经网络　中介绍了 Dropout，而且在整本书中都使用了 Dropout。就像第 2 章一样，在下面的内容中将演示添加 Dropout 所带来的性能差异，这一次将使用 CIFAR10 数据集。

如何去做…

1）首先导入所用函数库，如下所示：

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
from keras.callbacks import EarlyStopping, TensorBoard,
ModelCheckpoint

from keras.datasets import cifar10
```

2）接下来，加载 CIFAR10 数据集并对其进行预处理：

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
validation_split = 0.1
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=validation_split, random_state=SEED)

X_train = X_train.astype('float32')
X_train /=255.
X_val = X_val.astype('float32')
X_val /=255.
X_test = X_test.astype('float32')
X_test /=255.

n_classes = 10
y_train = to_categorical(y_train, n_classes)
y_val = to_categorical(y_val, n_classes)
y_test = to_categorical(y_test, n_classes)
```

3）为了监控训练并防止过拟合，引入回调函数：

```
callbacks =[EarlyStopping(monitor='val_acc', patience=5,
verbose=2),
            ModelCheckpoint('checkpoints/{epoch:02d}.h5',
save_best_only=True),
            TensorBoard('~/notebooks/logs-lrscheduler',
write_graph=True, write_grads=True, write_images=True,
embeddings_freq=0, embeddings_layer_names=None,
embeddings_metadata=None),
            ]
```

4）接下来，定义模型结构并编译模型：

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
input_shape=X_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
# model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
# model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
# model.add(Dropout(0.5))
model.add(Dense(n_classes))
model.add(Activation('softmax'))
optimizer = SGD()
model.compile(loss='categorical_crossentropy',
optimizer=optimizer, metrics=['accuracy'])
```

5）开始训练：

```
n_epochs = 1000
batch_size = 128

history = model.fit(X_train, y_train, epochs=n_epochs,
batch_size=batch_size,
          validation_data=[X_val, y_val],
          verbose = 1, callbacks=callbacks)
```

6）现在，添加 Dropout 到所建模型结构中。在每个卷积块之后和全连接层之后执行此操作：

```
model_dropout = Sequential()
model_dropout.add(Conv2D(32, (3, 3), padding='same',
input_shape=X_train.shape[1:]))
model_dropout.add(Activation('relu'))
model_dropout.add(Conv2D(32, (3, 3)))
model_dropout.add(Activation('relu'))
model_dropout.add(MaxPooling2D(pool_size=(2, 2)))
model_dropout.add(Dropout(0.25))

model_dropout.add(Conv2D(64, (3, 3), padding='same'))
model_dropout.add(Activation('relu'))
model_dropout.add(Conv2D(64, (3, 3)))
model_dropout.add(Activation('relu'))
model_dropout.add(MaxPooling2D(pool_size=(2, 2)))
model_dropout.add(Dropout(0.25))

model_dropout.add(Flatten())
model_dropout.add(Dense(512))
model_dropout.add(Activation('relu'))
model_dropout.add(Dropout(0.5))
model_dropout.add(Dense(n_classes))
model_dropout.add(Activation('softmax'))
optimizer = Adam()
model_dropout.compile(loss='categorical_crossentropy',
optimizer=optimizer, metrics=['accuracy'])
```

7）再次从头开始训练：

```
n_epochs = 1000
batch_size = 128

history_dropout = model_dropout.fit(X_train, y_train,
epochs=n_epochs, batch_size=batch_size,
          validation_data=[X_val, y_val],
          verbose = 1, callbacks=callbacks)
```

8）绘制没有 Dropout 模型的训练和验证精度：

```
aplt.plot(np.arange(len(history.history['acc'])),
history.history['acc'], label='training')
plt.plot(np.arange(len(history.history['val_acc'])),
history.history['val_acc'], label='validation')
plt.title('Accuracy of model without dropouts')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.legend(loc=0)
plt.show()
```

从图 12.6 中可以看到模型显然对训练数据进行了过拟合。

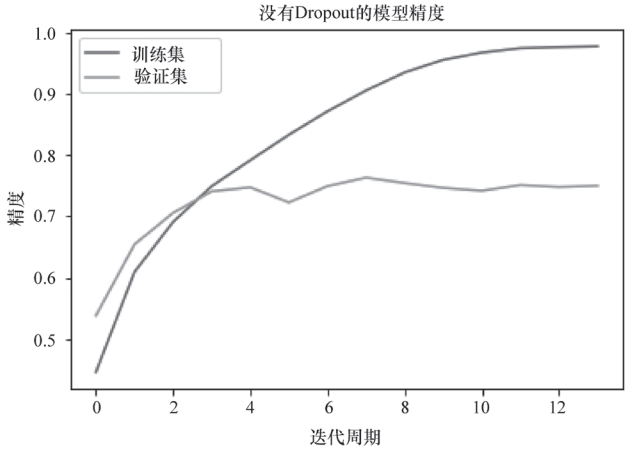


图 12.6　没有 Dropout 模型的精度

9）最后，具有 Dropout 模型的训练和验证精度：

```
plt.plot(np.arange(len(history_dropout.history['acc'])),
history_dropout.history['acc'], label='training')
plt.plot(np.arange(len(history_dropout.history['val_acc'])),
history_dropout.history['val_acc'], label='validation')
plt.title('Accuracy of model with dropouts')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.legend(loc=0)
plt.show()
```

现在，可以看到，由于 Dropout，所建模型能够更好地泛化，并且不会过拟合训练数据，然而还有改进的空间，如图 12.7 所示。

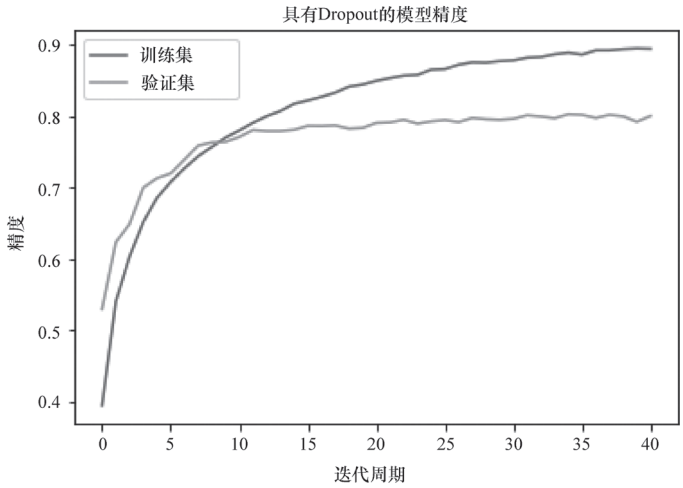


图 12.7　有 Dropout 模型的精度

12.9 通过数据增广使模型更加鲁棒

提高用于计算机视觉任务网络性能的常用方法是进行数据增广。通过在训练时间内使用数据增广，可以增加训练集的大小。因此，可以使模型对训练数据中的轻微变化更鲁棒。在第 7 章 计算机视觉 中，演示了一些数据增广技术。在下面的内容中，将使用 Keras 和它的 ImageDataGenerator 进行数据增广。

如何去做…

1）开始像往常一样导入所需的函数库：

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping, TensorBoard,
ModelCheckpoint

from keras.datasets import cifar10
```

2）加载并对训练数据和验证数据进行预处理，如下：

```
(X_train, y_train), (X_val, y_val) = cifar10.load_data()

X_train = X_train.astype('float32')/255.
X_val = X_val.astype('float32')/255.
n_classes = 10
y_train = keras.utils.to_categorical(y_train, n_classes)
y_val = keras.utils.to_categorical(y_val, n_classes)
```

3）接下来，按如下步骤来定义模型结构：

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
input_shape=X_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

```
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(n_classes))
model.add(Activation('softmax'))
```

4）定义 Adam 优化器并编译模型：

```
opt = Adam(lr=0.0001)
model.compile(loss='categorical_crossentropy',
optimizer=opt, metrics=['accuracy'])
```

5）在 Keras 中，可以使用 ImageDataGenerator 来轻松设置图像增广，如下所示：

```
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    horizontal_flip=True,
    vertical_flip=False)

datagen.fit(X_train)
```

6）接下来，设置回调函数：

```
callbacks = [EarlyStopping(monitor='val_acc', patience=5,
verbose=2), ModelCheckpoint('checkpoints/weights.{epoch:02d}-
'+str(batch_size)+'.hdf5', save_best_only=True),
TensorBoard('~/notebooks/logs-lrscheduler',
write_graph=True, write_grads=True, write_images=True,
embeddings_freq=0, embeddings_layer_names=None,
embeddings_metadata=None)
]
```

7）开始训练模型：

```
batch_size = 128
n_epochs = 1000
history = model.fit_generator(datagen.flow(X_train, y_train,
                                batch_size=batch_size),
                     steps_per_epoch=X_train.shape[0] // batch_size,
                     epochs=epochs,
                     validation_data=(X_val, y_val),
                     callbacks=callbacks
                    )
```

12.10 利用 TTA 来提高精度

虽然训练期间的数据增广是众所周知的且广泛使用的技术，但大多数人对测试时间增广（TTA）不太了解。在使用 TTA 时，可以将训练模型用不同的视图分类，例如图像。如果翻转或轻微旋转图像，训练模型能够做出更准确的预测。使用 TTA 可以看作多个模型的集合。可以选择平均概率或任何其他综合技术来组合每个单独预测的结果。

第 13 章
网络内部构造

本章提供了关于神经网络内部构造的一系列内容，这包括张量分解、权重初始化、拓扑存储、瓶颈特征和相应的嵌入。

本章将包含以下内容：

- 用 TensorBoard 可视化训练过程；
- 用 TensorBoard 可视化网络结构；
- 分析网络权重等；
- 冻结层；
- 存储网络结构并训练权重。

13.1 简介

在本书中，专注于为深度学习提供构建基础，并为实际问题提供实战解决方案。在本章中，要关注这些强大的深度学习网络背后的细节。例如，将演示如何使用 TensorBoard 分析训练后的权重和可视化训练过程。

13.2 用 TensorBoard 可视化训练过程

在前面的内容中，演示了如何用 Keras 建立 TensorBoard。但是正如已经提到的，TensorBoard 也可以用于 TensorFlow 等。在本节内容中，将向读者展示如何在分类 Fashion-MNIST 数据集时使用 TensorBoard 和 TensorFlow。

如何去做…

1）首先导入 TensorFlow 并加载一个 MNIST 数据集的工具，如下所示：

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

2）接下来，确定 Fashion-MNIST 数据集并进行加载：

```
mnist = input_data.read_data_sets('Data/fashion', one_hot=True)
```

3）为输入数据创建占位符：

```
n_classes = 10
input_size = 784

x = tf.placeholder(tf.float32, shape=[None, input_size])
y = tf.placeholder(tf.float32, shape=[None, n_classes])
```

4）在确定网络结构之前，定义一些将在模型中多次使用的函数。从一个创建并初始化权重的函数开始：

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)
```

5）为偏置创建一个类似的函数：

```
def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

6）在网络结构中，使用包含一个 Max-pooling 层的多个卷积块：

```
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1], padding='SAME')
```

7）接下来，定义完整的网络结构。将使用三个卷积块，然后是两个全连接网络层：

```
W_conv1 = weight_variable([7, 7, 1, 100])
b_conv1 = bias_variable([100])
x_image = tf.reshape(x, [-1,28,28,1])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

W_conv2 = weight_variable([4, 4, 100, 150])
b_conv2 = bias_variable([150])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

W_conv3 = weight_variable([4, 4, 150, 250])
b_conv3 = bias_variable([250])
h_conv3 = tf.nn.relu(conv2d(h_pool2, W_conv3) + b_conv3)
h_pool3 = max_pool_2x2(h_conv3)
```

```
W_fc1 = weight_variable([4 * 4 * 250, 300])
b_fc1 = bias_variable([300])
h_pool3_flat = tf.reshape(h_pool3, [-1, 4*4*250])
h_fc1 = tf.nn.relu(tf.matmul(h_pool3_flat, W_fc1) + b_fc1)
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = weight_variable([300, n_classes])
b_fc2 = bias_variable([n_classes])
y_pred = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

8）接下来，需要按如下步骤提取交叉熵：

```
with tf.name_scope('cross_entropy'):
    diff = tf.nn.softmax_cross_entropy_with_logits(labels=y,
logits=y_pred)
    with tf.name_scope('total'):
        cross_entropy = tf.reduce_mean(diff)
tf.summary.scalar('cross_entropy', cross_entropy)
```

9）训练期间将使用 AdamOptimizer 优化器，学习率为 0.001：

```
learning_rate = 0.001
train_step =
tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
```

10）为了跟踪进度，需要提取精度，如下所示：

```
with tf.name_scope('accuracy'):
    correct_prediction = tf.equal(tf.argmax(y_pred, 1),
tf.argmax(y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))
tf.summary.scalar('accuracy', accuracy)
```

11）创建一个交互式的 TensorFlow 会话：

```
sess = tf.InteractiveSession()
```

12）为 TensorBoard 设置 TensorFlow 的摘要编写器：

```
log_dir = 'tensorboard-example'
merged = tf.summary.merge_all()
train_writer = tf.summary.FileWriter(log_dir + '/train',
sess.graph)
val_writer = tf.summary.FileWriter(log_dir + '/val')
```

13）接下来，在开始训练之前定义一些超参数：

```
n_steps = 1000
batch_size = 128
dropout = 0.25
evaluate_every = 10
```

14）最后，开始训练：

```
tf.global_variables_initializer().run()
for i in range(n_steps):
    x_batch, y_batch = mnist.train.next_batch(batch_size)
    summary, _, train_acc = sess.run([merged, train_step,
    accuracy], feed_dict={x: x_batch, y: y_batch, keep_prob:
    dropout})
    train_writer.add_summary(summary, i)
    if i % evaluate_every == 0:
        summary, val_acc = sess.run([merged, accuracy],
        feed_dict={x: mnist.test.images, y: mnist.test.labels,
        keep_prob: 1.0})
        val_writer.add_summary(summary, i)
        print('Step {:04.0f}: train_acc: {:.4f}; val_acc
        {:.4f}'.format(i, train_acc, val_acc))
train_writer.close()
val_writer.close()
```

15）连接 TensorBoard，打开一个新的终端窗口。如果登录到服务器，请确保启动一个到服务器的新连接，并使用不同的端口进行 SSH 通道连接，例如使用 GCP 时（使用自己的设置替换实例和名称）：

```
gcloud compute ssh --ssh-flag="-L 6006:localhost:6006"  --zone
"instance-zone" "instance-name"
```

16）登录后，可以启动 TensorBoard 连接，如下所示：

```
tensorboard --logdir='~/tensorflow-example'
```

终端将输出一个地址，通过这个地址可以连接 TensorBoard，例如 http：// instance-name：6006。在这里的例子中，因为已经启用了 SSH 通道，所以可以利用本地浏览器并使用地址 http：// localhost：6006 / 来访问仪表板。在 TensorFlow 仪表板中，可以跟踪模型的进度。在图 13.1 中，可以看到训练完成后的输出。

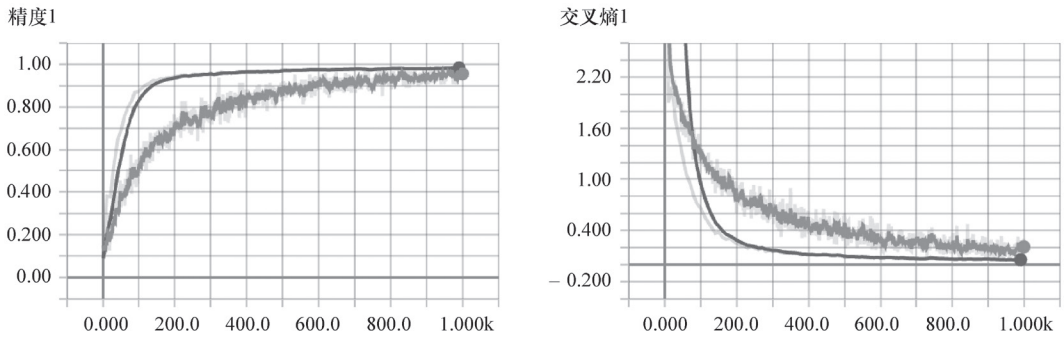


图 13.1　迭代 1000 次以后的训练（橙色）和验证（蓝色）结果

13.3　用 TensorBoard 可视化网络结构

正如在本书中所展示的那样，在使用 Keras 框架时，很容易输出网络结构的简单摘要。但是使用 TensorBoard，可以使用不同的设置来显示网络结构。在图 13.2 中，可以看到这样一个可视化的例子。

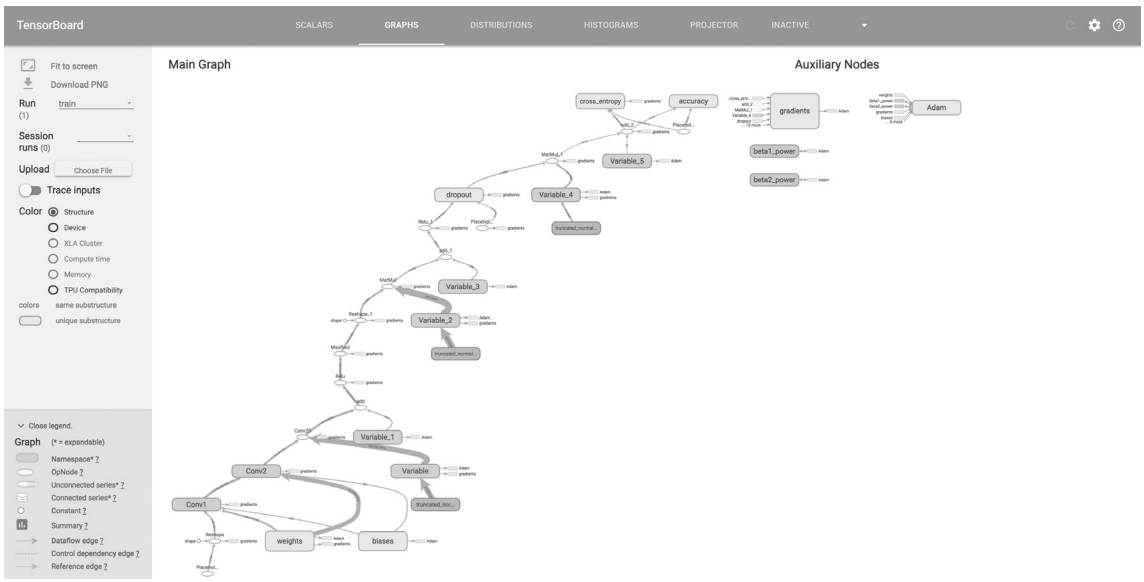


图 13.2　TensorBoard 图形可视化示例

13.4　分析网络权重等

在之前的内容中，着重于可视化损失和度量。但是使用 TensorBoard，还可以跟踪权重变化过程。仔细观察权重可以帮助理解模型是如何工作和学习的。

如何去做…

1）从导入 TensorFlow 开始，如下所示：

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

2）现在，可以用一行代码加载 Fashion-MNIST 数据集：

```
mnist = input_data.read_data_sets('Data/fashion', one_hot=True)
```

3）在继续之前，需要为模型设置占位符：

```
n_classes = 10
input_size = 784

x = tf.placeholder(tf.float32, shape=[None, input_size])
y = tf.placeholder(tf.float32, shape=[None, n_classes])
keep_prob = tf.placeholder(tf.float32)
```

4）定义四个函数来帮助构建网络结构：

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1], padding='SAME')
```

5）要将统计数据写入 TensorBoard，定义一个提取这些参量的函数：

```
def summary_variable(var):
    with tf.name_scope('summaries'):
        mean = tf.reduce_mean(var)
        tf.summary.scalar('mean', mean)
        with tf.name_scope('stdev'):
```

```
            stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
        tf.summary.scalar('stdev', stddev)
        tf.summary.scalar('max', tf.reduce_max(var))
        tf.summary.scalar('min', tf.reduce_min(var))
        tf.summary.histogram('histogram', var)
```

6）接下来，定义网络结构。对于前两个卷积块，这里想在训练期间提取权重和统计数据。操作如下：

```
x_image = tf.reshape(x, [-1,28,28,1])
with tf.name_scope('weights'):
    W_conv1 = weight_variable([7, 7, 1, 100])
    summary_variable(W_conv1)
with tf.name_scope('biases'):
    b_conv1 = bias_variable([100])
    summary_variable(b_conv1)

with tf.name_scope('Conv1'):
    h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
    h_pool1 = max_pool_2x2(h_conv1)
    tf.summary.histogram('activations', h_conv1)
    tf.summary.histogram('max_pool', h_pool1)

with tf.name_scope('weights'):
    W_conv2 = weight_variable([4, 4, 100, 150])
    summary_variable(W_conv2)
with tf.name_scope('biases'):
    b_conv2 = bias_variable([150])
    summary_variable(b_conv2)

with tf.name_scope('Conv2'):
    h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
    h_pool2 = max_pool_2x2(h_conv2)
    tf.summary.histogram('activations', h_conv2)
    tf.summary.histogram('max_pool', h_pool2)
```

7）对于其余网络层，不存储任何信息：

```
W_conv3 = weight_variable([4, 4, 150, 250])
b_conv3 = bias_variable([250])
h_conv3 = tf.nn.relu(conv2d(h_pool2, W_conv3) + b_conv3)
h_pool3 = max_pool_2x2(h_conv3)

W_fc1 = weight_variable([4 * 4 * 250, 300])
b_fc1 = bias_variable([300])
h_pool3_flat = tf.reshape(h_pool3, [-1, 4*4*250])
h_fc1 = tf.nn.relu(tf.matmul(h_pool3_flat, W_fc1) + b_fc1)
```

```
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = weight_variable([300, n_classes])
b_fc2 = bias_variable([n_classes])
y_pred = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

8）但是需要保存交叉熵和精度：

```
with tf.name_scope('cross_entropy'):
    diff = tf.nn.softmax_cross_entropy_with_logits(labels=y,
logits=y_pred)
    with tf.name_scope('total'):
        cross_entropy = tf.reduce_mean(diff)
tf.summary.scalar('cross_entropy', cross_entropy)

with tf.name_scope('accuracy'):
    correct_prediction = tf.equal(tf.argmax(y_pred, 1),
tf.argmax(y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))
tf.summary.scalar('accuracy', accuracy)
```

9）设置学习率并定义优化器：

```
learning_rate = 0.001
train_step =
tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
```

10）建立一个会话，并创建文件编写器：

```
sess = tf.InteractiveSession()
log_dir = 'tensorboard-example-weights'
merged = tf.summary.merge_all()
train_writer = tf.summary.FileWriter(log_dir + '/train',
sess.graph)
val_writer = tf.summary.FileWriter(log_dir + '/val')
```

11）接下来，定义超参数：

```
n_steps = 1000
batch_size = 128
dropout = 0.25
evaluate_every = 10
n_val_steps = mnist.test.images.shape[0] // batch_size
```

12）开始训练：

```
tf.global_variables_initializer().run()
for i in range(n_steps):
    x_batch, y_batch = mnist.train.next_batch(batch_size)
    summary, _, train_acc = sess.run([merged, train_step,
    accuracy], feed_dict={x: x_batch, y: y_batch,
    keep_prob: dropout})
    train_writer.add_summary(summary, i)

    if i % evaluate_every == 0:
        val_accs = []
        for j in range(n_val_steps):
            x_batch, y_batch = mnist.test.next_batch(batch_size)
            summary, val_acc = sess.run([merged, accuracy],
            feed_dict={x: x_batch, y: y_batch, keep_prob: 1.0})
            val_writer.add_summary(summary, i)
            val_accs.append(val_acc)
        print('Step {:04.0f}: train_acc: {:.4f}; val_acc
        {:.4f}'.format(i, train_acc, sum(val_accs)/len(val_accs)))
train_writer.close()
val_writer.close()
```

有关如何连接 TensorBoard 的更多信息，请参阅使用 TensorBoard 进行可视化训练相关内容。图 13.3 显示了 TensorBoard 中的输出示例。

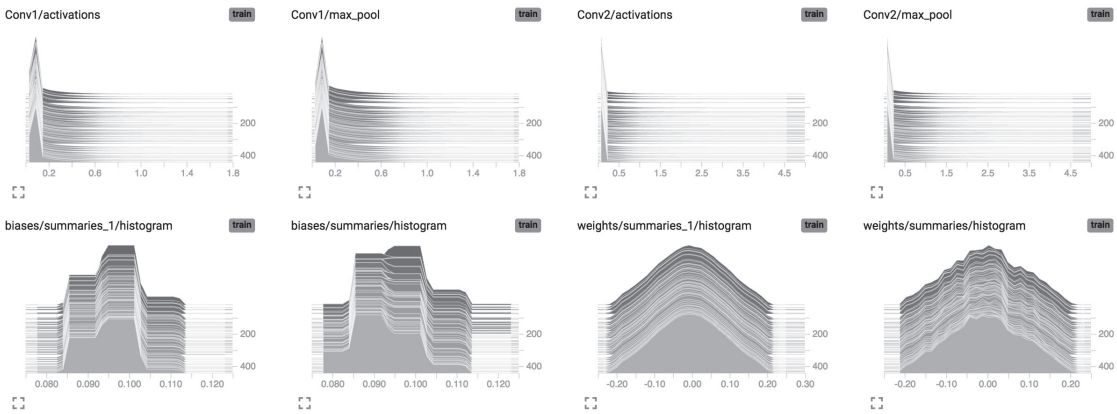


图 13.3　在 TensorBoard 中可视化权重和偏置

这些关于网络权重的统计数据可能非常有趣，有助于理解发生了什么。如果读者认为模型没有得到正确的训练，这些可视化可以让读者深入了解其中某网络层存在问题的潜在原因。

13.5 冻结层

有时（例如，在使用预训练网络时），希望冻结一些层。可以做到的是明确一些网络层（大多数情况下是靠前的几层，也称为网络的底层）作为特征提取具有一定的价值。在下面的内容中，将演示如何在训练之后冻结网络的一部分，并且只训练剩余的网络子集。

如何去做…

1）首先，加载所用函数库，如下：

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

2）在 TensorFlow 中，加载 MNIST 数据集非常简单：

```
mnist = input_data.read_data_sets('Data/mnist', one_hot=True)
```

3）接下来，定义占位符：

```
n_classes = 10
input_size = 784

x = tf.placeholder(tf.float32, shape=[None, input_size])
y = tf.placeholder(tf.float32, shape=[None, n_classes])
keep_prob = tf.placeholder(tf.float32)
```

4）在网络结构中定义了一些想要重复使用的函数：

```
def weight_variable(shape, name='undefined'):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial, name=name)

def bias_variable(shape, name='undefined'):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial, name=name)
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1], padding='SAME')
```

5）现在定义网络结构：

```
x_image = tf.reshape(x, [-1,28,28,1])

W_conv1 = weight_variable([7, 7, 1, 100], name='1st_layer_weights')
b_conv1 = bias_variable([100], name='1st_layer_bias')
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

W_conv2 = weight_variable([4, 4, 100, 150])
b_conv2 = bias_variable([150])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

W_conv3 = weight_variable([4, 4, 150, 250])
b_conv3 = bias_variable([250])
h_conv3 = tf.nn.relu(conv2d(h_pool2, W_conv3) + b_conv3)
h_pool3 = max_pool_2x2(h_conv3)

W_fc1 = weight_variable([4 * 4 * 250, 300])
b_fc1 = bias_variable([300])
h_pool3_flat = tf.reshape(h_pool3, [-1, 4*4*250])
h_fc1 = tf.nn.relu(tf.matmul(h_pool3_flat, W_fc1) + b_fc1)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = weight_variable([300, n_classes])
b_fc2 = bias_variable([n_classes])
y_pred = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

diff = tf.nn.softmax_cross_entropy_with_logits(labels=y,
logits=y_pred)
cross_entropy = tf.reduce_mean(diff)
correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y,
1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

6）显示输出可训练变量的名字：

```
trainable_vars = tf.trainable_variables()

for i in range(len(trainable_vars)):
    print(trainable_vars[i])
```

7）定义优化器时，可以设置在训练时应该包含哪些变量：

```
vars_train = [var for var in trainable_vars if '1st_' in var.name]

learning_rate = 0.001
train_step =
tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy,
var_list=vars_train)
```

也不使用变量的名称，而是在 TensorFlow 中显式设置可训练的参数。

8）接下来，设置想要使用的超参数并开始训练：

```
n_steps = 10
batch_size = 128
dropout = 0.25
evaluate_every = 10

sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
for i in range(n_steps):
    x_batch, y_batch = mnist.train.next_batch(batch_size)
    _, train_acc = sess.run([train_step, accuracy], feed_dict={x:
    x_batch, y: y_batch, keep_prob: dropout})
    print('Step {:04.0f}: train_acc: {:.4f}'.format(i, train_acc))
```

13.6 存储网络结构并训练权重

在大多数深度学习框架中，存储网络结构和训练权重是直接的。因为这可能非常重要，所以将演示如何使用 TensorFlow 在下面的内容中存储模型。

如何去做…

1）从导入函数库文件开始：

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

2）接下来，加载 MNIST 数据：

```
mnist = input_data.read_data_sets('Data/mnist', one_hot=True)
```

3）定义 TensorFlow 占位符，如下：

```
n_classes = 10
input_size = 784

x = tf.placeholder(tf.float32, shape=[None, input_size])
y = tf.placeholder(tf.float32, shape=[None, n_classes])
keep_prob = tf.placeholder(tf.float32)
```

4）为方便起见，创建函数来建立深度学习网络：

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1], padding='SAME')
```

5）现在可以定义完整的网络结构：

```
x_image = tf.reshape(x, [-1,28,28,1])

W_conv1 = weight_variable([7, 7, 1, 100])
b_conv1 = bias_variable([100])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

W_conv2 = weight_variable([4, 4, 100, 150])
b_conv2 = bias_variable([150])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

W_conv3 = weight_variable([4, 4, 150, 250])
b_conv3 = bias_variable([250])
h_conv3 = tf.nn.relu(conv2d(h_pool2, W_conv3) + b_conv3)
h_pool3 = max_pool_2x2(h_conv3)

W_fc1 = weight_variable([4 * 4 * 250, 300])
b_fc1 = bias_variable([300])
h_pool3_flat = tf.reshape(h_pool3, [-1, 4*4*250])
h_fc1 = tf.nn.relu(tf.matmul(h_pool3_flat, W_fc1) + b_fc1)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = weight_variable([300, n_classes])
b_fc2 = bias_variable([n_classes])
y_pred = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

6）需要以下定义来训练所建网络并确定性能：

```
diff = tf.nn.softmax_cross_entropy_with_logits(labels=y,
logits=y_pred)
cross_entropy = tf.reduce_mean(diff)
correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y,
1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

7）接下来定义优化器：

```
learning_rate = 0.001
train_step =
tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
```

8）现在可以定义超参数：

```
n_steps = 25
batch_size = 32
dropout = 0.25
evaluate_every = 10
n_val_steps = mnist.test.images.shape[0] // batch_size
```

9）为了在训练中保存模型，需要创建一个 TensorFlow 存入变量。因为不想存储太多的模型，所以将变量 max_to_keep 设置为 5：

```
saver = tf.train.Saver(max_to_keep=5)
save_dir = 'checkpoints/'
```

10）现在可以创建会话并开始训练。根据验证精度存储最佳模型的权重：

```
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
best_val = 0.0
for i in range(n_steps):
    x_batch, y_batch = mnist.train.next_batch(batch_size)
    _, train_acc = sess.run([train_step, accuracy], feed_dict={x:
    x_batch, y: y_batch, keep_prob: dropout})
    if i % evaluate_every == 0:
        val_accs = []
        for j in range(n_val_steps):
            x_batch, y_batch = mnist.test.next_batch(batch_size)
            val_acc = sess.run(accuracy, feed_dict={x: x_batch, y:
            y_batch, keep_prob: 1.0})
            val_accs.append(val_acc)
```

```
        print('Step {:04.0f}: train_acc: {:.4f}; val_acc:
        {:.4f}'.format(i, train_acc, sum(val_accs)/len(val_accs)))
        if val_acc > best_val:
            saver.save(sess, save_dir+'best-model', global_step=i)
            print('Model saved')
            best_val = val_acc
            saver.save(sess, save_dir+'last-model'))
```

11）现在，如果想使用已存储模型的权重，可以从一个检查点进行加载：

```
with tf.Session() as sess:
    new_saver = tf.train.import_meta_graph(save_dir+'last-
model.meta')
    new_saver.restore(sess, save_dir+'last-model')
    for i in range(35):
        x_batch, y_batch = mnist.train.next_batch(batch_size)
        _, train_acc = sess.run([train_step, accuracy],
        feed_dict={x: x_batch, y: y_batch, keep_prob: dropout})
```

第 14 章
预训练模型

本章提供了基于流行深度学习模型的一系列内容，如 VGG-16 和 Inception V4：

- 使用 GoogLeNet / Inception 进行大规模视觉识别；
- 用 ResNet 提取瓶颈特征；
- 对新类别使用预训练的 VGG 模型；
- 用 Xception 细调。

14.1 简介

在过去的几年里，许多改变游戏规则的网络结构已提出并公开发布，它们中的大多数是开源代码或公布权重。如果后者不是这种情况，其他人则执行网络结构并分享权重。因此，许多深度学习框架可以直接访问流行的模型及其权重。在本章中，将演示如何利用这些预训练的权重。这些模型中的大多数已经在比赛中使用的大型图像数据集（例如 ImageNet 数据集）上进行了训练。该数据集已发布，且用于 **ImageNet 大规模视觉识别挑战赛**（ILSVRC）。通过利用这些预训练的权重，可以获得良好的结果并减少训练时间。

14.2 使用 GoogLeNet/Inception 进行大规模视觉识别

在 2014 年，Google 公司发布了 Going Deeper with Convolutions（越来越深的卷积网络）（https：//arxiv.org/abs/1409.4842）论文，介绍了 GoogLeNet 架构。随后，更新的版本（2015 年 https：//arxiv.org/abs/1512.00567）以 Inception 的名义发布。在这些 GoogLeNet / Inception 模型中，在堆叠并输入到下一层之前，多个卷积层可并行应用。该网络结构的一大优点是计算成本较低，训练权重的文件要小得多。在本节内容中，将演示如何在 Keras 中加载 InceptionV3 权重并应用模型对图像进行分类。

如何去做…

1）Keras 有一些使用预训练模型的好工具。从导入函数库和工具开始，如下所示：

```
import numpy as np

from keras.applications.inception_v3 import InceptionV3
from keras.applications import imagenet_utils
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
```

请注意，可以使用 Keras 中包含的任何预训练模型替换 VGG16。

2）接下来，加载 InceptionV3 模型，如下：

```
pretrained_model = InceptionV3
model = pretrained_model(weights="imagenet")
```

3）InceptionV3 模型是在 ImageNet 数据集上进行训练的，因此如果想要提供新的图像给模型，应该使用相同的输入规模：

```
input_dim = (299, 299)
```

4）使用 Keras 中包含的默认 ImageNet 预处理技术：

```
preprocess = imagenet_utils.preprocess_input
```

5）下一步，加载一个示例图像并对其进行预处理：

```
image = load_img('Data/dog_example.jpg', target_size=input_dim)
image = img_to_array(image)
image = image.reshape((1, *image.shape))
image = preprocess(image)
```

6）经过这些简单的步骤，准备好预测示例图像的类别，如下所示：

```
preds = model.predict(image)
```

7）为了预测解码值，可以使用 Keras imagenet_utils.decode_predictions 预测方法：

```
preds_decoded = imagenet_utils.decode_predictions(preds)
```

8）最后，呈现这些值：

```
preds_decoded
```

9）输出应该如图 14.1 所示。

```
In [6]: preds_decoded

Out[6]: [[('n02109961', 'Eskimo_dog', 0.59557849),
          ('n02110185', 'Siberian_husky', 0.40070605),
          ('n02110063', 'malamute', 0.0032684309),
          ('n02091467', 'Norwegian_elkhound', 0.00015924724),
          ('n03218198', 'dogsled', 9.0920155e-05)]]
```

图 14.1　示例图像上的 VGG16 的预测和概率

用 16 行代码检索任意图像上的预测是超乎寻常的。但是，只有当目标类包含在原始 ImageNet 数据集（1000 个类）中时，才能使用此模型。

14.3　用 ResNet 提取瓶颈特征

ResNet 架构于 2015 年在 Deep Residual Learning for Image Recognition（图像识别的深度残差学习，http://arxiv.org/abs/1512.03385）论文中进行了介绍。ResNet 与 VGG 有不同的网络结构，它由堆叠在一起的微架构组成。ResNet 在 2015 年赢得了 ILSVRC，并在 ImageNet 数据集上超越了人类的表现。在下面的内容中，将演示如何利用 ResNet50 权重来提取瓶颈特征。

如何去做…

1）首先导入所用的 Keras 工具：

```
from keras.models import Model
from keras.applications.resnet50 import ResNet50

from keras.applications.resnet50 import preprocess_input
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications import imagenet_utils
```

2）接下来，加载 ImageNet 权重的 ResNet50 模型：

```
resnet_model = ResNet50(weights='imagenet')
```

3）对于这个例子，将在 ResNet 实现中提取最终的平均池化层。这个层叫做 avg_pool。如果想从另一层提取特征，可以在 https：//github.com/fchollet/keras/blob/make/keras/applications/resc50.py 查找 ResNet50 的 Keras 实现：

```
model = Model(inputs=resnet_model.input,
outputs=resnet_model.get_layer('avg_pool').output)
```

4）需要确保想要测试的例子的输入规模是 224 × 224，并且将默认的预处理技术应用于 ImageNet：

```
input_dim = (224, 224)
image = load_img('Data/dog_example.jpg', target_size=input_dim)
image = img_to_array(image)
image = image.reshape((1, *image.shape))
image = preprocess_input(image)
```

5）提取示例图像的特征：

```
avg_pool_features = model.predict(image)
```

6）使用此模型提取的特征可用于堆叠。例如，可以在这些特征（和其他特征）之上训练 Xgboost 模型以进行最终分类。

14.4 对新类别使用预训练的 VGG 模型

2014 年公开发表了论文 Very Deep Convolutional Networks for Large-Scale Image Recognition（用于大规模图像识别的超深度卷积网络，https：//arxiv.org/abs/1409.1556）。那时，VGG16 和 VGG19 这两个模型都被认为是超深度结构，分别有 16 层和 19 层。除了输入和输出层以外，还包括权重以及一些最大池化层。VGG 的网络结构将多个 3 × 3 的卷积层叠加在一起。VGG16 网络结构总共有 13 个卷积层和 3 个全连接网络层。VGG19 变体有 16 个卷积层和相同的 3 个全连接网络层。在下面的内容中，将使用 VGG16 的瓶颈特征，并在其上添加自己的网络层。将冻结其原始模型权重，只训练顶层。

如何去做…

1）首先导入所用函数库，如下所示：

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

from keras.models import Model
from keras.applications.vgg16 import VGG16
from keras.layers import Dense, GlobalAveragePooling2D
from keras.optimizers import Adam

from keras.applications import imagenet_utils
from keras.utils import np_utils
from keras.callbacks import EarlyStopping
```

2）在这个方案中，将使用 Keras 的 CIFAR10 数据集：

```
from keras.datasets import cifar10
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

3）在将数据导入模型之前，需要对数据进行预处理：

```
n_classes = len(np.unique(y_train))
y_train = np_utils.to_categorical(y_train, n_classes)
y_test = np_utils.to_categorical(y_test, n_classes)

X_train = X_train.astype('float32')/255.
X_test = X_test.astype('float32')/255.
```

4）将使用 VGG16 模型并加载没有顶层神经元密集层的权重：

```
vgg_model = VGG16(weights='imagenet', include_top=False)
```

5）关于 Keras 的好处是，当第一次使用权重时，权重将自动下载，如图 14.2 所示。

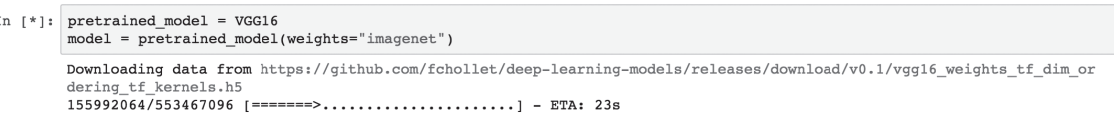


图 14.2　Keras 自动下载权重

6）在这个 VGG16 模型之上，将添加一个神经元密集层和一个输出层：

```
x = vgg_model.output
 x = GlobalAveragePooling2D()(x)
 x = Dense(512, activation='relu')(x)
 out = Dense(10, activation='softmax')(x)
```

7）将这两个模型组合，如下所示：

```
model = Model(inputs=vgg_model.input, outputs=out)
```

8）可以用下面的代码冻结 VGG16 模型的权重：

```
for layer in vgg_model.layers:
 layer.trainable = False
```

9）接下来，用 Adam 优化器编译模型并输出模型的摘要：

```
opt = Adam()
model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
model.summary()
```

10）从摘要中可以看出，所建模型总共有超过 1500 万个参数。然而通过冻结 VGG16 层，最终得到 535562 个可训练参数。

11）定义一个早停方法的回调函数，以防止过拟合发生：

```
callbacks = [EarlyStopping(monitor='val_acc', patience=5,
verbose=0)]
```

12）现在准备训练所建的模型：

```
n_epochs = 50
batch_size = 512
history = model.fit(X_train, y_train, epochs=n_epochs,
batch_size=batch_size, validation_split=0.2, verbose=1,
callbacks=callbacks)
```

13）绘制训练精度和验证精度曲线：

```
plt.plot(np.arange(len(history.history['acc'])),
history.history['acc'], label='training')
plt.plot(np.arange(len(history.history['val_acc'])),
history.history['val_acc'], label='validation')
 plt.title('Accuracy')
 plt.xlabel('batches')
 plt.ylabel('accuracy ')
 plt.legend(loc=0)
 plt.show()
```

14）从图 14.3 可以看出，模型的最终验证精度在 60%左右。

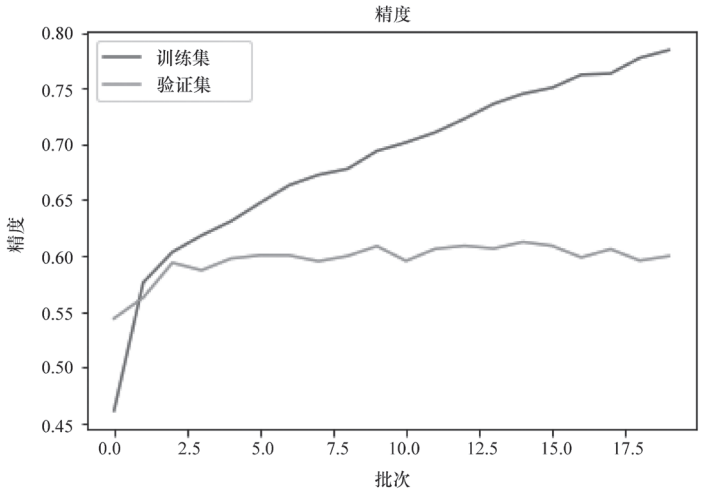


图 14.3　使用 VGG16 网络权重后的训练结果

14.5 用 Xception 细调

Xception 网络由 Keras 的创建者 François Chollet 在其论文“Xception：具有深度可分离卷积的深度学习”中提出并加以实现。原文参见“Xception: Deep Learning with Depthwise Separable Convolutions”（https://arxiv.org/abs/1610.02357）中设计制作。Xception 是 Inception 架构的扩展，其中 Inception 模块被深度可分离的卷积代替。在之前的内容中，只关注顶层的训练，同时保持原始权重在训练期间冻结。但也可以选择以较小的学习率来训练所有权重，这就是所谓的**微调**。这种技术可以通过去除原始网络中一些偏大权重来使模型的性能有小幅的提升。

如何去做…

1）首先，从导入所有需要的函数库开始，如下所示：

```
import numpy as np
from keras.models import Model
from keras.applications import Xception
from keras.layers import Dense, GlobalAveragePooling2D
from keras.optimizers import Adam

from keras.applications import imagenet_utils
from keras.utils import np_utils
from keras.callbacks import EarlyStopping
```

2）接下来，加载数据集：

```
from keras.datasets import cifar10
 (X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

3）在将数据提供给模型之前，需要对数据进行预处理：

```
n_classes = len(np.unique(y_train))
y_train = np_utils.to_categorical(y_train, n_classes)
y_test = np_utils.to_categorical(y_test, n_classes)

X_train = X_train.astype('float32')/255.
X_test = X_test.astype('float32')/255.
```

4）加载 Xception 模型而不使用 Keras 全连接顶层：

```
xception_model = Xception(weights='imagenet', include_top=False)
```

5）接下来，定义添加到 Xception 模型的网络层：

```
x = xception_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(512, activation='relu')(x)
out = Dense(10, activation='softmax')(x)
```

6）将其组合成一个模型：

```
model = Model(inputs=xception_model.input, outputs=out)
```

7）需要确保 Xception 模型的原始权重处于冻结状态。在微调之前首先训练顶层：

```
for layer in xception_model.layers:
 layer.trainable = False
```

8）现在可以定义优化器并编译模型：

```
opt = Adam()
model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
model.summary()
```

9）为了防止过拟合，使用早停方法：

```
callbacks = [EarlyStopping(monitor='val_acc', patience=5,
verbose=0)]
```

10）开始训练顶层：

```
n_epochs = 100
batch_size = 512
history = model.fit(X_train, y_train, epochs=n_epochs,
batch_size=batch_size, validation_split=0.2, verbose=1,
callbacks=callbacks)
```

11）之后，可以开始微调。首先，需要知道想要使用哪个网络层进行微调，所以按如下步骤罗列网络层名称：

```
for i, layer in enumerate(model.layers):
 print(i, layer.name)
```

12）调整最后两个 Xception 块，从第 115 层开始：

```
for layer in model.layers[:115]:
 layer.trainable = False
for layer in model.layers[115:]:
 layer.trainable = True
```

13）之后，训练所建模型来训练其他网络层。学习率在这里是非常重要的，当学习率太大时，完全忽略了预训练的权重。如果原来的权重不适合当前的任务，就很有必要对权重进行预训练：

```
opt_finetune = Adam()
model.compile(optimizer=opt_finetune,
loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

14）最后，可以开始微调所建模型：

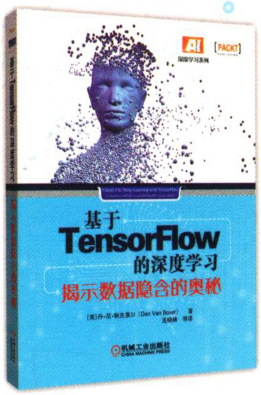
```
history_finetune = model.fit(X_train, y_train, epochs=n_epochs,
batch_size=batch_size, validation_split=0.2, verbose=1,
callbacks=callbacks)
```

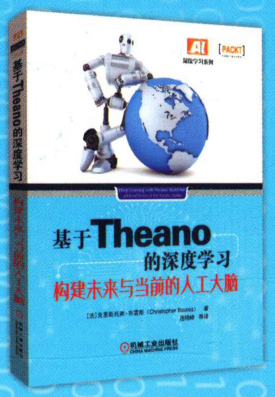
在本章中，演示了如何利用预训练的深度学习模型。这些预训练模型的权重可以用于类似的图像分类任务或少数相关的任务。

系列图书如下

基于H2O的机器学习实用方法：

一种强大的可扩展的人工智能和深度学习技术





关于本书

本书针对所提出的问题提供技术解决方案，并提供对这些解决方案的详细解释。此外，还讨论了使用TensorFlow、PyTorch、Keras和CNTK等流行开源框架针对实际问题解决方案相应的优缺点。本书也介绍了人工神经网络基本概念及其相关技术，包括经典的网络拓扑等。本书主要目的是为Python程序员提供较为详细的实战方案，以便将深度学习应用于常见和不常见实际问题场景。

本书包括14章：1 编程环境、GPU 计算、云解决方案和深度学习框架；2 前馈神经网络；3 卷积神经网络；4 递归神经网络；5 强化学习；6 生成对抗网络；7 计算机视觉；8 自然语言处理；9 语音识别和视频分析；10 时间序列和结构化数据；11 游戏智能体和机器人；12 超参数选择、调优和神经网络学习；13 网络内部构造；14 预训练模型。

本书特色

- 提供训练不同神经网络模型并调整模型以期获得最佳性能的实战方案；
- 使用诸如TensorFlow、Caffe、Keras、Theano的Python框架进行自然语言处理、计算机视觉识别等；
- Python深度学习中的常见以及不常见问题的解决指南。

阅读本书将会学到的内容

- 在Python中实现不同的人工神经网络模型；
- 选择诸如PyTorch、TensorFlow、MXNet和Keras等最优的Python开源框架来进行深度学习；
- 应用神经网络内部细节相关的提示和技巧，以提高学习成效；
- 巩固机器学习原理并将其应用于深度学习领域；
- 重用Python代码段并将其应用于解决日常问题；
- 评估每个解决方案的成本/收益和性能影响。

Packt>

地址：北京市百万庄大街22号
邮政编码：100037
电话服务
服务咨询热线：010-88361066
读者购书热线：010-68326294
010-88379203
网络服务
机工官网：www.cmpbook.com
机工官博：weibo.com/cmp1952
金书网：www.golden-book.com
教育服务网：www.cmpedu.com
封面无防伪标均为盗版

机械工业出版社微信公众号　传播电类内容提升专业知识　关注电类行业动向　聚焦前沿科技

上架指导 人工智能/深度学习
ISBN 978-7-111-59872-5
策划编辑◎顾 谦 / 封面设计◎ 子时文化 ZiShi Culture

ISBN 978-7-111-59872-5
9 787111 598725 >
定价：79.00元