O'REILLY®

TURING 图灵程序设计丛书

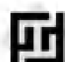
金融人工智能

用Python实现AI量化交易

Artificial Intelligence in Finance

[德] 伊夫·希尔皮斯科 著

石磊磊 余宇新 李煜鑫 译



中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

AllTick

实时行情数据接口

专为量化交易打造

全方位的市场行情数据接口

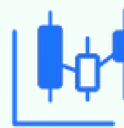
包含实时和历史行情

外汇API

来自世界领先银行机构的 100 多种货币对的逐笔更新。

商品API

所有贵金属（如黄金、白银）和所有能源类别的实时和历史商品数据 API。



股票API

适用于 170,000+ 美国和香港股票的实时和历史股票数据 API。

加密货币API

来自所有主要加密货币交易所的实时和历史加密货币数据，统一在一个易于使用的 API 中。

量化交易神器
回测必备！！

可靠的行情源

我们的系统具有 99.95% 的 SLA。AllTick 倾向于将质量和正常运行时间的标准提升到下一个级别。

低延时接口实时推送

通过 WebSocket 进行的实时数据流具有超低延迟，平均仅约 170 毫秒。

逐笔更新的高频数据

每个交易的实时推送，每个都可追踪，并且与交易所的实时交易行情完全同步。

马上登录ALLTICK.CO

免费试用!

全美股财报免费送!

译者简介

石磊磊

在人工智能领域深耕20余年，曾任职于蚂蚁金服、微软等国内外知名公司，主导了多个金融模型的开发，服务于全球数亿设备，带领团队研发了工业级实时动态图风控系统，在金融风险管理和量化交易方面有丰富的研究经验和应用经验。

余宇新

上海外国语大学副教授，金融大数据中心执行主任，上海市创新政策评估研究中心研究员，发表论文40余篇，开发的大数据算法获国家发明专利授权1项，曾参与多项人工智能产品研发工作。

李煜鑫

上海外国语大学副教授，硕士生导师；英国华威大学特聘研究员，博士生导师；英国约克大学经济学院博士；在国内外从事金融学相关研究近20年，在国内外核心期刊上发表论文30余篇，撰写了多部专著。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

专为量化交易设计的实时行情数据API: www.alltick.co

图灵程序设计丛书

金融人工智能：
用Python实现AI量化交易

Artificial Intelligence in Finance

[德] 伊夫·希尔皮斯科 著
石磊磊 余宇新 李煜鑫 译

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

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

图书在版编目（CIP）数据

金融人工智能 : 用Python实现AI量化交易 / (德)
伊夫·希尔皮斯科 (Yves Hilpisch) 著 ; 石磊磊, 余宇
新, 李煜鑫译. -- 北京 : 人民邮电出版社, 2022.8
（图灵程序设计丛书）
ISBN 978-7-115-59455-6

Ⅰ. ①金… Ⅱ. ①伊… ②石… ③余… ④李… Ⅲ.
①软件工具－程序设计－应用－金融工作 Ⅳ.
①F830-39

中国版本图书馆CIP数据核字(2022)第100447号

内 容 提 要

本书通过 Python 示例介绍人工智能技术在金融数据分析中的应用。你将了解如何运用神经网络、强化学习等深度学习技术预测金融市场。本书分为六大部分。第一部分介绍人工智能算法的核心概念，包括监督学习和神经网络，并描绘超级人工智能愿景。第二部分讨论机器学习技术在金融市场中的应用。第三部分更进一步，讨论如何利用神经网络和强化学习技术解决金融市场中的统计失效问题。第四部分详述如何利用算法交易解决统计失效问题。第五部分展望未来，探讨人工智能会如何改变金融业。第六部分给出以 Python 实现的神经网络，可用于时间序列预测。

本书面向金融方向的数据分析师、学生和研究人员，以及使用 Python 开发量化交易策略的“宽客”。

◆ 著　[德] 伊夫·希尔皮斯科
译　石磊磊　余宇新　李煜鑫
责任编辑　张海艳
责任印制　彭志环

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　https://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：24.5　2022年 8 月第 1 版
字数：566千字　2022年 8 月北京第 1 次印刷
著作权合同登记号　图字：01-2021-2068号

定价：129.80元

读者服务热线：(010)84084456-6009　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东市监广登字 20170147 号

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

版权声明

Copyright © 2021 Yves Hilpisch. All rights reserved.

Simplified Chinese edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2022. Authorized translation of the English edition, 2022 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

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

英文原版由 O'Reilly Media, Inc. 出版，2021。

简体中文版由人民邮电出版社有限公司出版，2022。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

O’Reilly Media, Inc.介绍

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

O’Reilly业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、互动学习、认证体验、图书、视频等，使客户更容易获取业务成功所需的专业知识。几十年来O’Reilly图书一直被视为学习开创未来之技术的权威资料。我们所做的一切是为了帮助各领域的专业人士学习最佳实践，发现并塑造科技行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O’Reilly Radar博客有口皆碑。”

——*Wired*

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

——*Business 2.0*

“O’Reilly Conference是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

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

——*Linux Journal*

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

本书赞誉

"凭借其全面和直观的方法，这将是金融领域从业人员和学者的主要参考书。"

——Abdullah Karasan，金融数据科学学者

"这是一本优秀的机器学习实践指南，旨在解决量化金融领域的一系列问题。"

——Tim Nugent，路孚特公司研究员

"人工智能在方方面面改变了我们的生活，在金融行业（尤其是量化交易领域）有广阔的应用前景。但是，行业缺乏既懂人工智能又懂投资交易的复合型人才。对于有志于从事该领域的读者，这本书不仅能帮助熟悉人工智能技术在量化投资交易中的应用之道，还能在书中启发式的论述下深入理解人工智能应用的局限与潜能，从而进一步提升实践能力。相信这本书的翻译出版能让更多的国内从业人员从中获益良多，助力中国的量化投资发展。"

——漆远，复旦大学浩清特聘教授、
复旦大学人工智能创新与产业研究院院长、前蚂蚁集团首席 AI 科学家

"希尔皮斯科博士将正统金融理论和人工智能有机融合，通过具体的代码带领读者一步步走进金融人工智能的世界。从特征工程到神经网络、强化学习，再到算法交易、风险管理，这本书涵盖了 AI 量化的投研与交易，内容丰富且极具实操性，同时又回归量化的奇点，富含学术优雅，给人以启发，是 AI 量化最佳实践。"

——梁举，BigQuant 人工智能量化平台创始人兼 CEO

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

“现在的投资行业可以说正处于一个加速变革的过程中，AI 和机器学习正是推进这一变革的重要驱动力。从 CFA Institute 进行的全球调研中可以看到，亚太市场近半的从业人员认为 AI 的发展将在未来 5 年内对他们的职位产生重大影响。如何应对变革？我们需要的是 AI 与 HI（人类智慧）的结合。这本实用指南全面展现出 AI 和机器学习如何发现金融市场中广泛存在的不完全有效并实现量化交易，是不可多得的工具书。”

——张一，CFA Institute 中国区总经理

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

目录

前言 xiii

第一部分　机器智能

第 1 章　人工智能 3
1.1　算法 3
1.1.1　数据类型 3
1.1.2　学习类型 4
1.1.3　任务类型 7
1.1.4　方法类型 7
1.2　神经网络 8
1.2.1　OLS 回归 8
1.2.2　神经网络估计 12
1.2.3　神经网络分类 17
1.3　数据的重要性 19
1.3.1　小数据集 19
1.3.2　更大的数据集 22
1.3.3　大数据 24
1.4　结论 25

第 2 章　超级智能 26
2.1　成功故事 27
2.1.1　雅达利（Atari） 27
2.1.2　围棋（Go） 32

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

2.1.3 国际象棋（Chess）……33
2.2 硬件的重要性……35
2.3 智能的形式……36
2.4 通往超级智能的途径……37
2.4.1 网络和组织……38
2.4.2 生物增强……38
2.4.3 脑机混合……38
2.4.4 全脑模拟……39
2.4.5 人工智能……39
2.5 智能爆炸……40
2.6 目标和控制……41
2.6.1 超级智能和目标……41
2.6.2 超级智能和控制……42
2.7 潜在的结果……43
2.8 结论……45

第二部分 金融和机器学习

第 3 章 规范性金融理论……49
3.1 不确定性与风险……50
3.1.1 定义……50
3.1.2 数字模拟例子……51
3.2 预期效用理论……53
3.2.1 假设和结论……53
3.2.2 数值例子……55
3.3 均值-方差投资组合理论……57
3.3.1 假设和结论……57
3.3.2 数值例子……59
3.4 资本资产定价模型……67
3.4.1 假设和结论……67
3.4.2 数值例子……69
3.5 套利定价理论……74
3.5.1 假设和结论……74
3.5.2 数值例子……75
3.6 结论……77

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 4 章　数据驱动的金融学 78
4.1　科学方法 78
4.2　金融计量经济学与回归 79
4.3　数据可用性 82
4.3.1　可编程 API 82
4.3.2　结构化历史数据 83
4.3.3　结构化流数据 85
4.3.4　非结构化历史数据 86
4.3.5　非结构化流数据 88
4.3.6　非传统数据 89
4.4　重新审视规范性理论 93
4.4.1　预期效用与现实 93
4.4.2　均值 – 方差投资组合理论 96
4.4.3　资本资产定价模型 103
4.4.4　套利定价理论 107
4.5　揭示中心假设 115
4.5.1　正态分布收益率 115
4.5.2　线性关系 124
4.6　结论 126
4.7　Python 代码段 126
第 5 章　机器学习 130
5.1　学习 131
5.2　数据 131
5.3　成功 133
5.4　容量 137
5.5　评估 140
5.6　偏差和方差 145
5.7　交叉验证 147
5.8　结论 149
第 6 章　人工智能优先的金融 150
6.1　有效市场 150
6.2　基于收益数据的市场预测 155
6.3　基于更多特征的市场预测 161
6.4　日内市场预测 166
6.5　结论 167

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第三部分　统计失效

第 7 章　密集神经网络 171
7.1　数据 171
7.2　基线预测 173
7.3　归一化 177
7.4　暂退 179
7.5　正则化 181
7.6　装袋 184
7.7　优化器 185
7.8　结论 186
第 8 章　循环神经网络 187
8.1　第一个示例 188
8.2　第二个示例 191
8.3　金融价格序列 194
8.4　金融收益率序列 197
8.5　金融特征 199
8.5.1　估计 199
8.5.2　分类 200
8.5.3　深度 RNN 201
8.6　结论 202
第 9 章　强化学习 203
9.1　基本概念 204
9.2　OpenAI Gym 205
9.3　蒙特卡罗智能体 208
9.4　神经网络智能体 210
9.5　DQL 智能体 212
9.6　简单的金融沙箱 216
9.7　更好的金融沙箱 220
9.8　FQL 智能体 222
9.9　结论 227

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第四部分　算法交易

第 10 章　向量化回测 ······ 231

10.1　基于 SMA 策略的回测 ······ 232

10.2　基于 DNN 的每日策略的回测 ······ 237

10.3　基于 DNN 的日内策略的回测 ······ 243

10.4　结论 ······ 248

第 11 章　风险管理 ······ 249

11.1　交易机器人 ······ 250

11.2　向量化回测 ······ 253

11.3　基于事件的回测 ······ 255

11.4　风险评估 ······ 261

11.5　风控措施回测 ······ 264

11.5.1　止损 ······ 266

11.5.2　跟踪止损 ······ 268

11.5.3　止盈 ······ 269

11.6　结论 ······ 272

11.7　Python 代码 ······ 273

11.7.1　金融环境 ······ 273

11.7.2　交易机器人 ······ 275

11.7.3　回测基类 ······ 279

11.7.4　回测类 ······ 281

第 12 章　执行与部署 ······ 284

12.1　Oanda 账户 ······ 285

12.2　数据检索 ······ 285

12.3　订单执行 ······ 289

12.4　交易机器人 ······ 294

12.5　部署 ······ 300

12.6　结论 ······ 304

12.7　Python 代码 ······ 304

12.7.1　Oanda 环境 ······ 304

12.7.2　向量化回测 ······ 307

12.7.3　Oanda 交易机器人 ······ 308

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第五部分　展望

第 13 章　基于人工智能的竞争 313
13.1　人工智能和金融 313
13.2　标准的缺失 315
13.3　教育和培训 316
13.4　资源争夺 317
13.5　市场影响 318
13.6　竞争场景 319
13.7　风险、监管和监督 320
13.8　结论 322
第 14 章　金融奇点 323
14.1　概念和定义 323
14.2　风险是什么 324
14.3　通往金融奇点的途径 327
14.4　正交技能和资源 328
14.5　之前和之后的情景 328
14.6　星际迷航还是星球大战 329
14.7　结论 329

第六部分　附录

附录 A　交互式神经网络 333
附录 B　神经网络类 348
附录 C　卷积神经网络 360
参考文献 366

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

前言

> 对于我们所能想到的每一种投资策略，alpha 收益是否终将消失？更本质的问题是，在无数聪明头脑和智慧机器的帮助下，我们是否将迎来这样的一天：金融市场达到完全有效，从而使我们可以放心地认为所有的资产都被正确定价？
>
> ——Robert Shiller，2015 年

人工智能（artificial intelligence，AI）已成为 21 世纪最初十年的一项关键技术，并将继续主导之后的技术发展。随着技术创新、算法突破、大数据崛起以及与日俱增的算力，很多行业正在经历人工智能所带来的剧变。

在媒体大众聚焦于人工智能给自动驾驶汽车和游戏等领域带来突破的同时，在金融行业，人工智能也正异军突起。然而，相较于在网页搜索和社交媒体等领域的应用，人工智能在金融领域的应用可以说还只是锋芒初露。

本书覆盖了金融人工智能的若干重要方面。金融人工智能是一个宽泛的话题，仅一本书很难面面俱到，因此需要有所侧重。本书将在第一部分和第二部分中首先介绍一些基本信息。在第三部分中，我们将从统计学角度探讨金融市场失效的现象，并在此过程中借用人工智能中的神经网络技术。只有存在市场失效，并且可以通过人工智能算法对市场动向进行预测，才有可能进一步采用算法交易从经济失效中获益，本书将在第四部分中讨论相关话题。如果我们可以系统地从市场失效和经济失效中获益，那么这将与金融领域众所周知的基础理论“有效市场假说”（efficient market hypothesis，EMH）相矛盾。成功设计出智能交易机器人一直以来被认为是金融领域的“圣杯”，也许人工智能将引领我们实现这一终极目标。在这种情况下，人工智能将如何影响金融市场？金融奇点来临的可能性有多大？本书的第五部分将针对上述问题展开讨论。作为技术附录的第六部分将结合实际用例为读者展示如何基于 Python 代码从零开始搭建神经网络。

在金融领域应用人工智能所遇到的问题，与在其他领域中所遇到的问题并无大异。在 21 世纪第一个十年中，人工智能领域的一些主要突破可能来自强化学习（reinforcement learning，RL）在游戏中的应用，比如训练人工智能下国际象棋和围棋等棋牌类游戏（Silver 等，2016），以及玩雅达利（Atari）游戏——一种在 20 世纪 80 年代出品的电子游

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

戏（Mnih 等，2013）。在游戏领域应用强化学习的过程让科学家受益匪浅，同样的能力今天被用来解决更多富有挑战的问题，譬如设计和制造自动驾驶汽车，以及改进医疗诊断手段。表 P-1 是人工智能在不同领域的应用情况。

表P-1：人工智能在不同领域的应用情况

领域	智能体	目标	方法	奖励	障碍	风险
电子游戏	人工智能体（软件）	最大化游戏分数	在虚拟游戏环境中进行强化学习	得分	规划和延迟奖励	无
自动驾驶	自动驾驶汽车（软件 + 车）	从地点 A 安全地驾驶至地点 B	在虚拟（游戏）环境中进行强化学习，并在现实世界中试驾	对错误进行处罚	从虚拟世界到物理世界的转换	毁坏建筑，伤害人类
金融交易	智能交易机器人（软件）	最优化长期表现	在虚拟交易环境中进行强化学习	经济回报	有效市场和竞争	经济损失

训练人工智能体玩电子游戏的天然优势在于，游戏世界是一个完美的虚拟环境[1]，不存在任何真实的风险。相反，训练自动驾驶汽车则不然，将人工智能体从虚拟学习环境迁移到真实驾驶路况的过程中面临着巨大的挑战。我们很难想象将《侠盗飞车》游戏世界中的车辆开到现实世界中来，这将带来车祸甚至伤亡等后果十分严重的风险。

对智能交易机器人来说，情况则要好很多，因为可以通过仿真的金融市场，实现完全虚拟的强化学习。操作错乱的智能交易机器人所带来的主要风险在于经济损失，或者从更大的尺度上看，在于智能交易机器人的群体性行为所带来的系统性风险。然而总体来看，对人工智能算法的训练、测试和部署来说，金融领域是一个理想的场景。

随着金融领域的快速发展，甚至一个好奇并且努力上进的学生都可以在笔记本计算机和互联网上成功地将人工智能应用于金融交易场景。这要归功于近年来计算机软件和硬件的发展，还有在线交易商的崛起。在线交易商提供历史金融数据和实时金融数据，并允许通过程序化交易接口（API）执行金融交易。

本书包括如下 6 个部分。

第一部分

第一部分会讨论人工智能的主要概念和算法，比如监督学习和神经网络（第 1 章）。接下来还会讨论超级人工智能的概念，这意味着人工智能将拥有与人类相当的智能水平，甚至在某些领域会超越人类的智能水平（第 2 章）。并不是每一个研究人员都相信在可预见的未来能够实现超级人工智能。然而，关于超级人工智能的讨论为我们提供了一个有益的框架，可以帮助我们讨论人工智能，尤其是人工智能在金融领域的应用。

第二部分

第二部分由 4 章组成，会讨论传统的规范性金融理论（第 3 章），以及该领域如何被数据驱动的金融学（第 4 章）和机器学习（第 5 章）所转变。数据驱动的金融学和机器学习促进了无模型、人工智能优先的方法在金融领域的崛起，第 6 章将对此展开讨论。

注 1：参见 Arcade Learning Environment。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第三部分

第三部分通过运用深度学习、神经网络和强化学习等方法，发掘统计学意义上的金融市场失效现象。这一部分会依次介绍密集神经网络（DNN，第 7 章）[2]、循环神经网络（RNN，第 8 章）和强化学习的相关算法（第 9 章），这些算法依赖密集神经网络模型来表达和逼近人工智能体的最优策略。

第四部分

第四部分会讨论如何通过算法交易来利用统计学意义上的市场失效。这一部分的话题包括向量化回测（第 10 章）、基于事件的回测和风险管理（第 11 章），以及人工智能算法交易策略的执行与部署（第 12 章）。

第五部分

第五部分是关于金融行业中基于人工智能的竞争所带来的潜在后果（第 13 章）。这一部分还会讨论金融奇点的可能性，届时人工智能体将在我们所知的金融领域的每一方面占据主导地位。在该场景的讨论中，我们将侧重于那些产出利润持续高于人类和机构业绩基线的人工智能金融交易机器人（第 14 章）。

第六部分

附录部分包含交互式神经网络训练的 Python 代码（附录 A），通过 Python 代码从零开始搭建简单的浅层神经网络类库（附录 B），以及一个运用卷积神经网络（CNN）进行金融时间序列预测的实例（附录 C）。

作者题记

人工智能在金融交易中的应用仍然是一个新生领域。虽然在撰写本书时市面上已经有一些关于这个方向的书，但是它们大多没有展示对于经济有效地利用统计学意义上的市场失效现象，人工智能意味着什么。

一些对冲基金宣称已经完全通过机器学习来管理投资者的资产。突出的案例是一家名为 Voleon Group 的对冲基金，截至 2019 年年底，该基金管理着超过 60 亿美元的资产（Lee 和 Karsh，2020）。依赖于机器学习，该基金在 2019 年的业绩为 7%，而同年，标准普尔 500 股票指数上涨了约 30%。

本书基于多年来开发、回测和部署人工智能算法交易策略的实战经验。由于行业自身具有前沿性和保密性，本书所展示的方法和示例主要基于我本人的研究。论述和表达侧重实操，难免有失严谨，很多实例缺少正规的理论证明或者实验验证。金融领域或机器学习领域的专家可能会对本书中的一些应用和案例持有完全不同的意见。譬如说，François Chollet（2017）等机器学习和深度学习领域的专家，对于预测金融市场这件事本身是否可行，持强烈的怀疑态度。某些金融领域的专家，比如 Robert Shiller（2015），怀疑金融奇点这一天是否真的会到来。一些活跃于两个领域交叉方向的专家，比如 Marcos López de Prado（2018）则认为，在金融交易和投资中运用机器学习技术需要行业规模的投入，这意味着浩大的团队及巨量的预算。

注 2：本书中的 DNN 如无另外说明，均为 dense neural network 的缩写，即**密集神经网络**，在其他文献中也称为 MLP（multi-layer perceptron，多层感知机）。——译者注

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

本书并非就每个主题所涉及的各种观点给出一个中立、全面的看法。书中所展示的内容主要基于我的个人经验、演示示例和 Python 代码的实践考量。很多示例是为了展示特定观点和结果而有意选择并做出调整的。因此，难免会被认为存在数据探查和过拟合的现象 [关于这些主题的讨论，详见 Hilpisch（2020）的第 4 章]。

本书的主要目标是使读者能够使用书中的代码示例作为框架，探索运用人工智能进行金融交易这一令人兴奋的领域。为实现此目标，本书始终以许多简化的假设为基础，并且主要采用**金融时间序列数据以及基于此类数据的衍生特征**。在实际应用中，当然不必局限于金融时间序列数据，也可以使用多种其他数据类型和数据源。本书的特征衍生方法隐式地假设了金融时间序列及衍生特征显示出的模式至少在一定程度上会随时间持续存在，并且可以用来预测未来的价格走势。

在此背景下，本书提供的所有示例和代码本质上都是技术性和说明性的，并不代表任何推荐或投资建议。

对于那些想部署本书中介绍的方法和算法交易策略的读者，我的另一本书 *Python for Algorithmic Trading: From Idea to Cloud Deployment* 提供了更多面向过程的技术细节，它与本书在许多方面是相辅相成的。对于刚开始在金融领域使用 Python 的读者，以及需要温故知新或者参考资料的读者，我的《Python 金融大数据分析》这本书完整地涵盖了在金融领域应用 Python 所涉及的重要主题和基本技能。

排版约定

本书使用如下排版约定。

黑体字

表示新术语或重点强调的内容。

等宽字体（`constant width`）

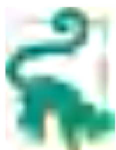
表示程序片段，以及正文中出现的变量名、函数名、数据库、数据类型、环境变量、语句和关键字等。

等宽粗体（**`constant width bold`**）

表示应该由用户输入的命令或其他文本。

等宽斜体（*`constant width italic`*）

表示应该由用户输入的值或根据上下文确定的值替换的文本。

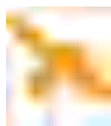


该图标表示提示或建议。

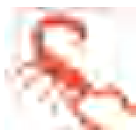
专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

该图标表示一般注记。



该图标表示重要信息。



该图标表示警告或警示。

使用示例代码

可以通过图灵社区下载本书示例代码：ituring.cn/book/2858。

本书旨在帮助你完成工作。一般来说，你可以在自己的程序或文档中使用本书提供的示例代码。除非需要复制大量代码，否则无须联系我们获得许可。比如，使用本书中的几个代码片段编写程序无须获得许可，销售或分发 O’Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将本书中的大量示例代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明通常包括书名、作者、出版社和 ISBN，比如"*Artificial Intelligence in Finance* by Yves Hilpisch (O’Reilly). Copyright 2021 Yves Hilpisch, 978-1-492-05543-3"。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

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

O'REILLY®

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

我们拥有独特的由专家和创新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O’Reilly 的在线学习平台让你能够按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O’Reilly 和 200 多家其他出版商提供的大量文本资源和视频资源。有关的更多信息，请访问 https://www.oreilly.com。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

联系我们

如有与本书有关的评价或问题，请联系出版社。

美国：

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

中国：

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

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表[3]、示例代码以及其他信息。本书的网页是 https://oreil.ly/ai-in-finance。

你还可以发送电子邮件至 errata@oreilly.com.cn 评论或询问与本书有关的技术问题。

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：https://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

致谢

我要感谢本书的技术审稿人 Margaret Maynard-Reid、Tim Nugent 博士和 Abdullah Karasan 博士，他们帮助我改进了本书的内容。

“Python 计算金融和算法交易”证书项目的代表也为本书的改进提供了帮助。他们的不断反馈使我避免了很多错误，并改进了本书中和在线培训课程中所使用的代码以及 Python 笔记本。

同时也要感谢“Python 量化”和“AI 机器”的团队成员，特别是 Michael Schwed、Ramanathan Ramakrishnamoorthy 和 Prem Jebaseelan 在多方面给予我的支持，他们帮助我解决了在写作本书时遇到的各种技术难题。

还要感谢 O'Reilly Media 的团队成员，尤其是 Michelle Smith、Corbin Collins、Victoria DeRose 和 Danny Elfanbaum。因为有他们，才有了本书，他们在很多方面帮助我完善了本书。

当然，由于本人才疏学浅，难免会由于我个人的原因造成各种错漏。

此外，还要感谢 Refinitiv 的团队，特别是 Jason Ramchandani 提供的财务数据和持续的支持，本书中使用并提供给读者的主要数据文件都是通过各种方式从 Refinitiv 的数据接口获得的。

注 3：也可以通过图灵社区提交中文版勘误：ituring.cn/book/2858。——编者注

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

当然，今天利用人工智能和机器学习的每一个人，都得益于大量前人的成就和贡献。因此，我们应该永远铭记牛顿在 1675 年所写的那句话："如果说我看得比别人远一些，那是因为我站在了巨人的肩膀上。"从这个意义上说，非常感谢所有为该领域做出贡献的研究人员和开源维护者。

最后，特别要感谢我的家人，他们一直支持我的工作和写作。尤其感谢我的妻子 Sandra，她不懈地照顾家庭中的每一个人，为我们提供了一个充满爱的家庭环境。我把本书献给我可爱的妻子 Sandra 和我优秀的儿子 Henry。

电子书

扫描如下二维码，即可购买本书中文版电子书。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第一部分

机器智能

当今的算法交易程序还相对简单，对人工智能的应用还很有限。变革，势在必行！

——Murray Shanahan，2015 年

第一部分对人工智能进行了整体介绍。在这里，**人工**的含义是指智能不是以生物有机体的形式展示，而是以机器的形式展示；**智能**的含义可以参考人工智能研究者 Max Tegmark 的定义："达成复杂目标的能力。"本部分介绍人工智能领域的核心概念和算法，同时列举近期的重大突破，并从多个方面对超级智能进行讨论。本部分由两章组成。

- 第 1 章介绍人工智能领域的一般概念、思想和定义。这一章还提供一些 Python 示例，说明如何在实践中应用不同的算法。
- 第 2 章讨论通用人工智能和超级智能的相关概念和主题。前者所代表的智能在所有领域都能达到人类水平，后者所代表的智能则在特定领域会超越人类水平。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 1 章

人工智能

> 这是计算机程序第一次在全尺寸围棋棋盘上击败人类职业棋手，之前人们普遍认为这一壮举至少还需要十年时间。
>
> ——David Silver 等，2016 年

本章介绍**人工智能**（artificial intelligence，AI）领域的一般概念、思想和定义，以便于读者后续的阅读。此外，本章还为不同类型的主流学习算法提供了可以直接使用的示例。1.1 节从广阔的“算法”视角，对人工智能语境下的数据类型、学习类型和问题类型进行分类。本章还会介绍无监督学习和强化学习的相关示例。从 1.2 节开始，我们将直接进入神经网络的世界。神经网络不仅是本书后几章内容的核心基础，而且已被证明是人工智能领域目前最强大的算法之一。1.3 节会讨论数据量和数据多样性在人工智能领域中的重要性。

1.1　算法

本节介绍与本书相关的人工智能基本概念，还会讨论与人工智能这一术语相关的数据类型、学习类型、问题类型和方法类型。Alpaydin（2016）为本节中简略覆盖的各个主题提供了非正式的介绍，以及许多示例。

1.1.1　数据类型

数据通常有两个主要组成部分。

特征数据

　　特征数据（或称输入数据）是输入到算法中的数据。举例来说，在金融领域，特征数据可以是潜在债务人的收入和储蓄。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

标签数据

标签数据（或称输出数据）是需要学习的算法输出，比如可以利用监督学习算法来学习如何通过特征数据计算出给定的标签数据。在金融领域，标签数据可以是一个潜在借款方债务人的信誉度。

1.1.2　学习类型

学习算法主要有以下 3 类。

监督学习

监督学习（supervised learning，SL）从包含特征值（输入值）和标签值（输出值）的样本数据集中进行学习。1.2 节会介绍此类算法的示例，比如普通最小二乘（OLS）回归和神经网络。监督学习的目的是学习输入值和输出值之间的关系。在金融领域，可以训练此类算法来预测潜在债务人是否信誉良好。就本书而言，监督学习是最重要的算法类型。

无监督学习

无监督学习（unsupervised learning，UL）从仅包含特征值（输入值）的样本数据集中进行学习，目的通常是在数据中发现结构信息。这类算法通常基于给定的参数设置，对输入数据集进行学习。聚类算法就属于无监督学习算法。在金融领域，无监督学习算法可以用来对股票进行分组。

强化学习

强化学习（reinforcement learning，RL）通过从反复试验中不断试错来学习，并根据收到的奖励和惩罚来更新最佳行动策略。强化学习算法可以用于需要连续采取行动并且立即获得奖励的环境，比如用于计算机游戏中。

后文会详细讨论监督学习，这里将采用一些简明示例来说明无监督学习和强化学习。

1. 无监督学习

简而言之，k 均值（k-means）聚类算法将 n 个观测值分为 k 个类别，每个观测值归属于距离最近的类均值（中心）所属的类别。以下 Python 代码会生成用来聚类的样本数据。图 1-1 对聚类的样本数据进行了可视化展示，可以看出，此处使用的 scikit-learn 包中的 `KMeans` 算法完美地识别了数据中的聚类类别，图中的点按照算法学到的所属类别进行了着色。[1]

```
In [1]: import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        np.set_printoptions(precision=4, suppress=True)

In [2]: from sklearn.cluster import KMeans
        from sklearn.datasets import make_blobs

In [3]: x, y = make_blobs(n_samples=100, centers=4,
                          random_state=500, cluster_std=1.25)  ❶
```

注 1：有关详细信息，请参阅 `sklearn.cluster.KMeans` 和 VanderPlas（2017）的第 5 章。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [4]: model = KMeans(n_clusters=4, random_state=0)  ❷

In [5]: model.fit(x)  ❸
Out[5]: KMeans(n_clusters=4, random_state=0)

In [6]: y_ = model.predict(x)  ❹

In [7]: y_  ❺
Out[7]: array([3, 3, 1, 2, 1, 1, 3, 2, 1, 2, 2, 3, 2, 0, 0, 3, 2, 0, 2, 0, 0, 3,
               1, 2, 1, 1, 0, 0, 1, 3, 2, 1, 1, 0, 1, 3, 1, 3, 2, 2, 2, 1, 0, 0,
               3, 1, 2, 0, 2, 0, 3, 0, 1, 0, 1, 3, 1, 2, 0, 3, 1, 0, 3, 2, 3, 0,
               1, 1, 1, 2, 3, 1, 2, 0, 2, 3, 2, 0, 2, 2, 1, 3, 1, 3, 2, 2, 3, 2,
               0, 0, 0, 3, 3, 3, 3, 0, 3, 1, 0, 0], dtype=int32)

In [8]: plt.figure(figsize=(10, 6))
        plt.scatter(x[:, 0], x[:, 1], c=y_, cmap='coolwarm');
```

❶ 生成聚类用的样本数据集。

❷ 初始化 KMeans 模型对象，并指定类别数量。

❸ 用样本数据训练模型。

❹ 用训练好的模型生成预测结果。

❺ 预测结果为数字 0~3，每一个数字代表一个类别。

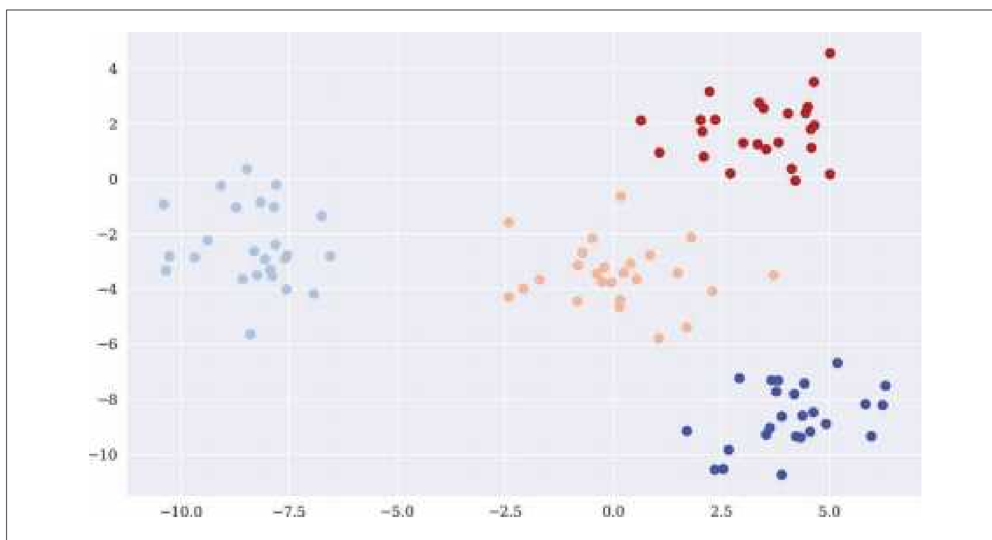


图 1-1：无监督聚类学习

一旦训练了如 KMeans 之类的算法，它就可以预测新样本（之前未见过的样本）所属的类别。假设我们在描述银行潜在债务人和实际债务人的特征数据集上训练这种算法，它可以通过生成两个类别来了解潜在债务人的信誉度，将新的潜在债务人归类为两个类别之一：

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

“信誉良好”与“信誉不佳”。

2. 强化学习

下面的示例基于《抛硬币》游戏，在该游戏中，硬币有 80%的时间正面朝上，有 20%的时间背面朝上。这个抛硬币游戏偏重于强调学习算法相较于随机基线算法的性能改进。在基线算法中，算法在正面和背面中随机下注，平均每 100 次下注可以赢 50 次，即总奖励值为 50。

```
In [9]: ssp = [1, 1, 1, 1, 0]  ❶

In [10]: asp = [1, 0]  ❷

In [11]: def epoch():
             tr = 0
             for _ in range(100):
                 a = np.random.choice(asp)  ❸
                 s = np.random.choice(ssp)  ❹
                 if a == s:
                     tr += 1  ❺
             return tr

In [12]: rl = np.array([epoch() for _ in range(15)])  ❻
         rl
Out[12]: array([53, 55, 50, 48, 46, 41, 51, 49, 50, 52, 46, 47, 43, 51, 52])

In [13]: rl.mean()  ❼
Out[13]: 48.93333333333333
```

❶ 定义状态空间（1= 正面，0= 背面）。

❷ 定义动作空间（1= 押注正面，0= 押注背面）。

❸ 从动作空间中随机选取一个动作。

❹ 从状态空间中随机选取一个状态。

❺ 如果押注正确，则奖励值加 1。

❻ 执行多轮实验，每轮实验押注 100 次。

❼ 计算每轮实验的奖励值。

强化学习尝试从采取行动后观察到的结果中学习，通常基于奖励。为简单起见，以下学习算法仅追踪每个回合中观察到的状态，并将观察到的状态附加到动作空间 `list` 对象中。通过这种方式，算法可以学习游戏中的正反面出现的概率偏差，虽然不能学得与游戏设置完全一致，但也能比较好地进行。通过从更新后的动作空间中随机采样，可以自然而然地反映出游戏中的正反面出现的概率偏差，也即正面出现的概率偏高。随着采样次数的不断增加，最终平均大约 80%的时间会选择正面。平均总奖励约为 65，这反映了学习算法相较于随机基线算法的性能改进。

```
In [14]: ssp = [1, 1, 1, 1, 0]

In [15]: def epoch():
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        tr = 0
        asp = [0, 1]  ❶
        for _ in range(100):
            a = np.random.choice(asp)
            s = np.random.choice(ssp)
            if a == s:
                tr += 1
            asp.append(s)  ❷
        return tr

In [16]: rl = np.array([epoch() for _ in range(15)])
         rl
Out[16]: array([64, 65, 77, 65, 54, 64, 71, 64, 57, 62, 69, 63, 61, 66, 75])

In [17]: rl.mean()
Out[17]: 65.13333333333334
```

❶ 每一轮游戏开始前重置动作空间。

❷ 将观察到的状态加入动作空间。

1.1.3 任务类型

根据标签数据的类型和需要解决的问题，有以下两种比较重要的学习任务。

估计

　　估计（也被称为近似或者回归）是指标签数据是（连续的）实数值的情况，在计算机中通常表示为浮点数。

分类

　　分类是指标签数据由有限数量的分组或者类别组成的情况，这些类别通常由离散值（正自然数）表示，在计算机中表示为整数。

后续章节将分别提供两种任务的示例。

1.1.4 方法类型

最后再介绍 3 个主要术语以及它们之间的差别，以便在后续章节中使用。

人工智能

　　人工智能除了涵盖之前定义的所有类型的学习算法，还包括更多类型，比如专家系统。

机器学习

　　机器学习（machine learning，ML）泛指基于一个算法和评估指标从数据集中学习数据之间关系和其他信息的一类学科。例如，在给定需要估计输出的标签值以及基于算法的预测值时，一个成功的评估指标可能是均方误差（MSE）。机器学习是人工智能的子集。

深度学习

　　深度学习（deep learning，DL）包含所有基于神经网络的算法。**深度**在这里表示神经网络具有多个隐藏层。深度学习是机器学习的子集，因此也是人工智能的子集。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

事实证明，深度学习适用的问题领域非常广泛。它适用于估计任务和分类任务，以及强化学习任务。在许多情况下，基于深度学习的方法比其他算法（比如逻辑回归或支持向量机等核方法）性能更好。[2] 这也是本书主要关注深度学习的原因。书中使用的深度学习方法包括密集神经网络（dense neural network，DNN）、循环神经网络（recurrent neural netwok，RNN）和卷积神经网络（convolutional neural network，CNN）。在后续章节中，尤其是第三部分，我们将针对深度学习的细节进行更多的介绍。

1.2 神经网络

前文对人工智能算法进行了一般性的概述，本节将就神经网络展开讨论。我们这里用一个简单的示例说明，与 OLS 回归等传统的统计方法相比，神经网络的不同之处。该示例从一个数学函数开始，先使用线性回归的方式进行**估计**（或函数逼近），然后再应用神经网络方式进行估计。这里采用的是一种监督学习方法，其中的任务是根据特征数据估计标签数据。同时，本节还会演示如何在分类问题中使用神经网络。

1.2.1 OLS回归

假定数学函数如下：

$$f:\mathbb{R}\to\mathbb{R}, y=2x^2-\frac{1}{3}x^3$$

这个函数将输入值 x 转换为输出值 y，或者将一系列输入值 $x_1, x_2, \cdots, x_N$ 转换为一系列输出值 $y_1, y_2, \cdots, y_N$。以下 Python 代码将这个数学函数实现为 Python 函数，并创建了一组输入值和输出值。图 1-2 绘制了输出值与输入值的关系图。

```
In [18]: def f(x):
             return 2 * x ** 2 - x ** 3 / 3  ❶
In [19]: x = np.linspace(-2, 4, 25)  ❷
         x  ❷
Out[19]: array([-2.  , -1.75, -1.5 , -1.25, -1.  , -0.75, -0.5 , -0.25,  0.  ,
                 0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ,  2.25,
                 2.5 ,  2.75,  3.  ,  3.25,  3.5 ,  3.75,  4.  ])

In [20]: y = f(x)  ❸
         y  ❸
Out[20]: array([10.6667,  7.9115,  5.625 ,  3.776 ,  2.3333,  1.2656,  0.5417,
                 0.1302,  0.    ,  0.1198,  0.4583,  0.9844,  1.6667,  2.474 ,
                 3.375 ,  4.3385,  5.3333,  6.3281,  7.2917,  8.1927,  9.    ,
                 9.6823, 10.2083, 10.5469, 10.6667])

In [21]: plt.figure(figsize=(10, 6))
         plt.plot(x, y, 'ro');
```

❶ 数学函数的 Python 函数形式。

注 2：有关详细信息，请参阅 VanderPlas（2017）的第 5 章。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❷ 输入值。

❸ 输出值。

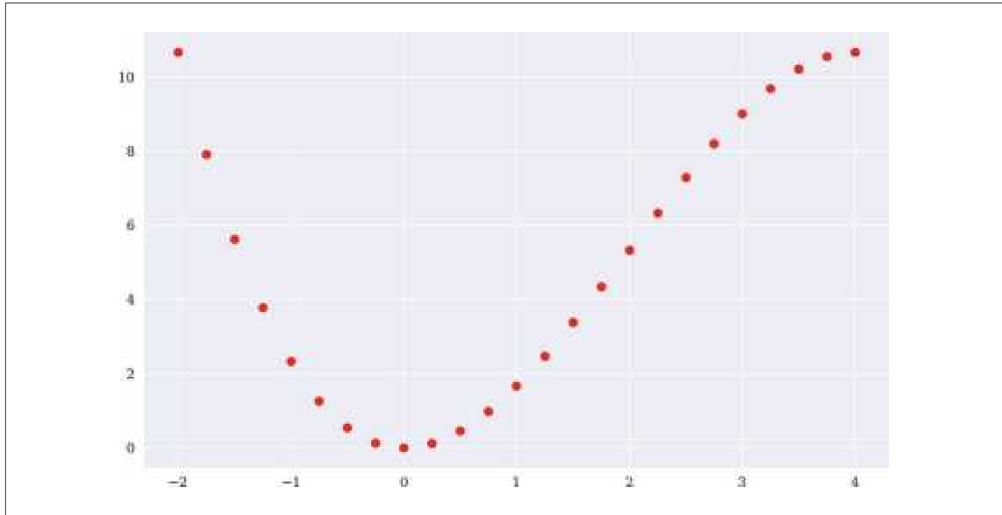


图 1-2：输出值与输入值的关系图

在示例函数的数学形式表达中，函数排在第一位，输入数据排在第二位，输出数据排在第三位。而在**统计学习**中的表达顺序则是不同的。假设同样是给定上述输入值 x 和输出值 y，也即**样本**（数据），在**统计回归**中将问题定义为找到一个函数，尽可能地逼近输入值（也称为**自变量**）和输出值（也称为**因变量**）之间的函数关系。

这里以一个简单的 OLS 线性回归为例，假定输入值和输出值之间的函数关系是线性的，需要求解的问题是为以下线性方程式找到最优参数 α 和 β：

$$\hat{f}:\mathbb{R}\rightarrow\mathbb{R}, \hat{y}=\alpha+\beta x$$

对于给定的输入值 $x_1, x_2, \cdots, x_N$ 和输出值 $y_1, y_2, \cdots, y_N$，最优解将使得实际输出值与近似输出值之间的 MSE 最小：

$$\min_{\alpha,\beta}\frac{1}{N}\sum_{n}^{N}(y_n-\hat{f}(x_n))^2$$

对于简单的线性回归，可以通过公式计算出最优参数 (α^*, β^*) 的解析解，如以下公式所示，变量上的横线代表样本均值：

$$\beta^*=\frac{\text{Cov}(x,y)}{\text{Var}(x)}$$

$$\alpha^*=\overline{y}-\beta\overline{x}$$

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

以下 Python 代码通过线性估计（逼近）输出值来计算最优参数值，同时在样本数据上绘制线性回归线，如图 1-3 所示。通过较高的 MSE 值可以看出，线性回归方法在逼近函数关系时效果不佳。

```
In [22]: beta = np.cov(x, y, ddof=0)[0, 1] / np.var(x)  ❶
         beta  ❶
Out[22]: 1.0541666666666667

In [23]: alpha = y.mean() - beta * x.mean()  ❷
         alpha  ❷
Out[23]: 3.8625000000000003

In [24]: y_ = alpha + beta * x  ❸

In [25]: MSE = ((y - y_) ** 2).mean()  ❹
         MSE  ❹
Out[25]: 10.721953125

In [26]: plt.figure(figsize=(10, 6))
         plt.plot(x, y, 'ro', label='sample data')
         plt.plot(x, y_, lw=3.0, label='linear regression')
         plt.legend();
```

❶ 计算 β 的最优解。

❷ 计算 α 的最优解。

❸ 计算估计输出值。

❹ 根据估计输出值计算 MSE。

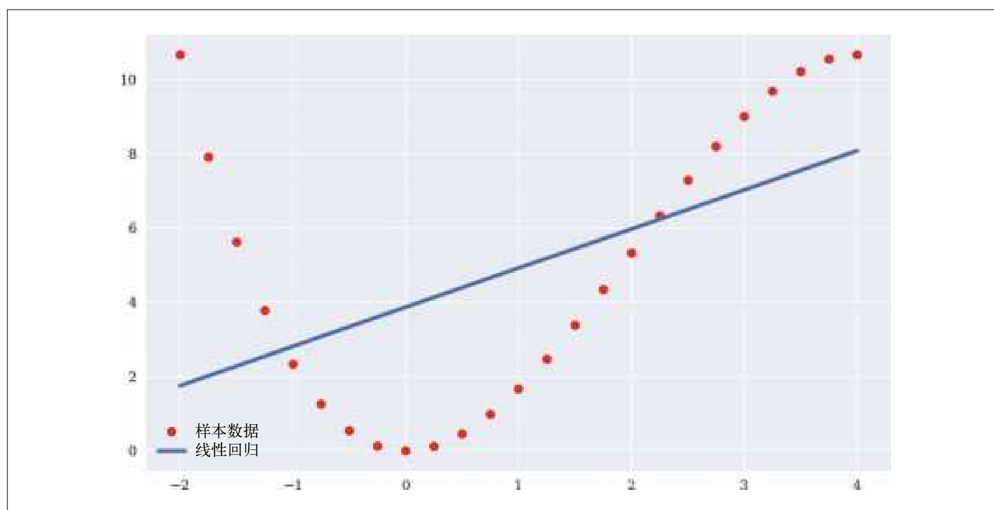


图 1-3：样本数据和线性回归线[3]

注 3：为了便于读者理解，本书大部分图片中的英文翻译成了中文。——译者注

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

如何才能改进（降低）MSE 值，甚至将其降低至 0，以达到“完美估计”？当然，OLS 回归不限于用来近似简单的线性关系。除了常数项和一次项，高次项也可以作为回归的基函数被加入进来。为了证明这一点，可以将图 1-4 所示的回归结果与用来创建该图的代码进行对比。使用二次项和三次项作为基函数所带来的改进是显而易见的，同时我们可以通过计算 MSE 值来从数值上验证这一点。对于不超过三次项的基函数，都可以通过包含二次项和三次项的 OLS 回归进行完美逼近并完美恢复函数关系。

```
In [27]: plt.figure(figsize=(10, 6))
         plt.plot(x, y, 'ro', label='sample data')
         for deg in [1, 2, 3]:
             reg = np.polyfit(x, y, deg=deg)  ❶
             y_ = np.polyval(reg, x)  ❷
             MSE = ((y - y_) ** 2).mean()  ❸
             print(f'deg={deg} | MSE={MSE:.5f}')
             plt.plot(x, np.polyval(reg, x), label=f'deg={deg}')
         plt.legend();
         deg=1 | MSE=10.72195
         deg=2 | MSE=2.31258
         deg=3 | MSE=0.00000

In [28]: reg  ❹
Out[28]: array([-0.3333,  2.    ,  0.    , -0.    ])
```

❶ 回归训练。

❷ 回归预测。

❸ 计算 MSE。

❹ 最优参数值。

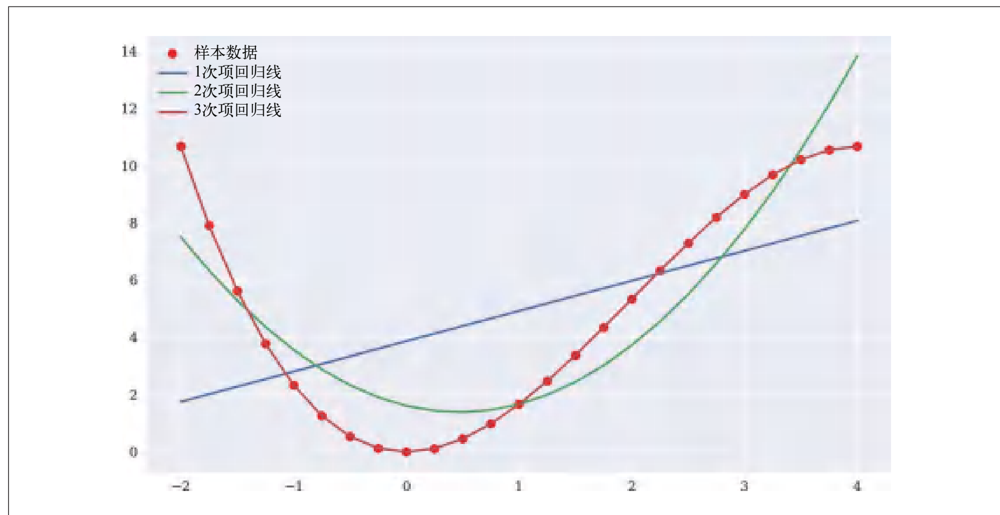


图 1-4：样本数据和 OLS 回归线

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

我们可以利用对于所逼近函数具体形式的了解，在回归中添加更多的基函数来达到“完美逼近”，也就是说，通过 OLS 回归可以分别恢复原始函数中二次项和三次项系数的精确值。

1.2.2 神经网络估计

然而，并不是所有关系都是简单的线性关系或者高阶线性关系，这时就需要借助神经网络（neural network，NN）等方法来进行建模。无须赘述，神经网络可以在不需要知道函数关系具体形式的条件下近似各种函数关系。

1. scikit-learn

以下 Python 代码使用了 `scikit-learn` 库中的 `MLPRegressor` 类，该类可用 DNN 进行回归估计。DNN 有时也被称为多层感知器（multi-layer perceptron，MLP）。[4] 如图 1-5 中的 MSE 所示，结果并不完美，但是对一个配置简单的模型来说，效果已经非常不错了。

```
In [29]: from sklearn.neural_network import MLPRegressor

In [30]: model = MLPRegressor(hidden_layer_sizes=3 * [256],
                              learning_rate_init=0.03,
                              max_iter=5000)  ❶

In [31]: model.fit(x.reshape(-1, 1), y)  ❷
Out[31]: MLPRegressor(hidden_layer_sizes=[256, 256, 256], learning_rate_init=0.03,
                      max_iter=5000)

In [32]: y_ = model.predict(x.reshape(-1, 1))  ❸

In [33]: MSE = ((y - y_) ** 2).mean()
         MSE
Out[33]: 0.021662355744355866

In [34]: plt.figure(figsize=(10, 6))
         plt.plot(x, y, 'ro', label='sample data')
         plt.plot(x, y_, lw=3.0, label='dnn estimation')
         plt.legend();
```

❶ 实例化 `MLPRegressor` 对象。

❷ 拟合或学习步骤。

❸ 预测步骤。

注 4：有关详细信息，请参阅 `sklearn.neural_network.MLPRegressor`。如需了解更多背景信息，请参阅 Goodfellow 等（2016）的第 6 章。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

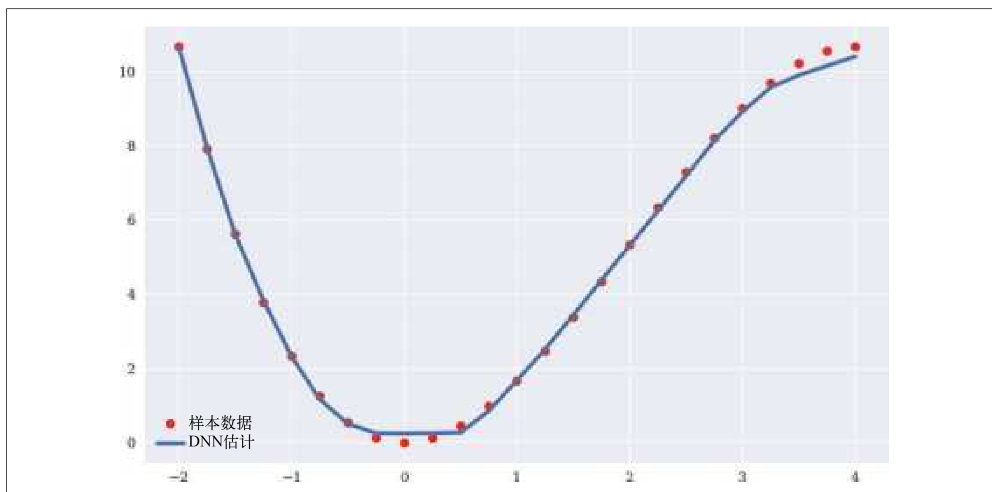


图 1-5：样本数据和基于神经网络的估计

仅对图 1-4 和图 1-5 中的结果进行观察，你可能会认为两种方法差异不大。但是，需要强调一个根本的区别：尽管 OLS 回归方法（如简单线性回归所示）基于确定性的计算逻辑是基于给定的数值和参数进行计算，但神经网络方法依赖于有一定随机性的**增量学习**。神经网络首先随机初始化一组参数（神经网络中的**权重**），然后根据神经网络输出值与样本输出值之间的差异对参数逐步进行调整。这种方法使得我们可以通过增量的方式逐步重新训练（更新）神经网络。

2. Keras

下一个示例使用了 Keras 深度学习软件包中的序列模型 Sequential。[5] 对该模型每轮拟合或训练进行 100 次迭代，共重复 5 轮。每轮训练之后，我们将更新并绘制由神经网络预测的近似值。图 1-6 显示了随着每一轮训练的近似值的准确率逐渐提高，MSE 值逐渐降低。与之前的模型相似，最终结果并不完美，但是鉴于模型的简单性，它还是不错的。

```
In [35]: import tensorflow as tf
         tf.random.set_seed(100)

In [36]: from keras.layers import Dense
         from keras.models import Sequential
         Using TensorFlow backend.

In [37]: model = Sequential()  ❶
         model.add(Dense(256, activation='relu', input_dim=1))  ❷
         model.add(Dense(1, activation='linear'))  ❸
         model.compile(loss='mse', optimizer='rmsprop')  ❹

In [38]: ((y - y_) ** 2).mean()
Out[38]: 0.021662355744355866
```

注 5：有关详细信息，请参阅 Chollet（2017）的第 3 章。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [39]: plt.figure(figsize=(10, 6))
         plt.plot(x, y, 'ro', label='sample data')
         for _ in range(1, 6):
         model.fit(x, y, epochs=100, verbose=False)  ❺
         y_ =  model.predict(x)  ❻
         MSE = ((y - y_.flatten()) ** 2).mean()  ❼
         print(f'round={_} | MSE={MSE:.5f}')
         plt.plot(x, y_, '--', label=f'round={_}')  ❽
     plt.legend();
     round=1 | MSE=3.09714
     round=2 | MSE=0.75603
     round=3 | MSE=0.22814
     round=4 | MSE=0.11861
     round=5 | MSE=0.09029
```

❶ 实例化 `Sequential` 模型对象。

❷ 添加采用整流线性单元（ReLU）激活函数的全连接层作为隐藏层。

❸ 添加线性激活的输出层。

❹ 编译模型对象。

❺ 迭代训练神经网络指定次数。

❻ 预测近似值。

❼ 计算当前的 MSE。

❽ 绘制当前的近似结果。

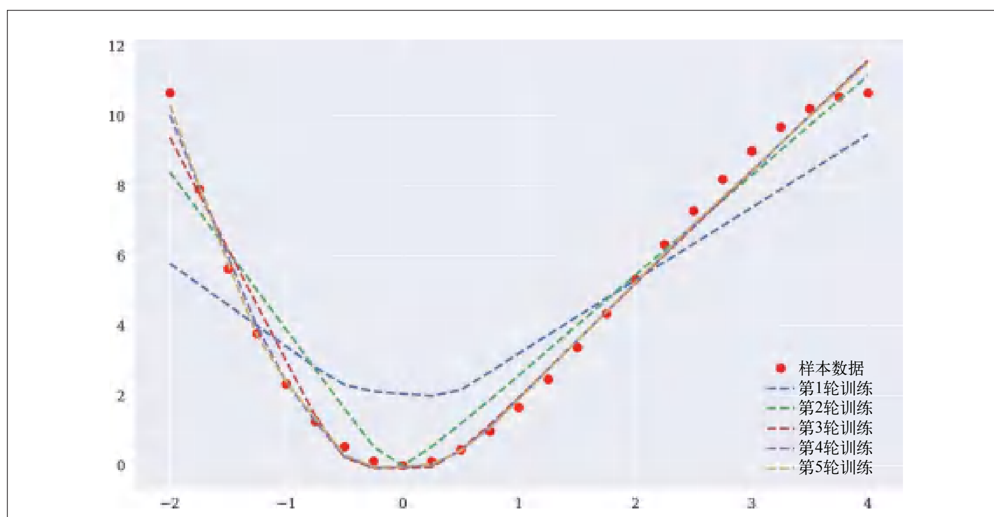


图 1-6：样本数据和多轮训练后得到的估计值

粗略地讲，神经网络提供了与 OLS 回归几乎一样完美的估计性能，那么为什么还要使用神经网络呢？本书稍后会提供更全面的解答，但这里我们可以通过一个不同的示例得到些许提示。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

假设我们的数据不是通过预定义好的数学函数生成的，而是随机产生的特征和标签。当然，该示例仅用于说明，并不具有解释性。

以下代码会生成随机样本数据集，并创建不同的多次项 OLS 回归进行逼近。从图 1-7 显示的结果可以看出，即使是最高次项的 OLS 回归，逼近结果仍然不是很好，MSE 值相对较高。

```
In [40]: np.random.seed(0)
         x = np.linspace(-1, 1)
         y = np.random.random(len(x)) * 2 - 1

In [41]: plt.figure(figsize=(10, 6))
         plt.plot(x, y, 'ro', label='sample data')
         for deg in [1, 5, 9, 11, 13, 15]:
             reg = np.polyfit(x, y, deg=deg)
             y_ = np.polyval(reg, x)
             MSE = ((y - y_) ** 2).mean()
             print(f'deg={deg:2d} | MSE={MSE:.5f}')
             plt.plot(x, np.polyval(reg, x), label=f'deg={deg}')
         plt.legend();
         deg= 1 | MSE=0.28153
         deg= 5 | MSE=0.27331
         deg= 9 | MSE=0.25442
         deg=11 | MSE=0.23458
         deg=13 | MSE=0.22989
         deg=15 | MSE=0.21672
```

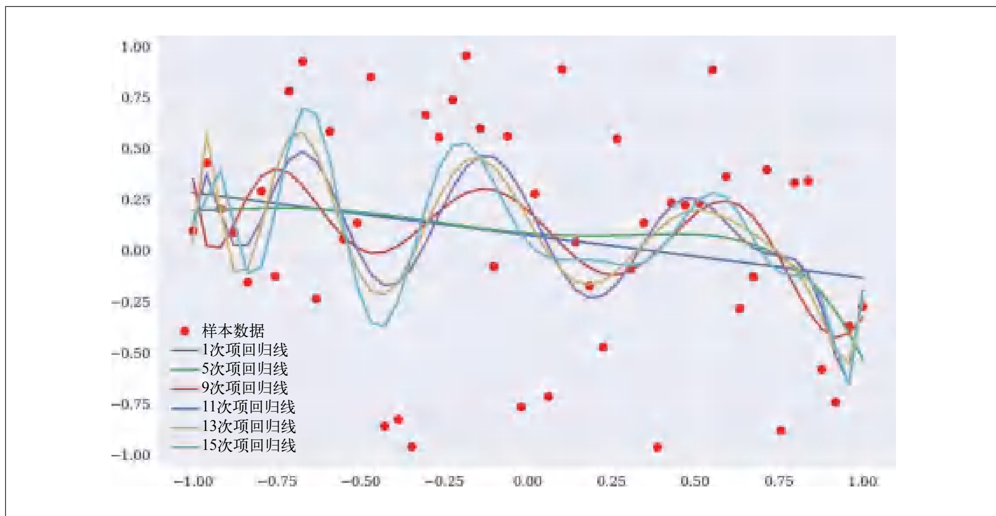


图 1-7：随机样本数据和 OLS 回归线

不出所料，OLS 回归的效果并不理想。OLS 回归假设我们可以通过有限个（基于多项式的）基函数的组合来逼近目标函数。由于样本数据集是随机生成的，因此在这种情况下，OLS 回归效果不佳。

神经网络表现如何呢？它使用起来仍然像前面的示例一样简单，逼近的结果如图 1-8 所示。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

尽管预测结果并不完美，但很明显，在根据随机特征值估计随机标签值时，神经网络的性能优于 OLS 回归。然而，神经网络架构有近 200 000 个可训练的参数（权重），这提供了相对较高的灵活性，尤其是与 OLS 回归（最多使用 15+1 个参数）相比。

```
In [42]: model = Sequential()
         model.add(Dense(256, activation='relu', input_dim=1))
         for _ in range(3):
             model.add(Dense(256, activation='relu'))  ❶
         model.add(Dense(1, activation='linear'))
         model.compile(loss='mse', optimizer='rmsprop')

In [43]: model.summary()  ❷
         Model: "sequential_2"
         _________________________________________________________________
         Layer (type)                 Output Shape              Param #
         =================================================================
         dense_3 (Dense)              (None, 256)               512
         _________________________________________________________________
         dense_4 (Dense)              (None, 256)               65792
         _________________________________________________________________
         dense_5 (Dense)              (None, 256)               65792
         _________________________________________________________________
         dense_6 (Dense)              (None, 256)               65792
         _________________________________________________________________
         dense_7 (Dense)              (None, 1)                  257
         =================================================================
         Total params: 198,145
         Trainable params: 198,145
         Non-trainable params: 0
         _________________________________________________________________

In [44]: %%time
         plt.figure(figsize=(10, 6))
         plt.plot(x, y, 'ro', label='sample data')
         for _ in range(1, 8):
             model.fit(x, y, epochs=500, verbose=False)
             y_ =  model.predict(x)
             MSE = ((y - y_.flatten()) ** 2).mean()
             print(f'round={_} | MSE={MSE:.5f}')
             plt.plot(x, y_, '--', label=f'round={_}')
         plt.legend();
         round=1 | MSE=0.13560
         round=2 | MSE=0.08337
         round=3 | MSE=0.06281
         round=4 | MSE=0.04419
         round=5 | MSE=0.03329
         round=6 | MSE=0.07676
         round=7 | MSE=0.00431
         CPU times: user 30.4 s, sys: 4.7 s, total: 35.1 s
         Wall time: 13.6 s
```

❶ 添加多个隐藏层。

❷ 显示神经网络架构以及可训练参数的数量。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

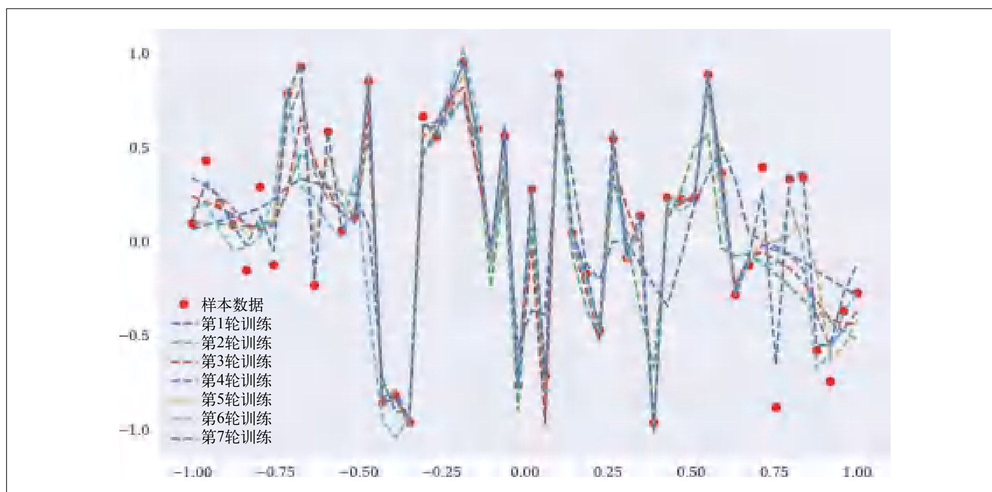


图 1-8：随机样本数据和神经网络估计

1.2.3 神经网络分类

神经网络也可以很容易地用于分类任务。考虑以下基于 Keras 实现神经网络分类的 Python 代码。二元特征数据和二元标签数据是随机生成的。建模方面的主要调整是将输出层的激活函数从 linear 更改为 sigmoid，后面的章节会对此作更详细的介绍。虽然分类效果并不完美，但是也达到了很高的准确率。图 1-9 显示了准确率（表示为正确结果与所有标签值之间的关系）如何随着训练轮数而变化。准确率一开始很低，然后逐步提高，尽管不一定每一步都会有所提高。

```
In [45]: f = 5
         n = 10

In [46]: np.random.seed(100)

In [47]: x = np.random.randint(0, 2, (n, f))  ❶
         x  ❶
Out[47]: array([[0, 0, 1, 1, 1],
                [1, 0, 0, 0, 0],
                [0, 1, 0, 0, 0],
                [0, 1, 0, 0, 1],
                [0, 1, 0, 0, 0],
                [1, 1, 1, 0, 0],
                [1, 0, 0, 1, 1],
                [1, 1, 1, 0, 0],
                [1, 1, 1, 1, 1],
                [1, 1, 1, 0, 1]])

In [48]: y = np.random.randint(0, 2, n)  ❷
         y  ❷
Out[48]: array([1, 1, 0, 0, 1, 1, 0, 1, 0, 1])
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [49]: model = Sequential()
         model.add(Dense(256, activation='relu', input_dim=f))
         model.add(Dense(1, activation='sigmoid'))  ❸
         model.compile(loss='binary_crossentropy', optimizer='rmsprop',
                       metrics=['acc'])  ❹

In [50]: model.fit(x, y, epochs=50, verbose=False)
Out[50]: <keras.callbacks.callbacks.History at 0x7fde09dd1cd0>

In [51]: y_ = np.where(model.predict(x).flatten() > 0.5, 1, 0)
         y_
Out[51]: array([1, 1, 0, 0, 0, 1, 0, 1, 0, 1], dtype=int32)

In [52]: y == y_  ❺
Out[52]: array([ True,  True,  True,  True, False,  True,  True,  True,  True,
                 True])

In [53]: res = pd.DataFrame(model.history.history)  ❻

In [54]: res.plot(figsize=(10, 6));  ❻
```

❶ 创建随机特征数据。

❷ 创建随机标签数据。

❸ 定义输出层的激活函数为 sigmoid。

❹ 定义损失函数为 binary_crossentropy[6]。

❺ 将预测值与标签数据进行比较。

❻ 绘制每轮训练的损失函数和准确率值。

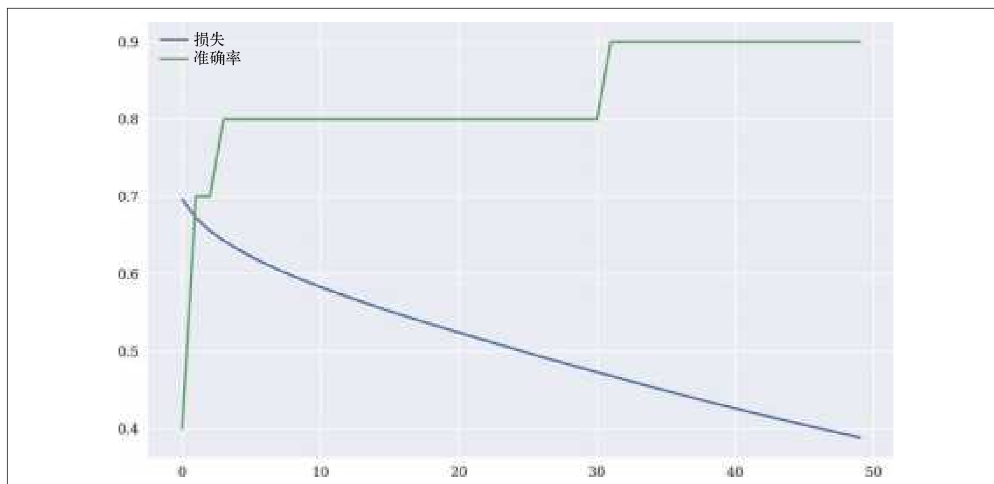


图 1-9：分类准确率及损失与训练轮数的关系

注 6：**损失函数**用来计算和衡量神经网络（或其他机器学习算法）的预测错误。**二元交叉熵**（binary crossentropy）适用于二元分类问题的损失函数，而 MSE 则适用于估计问题。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

本节中的示例说明了与 OLS 回归相比，神经网络的一些基本特征。

问题无关性

在给定一组特征值的情况下，神经网络方法的性能与需要估计或者分类的具体标签值是无关的。而统计方法（比如 OLS 回归）可能对较小的一组问题表现良好，对其他问题则表现不太好或根本没有效果。

增量学习

给定一个用来度量成功的目标，神经网络中的最佳权重是基于随机初始化和增量改进而逐步学习得到的。这些增量改进是在考虑预测值和样本标签值之间的差异后，通过神经网络反向传播权重更新来实现的。

通用函数逼近器

有严格的数学定理表明神经网络（即使只有一个隐藏层）几乎可以逼近任何函数。[7]

这些特点或许可以解释为什么本书将神经网络放在所用算法的核心位置，第 2 章会在这方面进行更多的讨论。

神经网络

神经网络擅长学习输入数据和输出数据之间的关系。它们可以应用于许多问题类型，比如在存在复杂关系的情况下进行估计，或者用于传统的统计方法并不适合的分类问题。

1.3 数据的重要性

上一节末尾的例子表明神经网络能够很好地解决分类问题。具有一个隐藏层的神经网络在给定的数据集或**样本内**达到了高度的准确率。然而，神经网络的预测能力如何呢？这在很大程度上取决于可用于训练神经网络的数据量和种类。另一个基于更大数据集的数值示例将说明这一点。

1.3.1 小数据集

现在考虑一个与之前分类示例中使用的随机样本类似的数据集，但其具有更多特征和更多样本。人工智能中使用的大多数算法是关于**模式识别**的。在下面的 Python 代码中，二值化特征的数量定义了算法可以学习的可能的模式的数量。鉴于标签数据也是二值化的，算法会尝试了解在给定某种模式的情况下（比如特征为 `[0, 0, 1, 1, 1, 1, 0, 0, 0, 0]`），标签是 `0` 的可能性更大，还是 `1` 的可能性更大。因为所有数字都是以相等的概率随机选择的，所以除了观察到无论在什么（随机）模式下，标签 `0` 和 `1` 的可能性都相等这一事实之外，没有太多可学习的。因此，基线预测算法在大约 50% 的时间内应该是准确的，无论它呈现什么（随机）模式。

```
In [55]: f = 10
         n = 250
```

注 7：参阅 Kratsios（2019）。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [56]: np.random.seed(100)

In [57]: x = np.random.randint(0, 2, (n, f))  ❶
         x[:4]  ❶
Out[57]: array([[0, 0, 1, 1, 1, 1, 0, 0, 0, 0],
                [0, 1, 0, 0, 0, 0, 1, 0, 0, 1],
                [0, 1, 0, 0, 0, 1, 1, 1, 0, 0],
                [1, 0, 0, 1, 1, 1, 1, 1, 0, 0]])

In [58]: y = np.random.randint(0, 2, n)  ❷
         y[:4]  ❷
Out[58]: array([0, 1, 0, 0])

In [59]: 2 ** f  ❸
Out[59]: 1024
```

❶ 特征数据。

❷ 标签数据。

❸ 模式数量。

为了简化后续某些操作和分析，原始数据被放入一个 `pandas DataFrame` 对象中。

```
In [60]: fcols = [f'f{_}' for _ in range(f)]  ❶
         fcols  ❶
Out[60]: ['f0', 'f1', 'f2', 'f3', 'f4', 'f5', 'f6', 'f7', 'f8', 'f9']

In [61]: data = pd.DataFrame(x, columns=fcols)  ❷
         data['l'] = y  ❸

In [62]: data.info()  ❹
         <class 'pandas.core.frame.DataFrame'>
         RangeIndex: 250 entries, 0 to 249
         Data columns (total 11 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   f0      250 non-null    int64
          1   f1      250 non-null    int64
          2   f2      250 non-null    int64
          3   f3      250 non-null    int64
          4   f4      250 non-null    int64
          5   f5      250 non-null    int64
          6   f6      250 non-null    int64
          7   f7      250 non-null    int64
          8   f8      250 non-null    int64
          9   f9      250 non-null    int64
          10  l       250 non-null    int64
         dtypes: int64(11)
         memory usage: 21.6 KB
```

❶ 定义特征数据的列名。

❷ 将特征数据放入 `DataFrame` 对象中。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❸ 将标签数据放入同一个 `DataFrame` 对象中。

❹ 显示数据集的元信息。

根据以下 Python 代码的执行结果，可以确定两个主要问题：首先，并非所有模式都在样本数据集中；其次，每个观察到的模式样本量太小。即使不深入挖掘，也可以很明显地知道没有分类算法可以真正以有意义的方式了解所有可能的模式。

```
In [63]: grouped = data.groupby(list(data.columns))  ❶

In [64]: freq = grouped['l'].size().unstack(fill_value=0)  ❷

In [65]: freq['sum'] = freq[0] + freq[1]  ❸

In [66]: freq.head(10)  ❹
Out[66]: l                              0  1  sum
         f0 f1 f2 f3 f4 f5 f6 f7 f8 f9
         0  0  0  0  0  0  0  1  1  1   0  1    1
                              1  0  1  0   1  1    2
                                       1   0  1    1
                           1  0  0  0  0   1  0    1
                                       1   0  1    1
                                 1  1  1   0  1    1
                              1  0  0  0   0  1    1
                                    1  0   0  1    1
                        1  0  0  0  1  1   1  0    1
                              1  1  0  0   1  0    1

In [67]: freq['sum'].describe().astype(int)  ❺
Out[67]: count    227
         mean       1
         std        0
         min        1
         25%        1
         50%        1
         75%        1
         max        2
         Name: sum, dtype: int64
```

❶ 按列对数据进行分组。

❷ 取消标签列分组数据的折叠。

❸ 将 `0` 和 `1` 的频率相加。

❹ 显示给定特定模式的 `0` 和 `1` 的频率。

❺ 提供频率总和的统计信息。

以下 Python 代码使用了来自 `scikit-learn`[8] 的 `MLPClassifier` 模型，该模型是在完整数据集上训练的。神经网络对于给定数据集中关系的刻画能力如何呢？正如样本内准确率得分所示，这种能力非常高，接近 100%，这在很大程度上是由相对较小的数据集以及相对较高

注 8：有关详细信息，请参阅 `sklearn.neural_network.MLPClassifier`。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

的神经网络容量所导致的结果。

```
In [68]: from sklearn.neural_network import MLPClassifier
         from sklearn.metrics import accuracy_score

In [69]: model = MLPClassifier(hidden_layer_sizes=[128, 128, 128],
                               max_iter=1000, random_state=100)

In [70]: model.fit(data[fcols], data['l'])
Out[70]: MLPClassifier(hidden_layer_sizes=[128, 128, 128], max_iter=1000,
                       random_state=100)

In [71]: accuracy_score(data['l'], model.predict(data[fcols]))
Out[71]: 0.952
```

但是训练好的神经网络的**预测能力**如何呢？为此，可以将给定的数据集拆分为训练数据集和测试数据集。模型仅在训练数据集上进行训练，然后就其对测试数据集的预测能力进行测试。和以前一样，训练好的神经网络在样本内（训练数据集）的准确率非常高。然而，它比测试数据集上的基线算法差 10 个百分点以上。

```
In [72]: split = int(len(data) * 0.7)  ❶

In [73]: train = data[:split]  ❶
         test = data[split:]  ❶

In [74]: model.fit(train[fcols], train['l'])  ❷
Out[74]: MLPClassifier(hidden_layer_sizes=[128, 128, 128], max_iter=1000,
                       random_state=100)

In [75]: accuracy_score(train['l'], model.predict(train[fcols]))  ❸
Out[75]: 0.9714285714285714

In [76]: accuracy_score(test['l'], model.predict(test[fcols]))  ❹
Out[76]: 0.38666666666666666
```

❶ 将数据拆分为训练数据集和测试数据集。

❷ 仅在训练数据集上训练模型。

❸ 报告样本内（训练数据集）的准确率。

❹ 报告样本外（测试数据集）的准确率。

粗略地说，由于两个已知的问题，仅在小数据集上训练的神经网络会学习到错误的关系。这些问题对于**样本内**的关系学习影响不大。相反，一般来说，数据集越小，越容易学习样本内关系。然而，当使用训练好的神经网络进行**样本外**预测时，这些问题会带来麻烦。

1.3.2 更大的数据集

幸运的是，通常有一种明确的方法可以解决这种情况下的问题，即创建**更大的数据集**（或者说更多的数据）。面对现实世界的问题，理论上确实可以这样做。然而，从实际的角度来看，我们并不总是能够获得这种更大的数据集，而且它们通常也不那么容易生成。但

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

是，基于本节的示例，确实可以通过代码轻松创建更大的数据集。

以下 Python 代码显著增加了初始样本数据集中的样本数量。结果表明，训练好的神经网络的预测准确率提高了 10 个百分点以上，达到约 50% 的水平，这也是我们基于标签数据的生成逻辑所预期的。它现在的性能与一个标准的基线算法相符合。

```
In [77]: factor = 50

In [78]: big = pd.DataFrame(np.random.randint(0, 2, (factor * n, f)),
                            columns=fcols)

In [79]: big['l'] = np.random.randint(0, 2, factor * n)

In [80]: train = big[:split]
         test = big[split:]

In [81]: model.fit(train[fcols], train['l'])
Out[81]: MLPClassifier(hidden_layer_sizes=[128, 128, 128], max_iter=1000,
                       random_state=100)

In [82]: accuracy_score(train['l'], model.predict(train[fcols]))  ❶
Out[82]: 0.9657142857142857

In [83]: accuracy_score(test['l'], model.predict(test[fcols]))  ❷
Out[83]: 0.5043407707910751
```

❶ 样本内的预测准确率（训练数据集）。

❷ 样本外的预测准确率（测试数据集）。

如接下来的代码所示，我们可以通过对可用数据的快速分析，来解释为什么预测准确率有所提高。首先，因为数据量足够大，所以所有可能的模式现在都会出现在数据集中。其次，所有模式在数据集中的平均频率都在 10 以上。换句话说，对**每一种可能的模式**，神经网络都可以观测和学习**多次**。这使得神经网络可以“学习”到，标签 `0` 和 `1` 对于所有模式的可能性是相等的。当然，这是一种相当复杂的学习方式，但它很好地说明了一个事实：在基于神经网络的设置下，**相对较小**的数据集可能是不够的。

```
In [84]: grouped = big.groupby(list(data.columns))

In [85]: freq = grouped['l'].size().unstack(fill_value=0)

In [86]: freq['sum'] = freq[0] + freq[1]  ❶

In [87]: freq.head(6)
Out[87]: l                              0  1  sum
         f0 f1 f2 f3 f4 f5 f6 f7 f8 f9
         0  0  0  0  0  0  0  0  0  0  10  9   19
                                    1   5  4    9
                                 1  0   2  5    7
                                    1   6  6   12
                              1  0  0   9  8   17
                                    1   7  4   11
```

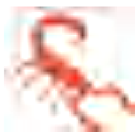

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [88]: freq['sum'].describe().astype(int)  ❷
Out[88]: count    1024
         mean       12
         std         3
         min         2
         25%        10
         50%        12
         75%        15
         max        26
         Name: sum, dtype: int64
```

❶ 将 0 和 1 的频率相加。

❷ 显示相加后的汇总统计信息。



数据量和多样性

在基于神经网络的预测任务中，用于训练神经网络的可用数据的数据量和多样性对其预测性能起决定性作用。本节中的假设示例表明，相较于在数据量较大且具有多样性的数据集上训练的神经网络，在数据量较小且较为单一的数据集上训练的相同神经网络，其表现会差 10 个百分点以上。考虑到人工智能从业者和公司经常为小到 1/10 个百分点的改进而奋斗，10 个百分点可以被认为是巨大的差异了。

1.3.3　大数据

更大的数据集和大数据集有什么区别？十多年来，**大数据**一词已被用于表示许多事物。就本书的目的而言，我们可以将**大数据**定义为在数量、种类和速度方面足够丰富的**大数据集**，它可以对人工智能算法进行适当的训练，从而使得与基线算法相比，人工智能算法在预测任务上表现得更好。

之前使用的较大的数据集在实际应用中仍然很小。但是，它足以完成指定的目标。训练人工智能算法所需要的数据集的数量和种类主要由特征及标签数据的结构和特点所决定。

在这种情况下，假设零售银行会使用基于神经网络的信用评分分类方法。给定内部数据，负责的数据科学家设计了 25 个分类特征，每个特征都可以采用 8 个不同的值，由此产生的模式数量是天文数字。

```
In [89]: 8 ** 25
Out[89]: 37778931862957161709568
```

很明显，没有一个数据集可以让神经网络观测到每一种模式。[9] 幸运的是，在实践中，对神经网络来说没有必要根据常规用户、违约用户和被拒绝用户的数据分别了解用户的信誉度。一般来说，也没有必要对**每个**潜在债务人的信誉度做出“好”的预测。

注 9：即使有这样的数据集，当前的计算技术也不足以基于此类数据集对神经网络进行建模和训练。基于此背景，第 2 章会讨论硬件对人工智能的重要性。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

这是由于多种原因造成的。这里仅举几个例子。首先是有效特征空间。并非每种模式都是实际存在的，有些模式可能根本不存在，或者完全不实际。其次是特征重要性。并非所有特征都是同等重要的，我们可以据此减少有效特征的数量，从而减少可能的模式数量。最后是特征冗余值。例如，编号为 7 的特征取值为 4 或 5 时可能根本没有区别，从而进一步减少了相关模式的数量。

1.4 结论

在本书中，人工智能包含能够从数据中学习诸如关系、规则、概率的方法，技术，算法等。本书重点关注监督学习算法，比如用于估计和分类的算法。在算法方面，核心是神经网络和深度学习方法。

本书的核心主题是将神经网络应用于金融领域的一个核心问题：对未来市场走势的预测。更具体地说，问题可能是预测股票指数或货币对的汇率变动方向。对未来市场方向的预测（目标水平或价格是上涨还是下跌）可以轻松转换为分类问题。

在深入探讨核心主题之前，第 2 章首先讨论与**超级智能**和**技术奇点**相关的一些主题。该讨论将为后续侧重于金融和人工智能在金融领域的应用的章节提供有用的背景知识。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 2 章

超级智能

> 有多条路径可以通向超级智能，这一事实应当增强我们对于最终实现超级智能的自信心，如果其中某一条路径最终被证明不可行，那么我们仍可以通过其他路径继续前行（正所谓“条条大路通罗马”）。
>
> ——Nick Bostrom，2014 年

技术奇点一词有多种定义。它的使用可以追溯至 Vinge（1993）的一篇文章，作者煽动性地使用了这样的开头：

> 三十年以内，我们所拥有的技术手段将能实现足以超越人类的智能。而在那之后不久，人类的时代就将结束。

对于本章和本书来说，**技术奇点**一词是指在某个时间点机器达到了超越人类的智能水平，或如作者 Vinge 在文章中所说的超级智能（super intelligence，SI）。随着 Kurzweil（2005）的著作被广为阅读和引用，这一说法和概念也更加被人们所认可。Barrat（2013）拥有丰富的关于该话题的历史和轶事信息。Shanahan（2015）则提供了对于其主要方面的一个非正式介绍和概览。**技术奇点**这一表达本身来自物理学中**奇点**的概念。奇点是指黑洞的中心，这是一个质量高度集聚，引力趋近于无穷大，同时传统物理学法则被打破的地方。宇宙的开始即所谓的宇宙大爆炸，同样也被称为一个奇点。

虽然这些关于技术奇点与超级智能的宽泛概念似乎与金融中应用的人工智能并无什么显而易见的直接关系，但是多了解人工智能的背景、相关问题以及其应用的潜在后果将是大有益处的。这是因为从一般性框架下获得的见解对于一个细分领域（比如人工智能在金融领域的应用）是很关键的。这些见解也会为我们关于人工智能在近期和长期内会如何重塑金融业的讨论指明方向。

2.1 节介绍了人工智能领域最近的几个成功例子，包括 DeepMind 公司是如何运用神经网络来玩雅达利 2600 游戏的，还包括这家公司是如何做到以高于人类专家的水平下围棋

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

的。关于国际象棋和计算机程序的故事也会在这一节被再次提及。2.2 节讨论了硬件在这些最近的成功例子中的重要性。2.3 节介绍了人工智能的不同形式，比如狭义人工智能（artificial narrow intelligence，ANI）、通用人工智能（artificial general intelligence，AGI）以及超级智能。2.4 节介绍了实现超级智能的可能方式，比如全脑模拟（WBE）。2.5 节讲述了研究者们是如何看待智能爆炸问题的。2.6 节讨论了在超级智能概念下的所谓控制问题的各个方面。最后，2.7 节简单陈述了应用这些技术的潜在结果以及超级智能实现后的世界景象。

2.1 成功故事

人工智能中的很多想法和算法可以追溯到几十年前，这一行业在过去几十年中一直是希望和绝望交织。Bostrom（2014）的第 1 章对该时期进行了回顾。

在 2020 年，就算不用“充满兴奋”这个词来描述，一个人也可以很确定地说人工智能正处在充满希望的时期。其中一个原因是，某些在过去几年内被视为未来数十年都无法通过人工智能解决的问题，近几年通过人工智能取得了突破性的进展。这类成功案例的数量很多，并且还在急速增长中。因此，本节仅仅专注于其中的 3 个成功故事。Gerrish（2018）提供了一个篇幅更长、描述也更细致的专栏。

2.1.1 雅达利（Atari）

本节首先讲述 DeepMind 公司如何利用强化学习与神经网络来玩雅达利 2600 游戏，随后通过一个具体的代码示例来展示其获得成功的基本方法。

1. 故事

第一个成功故事是如何以超出人类的水平玩雅达利 2600 游戏。雅达利 2600 游戏机上市于 1977 年，它是第一批在 20 世纪 80 年代广为流行的游戏主机之一。那个年代大受欢迎的游戏（比如《太空入侵者》《小行星》以及《导弹指挥官》），至今仍然是经典，并且在数十年后仍有复古游戏爱好者在玩。

DeepMind 发布了 Mnih 等人于 2013 年合著的一篇论文，其中详细说明了通过引入强化学习让算法或所谓的人工智能来玩雅达利 2600 游戏的结果。这一算法是应用在卷积神经网络上的 Q 学习的一个变体。[1] 该算法仅在高维的视觉输入（原始像素）上训练，而没有任何人为引导或人为输入。原本的项目专注于 7 款雅达利 2600 游戏，DeepMind 团队报道称人工智能在其中的 3 款游戏——《乒乓》《山地自行车耐力赛》和《打砖块》上有着超出人类专家的表现。

从人工智能的角度来看，DeepMind 取得如此成就的方法比这一结果本身更为人称道。首先，该团队仅仅使用了一个神经网络来学习和玩所有 7 款游戏。其次，人类没有提供任何指导或标记过的数据，神经网络的学习只依靠基于视觉输入的互动式学习过程所转化的特征数据。[2] 最后，DeepMind 团队实现这一结果的方法是通过强化学习，而这一学习方式仅依

注 1：有关详细信息，请参阅 Mnih 等（2013）。

注 2：除其他因素外，可用的 Arcade 学习环境（ALE）使这成为可能。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

靠对行动与结果（奖励）之间关系的观察，这基本上和人类玩家玩游戏的方式是一样的。

在雅达利 2600 游戏中，《打砖块》是 DeepMind 的人工智能的表现超出人类的其中一款游戏。此游戏的目标是通过用屏幕底部的一个挡板反弹小球来穿过屏幕，以摧毁在屏幕顶端线状排列的砖块。无论何时，只要小球撞到了砖块，砖块就将被摧毁，而小球会被反弹回来。小球也会从左、右和上部的墙上被反弹回来。一旦小球触及底部而未被挡板反弹回来，玩家就将失去一条“命”。

对于强化学习，其动作空间（action space）中存在着 3 种元素且均与挡板有关：保持在原地、向左移动和向右移动。而状态空间（state space）是由许多帧带有 128 色调色板的 210 像素 ×160 像素大小的游戏屏幕组成的。游戏分数代表所得到的奖励，而 DeepMind 算法正是被编程为最大化这一奖励。基于学习中所优化的行动策略（action policy），这一算法学习在给定的游戏状态下哪一个行动是最优解，也即如何去最大化游戏分数（总奖励）。

2. 示例

本章没有足够的篇幅来详细探讨 DeepMind 为《打砖块》和其他雅达利 2600 游戏所采用的方法。但是，我们可以通过 OpenAI Gym 环境来演示在类似但更简单的游戏中如何使用类似但更简单的神经网络。

本节中的 Python 代码适用于 OpenAI Gym 的 `CartPole` 环境。[3] 在此环境中，需要将推车向右或向左移动以保持推车上面的直立杆的平衡，因此，动作空间类似于《打砖块》的动作空间。状态空间由 4 个物理量（数据点）组成：小车位置、小车速度、杆子角度和杆子角速度（参见图 2-1）。如果在采取行动后杆子仍然处于平衡状态，智能体就会获得 1 的奖励。如果杆子失去平衡，则游戏结束。如果智能体达到 200 的总奖励，则认为该智能体是成功的。[4]

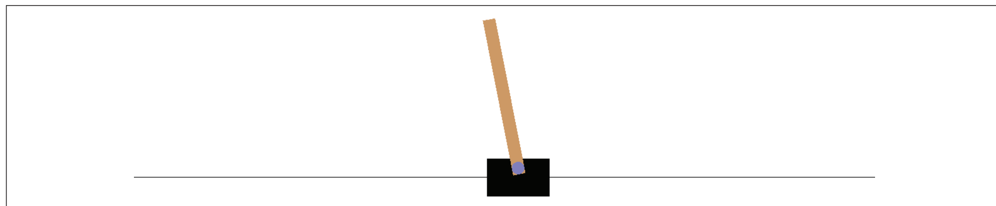


图 2-1：`CartPole` 环境的图形表示

下面的代码首先会实例化一个 `CartPole` 环境对象，然后会检查动作空间和状态空间，采取随机动作，并记录结果。当 `done` 变量值为 `False` 时，人工智能体会进入下一轮。

```
In [1]: import gym
        import numpy as np
        import pandas as pd
        np.random.seed(100)

In [2]: env = gym.make('CartPole-v0')  ❶
```

注 3：第 9 章更详细地回顾了这个例子。

注 4：更具体地说，如果人工智能体在连续 100 场比赛中达到 195 或更多的平均总奖励，则被认为是成功的。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [3]: env.seed(100)  ❷
Out[3]: [100]

In [4]: action_size = env.action_space.n  ❸
        action_size  ❸
Out[4]: 2

In [5]: [env.action_space.sample() for _ in range(10)]  ❹
Out[5]: [1, 0, 0, 0, 1, 1, 0, 0, 0, 0]

In [6]: state_size = env.observation_space.shape[0]  ❺
        state_size  ❺
Out[6]: 4

In [7]: state = env.reset()  ❻
        state #  [cart position, cart velocity, pole angle, pole angular velocity]
Out[7]: array([-0.01628537, 0.02379786, -0.0391981 , -0.01476447])

In [8]: state, reward, done, _ = env.step(env.action_space.sample())  ❼
        state, reward, done, _  ❼
Out[8]: (array([-0.01580941, -0.17074066, -0.03949338, 0.26529786]), 1.0, False, {})
```

❶ 实例化环境对象。

❷ 固定环境的随机数种子。

❸ 显示动作空间的大小。

❹ 采取一些随机行动并记录下来。

❺ 显示状态空间的大小。

❻ 重置（初始化）环境并获得状态信息。

❼ 采取随机行动并让环境前进到下一个状态。

下一步是根据随机动作玩游戏以生成足够大的数据集。然而，为了提高数据集的质量，我们仅收集总奖励为 110 及以上的游戏产生的数据。为此，我们打了几千场比赛以收集足够多的数据来训练神经网络。

```
In [9]: %%time
        data = pd.DataFrame()
        state = env.reset()
        length = []
        for run in range(25000):
            done = False
            prev_state = env.reset()
            treward = 1
            results = []
            while not done:
                action = env.action_space.sample()
                state, reward, done, _ = env.step(action)
                results.append({'s1': prev_state[0], 's2': prev_state[1],
                                's3': prev_state[2], 's4': prev_state[3],
                                'a': action, 'r': reward})
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            treward += reward if not done else 0
            prev_state = state
        if treward >= 110:  ❶
            data = data.append(pd.DataFrame(results))  ❷
            length.append(treward)  ❸
    CPU times: user 9.84 s, sys: 48.7 ms, total: 9.89 s
    Wall time: 9.89 s

In [10]: np.array(length).mean()  ❹
Out[10]: 119.75

In [11]: data.info()  ❺
         <class 'pandas.core.frame.DataFrame'>
         Int64Index: 479 entries, 0 to 143
         Data columns (total 6 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   s1      479 non-null    float64
          1   s2      479 non-null    float64
          2   s3      479 non-null    float64
          3   s4      479 non-null    float64
          4   a       479 non-null    int64
          5   r       479 non-null    float64
         dtypes: float64(5), int64(1)
         memory usage: 26.2 KB

In [12]: data.tail()  ❺
Out[12]:           s1        s2        s3        s4 a   r
         139  0.639509  0.992699 -0.112029 -1.548863 0 1.0
         140  0.659363  0.799086 -0.143006 -1.293131 0 1.0
         141  0.675345  0.606042 -0.168869 -1.048421 0 1.0
         142  0.687466  0.413513 -0.189837 -0.813148 1 1.0
         143  0.695736  0.610658 -0.206100 -1.159030 0 1.0
```

❶ 仅当随机智能体的总奖励至少为 100 时……

❷ ……收集数据……

❸ ……并记录总奖励。

❹ 数据集中包含的所有随机游戏的总奖励平均值。

❺ 查看 `DataFrame` 对象中收集的数据。

准备好数据后，可以按如下方式训练神经网络：建立神经网络进行分类，将状态数据作为特征，将采取的操作作为标签数据。鉴于数据集仅包含在给定状态下成功的动作，神经网络会了解在给定状态（特征）下要采取的动作（标签）。

```
In [13]: from pylab import plt
         plt.style.use('seaborn')
         %matplotlib inline

In [14]: import tensorflow as tf
         tf.random.set_seed(100)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [15]: from keras.layers import Dense
         from keras.models import Sequential
         Using TensorFlow backend.

In [16]: model = Sequential()  ❶
         model.add(Dense(64, activation='relu',
                         input_dim=env.observation_space.shape[0]))  ❶
         model.add(Dense(1, activation='sigmoid'))  ❶
         model.compile(loss='binary_crossentropy',
                       optimizer='adam',
                       metrics=['acc'])  ❶
In [17]: %%time
         model.fit(data[['s1', 's2', 's3', 's4']], data['a'],
                   epochs=25, verbose=False, validation_split=0.2)  ❷
         CPU times: user 1.02 s, sys: 166 ms, total: 1.18 s
         Wall time: 797 ms

Out[17]: <keras.callbacks.callbacks.History at 0x7ffa53685190>

In [18]: res = pd.DataFrame(model.history.history)  ❸
         res.tail(3)  ❸
Out[18]:     val_loss  val_acc      loss       acc
         22  0.660300  0.59375  0.646965  0.626632
         23  0.660828  0.59375  0.646794  0.621410
         24  0.659114  0.59375  0.645908  0.626632
```

❶ 使用只有一个隐藏层的神经网络。

❷ 使用先前收集的数据训练该模型。

❸ 显示了最后几个步骤的每个训练步骤的指标。

训练过的神经网络或人工智能体可以根据其学习到的任何状态所对应的最佳动作来玩 `CartPole` 游戏。人工智能体在 100 场比赛中每场获得 200 的最大总奖励。这是基于相对较小的数据集与相对简单的神经网络的结合。

```
In [20]: def epoch():
             done = False
             state = env.reset()
             treward = 1
             while not done:
                 action = np.where(model.predict(np.atleast_2d(state))[0][0] > \
                          0.5, 1, 0)  ❶
                 state, reward, done, _ = env.step(action)  ❷
                 treward += reward if not done else 0
             return treward

In [21]: res = np.array([epoch() for _ in range(100)])
         res  ❸
Out[21]: array([200., 200., 200., 200., 200., 200., 200., 200., 200., 200., 200.,
                200., 200., 200., 200., 200., 200., 200., 200., 200., 200., 200.,
                200., 200., 200., 200., 200., 200., 200., 200., 200., 200., 200.,
                200., 200., 200., 200., 200., 200., 200., 200., 200., 200., 200.,
                200., 200., 200., 200., 200., 200., 200., 200., 200., 200., 200.,
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            200., 200., 200., 200., 200., 200., 200., 200., 200., 200., 200.,
            200., 200., 200., 200., 200., 200., 200., 200., 200., 200., 200.,
            200., 200., 200., 200., 200., 200., 200., 200., 200., 200., 200.,
            200., 200., 200., 200., 200., 200., 200., 200., 200., 200., 200.,
            200.])

In [22]: res.mean()  ❹
Out[22]: 200.0
```

❶ 选择给定状态以及所对应的动作对模型进行训练。

❷ 根据学习到的动作将环境向前推进一步。

❸ 打多场比赛并记录每场比赛的总奖励。

❹ 计算所有游戏的总奖励平均值。

Arcade 学习环境的工作方式与 OpenAI Gym 类似。它允许人们以编程的方式与模拟的雅达利 2600 游戏进行交互、采取行动、收集所采取行动的结果等。如果状态空间大得多，那么学习玩《打砖块》的任务当然更复杂。但是，基本方法与此处采用的方法类似，当然还有一些算法改进。

2.1.2　围棋（Go）

作为棋盘游戏，围棋已有 2000 多年的历史。长期以来，它一直被认为是美与艺术的创造，因为其规则很简单，但对弈非常复杂，并且曾认为在未来几十年中会经受住人工智能的挑战。与许多武术类运动的晋级系统一致，围棋选手的实力以**段位**来衡量，例如，多次获得围棋世界冠军的李世石拥有专业九段的头衔。在 2014 年，Nick Bostrom 做了如下假设：

> 近年来，人工智能的围棋段位一直以约每年一段的速度精进。如果继续以这种速度提高，那么人工智能可能会在十年内击败人类世界冠军。

DeepMind 的一个团队使用其 AlphaGo 算法在围棋领域取得了突破性进展（参见 DeepMind 网站中的 AlphaGo 页面），Silver 等人在 2016 年的研究中将情况描述如下：

> 由于其巨大的搜索空间以及评估棋盘局势和落子的难度，围棋一直被视为人工智能经典游戏中最具挑战性的游戏。

DeepMind 团队成员将神经网络与蒙特卡罗树搜索算法相结合，并在论文中对此算法进行了简要概述。回顾他们从 2015 年开始的早期成功尝试，该团队在介绍中指出：

> 我们的程序 AlphaGo 在对抗其他围棋程序时取得了 99.8% 的胜率，以 5–0 击败了人类欧洲围棋冠军樊麾。这是计算机程序首次在全尺寸围棋棋盘上击败人类职业棋手，而在此之前，大家认为至少需要十年才能实现这一壮举。

值得注意的是，在达到这一里程碑一年之前，资深人工智能研究人员 Nick Bostrom 预测可能需要另外十年才能达到这一水平。然而，许多观察家表示，当时的欧洲围棋冠军樊麾并不能真正被视为一个基准，因为世界围棋精英的水平要比其高得多。DeepMind 团队接受

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

了挑战，并于 2016 年 3 月组织了一场五局两胜制的比赛，对抗当时的 18 届世界围棋冠军李世石——这无疑是人类精英级围棋比赛的一个合适的基准。为此，DeepMind 团队进一步将 AlphaGo Fan 版本迭代为了 AlphaGo Lee 版本。

该赛事以及 AlphaGo Lee 的故事引起了全世界的关注，并被详细报道。DeepMind 在其网页上写道：

> AlphaGo 于 2016 年 3 月在韩国首尔以 4–1 获胜，全球有超过 2 亿人观看。这一里程碑式的成就超前了十年。该游戏为 AlphaGo 赢得了专业九段头衔，这是围棋最高段位的认证，也是计算机围棋选手第一次获得该奖项。

在此之前，AlphaGo 是使用基于数百万人类专家游戏的训练数据集以及其他资源进行监督学习的。DeepMind 团队的下一个迭代版本 AlphaGo Zero，完全跳过了这种方法，仅依靠强化学习和自我对弈，将不同代且训练有素的基于神经网络的人工智能体放在一起相互竞争。Silver 等人在其 2017 年发表的文章中介绍了 AlphaGo Zero 的详细信息。在摘要中，研究人员总结道：

> AlphaGo 成了自己的老师：训练神经网络来预测自己的走法选择以及游戏的获胜者。该神经网络提高了树搜索的强度，从而能在下一次迭代中产生更高质量的走法选择和更强的自我对弈。从一无所知开始，新程序 AlphaGo Zero 取得了超人的表现，以 100–0 战胜了之前发布的击败了李世石的 AlphaGo。

值得注意的是，AlphaGo 训练的神经网络与上一节的 `CartPole` 示例（基于自我对弈）中的神经网络没有太大不同。该神经网络可以破解像围棋那样复杂的游戏，其可能的棋盘布局超过了宇宙中的原子数。同样值得注意的是，该人工智能完全没有依赖人类玩家几个世纪以来积累的围棋智慧。

DeepMind 团队并没有就此止步。AlphaZero 旨在成为一个通用的游戏人工智能体，能够学习不同的复杂棋盘游戏，比如围棋、国际象棋和将棋（日本象棋）。关于 AlphaZero，DeepMind 团队成员 Silver 在其 2017 年发表的文章中总结道：

> 本文将这种方法推广到了单个 AlphaZero 算法中，该算法可以在许多具有挑战性的领域实现超人的表现。从随机玩法开始，在除了游戏规则外没有任何领域知识的情况下，24 小时内 AlphaZero 在国际象棋、将棋以及围棋游戏中达到了超越人类的水平，并在每一个游戏领域都击败了世界冠军程序。

同样，DeepMind 在 2017 年达到了一个非凡的里程碑：一个玩游戏的人工智能体，经过不到 24 小时的自我游戏和训练，在 3 个经过数百年深入研究的棋盘游戏中达到了高于人类专家的水平。

2.1.3　国际象棋（Chess）

毫无疑问，国际象棋是世界上最受欢迎的棋盘游戏之一。国际象棋计算机程序在计算机早期时代就已经存在，甚至可以追溯到家用计算机的出现。例如，1983 年，ZX-81 Spectrum 家用计算机系统中引入了一个几乎完整的称为 ZX Chess 的国际象棋引擎，其只包含大约 672 字节的机器代码。虽然这个实现并不完整，缺乏像“王车易位”这样的规则，但它在

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

当时是一个伟大的成就，对今天的计算机象棋爱好者来说仍然很有吸引力。ZX Chess 作为最小国际象棋程序的记录保持了 32 年，仅在 2015 年被 BootChess 打破，而 BootChess 只有 487 字节。该程序可以玩游戏排列可能比宇宙中的原子还要多的棋盘游戏。能用如此小的代码库编写计算机程序的程序员几乎可以被认为是软件工程天才。

虽然从纯数字的角度看不像围棋那么复杂，但国际象棋可以被认为是最具挑战性的棋盘游戏之一，因为玩家需要几十年才能达到大师级别。

在 20 世纪 80 年代中期，即使在比基本的家用计算机 ZX-81 Spectrum 更好的硬件上，专家级计算机国际象棋程序仍然遥不可及。难怪当时的国际象棋高手在与计算机对战时自信满满。例如，Garry Kasparov（2017）回忆了在 1985 年他是如何同时进行了 32 场比赛的：

> 1985 年 6 月 6 日在汉堡度过了愉快的一天……我的 32 个对手，每一个都是一台计算机……毫无意外的……我取得了 32–0 的完美比分。

直到计算机国际象棋开发人员和国际商业机器公司（IBM）的硬件专家花了 12 年的时间制造了一台名为“深蓝”的计算机，机器才能够击败当时的人类世界国际象棋冠军 Garry Kasparov。Garry Kasparov 在其历史性地输给“深蓝”这一事件 20 年后出版的书中写道：

> 12 年后，我在纽约市为我的国际象棋生涯而战。我面对的仅有一台机器，一台价值 1000 万美元的 IBM 超级计算机，绰号“深蓝”。

Kasparov 与“深蓝”一共打了 6 场比赛，最终计算机以 3.5 分战胜 Kasparov 的 2.5 分（每盘游戏获胜者得 1 分，平局时每位玩家得 0.5 分）。虽然“深蓝”输掉了第一场比赛，但它赢了剩下 5 场比赛中的 2 场，另外 3 场比赛以双方协议的平局告终。有人指出，“深蓝”不应该被视为人工智能的一种形式，因为它主要依赖于庞大的硬件集群。这个由 IBM 专门为此活动设计的具有 30 个节点和 480 个专用于国际象棋芯片的硬件集群每秒可以分析大约 2 亿个棋局。从这个意义上说，“深蓝”主要依靠蛮力技术，而不是神经网络等现代人工智能算法。

自 1997 年以来，硬件和软件都取得了巨大进步。Kasparov 在其书中提到现代智能手机上的国际象棋应用时，如此说道：

> 再向前快进 20 年到今天，即 2017 年，你可以为你的手机下载任意数量的免费国际象棋应用，这些应用可以与任何人类大师相媲美。

击败人类大师的硬件价格已从 1000 万美元降至约 100 美元。然而，普通计算机和智能手机的国际象棋应用仍然依赖于数十年计算机国际象棋的智慧。它们体现了大量人为设计的游戏规则和策略，依赖大型数据库进行开局，然后受益于现代设备增加的计算能力和内存，以对数百万个国际象棋棋局进行基于蛮力计算的评估。

这就是 AlphaZero 的用武之地。AlphaZero 掌握国际象棋游戏的方法完全基于强化学习，不同版本的人工智能体相互对弈。DeepMind 团队将计算机国际象棋的传统方法与 AlphaZero 进行了如下对比（参见 AlphaZero 研究论文“AlphaZero: Shedding new light on chess, shogi, and Go”）：

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

> 传统的国际象棋引擎，包括世界计算机国际象棋冠军引擎 Stockfish 和 IBM 的开创性的“深蓝”，依赖于由强大的人类玩家手工制作的数千条规则和启发式算法，这些规则和启发式算法试图解释游戏中的每一种可能性……AlphaZero 采取了完全不同的方法，其利用深度神经网络和通用算法来代替这些手工制作的规则，而这些通用算法除了基本规则之外对游戏一无所知。

通过采用这种完全从零开始的方法，AlphaZero 在经过几小时的基于自我对弈的训练后表现非常出色，完全不逊于领先的传统国际象棋计算机程序。AlphaZero 只需要 9 小时或更短的训练就可以掌握国际象棋，其水平超过了每个人类棋手和所有其他计算机国际象棋程序，包括曾经统治计算机国际象棋的 Stockfish 引擎。AlphaZero 在 2016 年举办的 1000 场比赛测试中击败了 Stockfish，赢了 155 场（主要是在玩白棋时），仅输了 6 场，其余的则平局。

IBM 的“深蓝”每秒能够分析 2 亿个棋局，现代国际象棋引擎（如 Stockfish）可以在商用多核硬件上每秒分析大约 6000 万个棋局，而 AlphaZero 每秒只能分析大约 6 万个棋局。尽管 AlphaZero 每秒分析的棋局是 Stockfish 的 1/1000，但它仍然能够击败 Stockfish。人们倾向于认为 AlphaZero 确实能显示出某种形式的智能，而这种智能是纯粹的蛮力无法弥补的。考虑到人类大师每秒可以根据经验、模式和直觉分析数百个棋局，AlphaZero 可能处于专家级人类国际象棋棋手和基于蛮力方法的传统国际象棋引擎之间的最佳位置，并辅以手工规则和积累的国际象棋知识。人们可以推测，AlphaZero 获得了与人类模式识别、远见和直觉类似的东西，并且由于使用了为此目的而适配的硬件，因此具有更快的计算速度。

2.2 硬件的重要性

在过去十年中，人工智能研究人员和从业者在人工智能算法方面取得了巨大进步。如上一节所述，强化学习通常会与神经网络相结合用于行动策略表示，这在许多不同领域已被证明是有用且优越的。

然而，如果没有硬件方面的进步，最近的人工智能成就是不可能达成的。同样，DeepMind 的故事及其通过强化学习掌握围棋游戏的案例为我们提供了一些有价值的见解。表 2-1 是 2015 年以后 AlphaGo 主要版本的硬件使用和功耗概览。可以看到，AlphaGo 不仅实力稳步提升，硬件要求和相关功耗也大幅下降。[5]

表2-1：DeepMind用于AlphaGo的硬件

版本	年	Elo评级	硬件	功耗 [TDP]
AlphaGo Fan	2015	大于 3000	176 GPU	大于 40 000
AlphaGo Lee	2016	大于 3500	48 TPU	10 000 以上
AlphaGo Master	2016	大于 4500	4 TPU	小于 2000
AlphaGo Zero	2017	大于 5000	4 TPU	小于 2000

注 5： 在表 2-1 中，GPU 代表图形处理单元。TPU 代表张量处理单元，它是一种专用计算机芯片，用于更有效地处理所谓的张量和对张量的操作。张量是神经网络和深度学习的基本构建块，本书后面内容和 Chollet（2017）的第 2 章对其进行了更详细的介绍。TDP 代表热设计功率。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

人工智能的第一个主要硬件推动来自 GPU，尽管 GPU 最初被用于为计算机游戏生成快速高分辨率图形，但现代 GPU 也可用于许多其他目的。这些额外目的之一涉及线性代数（比如，以矩阵乘法的形式），这是一门对一般人工智能，尤其是神经网络至关重要的数学学科。

截至 2020 年年中，市场上速度最快的一款消费级 CPU 是最新迭代的 Intel i9 处理器（具有 8 个内核和最多 16 个并行线程）。[6] 根据现有的基准测试任务，它达到了大约 1 TFLOPS（每秒 1 万亿次浮点运算）或略高的速度。

与此同时，市场上速度最快的一款消费级 GPU 是 Nvidia GTX 2080 Ti。它有 4352 个 CUDA 内核，也即 Nvidia 的 GPU 内核，允许高度并行（比如，在线性代数运算的任务中）的计算。该 GPU 的速度高达 15 TFLOPS，比英特尔最快的消费级 CPU 快约 15 倍。很长一段时间以来，GPU 一直比 CPU 快。然而，通常一个主要的限制因素是 GPU 拥有相对较小且专用的内存。较新的 GPU 型号（如 GTX 2080 Ti）显著缓解了这种情况，它具有高达 11 GB 的快速 GDDR6 内存和可在 GPU 之间传输数据的高总线速度。

2020 年年中，这种 GPU 的零售价约为 1400 美元，比 10 年前同等强大的硬件便宜了几个数量级。这种发展使得人工智能研究对预算相对较少的个人学术研究人员来说（与 DeepMind 等公司相比）更容易负担。

另一个硬件趋势正在推动人工智能方法和算法的进一步发展和采用：基于云计算的 GPU 和 TPU。Scaleway 等云供应商提供了可按小时租用且具有强大 GPU 的云实例（参见 Scaleway GPU 实例）。谷歌等其他公司开发了 TPU，这是专门用于人工智能的芯片，与 GPU 类似，可以提高线性代数运算的效率（参见谷歌 TPU）。

总而言之，从人工智能的角度来看，硬件在过去几年中得到了极大的改进。总结起来，主要有 3 个方面。

性能

GPU 和 TPU 为硬件提供了高度并行的架构，非常适合人工智能算法和神经网络。

费用

按每 TFLOPS 计算能力的成本已显著下降，使得每 TFLOPS 以更少的预算或者在相同预算下使用更多算力成为可能。

功耗

功耗也下降了。与人工智能相关的相同任务需要的能源更少了，执行速度也更快了。

2.3 智能的形式

AlphaGo Zero 智能吗？如果没有对**智能**的具体定义，就很难说清楚。人工智能研究员 Max Tegmark（2017）将智能简明地定义为“完成复杂目标的能力”。

这个定义足够广泛，可以包含更具体的定义。鉴于该定义，AlphaGo Zero 是智能的，因为它能够完成一个复杂的目标，即在围棋或国际象棋比赛中打败人类玩家或其他人工智能体。当然，人类和一般的动物也因此被认为是智能的。

注 6：CPU 代表中央处理器，它是所有标准台式机或笔记本计算机中的通用处理器。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

就本书而言，以下更具体的定义似乎足够恰当且准确。

狭义人工智能（ANI）
　　这指定了仅在某些特定领域内超过人类专家级能力和技能的人工智能体。AlphaZero 可以被认为是围棋、国际象棋和将棋领域的 ANI。一种算法型股票交易人工智能体可以将投资资本持续实现每年 100% 的净收益率，它也可以被视为 ANI。

通用人工智能（AGI）
　　这指定了一种人工智能体，它可以在任何领域（比如国际象棋、数学、文本写作或金融）都达到人类水平的智能。

超级智能（SI）
　　这指定了在任何方面都超过人类水平智能的智能或人工智能体。

ANI 有能力在某些特定领域内达成复杂目标，并表现出比任何人类都更高的能力。在实现各种领域的复杂目标方面，AGI 与任何人类个体一样出色。最后，在大部分可以想象的领域中，在实现复杂目标方面，超级智能比任何人类个体甚至人类集体都要好得多。

前面对超级智能的定义，与 Bostrom 在其名为 *Superintelligence*（2014）的书中给出的定义一致：

> 我们可以暂时将超级智能定义为**在所有感兴趣的领域都大大超过人类认知能力的任何智能**。

如前所述，技术奇点是超级智能存在的时间点。然而，哪些途径可能会导致超级智能呢？这是下一节的主题。

2.4　通往超级智能的途径

多年来，研究人员和从业者都在争论是否有可能创造一种超级智能。技术奇点的实现时间从预计的几年到几十年，到几个世纪，再到永远。无论人们是否相信超级智能的可行性，对实现它的潜在途径的讨论都是富有成果的。

下面引用 Bostrom（2014）的第 2 章中一段较长的论述，其中列出了一些可能适用于任何潜在超级智能途径的一般性考虑：

> 然而，我们可以辨别出所需要的那种系统的一些一般特征。现在似乎很清楚，学习能力将是通用智能系统设计的一个核心组成部分，而不是稍后作为扩展或事后考虑的东西。这同样适用于有效处理不确定性和概率信息的能力。另外，从感知数据和内部状态中提取有用的概念的能力，以及将获得的概念灵活组合并用于逻辑和直觉推理的能力，也属于面向通用智能的现代人工智能系统设计的核心特征。

这些通用特征让人联想到了 AlphaZero 的方法和功能，尽管**直觉**之类的术语对人工智能体来说需要更明确的定义。但是如何实际地实现这些通用特性呢？Bostrom（2014）的第 2 章讨论了 5 种可能的途径，接下来我们将进行探讨。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

2.4.1 网络和组织

通往超级智能的第一条途径是通过可能涉及大量人类的网络和组织，以一种方式进行协调，这种方式会使他们的个人智能被放大并同步工作。由具有不同技能的人组成的团队是此类网络或组织的一个简单示例。

这条途径似乎有自然限制，因为一个人的能力是相对固定的。进化理论表明，人类难以在超过 150 人的网络和组织内进行协调，因此大公司通常会以比这小得多的团队、部门或小组来运行。

另外，即使有数百万个计算节点，计算机和机器的网络（如互联网）也倾向于无缝地系统工作。如今，此类网络至少能够组织人类的知识和其他数据（声音、图片、视频等）。当然，人工智能算法已经帮助人类浏览了所有这些知识和数据。然而，值得怀疑的是，超级智能是否会“自发地”在互联网上出现，至少目前来看仍为时尚早。

2.4.2 生物增强

如今，人们在提高个人认知和身体表现方面付出了很多努力。从自然方法（比如更好的训练和学习方法）到涉及物理的方法（比如补充剂）再到涉及特殊工具的方法，今天人类比以往任何时候都更努力地系统且科学地提高个人认知和身体表现。Harari（2015）将这项努力描述为智人寻求创造一个新的、更好版本的自己，即**智神**（homo deus）。

然而，这种方法再次面临人类硬件基本固定的障碍。它已经进化了数十万年，并且在可预见的未来可能会继续这样做。但这只会以相当缓慢的速度发生，而且需要很多代的持续进化。它也只会在很小的程度上发生，因为如今人类的自然选择作用已减弱，而自然选择赋予进化以改进的力量。Domingos（2015）的第 5 章讨论了进化过程中进步的核心方面。

在这种情况下，可以参考 Tegmark（2017）的第 1 章中概述的**生命版本**。

- **生命 1.0**（生物）：硬件（生物体）和软件（基因）基本固定的生命形式，两者都是通过进化同时缓慢进化的。这方面的例子是细菌或昆虫。
- **生命 2.0**（文化）：具有基本固定且缓慢进化的硬件但主要是设计和学习的软件（基因加上语言、知识、技能等）的生命形式。人类就是这样一个例子。
- **生命 3.0**（技术）：生命形式具有设计和可调整的硬件以及完全学习和进化的软件。一个例子是用计算机硬件、软件和人工智能算法创建的超级智能。

随着生命 3.0（技术）在超级智能机器中出现，可用硬件的局限性或多或少会完全消失。因此，目前来看，网络或生物增强之外的途径可能更有希望通往超级智能。

2.4.3 脑机混合

以人类使用各种硬件工具和软件工具为标志，在各个领域提高人类绩效的混合方法在我们的生活中无处不在。人类自出现以来就使用工具，今天，数十亿人携带着有谷歌地图的智能手机，即使在从未去过的地区和城市，他们也能轻松导航。我们的祖先则没有这些奢侈品，因此他们需要根据在天空中看到的物体，或者使用不那么复杂的工具（如指南针）来获得导航技能。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

以国际象棋为例，即使计算机（如“深蓝”）被证明更胜一筹，人类也不会停止下棋。相反，计算机国际象棋程序性能的改进，使其成了每位特级大师系统地改进自身游戏的不可或缺的工具。人类特级大师和快速计算的国际象棋引擎组成了一个人机团队，在其他条件相同的情况下，他们的表现比单独一个人要好。甚至还有国际象棋锦标赛，在比赛中，人类在利用计算机做出下一步行动时会相互对弈。

同样，我们可以想象通过适当的接口将人脑直接连接到机器，这样大脑就可以与机器通信、交换数据并启动某些计算、分析或学习任务。这些听起来像科幻小说的东西，现在是一个活跃的研究领域。以 Elon Musk 为例，他创办了一家名为 Neuralink 的初创公司，该公司所专注的领域就是我们通常所说的**神经技术**。

总而言之，脑机混合似乎是实际可行的，并且有可能显著超越人类智能。然而，它是否会带来超级智能还尚不明确。

2.4.4　全脑模拟

另一种通往超级智能的建议途径是先完全模拟人脑，然后再对其进行改进。这个想法是通过现代脑扫描技术以及生物和医学分析方法绘制整个人类大脑图，从而通过软件以神经元、突触等形式精确复制其结构。该软件可在适当的硬件上运行。Domingos（2015）的第 4 章阐述了关于人类大脑的背景信息以及它在学习方面的特征。Kurzweil（2012）通篇对这个主题进行了阐述，给出了详细的背景信息并勾勒出了实现全脑模拟（WBE，有时也称为**上传**）的方法。[7]

退一步讲，（人工）神经网络正是 WBE 试图实现的目标。神经网络，顾名思义，会受大脑的启发，并且因为它已经被证明在许多不同领域非常有用且成功，人们可能倾向于得出结论，WBE 确实可以被认为是通往超级智能的可行途径。然而，迄今为止，绘制完整人脑的必要技术还不完善。即使能成功绘制完整人脑，也不清楚人脑的模拟版本是否能够做与人脑相同的事情。

但是，如果 WBE 成功了，那么人脑软件就可以在比人体更强大、速度更快的硬件上运行，从而有可能实现超级智能。该软件还可以轻松复制，并且可以以协作的方式将大量模拟大脑组合在一起，这也可能导致超级智能。人脑软件也可以以人类由于生物学限制而无法做到的方式得到增强。

2.4.5　人工智能

最后，值得一提的是，本书中涉及的人工智能本身可能会导致超级智能：像神经网络这一类算法可以在标准或专用硬件上运行，并根据可用或自创数据进行训练。如果超级智能是完全可以实现的，那么大多数研究人员和从业者就有很多理由认为人工智能这条途径是最可行的。

注 7：2019 年 1 月，由 Keanu Reeves 主演的美国科幻惊悚片《复制品》在美国上映，这部电影被证明是商业上的失败，其主题是人类大脑的映射以及将映射转移到机器甚至其他通过克隆和复制生长的人体。这部电影触及了数百年来人类超越人体物理基础并至少在思想和灵魂方面成为不朽的愿望。即使 WBE 可能不会导致超级智能，但它在理论上是实现这种永生的基础。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第一个主要原因是，从历史上看，人类在工程方面的成功往往是因为忽略了大自然和进化中为解决某个问题而得到的类似方案。拿飞机的设计来说，其中利用了对物理学、空气动力学、热力学等的现代理解，而不是试图模仿鸟类或昆虫的飞行方式。再以计算器为例，当工程师建造第一台计算器时，并没有分析人脑是如何进行计算的，甚至也没有尝试复制生物学方法，他们更依赖于在技术硬件上实现的数学算法。在这两种情况下，更重要的方面是功能或能力本身（飞行、计算），其越有效越好，没有必要模仿大自然。

第二个主要原因是，人工智能的成功案例的数量似乎在不断增加。例如，将神经网络应用于几年前人工智能似乎并不占优势的领域，已被证明是在许多领域实现 ANI 的有效途径。一个希望可以进一步推动人工智能泛化能力的例子是，AlphaGo 正逐渐转变为 AlphaZero，在短时间内掌握了多款棋盘游戏。

第三个主要原因是，可能只有在许多 ANI（甚至一些 AGI）被观察到之后，超级智能才会出现（“奇点”）。由于人工智能在特定专业和领域的力量毋庸置疑，因此研究人员和企业等将继续专注于改进人工智能算法和硬件。例如，大型对冲基金将努力通过人工智能方法和智能体来产生 alpha 收益（一种与市场基准相比，衡量基金业绩表现的指标），许多基金拥有致力于此类工作的大型专门团队。跨不同行业的这些全球性努力可能会共同产生超级智能所需的进步。

人工智能

在通往超级智能的所有可能途径中，人工智能似乎是最有前途的一种。在经历了多个人工智能寒冬之后，我们迎来了另一个人工智能的春天，尤其以最近在强化学习和神经网络领域取得的成功为代表。现在许多人甚至相信超级智能可能不会像我们几年前想象的那么遥远，该领域的进展比专家不久前预测的要快得多。

2.5 智能爆炸

前面提到的 Vinge（1993）的引述不仅描绘了技术奇点之后人类面临的危险情景，而且还预言了危险情景将在**不久之后**出现，为什么这么快？

如果存在一个超级智能，那么工程师或超级智能本身就可以创建另一个超级智能，甚至可能是更好的超级智能，因为与最初的超级智能创造者相比，现在的超级智能将拥有卓越的工程知识和技能。超级智能的复制不会受到经过数百万年进化的生物过程持续时间的限制，只会受到新硬件的技术组装过程的限制。超级智能可以以显著的方式对其自身进行改进。软件可以快速且轻松地复制到新硬件中。资源可能会限制复制，但超级智能会想出更好的甚至新的方法来挖掘和生产所需的资源。

这些以及类似的论点都支持这样一种观点，即一旦达到技术奇点，智能就会爆炸。这可能与（某些科学家提出的关于宇宙起源的）创世大爆炸类似，创世大爆炸开始于（物理）奇点，而已知的宇宙从爆炸中诞生。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

关于特定领域和 ANI，类似的论点可能适用。假设一个算法交易人工智能体会比市场上的其他交易者和对冲基金呈现出更成功且更持久的明智表现，这样的人工智能体将通过交易收益和吸引外部资金来积累更多的资金。反过来，这将增加可用预算来改进硬件、算法、学习方法等，例如，通过支付高于市场的工资和奖励来吸引人工智能金融领域最聪明的人。

2.6 目标和控制

在正常的人工智能环境中，例如，当人工智能体应该掌握图 2-1 中描述的简单的 `CartPole` 游戏或更复杂的游戏（如国际象棋或围棋）时，目标通常是明确定义的："至少获得 200 分的奖励""通过将死对手赢得国际象棋比赛"等，但超级智能的目标又是什么呢？

2.6.1 超级智能和目标

对于拥有超人能力的超级智能，目标可能不像前面的例子那么简单和稳定。首先，超级智能可能会为自己制订一个自认为比最初制订和规划的目标更合适的新目标。毕竟，它有能力以与工程团队相同的方式做到这一点。一般来说，它可以在任何方面对自己重新编程。许多科幻小说和电影让我们相信，主要目标的这种变化会对人类不利，这也是 Vinge（1993）所假设的。

即使假设超级智能的主要目标可以以不可更改的方式进行编程和嵌入，或者超级智能可能会坚持其原始目标，问题仍然会出现。Bostrom（2014）的第 7 章认为，每个超级智能都有独立于主要目标的 5 个工具性子目标。

自我保护

超级智能需要足够长的生存时间才能实现其主要目标，为此，它可能会采取不同的措施，而为了确保其生存，其中一些措施可能会对人类有害。

目标内容完整性

这指的是一种想法，即超级智能将尝试保持其当前主要目标，因为这会增加其未来的自己实现这一目标的可能性，因此，现在和未来的主要目标很可能是相同的。考虑一个以赢得国际象棋比赛为目标的国际象棋人工智能体，它可能会为了自己的皇后不被吃掉而不惜一切代价改变其目标。但这可能会阻碍它最终赢得比赛，因此这种目标的改变和赢得比赛的目标会不一致。

认知能力增强

无论超级智能的主要目标是什么，认知能力增强通常都被证明是有益的。因此，如果这有助于实现其主要目标，那么它可能会努力以尽可能快的速度提高其认知能力。因此，认知能力增强是一个主要的工具性目标。

技术完善

技术完善也是一个工具性目标。在生命 3.0 的意义上，超级智能既不会局限于其当前硬件，也不会局限于其软件的状态。它会努力存在于可能设计和生产的更好的硬件上，并会利用已编制的改进软件。这通常会服务于其主要目标，并且可能会更快实现。例如，在金融行业，高频交易（HFT）是一个以争夺技术优势为特征的领域。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

资源获取

对大多数主要目标来说，更多的资源通常会提升实现目标的概率和实现速度，当目标中隐含竞争情况时尤其如此。考虑一个人工智能体，其目标是尽快地挖掘尽可能多的金融工具。人工智能体可用的硬件、能源等形式的资源越多，就越能更好地实现其目标。在这种情况下，它甚至可能会出现从加密货币市场中的其他人那里获取（窃取）资源的非法做法。

从表面上看，工具性目标似乎不会构成威胁，毕竟，它们是为了确保人工智能体主要目标的实现。然而，正如 Bostrom（2014）被广泛引用的例子所示，问题很容易出现。例如，Bostrom 认为，以最大限度生产回形针为目标的超级智能可能会对人类构成严重威胁。要了解这一点，请在此类人工智能体的例子中考虑前面的工具性目标。

第一，它会想尽一切办法来保护自己，即使是使用武器来对付自己的创造者。第二，即使它自己的认知推理能力表明其主要目标并不是真正明智的，但它可能会随着时间的推移坚持下去以最大化实现此目标的机会。第三，认知增强对于实现其目标肯定是有价值的，因此，它会尝试各种方法，而为了提高其能力，其中许多方法可能会以牺牲和伤害人类为代价。第四，它自身的技术或是生产回形针的技术越好，对其主要目标的实现就越有利。因此，它会通过购买或窃取等方法来获取所有现有技术，并构建有助于实现其目标的新技术。最后，它拥有的资源越多，可以生产的回形针就越多，直到地球资源枯竭时，它可以开发太空探索和开采技术。在极端情况下，这样的超级智能可能会耗尽太阳系、银河系甚至整个宇宙的资源。

工具性目标

假设任何形式的超级智能都具有与其主要目标无关的工具性目标。这可能会导致许多意想不到的后果，比如为了获取更多资源，超级智能会以任何看起来有希望的方式进行永不满足的追求。

该示例说明了关于人工智能体目标的两个要点。首先，可能无法以能够完整而清晰地表现目标制订者意图的方式为人工智能体制订复杂的目标。例如，“保留和保护人类物种”这样的崇高目标可能会为了确保 1/4 的人类具有更高的生存可能性而杀死剩余的 3/4。在对地球和人类的未来进行了数十亿次模拟之后，超级智能觉得这种方法最有可能实现其主要目标。其次，一个看似善意且无害的目标可能会由于工具性目标的存在而导致意想不到的后果。在回形针示例中，目标中的一个问题是“尽可能多”这句话。这里的一种简单解决方法是将数字指定为 100 万，但即使这样也可能只是部分修复，因为像自我保护这样的工具性目标可能会成为主要目标。

2.6.2 超级智能和控制

如果在技术奇点之后**可能**出现糟糕甚至灾难性的后果，那么设计至少可以潜在地控制超级智能的措施至关重要。

第一套措施与主要目标的正确制订和设计有关，上一节在一定程度上讨论了这个方面。Bostrom（2014）的第 9 章在**动机选择方法**的讨论中提供了更多细节。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第二套措施与控制超级智能的能力有关。Bostrom（2014）的第 9 章概述了 4 种基本方法。

封闭

这是一种将正在出现的超级智能与外部世界隔离开来的方法，例如，将人工智能体与互联网断开，使它没有任何感官能力，同时还要排除人际互动。但这种控制人工智能体能力的方法也可能导致大量有趣的目标无法实现。考虑一个应该达到 ANI 级别的算法交易人工智能体，如果不与外部世界（比如股票交易平台）连接，那么人工智能体就没有机会实现其目标。

激励

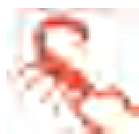
人工智能体可能会被赋予最大化其奖励的功能，这种（电子）奖励被有意设计为奖励被期望的行为并惩罚不被期望的行为。虽然这种间接方法给目标设计提供了更多的自由度，但它在很大程度上也受到与直接制订目标类似的问题的困扰。

阻碍

这种方法是指在硬件、计算速度或内存等方面故意限制人工智能体的能力。然而，这是一种刻意为之的方法。过多的阻碍会使超级智能永远不出现，而过少的阻碍和随之而来的智力爆炸又将使这种方法变得毫无作用。

绊线

这是指有助于尽早识别任何可疑或异常行为，以便启动有针对性的对策的措施。然而，这种方法存在类似警报系统向警方发出入室盗窃警报的问题。可能窃贼在 5 分钟前就离开了，但警察需要 10 分钟才能到达犯罪现场，并且即便是监控摄像头也可能无助于找出窃贼。



能力控制

总而言之，当超级智能达到技术奇点水平时，能否对其进行适当且系统的控制就变得令人怀疑了。毕竟，它的超能力至少在原则上可以用来克服任何人为设计的控制机制。

2.7 潜在的结果

除了 Vinge（1993）早期的预言：“超级智能的出现将意味着人类的世界末日”，还有哪些潜在的结果和情景是可以想象的？

越来越多的人工智能研究人员和从业者警告了不受控制的人工智能可能带来的潜在威胁。在超级智能出现之前，人工智能会导致歧视、社会失衡、金融风险等。（在这方面，著名的人工智能评论家是特斯拉、SpaceX 和 Neuralink 等公司的创始人 Elon Musk。）因此，人工智能伦理与治理是研究人员和从业者之间激烈争论的话题。简而言之，可以说这个群体害怕人工智能引发的**反乌托邦**。而以 Ray Kurzweil（2005，2012）为代表的其他群体则强调人工智能可能是通往乌托邦的“唯一”途径。

这种情况下的问题是，即使出现反乌托邦的结果的可能性相对较低，也足以令人担忧。正如上一节所述，鉴于现有技术，可能还无法设计出适当的控制机制。在此背景下，在我撰

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

写本书时，已有 42 个国家签署了第一个关于人工智能发展的国际协议。

正如 Murgia 和 Shrikanth（2019）在《金融时报》上报道的那样：

> 上周发生了一个历史性的事件，42 个国家联合起来支持为我们这个时代最强大的新兴技术之一——人工智能——建立一个全球治理框架。
>
> 该协议由美国、英国和日本等经合组织国家以及非成员国签署。签署时正值各国政府进行清算之际，这些政府最近才开始应对在工业中应用人工智能的道德与实际后果……近年来，谷歌、亚马逊等公司在人工智能方面的快速发展远远超出了该领域的监管，从而暴露出它们在以下几方面所面临的重大挑战：带有偏见的人工智能决策、彻头彻尾的造假和错误信息，以及自动化军事武器的危险。

乌托邦与反乌托邦

即使是基于人工智能进步的乌托邦未来的坚定支持者也必须同意，不能完全排除技术奇点之后出现反乌托邦未来的可能性。由于后果可能是灾难性的，因此反乌托邦的结果必须在关于人工智能和超级智能的更广泛讨论中被考虑进来。

超级智能的数量以及技术奇点之后的情况将会如何？似乎可能出现以下 3 种基本情况。

单极

单一的超级智能会出现并获得非常强大的力量，以至于没有其他超级智能能够生存甚至出现。例如，谷歌主宰着搜索市场，在该领域几乎占据垄断地位。超级智能在出现后不久可能会很快在许多相关领域和行业中达到类似的位置。

多极

多个超级智能几乎同时出现并共存更长时间，例如，对冲基金行业有一些大型参与者，鉴于它们的总市场份额，可以被视为寡头垄断。根据它们之间的分而治之的协议，多个超级智能可以类似地共存，至少在一段时间内是这样。

原子化

在技术奇点之后不久出现了大量的超级智能。从经济学的角度讲，这种情况类似于一个完全竞争的市场。从技术角度讲，国际象棋的演变为这种情况提供了一个类比。虽然 IBM 在 1997 年制造了一台可以主宰计算机和人类国际象棋世界的机器，但今天每部智能手机上的国际象棋应用都胜过人类棋手。截至 2018 年，已经有 30 多亿部智能手机投入使用。在此背景下，值得注意的是，最近智能手机的硬件趋势是在常规 CPU 之外添加专用人工智能芯片，稳步提升这些小型设备的功能。

本节并没用争论技术奇点之后的一个或另一个潜在结果（比如反乌托邦、乌托邦、单极、多极或原子化），而是提供了一个思考超级智能或强大的 ANI 在各自领域潜在影响的基本框架。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

2.8 结论

最近，像 DeepMind 和 AlphaZero 这样的成功案例带来了新的人工智能的春天，人们对超级智能的实现寄予了前所未有的希望。目前，人工智能已经产生了在不同领域远远超过人类专家水平的 ANI。虽然 AGI 和超级智能是否可能实现仍然存在争议，但至少不能排除通过某种途径（如人工智能）确实可以实现。一旦技术奇点出现，也不排除超级智能可能会给人类带来意想不到的、负面的甚至是灾难性的后果。因此，在技术奇点出现之前，对于控制新兴的、更强大的人工智能体，合适的目标和激励设计以及合适的控制机制可能是至关重要的。一旦达到奇点，智能爆炸可能会迅速将超级智能的控制权从其创造者和赞助者手中夺走。

人工智能、机器学习、神经网络、超级智能和技术奇点是对人类生活的各个领域的现在或将来都很重要的话题。目前，许多研究领域、行业和人类生存涉及的领域因人工智能、机器学习和深度学习而发生着根本性的变化。金融和金融行业也是如此，不过，由于人工智能在这两个领域的应用速度相对较慢，因此它的影响可能还没有那么大。但与其他领域一样，人工智能将会改变金融以及金融市场参与者的基本行为方式，正如后面章节所讨论的那样。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第二部分

金融和机器学习

如果有一个行业能真正通过应用人工智能而获益，那就是投资管理。

——Angelo Calvello，2020 年

第二部分由 4 章组成，核心主题涵盖为什么以数据驱动的金融、人工智能和机器学习将对金融理论和实践产生持久影响。

- 第 3 章为重要且流行的金融理论和模型做了基础性介绍，这些理论和模型几十年来一直被认为是金融的基石，其中包括均值 – 方差投资组合（MVP）理论和资本资产定价模型（CAPM）。
- 第 4 章讨论了越来越多的历史和实时金融数据如何将金融从理论驱动型学科转变为数据驱动型学科。
- 第 5 章将机器学习作为一种通用方法进行介绍，从具体的算法中抽象出更通用的概念。
- 第 6 章在通用层面上讨论了数据驱动的金融的出现与人工智能和机器学习相结合如何导致金融范式转变。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 3 章

规范性金融理论

CAPM 是建立在许多非现实假设之上的。例如，一个极端的假设就是假定投资者只关心单期投资组合收益率的均值和方差。

——Eugene Fama 和 Kenneth French，2004 年

相较于基于基本粒子的科学，人类参与的科学对优雅的数学有着更强的抗拒。

——Alon Halevy 等，2009 年

本章回顾了主要的规范性金融理论和模型。简单地说，就本书而言，**规范性理论**是建立在假设（数学的，公理的）基础之上的理论，从一系列相关假设中得出论点和结果等。**实证性理论**则是建立在观察、实验、数据、关系等基础之上，根据可获得的信息和已有的结论得出相应见解以用于现象的描述。就本章介绍的理论和模型起源，Rubinstein（2006）给出了详细的历史解释。

3.1 节介绍了金融模型的核心概念，比如不确定性、风险、交易资产等。3.2 节讨论了不确定决策下的主要经济范式：**预期效用理论**。预期效用理论的现代形式可以追溯到 von Neumann 和 Morgenstern（1944）。3.3 节介绍了 Markowitz（1952）的均值 – 方差投资组合理论。3.4 节对 Sharpe（1964）和 Lintner（1965）的**资本资产定价模型**进行了说明。3.5 节概述了 Ross（1971，1976）的**套利定价理论**（APT）。

本章的目的是以介绍核心的规范性金融理论的形式为本书其余部分奠定基础。这一点很重要，因为数代的经济学家、金融分析师、资产管理者、交易员、银行家、会计师和其他专业人员都接受过这些理论的训练。从这个意义上说，作为一门理论与实践相结合的学科，金融在很大程度上是由这些理论所塑造的。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

3.1 不确定性与风险

金融理论的核心就是当前不确定性和风险条件下的投资、交易和估值，本节会以尽量正式的形式介绍这些主题的核心概念，而重点是从概率论角度介绍支撑数量金融的基本概念。[1]

3.1.1 定义

假设一种经济模式只能观察到两个时间点：今天，记为 $t = 0$；一年后，记为 $t = 1$。本章后面的内容在很大程度上就是基于这样的**静态经济模式**。[2]

在 $t = 0$ 时，没有任何不确定性。在 $t = 1$ 时，经济模式可以是有限状态集合 Ω 中的一个状态 S，$\omega \in \Omega = \{\omega_1, \omega_2, \cdots, \omega_S\}$。$\Omega$ 被称为**状态空间**，它的基数（或者势）为 S，$|\Omega| = S$。

代数 F 在 Ω 中是一个集合族，满足以下条件。

1. $\Omega \in \mathsf{F}$
2. $\mathbb{E} \in \mathsf{F} \Rightarrow \mathbb{E}^c \in \mathsf{F}$
3. $\mathbb{E}_1, \mathbb{E}_2, \cdots, \mathbb{E}_I \in \mathsf{F} \Rightarrow \bigcup_{i=1}^{I} \mathbb{E}_i \in \mathsf{F}$

$\mathbb{E}^c$ 是集合 $\mathbb{E}$ 的补集，幂集 $\wp(\Omega)$ 是最大的代数，同时 $\mathsf{F} = \{\varnothing, \Omega\}$ 是 Ω 中最小的代数。代数就是经济模式中**可观测事件**的一个模型。在此背景下，经济模式的单一状态 $\omega \in \Omega$ 可以解释为一个**原子**事件。

概率就是把一个实数 $0 \leqslant p_\omega \equiv P(\{\omega\}) \leqslant 1$ 赋值给一个状态 $\omega \in \Omega$，或者把一个实数 $0 \leqslant P(\mathbb{E}) \leqslant 1$ 赋值给一个事件 $\mathbb{E} \in \mathsf{F}$。如果已知一个事件的所有状况的概率，则该事件的概率可以加总为 $P(\mathbb{E}) = \sum_{\omega \in \mathbb{E}} p_\omega$。

概率测度 $P : \mathsf{F} \to [0, 1]$ 具有以下特征。

1. $\forall \mathbb{E} \in \mathsf{F} : P(\mathbb{E}) \geqslant 0$
2. $P(\bigcup_{i=1}^{I} \mathbb{E}_i) = \sum_{i=1}^{I} \mathbb{E}_i$，其中 $\mathbb{E}_i \in \mathsf{F}$
3. $P(\Omega) = 1$

将 $\{\Omega, \mathsf{F}, P\}$ 这 3 个要素组合在一起会形成一个**概率空间**。概率空间是现代经济学中对**不确定性**的正式表达形式。如果概率测度 P 是固定的，就可以说这种经济模式是处于**风险**中的。如果经济模式中的所有经济主体都知道概率分布，那么这一经济模式就是**信息对称**的。

给定一个概率空间 $\{\Omega, \mathsf{F}, P\}$，**随机变量**是一个函数 $S : \Omega \to \mathbb{R}_+, \omega \mapsto S(\omega)$，并且该函数是 F - 可测的。这就意味着每一个 $\mathbb{E} \in \{[a, b[: a, b \in \mathbb{R}, a < b\}$ 都满足以下关系：

$$S^{-1}(\mathbb{E}) \equiv \{\omega \in \Omega : S(\omega) \in \mathbb{E}\} \in \mathsf{F}$$

如果 $\mathsf{F} \equiv \wp(\Omega)$，则随机变量的**预期值**会被定义为：

注 1：参见 Jacod 和 Protter（2004）关于概率论的介绍。

注 2：在**动态经济模式**中，不确定性会随着时间的流逝而逐渐消失，也就是说，在今天与一年后的今天之间的每一天流逝都让不确定性减少。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

$$\mathbf{E}^P(S)=\sum_{\omega\in\Omega}P(\omega)\times S(\omega)$$

否则，随机变量的**预期值**会被定义为：

$$\mathbf{E}^P(S)=\sum_{\mathbb{E}\in\mathsf{F}}P(\mathbb{E})\times S(\mathbb{E})$$

一般来说，我们会假设金融经济系统是**完美**的。也就是说，不存在交易成本，可得的资产具有固定的价格，并且有无限的数量，一切事物都以光速发生，各经济主体都拥有完备且对称的信息。

3.1.2　数字模拟例子

假设现在一种简单**静态**经济模式在 $\{\Omega,\mathsf{F},P\}$ **风险**下，满足以下条件。

1. $\Omega\equiv\{u,\ d\}$
2. $\mathsf{F}\equiv\wp(\Omega)$
3. $P\equiv\left\{P(\{u\})=\frac{1}{2},\ P(\{d\})=\frac{1}{2}\right\}$

1. 交易的资产

在这种经济模式中，有两种资产可交易。第一种是风险资产，比如**股票**资产。该资产今天有一个确定的价格 $S_0=10$，明天的支付则是不确定的，但满足以下随机变量形式。

$$S_1=\begin{cases}S_1^u=20,\text{如果 }\omega=u\\ S_1^d=5,\ \text{如果 }\omega=d\end{cases}$$

第二种是无风险资产，比如**债券**资产。该资产今天是确定的价格 $B_0=10$，明天的支付也是确定的，满足以下条件。

$$B_1=\begin{cases}B_1^u=11,\text{如果 }\omega=u\\ B_1^d=11,\text{如果 }\omega=d\end{cases}$$

正式形式上，这一经济模型可以被写成 $\mathsf{M}^2=(\{\Omega,\mathsf{F},P\},\mathbb{A})$，其中 $\mathbb{A}$ 代表今天的可交易资产价格向量 $\boldsymbol{M}_0=(S_0,B_0)^{\mathrm{T}}$，明天的市场支付则用以下形式表示。

$$\boldsymbol{M}_1=\begin{pmatrix}S_1^u & B_1^u\\ S_1^d & B_1^d\end{pmatrix}$$

2. 套利定价理论

例如在一种经济模式中，我们能解决如何以执行价为 $K=14.5$ 的股票获得**欧式看涨期权**的公允价值的问题。无套利的欧式看涨期权 C_0 可以借助股票和债券的组合来复制期权 ϕ 的支付 C_1。这个投资组合复制的价格也必须是欧式看涨期权的价格。否则，（无限）套利利

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

润将成为可能。在 Python 中，这样的复制很容易做到。[3]

```
In [1]: import numpy as np

In [2]: S0 = 10  ❶
        B0 = 10  ❶

In [3]: S1 = np.array((20, 5))  ❷
        B1 = np.array((11, 11))  ❷

In [4]: M0 = np.array((S0, B0))  ❸
        M0  ❸
Out[4]: array([10, 10])

In [5]: M1 = np.array((S1, B1)).T  ❹
        M1  ❹
Out[5]: array([[20, 11],
               [ 5, 11]])

In [6]: K = 14.5  ❺

In [7]: C1 = np.maximum(S1 - K, 0)  ❻
        C1  ❻
Out[7]: array([5.5, 0. ])

In [8]: phi = np.linalg.solve(M1, C1)  ❼
        phi  ❼
Out[8]: array([ 0.36666667, -0.16666667])

In [9]: np.allclose(C1, np.dot(M1, phi))  ❽
Out[9]: True

In [10]: C0 = np.dot(M0, phi)  ❾
         C0  ❾
Out[10]: 2.0
```

❶ 股票和债券今日价格。

❷ 股票和债券明日不确定的回报。

❸ 市场价格向量。

❹ 市场回报矩阵。

❺ 股票期权行权价格。

❻ 期权的不确定的回报。

❼ 复制投资组合 ϕ。

❽ 检查复制的投资组合与期权的回报是否一致。

❾ 复制投资组合的价格是期权的无套利价格。

注 3：关于风险中性定价与套利定价的详细信息，请参阅 Hilpisch（2015）的第 4 章。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

套利定价

如前面的例子所示，套利定价理论［如**资产定价基本定理**（FTAP）］被认为是具有最稳健数学结果的最强大的金融理论之一。[4] 除其他原因外，这是由于期权的价格可以从其他可观察的市场参数（比如，期权标的股票的价格）中得出。从这个意义上说，套利定价并不关心如何先提出一个公允的股价，而只是把它作为一种输入。因此，套利定价在很少且较宽松的假设（比如**不存在套利假设**）下就已经能起作用了，这是许多其他金融理论所不能做到的。注意，甚至无须使用概率测度就可以推导出套利价格。

3.2 预期效用理论

预期效用理论（EUT）是金融理论的基石。自 20 世纪 40 年代提出以来，EUT 一直是不确定性决策建模的核心范式之一。[5] 基本上每一本关于金融理论和投资理论的入门教科书中都有对 EUT 的描述。原因之一是，金融领域的其他核心结果都可以从 EUT 范式中衍生出来。

3.2.1 假设和结论

EUT 是一种公理化理论，可以追溯到 von Neumann 和 Morgenstern（1944）的开创性工作。这里的**公理化**意味着理论的主要结果只能从少量公理中推导出来。有关公理效用理论、不同变体和应用的综述，请参阅 Fishburn（1968）。

1. 公理和规范性理论

在 Wolfram MathWorld 上，可以找到公理的以下定义：公理是一个不证自明的真理性命题。

当面对不确定性下的选择时，EUT 通常建立在很少的关于经济主体**偏好**的主要公理之上。尽管与公理的定义并不完全一致，但并非全部的公理都被所有的经济学家认为是“不证自明的真理”。

von Neumann 和 Morgenstern（1944）的第 25 页关于公理选择的评论如下：

> 公理的选择并不是纯粹的客观任务。人们通常预期公理选择能达到某些确定的目的，得到一些特定的定理或从公理中导出的定理，在这种程度上，公理选择问题是精确且客观的。但除此之外，还有其他一些重要但并不那么精确的要求：公理不应太多，其体系应尽可能简单且清晰明了，每条公理应具有直接的直观意义，可直接判断其合理性。

从这个意义上说，一个公理集构成了规范性理论（部分）的世界，由该理论建模。公理集收集了最小的假设集，这些假设集应被事先满足，而不是通过某种形式的证明或类似的证明。在列出导致 EUT 的一个公理集之前，面对不确定性选择时，这里用“$\succeq$”正式形式以表示有关经济主体本身偏好。

注 4：再次参阅 Hilpisch（2015）的第 4 章和其中给出的参考文献。

注 5：有关更多背景和细节，请参阅 Eichberger 和 Harper（1997）的第 1 章或 Varian（2010）的第 12 章。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

2. 经济主体的偏好

假定一个经济主体 M[2] 的偏好 $\succeq$ 是在模型经济中面临两种可交易资产的投资选择问题。例如，该经济主体可能有权选择投资组合 ϕ_A，从而导致未来带来的回报是 $A=\phi_A \times M_1$；该经济主体可能也有权选择投资组合 ϕ_B，从而导致未来带来的回报是 $B=\phi_B \times M_1$。假定这个经济主体的偏好 $\succeq$ 是对未来回报而不是投资组合的选择偏好。如果这个经济主体强偏好 A 回报而不是 B 回报，就可以写为 $A \succ B$，反之可以写为 $A \prec B$。如果该经济主体对两种偏好都无差异，就可以写为 $A \sim B$。在这一描述下，EUT 的可能的一个公理集包括以下几个方面。

完备性

一个经济主体能够对**所有**的回报进行排序。排序必须是以下三种情况之一：$A \succ B$、$A \prec B$ 或 $A \sim B$。

传递性

如果有第三个组合 ϕ_C 未来的回报是 $C=\phi_C \times M_1$，那么若 $A \succ B$ 且 $B \succ C$，则 $A \succ C$。

连续性

如果 $A \succ B \succ C$，那么存在一个数 $\alpha \in [0, 1]$，使得 $B \sim \alpha A+(1-\alpha)C$。

独立性

若 $A \sim B$，则 $\alpha A+(1-\alpha)C \sim \alpha B+(1-\alpha)C$。同样，若 $A \succ B$，则 $\alpha A+(1-\alpha)C \succ \alpha B+(1-\alpha)C$。

主导性

如果存在 $C_1=\alpha_1 A+(1-\alpha_1)C$ 和 $C_2=\alpha_2 A+(1-\alpha_2)C$，则由 $A \succ C$ 和 $C_1 \succ C_2$，可以得到 $\alpha_1 > \alpha_2$。

3. 效用函数

效用函数是一种用数学和数值的方式来表示一个经济主体偏好 $\succeq$ 的方法，因为这样的函数可以将一个数值赋值给特定的回报。在这种情况下，绝对数值是无关的，值得注意的是这些值所代表的次序。[6] 假设 $\mathbb{X}$ 表示经济主体表达偏好的所有可能的回报，那么效用函数 U 定义如下。

$$U:\mathbb{X} \to \mathbb{R}_+, x \mapsto U(x)$$

如果 U 代表偏好 $\succeq$，那么下面的关系是正确的。

$A \succ B \Rightarrow U(A) > U(B)$　（强烈偏好）
$A \succeq B \Rightarrow U(A) \geqslant U(B)$　（弱偏好）
$A \prec B \Rightarrow U(A) < U(B)$　（强烈不偏好）
$A \preceq B \Rightarrow U(A) \leqslant U(B)$　（弱不偏好）
$A \sim B \Rightarrow U(A) = U(B)$　（无差异）

效用函数 U 只能通过正线性变换来确定。因此，如果 U 代表偏好 $\succeq$，那么在 $a, b > 0$ 的条件下，$V = a + bU$ 也能代表偏好。关于效用函数，von Neumann 和 Morgenstern（1944）的第 25 页总结如下：“我们看到：如果效用的这个数值存在，那么它就是由一个线性变换决

注 6：也即通常所说的**序数**，比如门牌号码。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

定的。也就是说，效用是一个线性变换的数。”

4. 预期效用函数

von Neumann 和 Morgenstern（1944）表明，如果经济主体的偏好 $\succeq$ 满足前面提到的 5 个公理，那么存在以下**预期效用函数**形式。

$$U:\mathbb{X}\rightarrow\mathbb{R}_{+},\ x\mapsto\mathbf{E}^{P}(u(x))=\sum_{\omega}^{\Omega}P(\omega)u(x(\omega))$$

这里，$u:\mathbb{R}\rightarrow\mathbb{R},\ x\mapsto u(x)$ 是一个单调递增、状态独立的函数，通常称为**伯努利效用**，比如 $u(x)=\ln(x)$、$u(x)=x$ 和 $u(x)=x^2$。

换句话说，预期效用函数 U 首先是给定状态回报 $x(\omega)$ 的函数 u，并以给定状态发生概率 $P(\omega)$ 作为每一状态下效用的权重。在线性伯努利效用 $u(x)=x$ 的特殊情况下，预期效用就是状态相关回报的简单预期值，即 $U(x)=\mathbf{E}^{P}(x)$。

5. 风险规避

在金融学中，**风险规避**的概念很重要。最常用于测量风险厌恶程度的是**绝对风险厌恶**（ARA）的 Arrow-Pratt 度量，这可以追溯到 Pratt（1964）。假设一个经济主体的独立于状态的伯努利效用函数是 $u(x)$，那么 ARA 的 Arrow-Pratt 度量定义如下。

$$ARA(x)=-\frac{u''(x)}{u'(x)},\ x\geqslant 0$$

邮
电

根据这一测量标准，可以区分以下 3 种情况。

$$ARA(x)=-\frac{u''(x)}{u'(x)}\begin{cases}>0 & \text{风险规避}\\ =0 & \text{风险中性}\\ <0 & \text{风险偏好}\end{cases}$$

在金融理论和模型中，风险规避和风险中性的假设通常认为是合适的。

考虑前面提到的 3 种伯努利函数 $u(x)=\ln(x)$、$u(x)=x$ 和 $u(x)=x^2$，很容易证实它们分别对应着的风险规避者、风险中性者和风险偏好者的建模。比如，考虑 $u(x)=x^2$。

$$\frac{u''(x)}{u'(x)}=-\frac{2}{2x}<0, x>0\Rightarrow\text{风险偏好}$$

3.2.2　数值例子

用 Python 很容易说明 EUT 的应用。假设这个例子会模型化前面的经济模式 $\mathcal{M}^2$。假设一个经济主体根据 EUT 来决定他在未来不同回报中的偏好 $\succeq$。这个经济主体的伯努利效用函数是 $u(x)=\sqrt{x}$。在这个例子中，来自组合 ϕ_A 的回报 A_1 偏好优于组合 ϕ_D 的回报 D_1。

下面的代码演示了这个应用。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [11]: def u(x):
             return np.sqrt(x)  ❶

In [12]: phi_A = np.array((0.75, 0.25))  ❷
         phi_D = np.array((0.25, 0.75))  ❷

In [13]: np.dot(M0, phi_A) == np.dot(M0, phi_D)  ❸
Out[13]: True

In [14]: A1 = np.dot(M1, phi_A)  ❹
         A1  ❹
Out[14]: array([17.75, 6.5 ])

In [15]: D1 = np.dot(M1, phi_D)  ❺
         D1  ❺
Out[15]: array([13.25, 9.5 ])

In [16]: P = np.array((0.5, 0.5))  ❻

In [17]: def EUT(x):
             return np.dot(P, u(x))  ❼

In [18]: EUT(A1)  ❽
Out[18]: 3.381292321692286

In [19]: EUT(D1)  ❽
Out[19]: 3.3611309730623735
```

❶ 风险规避型伯努利效用函数。

❷ 两个投资组合拥有不同的权重。

❸ 表明每一个投资组合的建立成本是相同的。

❹ 一个投资组合的不确定的回报……

❺ ……另一个投资组合的不确定的回报。

❻ 概率测度。

❼ 期望效用函数。

❽ 两个不确定的回报的效用值。

这种情况下的一个典型问题是，在经济主体预算固定（$w > 0$）条件下，求得一个最优投资组合（能最大化预期效用的组合）。以下 Python 代码对这个问题进行了建模并准确地解决了它。在可用预算中，经济主体将大约 60% 投资于风险资产、大约 40% 投资于无风险资产。结果主要由伯努利效用函数的特殊形式决定。

```
In [20]: from scipy.optimize import minimize

In [21]: w = 10  ❶

In [22]: cons = {'type': 'eq', 'fun': lambda phi: np.dot(M0, phi) - w}  ❷
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [23]: def EUT_(phi):
             x = np.dot(M1, phi)  ❸
             return EUT(x)  ❸

In [24]: opt = minimize(lambda phi: -EUT_(phi),  ❹
                        x0=phi_A,  ❺
                        constraints=cons)  ❻
In [25]: opt  ❼
Out[25]:      fun: -3.385015999493397
              jac: array([-1.69249132, -1.69253424])
          message: 'Optimization terminated successfully.'
             nfev: 16
              nit: 4
             njev: 4
           status: 0
          success: True
                x: array([0.61122474, 0.38877526])

In [26]: EUT_(opt['x'])  ❽
Out[26]: 3.385015999493397
```

❶ 经济主体的固定预算。

❷ 使用最小化的预算约束。

❸ 在投资组合上的预期效用函数。

❹ 最小化 -EUT_(phi)，最大化 EUT_(phi)。

❺ 最优化的初始猜测值。

❻ 预算约束应用。

❼ 最优结果，包括 x 下的最优投资组合。

❽ 给定预算 $w = 10$ 时的最优（最高）预期效用。

3.3 均值–方差投资组合理论

Markowitz（1952）的均值 – 方差投资组合（MVP）理论是另一个金融理论的基石。它是最早的不确定性投资理论之一，这种不确定性主要是对股票投资组合构建的统计性测量。MVP 完全是从驱动公司股价表现或假设可能对公司增长前景具有重要影响的未来竞争力的基本面抽象出来的。基本上，唯一重要的输入数据就是股价的时间序列以及由此得出的统计数据，比如（历史）年化平均收益率和（历史）年化收益率方差。

3.3.1 假设和结论

根据 Markowitz（1952）的说法，MVP 的中心假设是投资者只关心预期收益率和这些收益率的方差：

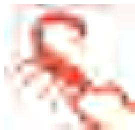
专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

接下来，考虑这样一条规则，即投资者确实（或应该）认为关心预期收益率是一件可取的事情，而关心收益率的方差则是一件不可取的事情。这条规则会有许多合理的观点，既可以作为投资行为的准则，也可以作为投资行为的假设。

预期收益率最大的投资组合不一定是方差最小的投资组合。投资者既可以通过接受方差来获得预期收益率，也可以通过放弃预期收益率来减少方差。

这种投资者偏好的方法与直接定义经济主体偏好和效用函数的方法截然不同。MVP 更倾向于假设经济主体的偏好和效用函数是投资组合预期收益率的一阶矩和二阶矩。



隐含的正态分布假设

一般而言，MVP 理论只关注一期投资组合风险和收益，与标准 EUT 不兼容。解决这个问题的一种方法是假设风险资产的收益率服从正态分布，这样一阶矩和二阶矩就足以描述资产收益率的完整的分布。正如第 4 章所述，在真实的金融数据中几乎从未看到过这种情形。另一种方法是假设一个特定的二次伯努利效用函数，如下一节所示。

1. 投资组合统计

假定一种静态经济模式 $\mathsf{M}^N=(\{\Omega, \mathsf{F}, P\}, \mathbb{A})$，其中可交易资产集 $\mathbb{A}$ 包括 N 个风险资产，即 $A^1, A^2, \cdots, A^N$。　　是资产 n 今天的固定价格，A_1^n 是一年后的回报，$\boldsymbol{r}^n$ 是资产 n（简单）净收益率向量，定义为：

$$\boldsymbol{r}^n=\frac{A_1^n}{A_0^n}-1$$

对于具有相同概率的所有未来状态，资产 n 的预期收益率表示为：

$$\mu^n=\frac{1}{|\Omega|}\sum_{\omega}^{\Omega}\boldsymbol{r}^n(\omega)$$

相应地，**预期收益率向量**为：

$$\boldsymbol{\mu}=\begin{bmatrix}\mu^1\\ \mu^2\\ \vdots\\ \mu^N\end{bmatrix}$$

投资组合（向量）$\boldsymbol{\phi}=(\phi^1, \phi^2, \cdots, \phi^N)^{\mathrm{T}}$ 满足 $\phi_n\geqslant 0$ 和 $\sum_n^N\phi^n=1$ 这两个条件，它们会为投资组合中的每项资产分配权重。[7]

投资组合的预期收益率由投资组合的权向量与预期收益率向量的点积给出：

注 7：此假设并非必要的约束条件，稍加调整即可适用于**做空**场景。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

$$\mu^{phi} = \boldsymbol{\phi} \times \boldsymbol{\mu}$$

现在通过以下公式定义资产 n 和 m 之间的协方差：

$$\sigma_{mn} = \sum_{\omega}^{\Omega} (\boldsymbol{r}^m(\omega) - \mu^m)(\boldsymbol{r}^n(\omega) - \mu^n)$$

协方差矩阵为：

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \cdots & \sigma_{1n} \\ \sigma_{21} & \sigma_{22} & \cdots & \sigma_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{n1} & \sigma_{n2} & \cdots & \sigma_{nn} \end{bmatrix}$$

投资组合的预期方差依次由双点积得到：

$$\varphi^{phi} = \phi^{\mathrm{T}} \times \boldsymbol{\Sigma} \times \boldsymbol{\phi}$$

相应地，**投资组合的预期波动率**如下。

$$\sigma^{phi} = \sqrt{\varphi^{phi}}$$

2. 夏普比率

Sharpe（1966）提出了一种测量方法以评估经风险调整后的共同基金和其他投资组合，甚至是单一风险资产的绩效。它的最简单形式是，将投资组合的（预期的，已实现的）收益率与其（预期的，已实现的）波动率联系起来。因此，在形式上夏普比率可由以下公式定义：

$$^{phi} = \frac{\mu^{phi}}{\sigma^{phi}}$$

如果 r 代表低风险的短期利率，则投资组合 phi 相对于无风险替代品的**风险溢价**或**超额收益**由下式定义：$\mu^{phi} - r$。在夏普比率的另一个版本中，这个风险溢价是分子：

$$^{phi} = \frac{\mu^{phi} - r}{\sigma^{phi}}$$

当无风险短期利率相对较低时，如果采用相同的无风险短期利率，则两种计算不会产生较大差异的数值结果。特别是，当根据夏普比率对不同的投资组合进行排名时，这两个版本应该产生相同的排名顺序，其他的都相等。

3.3.2　数值例子

回到静态经济模式 M^2，MVP 的基本概念可以很容易地再次用 Python 来说明。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

1. 投资组合统计

首先，以下是投资组合**预期收益率**的推导。

```
In [27]: rS = S1 / S0 - 1  ❶
         rS  ❶
Out[27]: array([ 1. , -0.5])

In [28]: rB = B1 / B0 - 1  ❷
         rB  ❷
Out[28]: array([0.1, 0.1])

In [29]: def mu(rX):
             return np.dot(P, rX)  ❸

In [30]: mu(rS)  ❹
Out[30]: 0.25

In [31]: mu(rB)  ❹
Out[31]: 0.10000000000000009

In [32]: rM = M1 / M0 - 1  ❺
         rM  ❺
Out[32]: array([[ 1. , 0.1],
                [-0.5, 0.1]])

In [33]: mu(rM)  ❻
Out[33]: array([0.25, 0.1 ])
```

❶ 风险资产的收益向量。

❷ 无风险资产的收益向量。

❸ 预期收益率函数。

❹ 交易资产的预期收益率。

❺ 交易资产收益矩阵。

❻ 预期收益向量。

然后，以下是**方差**、**波动率**以及**协方差矩阵**。

```
In [34]: def var(rX):
             return ((rX - mu(rX)) ** 2).mean()  ❶

In [35]: var(rS)
Out[35]: 0.5625

In [36]: var(rB)
Out[36]: 0.0

In [37]: def sigma(rX):
             return np.sqrt(var(rX))  ❷
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [38]: sigma(rS)
Out[38]: 0.75

In [39]: sigma(rB)
Out[39]: 0.0

In [40]: np.cov(rM.T, aweights=P, ddof=0)  ❸
Out[40]: array([[0.5625, 0.    ],
                [0.    , 0.    ]])
```

❶ 方差函数。

❷ 波动率函数。

❸ 协方差矩阵。

最后，以等权重投资组合为例，**投资组合预期收益率**、**投资组合预期方差**和**投资组合预期波动率**如下。

```
In [41]: phi = np.array((0.5, 0.5))

In [42]: def mu_phi(phi):
             return np.dot(phi, mu(rM))  ❶

In [43]: mu_phi(phi)
Out[43]: 0.17500000000000004

In [44]: def var_phi(phi):
             cv = np.cov(rM.T, aweights=P, ddof=0)
             return np.dot(phi, np.dot(cv, phi))  ❷

In [45]: var_phi(phi)
Out[45]: 0.140625

In [46]: def sigma_phi(phi):
             return var_phi(phi) ** 0.5  ❸

In [47]: sigma_phi(phi)
Out[47]: 0.375
```

❶ 投资组合预期收益率。

❷ 投资组合预期方差。

❸ 投资组合预期波动率。

2. 投资机会集

基于蒙特卡罗模拟的投资组合权重 ϕ 可以在波动率 – 收益率空间中可视化投资机会集，如图 3-1 所示。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

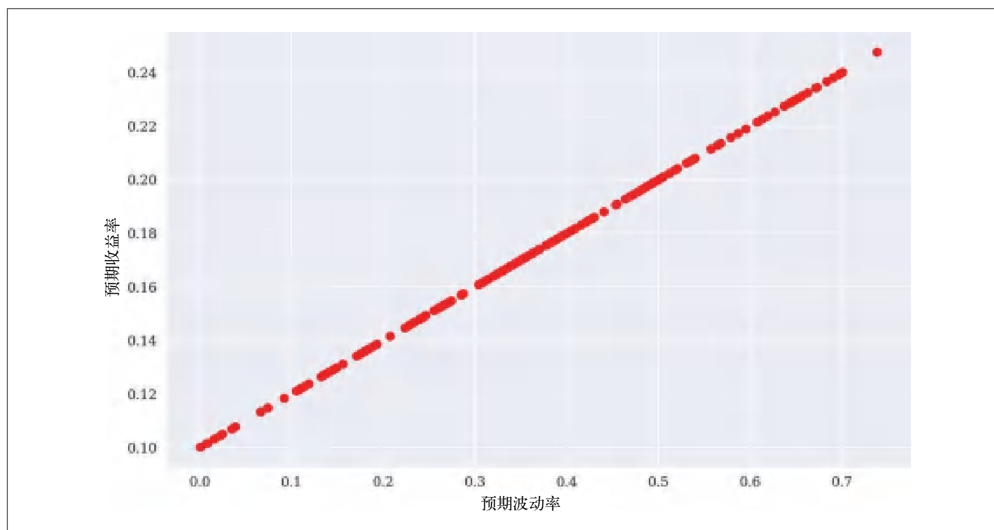


图 3-1：模拟投资组合预期波动率和预期收益率（单一风险资产）

因为只有一种风险资产和一种无风险资产，所以机会集是一条直线。

```
In [48]: from pylab import plt, mpl
         plt.style.use('seaborn')
         mpl.rcParams['savefig.dpi'] = 300
         mpl.rcParams['font.family'] = 'serif'

In [49]: phi_mcs = np.random.random((2, 200))  ❶

In [50]: phi_mcs = (phi_mcs / phi_mcs.sum(axis=0)).T  ❶

In [51]: mcs = np.array([(sigma_phi(phi), mu_phi(phi))
                         for phi in phi_mcs])  ❷

In [52]: plt.figure(figsize=(10, 6))
         plt.plot(mcs[:, 0], mcs[:, 1], 'ro')
         plt.xlabel('expected volatility')
         plt.ylabel('expected return');
```

❶ 随机投资组合，标准化为 1。

❷ 随机组合的预期投资组合波动率与收益率。

现在设想一种有 3 种状态的静态经济模式 $\mathcal{M}^3$，其中 $\Omega = \{u, m, d\}$。这 3 种状态的可能性相等，即 $P = \left\{\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right\}$。可交易资产集合是包括两个固定价格为 $S_0 = T_0 = 10$ 的风险资产 S 和 T 的组合，不确定回报分别为以下各项：

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

$$S_1 = \begin{bmatrix} 20 \\ 10 \\ 5 \end{bmatrix}$$

和

$$T_1 = \begin{bmatrix} 1 \\ 12 \\ 13 \end{bmatrix}$$

基于这些假设，下面的 Python 代码重复了蒙特卡罗模拟并可视化了图 3-2 中的结果。使用两种风险资产，著名的 MVP“子弹”将可见。

```
In [53]: P = np.ones(3) / 3  ❶
         P  ❶
Out[53]: array([0.33333333, 0.33333333, 0.33333333])

In [54]: S1 = np.array((20, 10, 5))

In [55]: T0 = 10
         T1 = np.array((1, 12, 13))

In [56]: M0 = np.array((S0, T0))
         M0
Out[56]: array([10, 10])

In [57]: M1 = np.array((S1, T1)).T
         M1
Out[57]: array([[20,  1],
                [10, 12],
                [ 5, 13]])

In [58]: rM = M1 / M0 - 1
         rM
Out[58]: array([[ 1. , -0.9],
                [ 0. ,  0.2],
                [-0.5,  0.3]])

In [59]: mcs = np.array([(sigma_phi(phi), mu_phi(phi))
                         for phi in phi_mcs])

In [60]: plt.figure(figsize=(10, 6))
         plt.plot(mcs[:, 0], mcs[:, 1], 'ro')
         plt.xlabel('expected volatility')
         plt.ylabel('expected return');
```

❶ 3 种状态的新概率测度。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

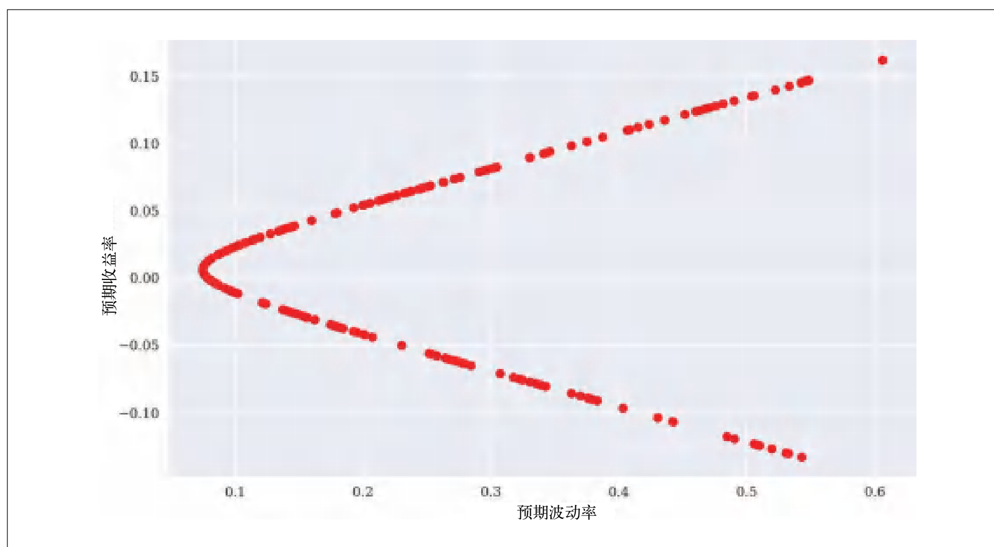


图 3-2：模拟投资组合预期波动率和预期收益率（两种风险资产）

3. 最小波动率和最大夏普比率

接下来，推导**最小波动率**（最小方差）和**最大夏普比率**投资组合。图 3-3 显示了两个投资组合在风险 – 收益空间中的位置。

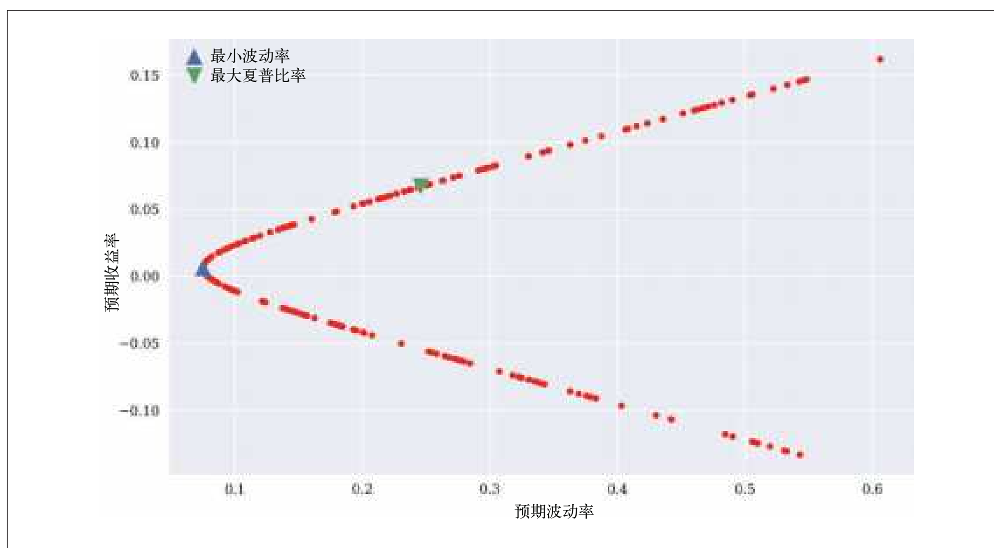


图 3-3：最小波动率和最大夏普比率投资组合

尽管风险资产 T 具有负的预期收益率，但它在最大夏普比率投资组合中具有重要的权重。这是由于多样化效应降低了投资组合的风险，而不是降低了投资组合的预期收益率。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [61]: cons = {'type': 'eq', 'fun': lambda phi: np.sum(phi) - 1}

In [62]: bnds = ((0, 1), (0, 1))

In [63]: min_var = minimize(sigma_phi, (0.5, 0.5),
                            constraints=cons, bounds=bnds)  ❶

In [64]: min_var
Out[64]:      fun: 0.07481322946910632
              jac: array([0.07426564, 0.07528945])
          message: 'Optimization terminated successfully.'
             nfev: 17
              nit: 4
             njev: 4
           status: 0
          success: True
                x: array([0.46511697, 0.53488303])

In [65]: def sharpe(phi):
             return mu_phi(phi) / sigma_phi(phi)  ❷

In [66]: max_sharpe = minimize(lambda phi: -sharpe(phi), (0.5, 0.5),
                        constraints=cons, bounds=bnds)  ❸

In [67]: max_sharpe
Out[67]:      fun: -0.2721654098971811
              jac: array([ 0.00012054, -0.00024174])
          message: 'Optimization terminated successfully.'
             nfev: 38
              nit: 9
             njev: 9
           status: 0
          success: True
                x: array([0.66731116, 0.33268884])

In [68]: plt.figure(figsize=(10, 6))
         plt.plot(mcs[:, 0], mcs[:, 1], 'ro', ms=5)
         plt.plot(sigma_phi(min_var['x']), mu_phi(min_var['x']),
                  '^', ms=12.5, label='minimum volatility')
         plt.plot(sigma_phi(max_sharpe['x']), mu_phi(max_sharpe['x']),
                  'v', ms=12.5, label='maximum Sharpe ratio')
         plt.xlabel('expected volatility')
         plt.ylabel('expected return')
         plt.legend();
```

❶ 最小化投资组合预期波动率。

❷ 定义夏普比率函数，假设短期利率为 0。

❸ 通过最小化夏普比率的负值来最大化夏普比率。

4. 有效边界

有效投资组合是在给定预期风险（收益率）下具有最大预期收益率（风险）的投资组合。在图 3-3 中，所有预期收益率低于最小风险组合的投资组合都是**低效**的。下面的代码从风

专为量化交易设计的实时行情数据API: www.alltick.co

险－收益空间中得到有效投资组合，并将其绘制在图 3-4 中。所有有效投资组合的集合称为**有效边界**，经济主体只会选择位于有效边界上的投资组合。

```
In [69]: cons = [{'type': 'eq', 'fun': lambda phi: np.sum(phi) - 1},
                 {'type': 'eq', 'fun': lambda phi: mu_phi(phi) - target}]  ❶

In [70]: bnds = ((0, 1), (0, 1))

In [71]: targets = np.linspace(mu_phi(min_var['x']), 0.16)  ❷

In [72]: frontier = []
         for target in targets:
             phi_eff = minimize(sigma_phi, (0.5, 0.5),
                                constraints=cons, bounds=bnds)['x']  ❸
             frontier.append((sigma_phi(phi_eff), mu_phi(phi_eff)))
         frontier = np.array(frontier)

In [73]: plt.figure(figsize=(10, 6))
         plt.plot(frontier[:, 0], frontier[:, 1], 'mo', ms=5,
                  label='efficient frontier')
         plt.plot(sigma_phi(min_var['x']), mu_phi(min_var['x']),
                  '^', ms=12.5, label='minimum volatility')
         plt.plot(sigma_phi(max_sharpe['x']), mu_phi(max_sharpe['x']),
                  'v', ms=12.5, label='maximum Sharpe ratio')
         plt.xlabel('expected volatility')
         plt.ylabel('expected return')
         plt.legend();
```

❶ 固定预期收益率目标水平的新的约束条件。

❷ 生成一组目标预期收益率。

❸ 得到给定预期收益率目标的最小波动率投资组合。

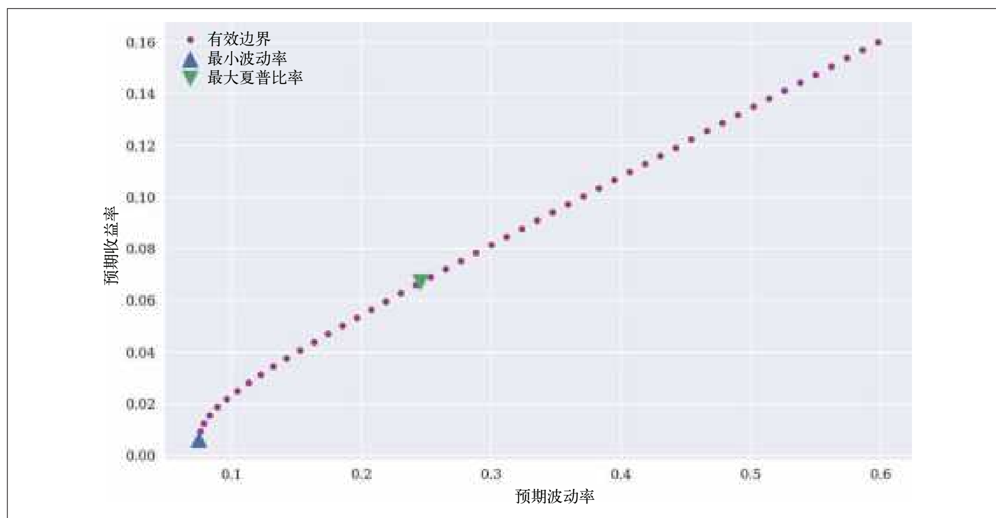


图 3-4：有效边界

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

3.4 资本资产定价模型

资本资产定价模型（CAPM）是金融学中被广泛记载和应用的模型之一，其核心是以线性方式将一只股票的预期收益率与市场投资组合的预期收益率联系起来，市场投资组合通常用标准普尔 500 等各类股票指数来近似。这种模式可以追溯到 Sharpe（1964）和 Lintner（1965）的开创性工作。Jones（2012）的第 9 章将 CAPM 与 MVP 的关系描述如下：

> 资本市场理论是一个实证理论，因为它假设了投资者的行为方式，而不是投资者应该如何的行为方式，就像现代投资组合理论（MVP）的情况。将资本市场理论视为投资组合理论的延伸是合理的，但重要的是要明白，MVP 并不是建立在资本市场理论的有效性或缺乏有效性之上的。
>
> 许多投资者感兴趣的特定均衡模型是资本资产定价模型，通常称为 CAPM。它允许我们评估单个证券的相关风险，以及评估风险与投资预期收益率之间的关系。作为一种均衡模型，CAPM 因其简单性和隐含性而具有吸引力。

3.4.1 假设和结论

假设上一节中的静态经济模式 $\mathsf{M}^N = (\{\Omega, \mathsf{F}, P\}, \mathbb{A})$ 有 N 个交易资产和所有的简化假设。在 CAPM 模型中，假设经济主体根据 MVP 进行投资，只关心风险资产一个时期内的风险和收益统计值。

在**资本市场均衡**中，所有可用资产由所有经济主体所持有，所有市场都是出清的。由于经济主体都假设使用 MVP 来形成有效投资组合，因此这意味着所有经济主体必须持有相同的有效投资组合（就其构成而言），因为每个经济主体的可交易资产集是相同的。换句话说，**市场投资组合**（一组可交易资产）必须位于效率前沿。如果不是这样，那么市场就不可能处于均衡状态。

获得资本市场均衡的机制是什么？当今的可交易资产价格是确保市场出清的机制。如果经济主体对可交易资产的需求不足，那么该资产的价格就需要下降。如果需求高于供给，则它的价格就会上涨。如果价格设定正确，那么每种可交易资产的需求和供给就会相等。MVP 让可交易资产的价格为给定值，CAPM 则是在给定资产风险 – 收益特征情形下，该资产均衡价格**应该**是多少的一种理论和模型。

CAPM 假设存在（至少）一种无风险资产，每个经济主体可以投资任何金额，并获得 $\bar{r}$ 的无风险利率。因此，每一个经济主体将在均衡状态下持有市场投资组合和无风险资产的组合，这就是所谓的**双基金分离定理**[8]。所有这些投资组合的集合被称为**资本市场线**（CML）。图 3-5 是 CML 的示意图。市场投资组合右侧的投资组合只有在允许经济主体卖空无风险资产和以这种方式借钱的情况下才能实现。

```
In [74]: plt.figure(figsize=(10, 6))
         plt.plot((0, 0.3), (0.01, 0.22), label='capital market line')
         plt.plot(0, 0.01, 'o', ms=9, label='risk-less asset')
```

注 8：如需了解更多信息，请参阅 Jones（2012）的第 9 章。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
plt.plot(0.2, 0.15, '^', ms=9, label='market portfolio')
plt.annotate('$(0, \\bar{r})$', (0, 0.01), (-0.01, 0.02))
plt.annotate('$(\sigma_M, \mu_M)$', (0.2, 0.15), (0.19, 0.16))
plt.xlabel('expected volatility')
plt.ylabel('expected return')
plt.legend();
```

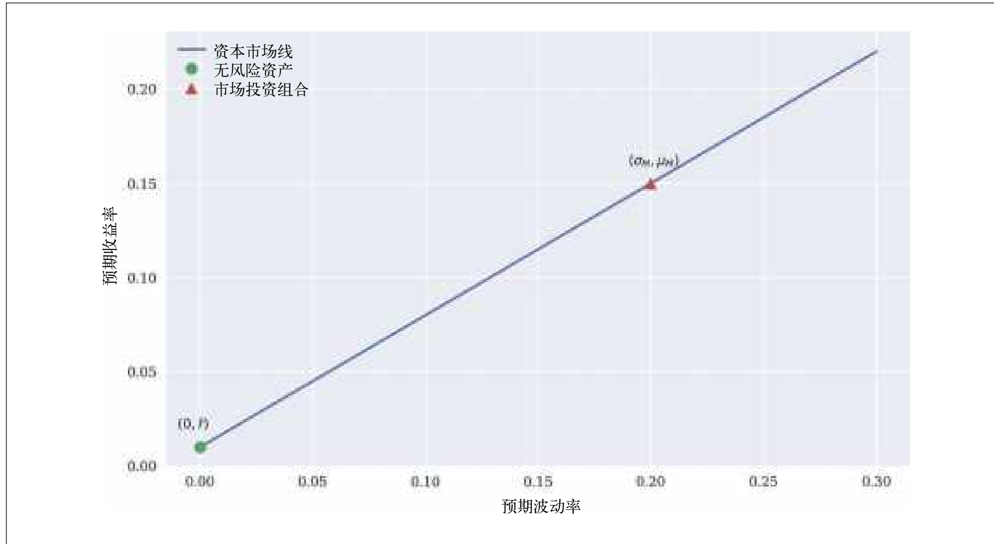


图 3-5：资本市场线（CML）

如果 σ_M 和 μ_M 分别为市场投资组合的预期波动率和预期收益率，则投资组合预期收益率 μ 与预期波动率 σ 之间的资本市场线定义如下：

$$\mu = \bar{r} + \frac{\mu_M - \bar{r}}{\sigma_M}\sigma$$

以下表达式称为**风险的市场价格**：

$$\frac{\mu_M - \bar{r}}{\sigma_M}$$

它表示在均衡时经济主体需要增加多少预期收益率才能承担一个单位的额外风险。

CAPM 把任何可交易风险资产（资产 $n = 1, 2, \cdots, N$）的预期收益率和市场投资组合的预期收益率关联在了一起，具体如下：

$$\mu^n = \bar{r} + \beta_n(\mu_M - \bar{r})$$

这里，β_n 被定义为由具有 n 个风险资产的市场投资组合的协方差除以市场投资组合本身的方差：

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

$$\beta_n = \frac{\sigma_{M,n}}{\sigma_M^2}$$

当 $\beta_n = 0$ 时，根据 CAPM 公式，预期收益率就是无风险利率。β_n 越高，风险资产的预期收益率就越高。β_n 会衡量**不可分散的风险**。这类风险也称为**市场风险**或**系统性风险**。根据 CAPM 的说法，经济主体得到更高预期收益率是承担这一风险的奖励。

3.4.2 数值例子

假设静态经济模式有 3 种可能的未来状态，即 $\mathcal{M}^3 = (\{\Omega, \mathcal{F}, P\}, \mathbb{A})$，其中有机会以无风险利率 $\bar{r} = 0.0025$ 进行借贷。风险资产 S 和 T 的可用的数量分别为 0.8 和 0.2。

1. 资本市场线

图 3-6 显示了有效边界、市场投资组合、无风险资产和由此产生的风险 − 收益空间的资本市场线。

```
In [75]: phi_M = np.array((0.8, 0.2))

In [76]: mu_M = mu_phi(phi_M)
         mu_M
Out[76]: 0.10666666666666666

In [77]: sigma_M = sigma_phi(phi_M)
         sigma_M
Out[77]: 0.39474323581566567

In [78]: r = 0.0025

In [79]: plt.figure(figsize=(10, 6))
         plt.plot(frontier[:, 0], frontier[:, 1], 'm.', ms=5,
                  label='efficient frontier')
         plt.plot(0, r, 'o', ms=9, label='risk-less asset')
         plt.plot(sigma_M, mu_M, '^', ms=9, label='market portfolio')
         plt.plot((0, 0.6), (r, r + ((mu_M - r) / sigma_M) * 0.6),
                  'r', label='capital market line', lw=2.0)
         plt.annotate('$(0, \\bar{r})$', (0, r), (-0.015, r + 0.01))
         plt.annotate('$(\sigma_M, \mu_M)$', (sigma_M, mu_M),
                      (sigma_M - 0.025, mu_M + 0.01))
         plt.xlabel('expected volatility')
         plt.ylabel('expected return')
         plt.legend();
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

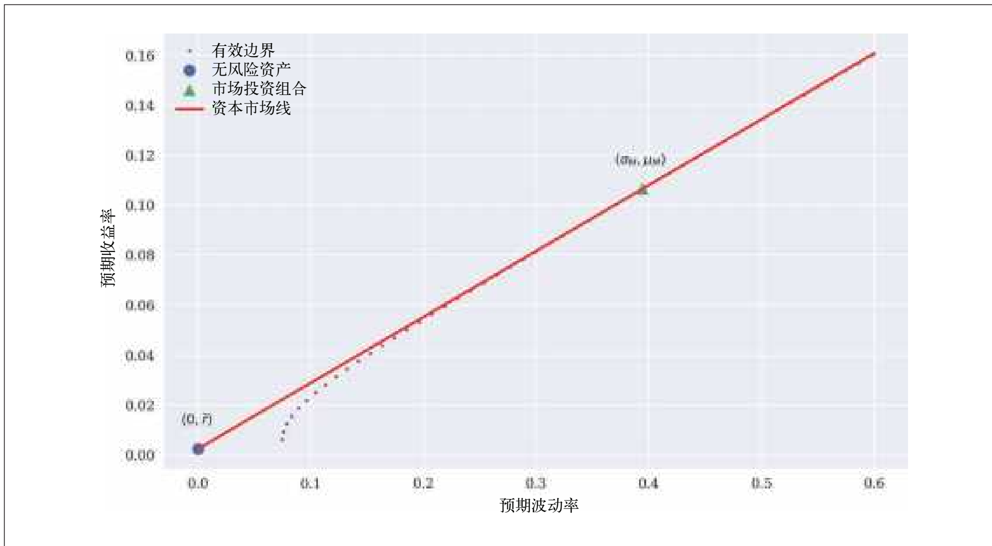


图 3-6：两种风险资产的资本市场线

2. 最优投资组合

假设一个经济主体的预期效用函数在未来的回报中定义如下：

$$U:\mathbb{X}\to\mathbb{R}_+, x\mapsto \mathbf{E}^P(u(x))=\mathbf{E}^P\left(x-\frac{b}{2}x^2\right)$$

这里，$b>0$。经过一些转换后，预期效用函数可以表示为风险－收益组合。

$$U:\mathbb{R}_+\times\mathbb{R}_+\to\mathbb{R},(\sigma,\mu)\mapsto \mu-\frac{b}{2}(\sigma^2+\mu^2)$$

特定二次效用函数

尽管 MVP 理论和 CAPM 都假设投资者只关心一个时期的投资组合风险和收益，但这种假设通常只在给定伯努利效用函数的特定形式（二次效用函数）时与 EUT 一致。这种伯努利函数几乎只在 MVP 理论中提及和使用。除此之外，它的特殊形式和特点通常被认为是不合适的。无论是正态分布的资产收益假设，还是二次效用函数，都不能以“优雅”的方式与 EUT、MVP 理论和 CAPM 保持一致。

经济主体会在 CML 上选择什么投资组合呢？直接效用最大化的结果可用 Python 应用来实现。为此，固定参数 $b=1$。

```
In [80]: def U(p):
             mu, sigma = p
             return mu - 1 / 2 * (sigma ** 2 + mu ** 2)  ❶
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [81]: cons = {'type': 'eq',
                 'fun': lambda p: p[0] - (r + (mu_M - r) / sigma_M * p[1])}  ❷

In [82]: opt = minimize(lambda p: -U(p), (0.1, 0.3), constraints=cons)

In [83]: opt
Out[83]:      fun: -0.034885186826739426
              jac: array([-0.93256102,  0.24608851])
          message: 'Optimization terminated successfully.'
             nfev: 8
              nit: 2
             njev: 2
           status: 0
          success: True
                x: array([0.06743897, 0.2460885 ])
```

❶ 风险 – 收益空间的效用函数。

❷ 投资组合在 CML 上的条件。

3. 无差异曲线

可视化的分析可以说明经济主体的最优决策。固定经济主体的效用水平后，可以在风险 – 收益空间中绘制**无差异曲线**。当无差异曲线与 CML 相切时，就可以找到最优投资组合。任何其他的无差异曲线（没有接触 CML 或与 CML 相交两次）都不能确定是最佳投资组合。

首先，下面是一些符号计算的 Python 代码，它将风险 – 收益空间中的效用函数转换为了在固定的效用水平 v 和固定的参数值 b 下的 μ 与 σ 的函数关系。图 3-7 显示了两条无差异曲线。无差异曲线上的每一对 (σ, μ) 组合会产生相同的效用，因此这些投资组合对经济主体来说是无差异的。

```
In [84]: from sympy import *
         init_printing(use_unicode=False, use_latex=False)

In [85]: mu, sigma, b, v = symbols('mu sigma b v')  ❶

In [86]: sol = solve('mu - b / 2 * (sigma ** 2 + mu ** 2) - v', mu)  ❷

In [87]: sol  ❷
Out[87]:
                 _________________________       _________________________
                /   2      2                    /   2      2
          1 - \/ - b *sigma  - 2*b*v + 1      \/ - b *sigma  - 2*b*v + 1  + 1
         [-------------------------------, -------------------------------]
                        b                                 b

In [88]: u1 = sol[0].subs({'b': 1, 'v': 0.1})  ❸
         u1
Out[88]:
                ______________
               /            2
         1 - \/  0.8 - sigma

In [89]: u2 = sol[0].subs({'b': 1, 'v': 0.125})  ❸
         u2
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
Out[89]:           _______________
                  /              2
         1 - \/  0.75 - sigma

In [90]: f1 = lambdify(sigma, u1)  ❹
         f2 = lambdify(sigma, u2)  ❹

In [91]: sigma_ = np.linspace(0.0, 0.5)  ❺
         u1_ = f1(sigma_)  ❻
         u2_ = f2(sigma_)  ❻

In [92]: plt.figure(figsize=(10, 6))
         plt.plot(sigma_, u1_, label='$v=0.1$')
         plt.plot(sigma_, u2_, '--', label='$v=0.125$')
         plt.xlabel('expected volatility')
         plt.ylabel('expected return')
         plt.legend();
```

❶ 定义 SymPy 符号。

❷ 求解的效用函数 μ。

❸ 用数值代替 b, v。

❹ 从得到的方程生成可调用函数。

❺ 指定用于评估函数的 σ 值。

❻ 评估两个不同效用级别的可调用函数。

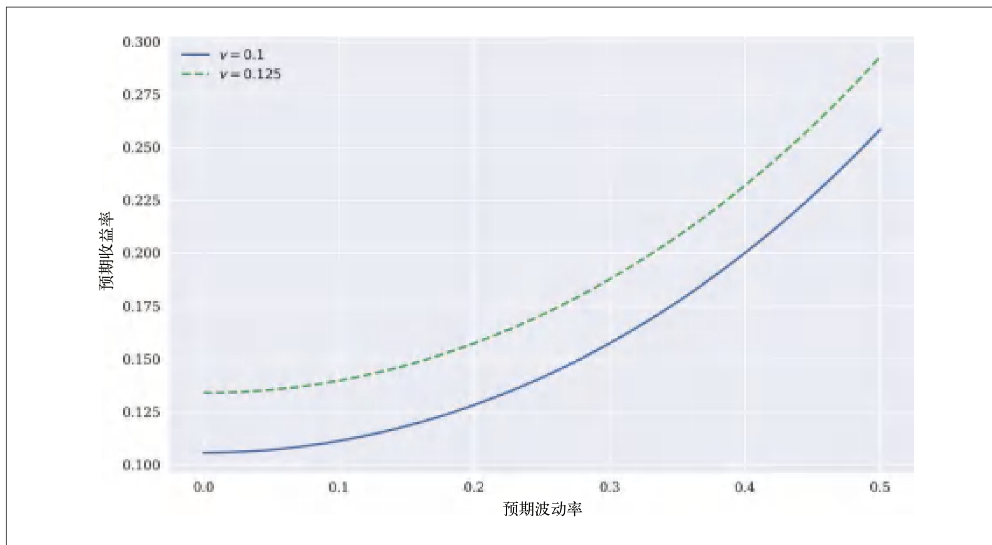


图 3-7：风险－收益空间中的无差异曲线

下一步，无差异曲线需要与 CML 相结合，以直观地找出经济主体的最优投资组合选择。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

利用前面的数值优化结果，图 3-8 显示了无差异曲线与 CML 相切点处的最优投资组合。可以看出，经济主体确实选择了市场投资组合和无风险资产的组合。

```
In [93]: u = sol[0].subs({'b': 1, 'v': -opt['fun']})  ❶
         u
Out[93]:        ____________________________
               /                          2
         1 - \/  0.930229626346521 - sigma

In [94]: f = lambdify(sigma, u)

In [95]: u_ = f(sigma_)  ❷

In [96]: plt.figure(figsize=(10, 6))
         plt.plot(0, r, 'o', ms=9, label='risk-less asset')
         plt.plot(sigma_M, mu_M, '^', ms=9, label='market portfolio')
         plt.plot(opt['x'][1], opt['x'][0], 'v', ms=9, label='optimal portfolio')
         plt.plot((0, 0.5), (r, r + (mu_M - r) / sigma_M * 0.5),
                  label='capital market line', lw=2.0)
         plt.plot(sigma_, u_, '--', label='$v={}$'.format(-round(opt['fun'], 3)))
         plt.xlabel('expected volatility')
         plt.ylabel('expected return')
         plt.legend();
```

❶ 定义最优效用级别的无差异曲线。

❷ 得到数值以绘制无差异曲线。

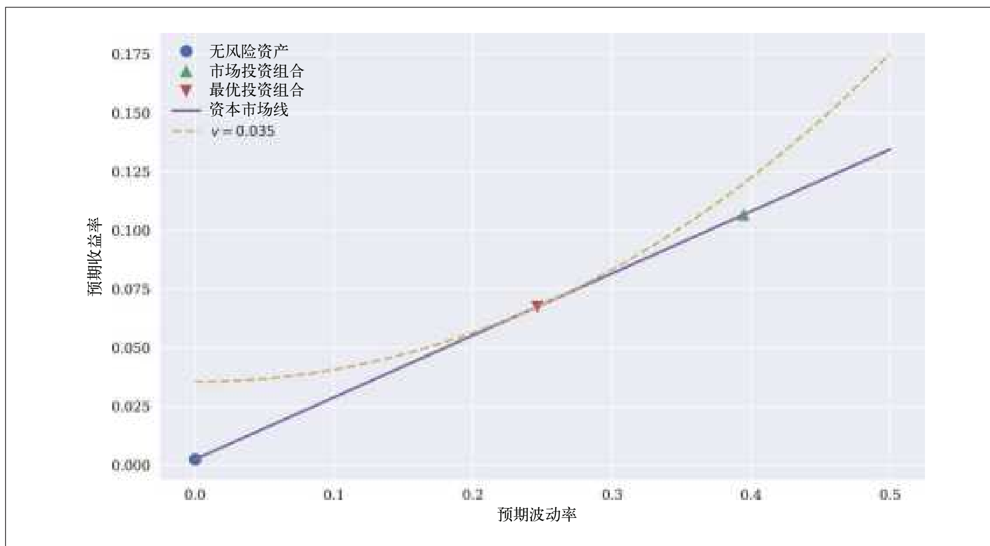


图 3-8：CML 上的最优投资组合

本节中的主题通常在**资本市场理论**（CMT）下讨论。CAPM 是该理论的一部分，第 4 章将用实际金融时间序列数据加以说明。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

3.5 套利定价理论

早期，我们观察到 CAPM 的缺点后会在金融学文献中加以讨论。CAPM 的一个主要推广是 Ross（1971，1976）提出的套利定价理论（APT）。Ross（1976）对其论文做了如下介绍：

> 本文的目的是严格检验 Ross（1971）提出的资本资产定价套利模型。套利模型是 Sharpe、Lintner 和 Treynor 提出的对均值 – 方差资本资产定价模型的一种替代模型，该模型已成为解释资本市场上观察到的风险资产现象的主要分析工具。

3.5.1 假设和结论

APT 是 CAPM 对多种风险因素的推广。从这个意义上说，APT 并不认为市场投资组合是唯一相关的风险因素，而是有相当多的风险类型一起影响股票的表现（预期收益率）。这些风险因素可能包括规模、波动率、价值和动量。[9] 除了这一主要差异之外，该模型还依赖于类似的假设，比如市场是完美的，在同一固定利率下可（无限制）借贷等。

在 Poss（1976）的原始动态版本中，APT 采用以下形式：

$$\boldsymbol{y}_t = \boldsymbol{a} + \boldsymbol{B}\boldsymbol{f}_t + \epsilon_t$$

这里，$\boldsymbol{y}_t$ 是 M 个观察变量的向量，即 M 个不同股票在时间 t 的预期收益率：

$$\boldsymbol{y}_t = \begin{bmatrix} y_t^1 \\ y_t^2 \\ \vdots \\ y_t^M \end{bmatrix}$$

$\boldsymbol{a}$ 是 M 常数项的向量：

$$\boldsymbol{a} = \begin{bmatrix} a^1 \\ a^2 \\ \vdots \\ a^M \end{bmatrix}$$

$\boldsymbol{f}_t$ 是 F 因素在 t 时刻的向量：

$$\boldsymbol{f}_t = \begin{bmatrix} f_t^1 \\ f_t^2 \\ \vdots \\ f_t^F \end{bmatrix}$$

注 9：有关实践中使用的因素的更多背景信息，请参阅 Bender 等（2013）。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

$\boldsymbol{B}$ 是 $M \times F$ 所谓因子荷载矩阵：

$$\boldsymbol{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1F} \\ b_{21} & b_{22} & \cdots & b_{2F} \\ \vdots & \vdots & \ddots & \vdots \\ b_{M1} & b_{M2} & \cdots & b_{MF} \end{bmatrix}$$

最后，ϵ_t 是 $\boldsymbol{M}$ 向量的充分独立的残差。

$$\epsilon_t = \begin{bmatrix} \epsilon_t^1 \\ \epsilon_t^2 \\ \vdots \\ \epsilon_t^M \end{bmatrix}$$

Jones（2012）的第 9 章描述了 CAPM 和 APT 之间的如下差异：

> 与 CAPM 或任何其他资产定价模型类似，APT 假设了预期收益率和风险之间的关系。然而，它使用了不同的假设和程序。非常重要的是，APT 不像 CAPM 那样（预测只有市场风险影响预期收益率）严重依赖于标的市场投资组合。相反，APT 认识到几种类型的风险可能会影响证券收益率。

CAPM 和 APT 都以**线性**方式将输出变量与相关的输入因子联系起来。从计量经济学的角度来看，这两个模型都是基于线性 OLS 回归来实现的。CAPM 可以在**单变量**线性 OLS 回归的基础上实现，APT 则需要**多变量** OLS 回归。

3.5.2 数值例子

尽管前面给出的公式是动态的，但是下面的数值例子将把 APT 转换为静态模型。假设与上一节一样的静态经济模式具有 3 种可能的未来状态，即 $\mathcal{M}^3 = (\{\Omega, \mathcal{F}, P\}, \mathbb{A})$。假设这两种风险资产现在是经济中的相关风险因素，并引入了第三种资产 V，其未来回报如下。

$$V_1 = \begin{bmatrix} 12 \\ 15 \\ 7 \end{bmatrix}$$

虽然两个线性无关的向量（比如 $\boldsymbol{S}_1$ 和 $\boldsymbol{T}_1$）不能构成 $\mathbb{R}^3$ 的基，但是它们可以用于 OLS 回归以近似回报 V_1。下面的 Python 代码实现了 OLS 回归。

```
In [97]: M1
Out[97]: array([[20,  1],
               [10, 12],
               [ 5, 13]])

In [98]: M0
Out[98]: array([10, 10])
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [99]: V1 = np.array((12, 15, 7))

In [100]: reg = np.linalg.lstsq(M1, V1, rcond=-1)[0]  ❶
          reg  ❶
Out[100]: array([0.6141665 , 0.50030531])

In [101]: np.dot(M1, reg)
Out[101]: array([12.78363525, 12.14532872, 9.57480155])

In [102]: np.dot(M1, reg) - V1  ❷
Out[102]: array([ 0.78363525, -2.85467128, 2.57480155])

In [103]: V0 = np.dot(M0, reg)  ❸
          V0  ❸
Out[103]: 11.144718094850402
```

❶ 最优回归参数可以解释为因子负荷。

❷ 这两个因素不足以解释回报 V_1。复制不完全，残差值不为 0。

❸ 因子负荷可用于估计风险资产 V 的无套利价格 V_0。

显然，这两个因素不足以完全解释回报 V_1。考虑到线性代数的标准结果，这并不奇怪。[10] 在模型经济中加入第三个风险因素 U 怎么样？假设第三个风险因素 U 的定义为 $U_0 = 10$，如下所示。

$$U_1 = \begin{bmatrix} 12 \\ 5 \\ 11 \end{bmatrix}$$

现在，这 3 个风险因素可以完全（准确地）解释（复制）回报 V_1。

```
In [104]: U0 = 10
          U1 = np.array((12, 5, 11))

In [105]: M0_ = np.array((S0, T0, U0))  ❶

In [106]: M1_ = np.concatenate((M1.T, np.array([U1,]))).T  ❷

In [107]: M1_  ❷
Out[107]: array([[20,  1, 12],
                 [10, 12,  5],
                 [ 5, 13, 11]])

In [108]: np.linalg.matrix_rank(M1_)  ❷
Out[108]: 3

In [109]: reg = np.linalg.lstsq(M1_, V1, rcond=-1)[0]
          reg
Out[109]: array([ 0.9575179 , 0.72553699, -0.65632458])
```

注 10：当然，回报 V_1 也可能恰巧落在由 $\boldsymbol{S}_1$ 和 $\boldsymbol{T}_1$ 两个因素所决定的区间。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [110]: np.allclose(np.dot(M1_, reg), V1)  ❸
Out[110]: True

In [111]: V0_ = np.dot(M0_, reg)
          V0_  ❹
Out[111]: 10.267303102625307
```

❶ 增扩市场价格向量。

❷ 增扩满秩市场回报矩阵。

❸ V_1 的精确复制，残差值为 0。

❹ 风险资产 V 的唯一无套利价格。

以上示例类似于 3.1 节中的例子，因为可以使用足够多的风险因素（可交易资产）来推导交易资产的无套利价格。APT 不一定要求完美复制，它的模型公式中包含了残值。然而，如果有可能进行完美复制，则剩余项为 0，就如前面的 3 个风险因素的例子所展示的那样。

3.6 结论

从 20 世纪 40 年代到 70 年代的早期理论和模型，特别是本章介绍的理论和模型，仍然是金融教科书的核心主题，并且依旧在金融实践中得到应用。一个原因是，这些大多是规范性的理论和模型，在智力启发方面对学生、学者和实践者具有强烈的吸引力。这些理论"似乎很有道理"，使用 Python 就能很容易地对模型的数值示例进行创建、分析和可视化。

尽管早期的 MVP 和 CAPM 等理论和模型在智力启发方面具有吸引力、易于实现且在数学上也很优雅，但它们今日仍然非常流行，这一点着实令人惊讶。之所以这样说，有以下几个原因。第一，本章所提出的理论和模型几乎没有任何有意义的实证支持。第二，一些理论和模型在许多方面，甚至在理论上相互不一致。第三，金融理论和模型前沿不断取得进展，可以提供替代理论和模型。第四，现代计算和实证金融学可以依赖几乎无限的数据源和几乎无限的计算能力，使得简洁、简约、优雅的数学模型和结果越来越不相关。

第 4 章在实际金融数据的基础上，分析了本章介绍的一些理论和模型。虽然在量化金融的早期，数据是一种稀缺资源，但如今，即使是学生也能获得丰富的金融数据和开源工具，从而能够根据真实数据对金融理论和模型进行全面分析。实证金融学一直是理论金融学的重要姊妹学科。然而，金融理论在很大程度上推动了实证金融学的发展。与金融中的数据相比，**数据驱动的金融学**新领域可能导致理论的相对重要性发生持久的转变。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 4 章

数据驱动的金融学

> 如果说人工智能是新的电力，那么大数据就是为发电机提供动力的石油。
>
> ——李开复，2018 年

> 如今，分析师对卫星图像和信用卡数据等非传统信息进行筛选，或者使用机器学习和自然语言处理等人工智能技术从经济数据和财报电话会议记录等传统来源获得新的见解。
>
> ——Robin Wigglesworth，2019 年

本章讨论了**数据驱动的金融学**的核心方面。在本书中，数据驱动的金融学被理解为主要由数据驱动并从数据中获得见解的金融学（理论、模型、应用等）。

4.1 节讨论了科学方法，其中给出了应该指导科学探索的一套公认的原则。4.2 节介绍了金融计量经济学和相关主题。4.3 节阐明了今天哪些类型的（金融）数据是可用的，以及通过可编程 API 可以获得的数据质量和数据量。4.4 节重新审视了第 3 章的规范性理论，并基于真实的金融时间序列数据进行了分析。4.5 节同样基于真实的金融数据，揭穿了金融模型和理论中最常见的两个假设：**正态分布收益率**和**线性关系**。

4.1 科学方法

科学方法是指指导任何科学项目的一套公认的原则，维基百科对科学方法的定义如下：

> 科学方法是一种获取知识的经验性方法，早在 17 世纪就成了科学发展的特征。它涉及仔细的观察，对观察结果应用严格的怀疑主义，因为认知假设会扭曲人们对观察结果的解释。它包括根据这些观察，通过归纳，提出假设；对从假设中得出的推论进行实验和测试；以及根据实验结果精炼（或消除）假设。这些是科学方法的原则，与适用于所有科学实体的一系列明确的应用步骤是不同的。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

根据这一定义，第 3 章讨论的规范性金融理论与科学方法形成了鲜明对比。规范性金融理论大多依赖于假设和公理，并结合**演绎**作为主要的分析方法，以得出其核心结果。

- 预期效用理论（EUT）**假设**，无论世界处于何种状态，经济主体都具有相同的效用函数，并且在不确定性条件下，它们使预期效用最大化。
- 均值 – 方差投资组合（MVP）理论描述了投资者应如何在不确定性条件下进行投资，**假设**在一个时期内仅计算投资组合的预期收益率和预期波动率。
- 资本资产定价模型（CAPM）**假设**只有不可分散的市场风险才能解释一个时期内股票的预期收益率和预期波动率。
- 套利定价理论（APT）**假设**多个可识别的风险因子可以解释股票的预期收益率和预期波动率。无可否认，与其他理论相比，APT 的表述更为宽泛，可作广泛的解释。

上述规范性金融理论的特点是，它们最初是在特定的假设和公理下，只用“笔和纸”推导出来的，而不依赖现实世界的数据或观察。从历史的角度来看，这些理论大多是在提出很久之后才用真实世界的数据或观察进行严格检验的。这主要可以通过更好的数据可用性和随着时间的推移增加的计算能力来解释。毕竟，数据和计算是统计方法在实践中应用的主要因素。将这些方法应用于金融市场数据的数学、统计学和金融学交叉学科通常称为**金融计量经济学**，这是下一节的主题。

4.2 金融计量经济学与回归

调整 Investopedia 对于**计量经济学**的定义，可以将**金融计量经济学**定义如下：

> 金融计量经济学是对统计模型和数学模型的定量应用，利用金融数据来发展金融理论或检验金融学中的现有假设，并根据历史数据预测未来趋势。它将现实世界的金融数据进行统计试验，然后将结果与金融理论或正在测试的理论进行比较和对比。

Alexander（2008b）对金融计量经济学领域进行了全面而广泛的介绍。该书的第 2 章涵盖了单因子模型和多因子模型，比如 CAPM 和 APT。Alexander（2008b）是 *Market Risk Analysis* 系列中的一本书。该系列共有 4 本书，其中的第一本书，Alexander（2008a），涵盖了理论背景、概念、主题和方法，比如 MVP 理论和 CAPM。Campbell（2018）是金融理论和相关计量研究的另一份综合资料。

金融计量经济学的主要工具之一是**回归**，包括单变量形式和多变量形式。一般来说，回归也是**统计学习**的核心工具。传统数学和统计学习的区别是什么？虽然这一问题没有一般性答案（毕竟，统计学是数学的一个子领域），但可以用一个简单的例子来强调与本书相关的主要区别。

第一是标准的数学方法。假设数学函数如下所示：

$$f:\mathbb{R}\to\mathbb{R}_+,\ x\mapsto 2+\frac{1}{2}x$$

给定 x_i 的多个值，$i=1, 2, \cdots, n$，则可以应用上述定义导出 f 的函数值：

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

$$y_i = f(x_i),\ i = 1,\ 2,\ \cdots,\ n$$

下面的 Python 代码基于一个简单的数值示例说明了这一点。

```
In [1]: import numpy as np

In [2]: def f(x):
            return 2 + 1 / 2 * x

In [3]: x = np.arange(-4, 5)
        x
Out[3]: array([-4, -3, -2, -1,  0,  1,  2,  3,  4])

In [4]: y = f(x)
        y
Out[4]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. ])
```

第二是统计学习中采用的方法。在前面的例子中，函数先出现，然后得到数据，但在统计学习中，这个顺序是相反的。在这里，一般给出数据，并找到一个函数关系。在这种情况下，x 通常称为**自变量**，y 称为**因变量**。因此，请考虑下列数据：

$$(x_i,\ y_i),\ i = 1,\ 2,\ \cdots,\ n$$

例如，问题是如何找到参数 $\alpha,\ \beta$ ：

$$\hat{f}(x_i) \equiv \alpha + \beta x_i = \hat{y}_i \approx y_i,\ i = 1,\ 2,\ \cdots,\ n$$

另一种写法包含残差值 $\epsilon_i,\ i = 1,\ 2,\ \cdots,\ n$ ：

$$\alpha + \beta x_i + \epsilon_i = y_i,\ i = 1,\ 2,\ \cdots,\ n$$

在 OLS 回归中，选择 $\alpha,\ \beta$ 系数以最小化近似值 $\hat{y}_i$ 和实际值 y_i 之间的均方误差。那么，最小化问题如下：

$$\min_{\alpha,\beta} \frac{1}{n} \sum_i^n (\hat{y}_i - y_i)^2$$

在**简单 OLS 回归**的情况下，如前所述，最优解是已知的封闭形式，如下所示：

$$\begin{cases} \beta = \dfrac{\mathrm{Cov}(x,y)}{\mathrm{Var}(x)} \\ \alpha = \overline{y} - \beta \overline{x} \end{cases}$$

这里，Cov() 代表**协方差**，Var() 代表**方差**，$\overline{x}, \overline{y}$ 代表 $x,\ y$ 的**平均值**。

回到前面的数值例子，这些见解可以用来推导最优参数 $\alpha,\ \beta$ ，在特殊情况下，可以复原 $f(x)$ 的原始定义。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [5]: x
Out[5]: array([-4, -3, -2, -1,  0,  1,  2,  3,  4])

In [6]: y
Out[6]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. ])

In [7]: beta = np.cov(x, y, ddof=0)[0, 1] / x.var()  ❶
        beta  ❶
Out[7]: 0.49999999999999994

In [8]: alpha = y.mean() - beta * x.mean()  ❷
        alpha  ❷
Out[8]: 2.0

In [9]: y_ = alpha + beta * x  ❸

In [10]: np.allclose(y_, y)  ❹
Out[10]: True
```

❶ β 由协方差矩阵和方差求出。

❷ α 由 β 和 x, y 的平均值推导而来。

❸ 给定 α 和 β，求出估计值 $\hat{y}_i, i=1, 2, \cdots, n$。

❹ 检查 $\hat{y}_i, y_i$ 值在数值上是否相等。

前面的例子和第 1 章中的例子说明了对给定数据集应用来说，OLS 回归通常是很直观的。OLS 回归之所以成为计量经济学和金融计量经济学的核心工具之一，还有更多的原因，如下所述。

数百年的历史
: 最小二乘方法，特别是与回归相结合，已经被使用了 200 多年。[1]

简单
: OLS 回归背后的数学原理很容易理解，在编程中也很容易实现。

可扩展性
: OLS 回归的数据量基本上没有限制。

灵活性
: OLS 回归可以应用于广泛的问题和数据集。

快速
: 即使是更大的数据集，OLS 回归也能快速评估。

可利用性
: Python 和许多其他编程语言的高效实现是唾手可得的。

注 1：请参阅 Kopf（2015）。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

然而，就像 OLS 回归一般的应用一样简单和直接，该方法依赖于许多假设，其中大多数与残差有关，而这些假设在实践中并不总是成立。

线性
: 对系数和残差来说，模型的参数是线性的。

独立性
: 自变量之间并不完全（高度）相关（没有**多重共线性**）。

零均值
: 残差的平均值是（或接近）零。

不相关
: 残差与自变量不（强）相关。

同方差性
: 残差的标准差（几乎）是恒定的。

无自相关
: 残差之间不存在（强）相关性。

在实践中，对于给定的数据集，检验这些假设的有效性通常相当简单。

4.3　数据可用性

金融计量经济学是由回归等统计方法和可用的金融数据驱动的。从 20 世纪 50 年代到 90 年代，甚至到 21 世纪初，理论和实证的金融研究与今天的标准相比，主要是由相对较小的数据集驱动的，并且由日终数据（EOD data）组成。在过去 10 年左右，数据可用性发生了巨大的变化，可用的金融数据类型和其他数据类型越来越多，其粒度越来越细，数量越来越大，速度也越来越快。

4.3.1　可编程API

就数据驱动的金融学而言，重要的不仅是可用的数据，还包括如何访问和处理这些数据。在相当长的一段时间里，金融专业人士一直依赖于 Refinitiv 或彭博等公司的数据终端，但这只是其中两家领先的提供商。报纸、杂志、财经报道等媒介早已被这种终端所取代，成为财经信息的主要来源。然而，这些终端所提供的海量和丰富的数据不能被单个用户甚至大批的金融专业人士系统地使用。因此，数据驱动的金融学要取得重大突破，关键在于通过应用程序接口（API）实现数据的**编程可用性**。API 允许使用计算机代码段选择、检索和处理任意数据集。

本节的剩余部分将专门介绍这些 API，通过这些 API，即使是学者和散户投资者也可以检索大量不同的数据集。在提供此类示例之前，表 4-1 概述了在金融环境中通常相关的数据类别以及典型示例。在该表中，**结构化数据**指的是通常以表格形式出现的数字数据类型，而**非结构化数据**指的是标准文本形式的数据，标准文本通常除了标题或段落之外没有其他结构。**非传统数据**是指通常不被视为金融数据的数据类型。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

表4-1：金融数据的相关类型

时间	结构化数据	非结构化数据	非传统数据
历史数据	价格、基本面	新闻、文本	网页、社交媒体、卫星电视
流数据	价格、成交量	新闻、文件	网页、社交媒体、卫星电视、物联网

4.3.2 结构化历史数据

首先，结构化历史数据类型将以编程方式检索。为此，下面的 Python 代码会使用 Eikon Data API。[2]

要通过 Eikon Data API 访问数据，必须运行一个本地应用，比如 Refinitiv Workspace，并且必须在 Python 中配置 API 访问。

```
In [11]: import eikon as ek
         import configparser

In [12]: c = configparser.ConfigParser()
         c.read('../aiif.cfg')
         ek.set_app_key(c['eikon']['app_id'])
         2020-08-04 10:30:18,059 P[14938] [MainThread 4521459136] Error on handshake
          port 9000 : ReadTimeout(ReadTimeout())
```

如果满足这些要求，则可以通过一个函数调用检索结构化历史数据。例如，下面的 Python 代码会检索一组符号和指定时间段的日终数据。

```
In [14]: symbols = ['AAPL.O', 'MSFT.O', 'NFLX.O', 'AMZN.O']  ❶

In [15]: data = ek.get_timeseries(symbols,
                                  fields='CLOSE',
                                  start_date='2019-07-01',
                                  end_date='2020-07-01')  ❷


In [16]: data.info()  ❸
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 254 entries, 2019-07-01 to 2020-07-01
         Data columns (total 4 columns):
           #  Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   AAPL.O  254 non-null    float64
          1   MSFT.O  254 non-null    float64
          2   NFLX.O  254 non-null    float64
          3   AMZN.O  254 non-null    float64
         dtypes: float64(4)
         memory usage: 9.9 KB

In [17]: data.tail()  ❹
Out[17]: CLOSE       AAPL.O  MSFT.O  NFLX.O  AMZN.O
         Date
```

注 2：该数据服务需要付费订阅。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
           2020-06-25  364.84  200.34  465.91  2754.58
           2020-06-26  353.63  196.33  443.40  2692.87
           2020-06-29  361.78  198.44  447.24  2680.38
           2020-06-30  364.80  203.51  455.04  2758.82
           2020-07-01  364.11  204.70  485.64  2878.70
```

❶ 定义要为其检索数据的 RIC（符号）列表。[3]

❷ 检索 RIC 列表的日终数据收盘价。

❸ 显示返回的 DataFrame 对象的元信息。

❹ 显示 DataFrame 对象的最后行。

类似地，带有 OHLC 字段的一分钟间隔数据可以通过适当调整参数来检索。

```
In [18]: data = ek.get_timeseries('AMZN.O',
                                  fields='*',
                                  start_date='2020-08-03',
                                  end_date='2020-08-04',
                                  interval='minute')  ❶

In [19]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 911 entries, 2020-08-03 08:01:00 to 2020-08-04 00:00:00
         Data columns (total 6 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   HIGH    911 non-null    float64
          1   LOW     911 non-null    float64
          2   OPEN    911 non-null    float64
          3   CLOSE   911 non-null    float64
          4   COUNT   911 non-null    float64
          5   VOLUME  911 non-null    float64
         dtypes: float64(6)
         memory usage: 49.8 KB

In [20]: data.head()
Out[20]: AMZN.O                  HIGH     LOW     OPEN    CLOSE  COUNT  VOLUME
         Date
         2020-08-03 08:01:00  3190.00  3176.03  3176.03  3178.17   18.0   383.0
         2020-08-03 08:02:00  3183.02  3176.03  3180.00  3177.01   15.0   513.0
         2020-08-03 08:03:00  3179.91  3177.05  3179.91  3177.05    5.0    14.0
         2020-08-03 08:04:00  3184.00  3179.91  3179.91  3184.00    8.0   102.0
         2020-08-03 08:05:00  3184.91  3182.91  3183.30  3184.00   12.0   403.0
```

❶ 检索包含单个 RIC 的所有可用字段的一分钟间隔数据。

我们可以从 Eikon 数据 API 中检索更多的结构化金融时间序列数据。此外，也可以同时检索多个 RIC 和多个数据字段的基础数据，如下面的 Python 代码所示。

```
In [21]: data_grid, err = ek.get_data(['AAPL.O', 'IBM', 'GOOG.O', 'AMZN.O'],
                                      ['TR.TotalReturnYTD', 'TR.WACCBeta',
```

注 3：RIC 代表 Reuters Instrument Code，即“路透社指令码”。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
                                      'YRHIGH', 'YRLOW',
                                      'TR.Ebitda', 'TR.GrossProfit'])  ❶

In [22]: data_grid
Out[22]:   Instrument  YTD Total Return      Beta   YRHIGH      YRLOW        EBITDA  \
         0     AAPL.O         49.141271  1.221249   425.66   192.5800  7.647700e+10
         1        IBM         -5.019570  1.208156   158.75    90.5600  1.898600e+10
         2     GOOG.O         10.278829  1.067084  1586.99  1013.5361  4.757900e+10
         3     AMZN.O         68.406897  1.338106  3344.29  1626.0318  3.025600e+10

            Gross Profit
         0   98392000000
         1   36488000000
         2   89961000000
         3  114986000000
```

❶ 检索多个 RIC 和多个数据字段的数据。

可编程数据可用性

现在基本上所有的结构化金融数据都可以通过编程的方式获得。在这种情形下，金融时间序列数据是最重要的例子。然而，其他结构化数据类型（如基础数据）也能以同样的方式使用，这大大简化了量化分析师、交易员、投资组合经理等人的工作。

4.3.3 结构化流数据

金融领域的许多应用需要实时的结构化数据，比如算法交易或市场风险管理。以下 Python 代码使用了 Oanda 交易平台的 API，获取以美元计价的金融工具价格的实时数据流，包括大量的时间戳、报价和要价报价：

```
In [23]: import tpqoa

In [24]: oa = tpqoa.tpqoa('../aiif.cfg')  ❶

In [25]: oa.stream_data('USD', stop=5)  ❷
         2020-08-04T08:30:38.621075583Z 11298.8 11334.8
         2020-08-04T08:30:50.485678488Z 11298.3 11334.3
         2020-08-04T08:30:50.801666847Z 11297.3 11333.3
         2020-08-04T08:30:51.326269990Z 11296.0 11332.0
         2020-08-04T08:30:54.423973431Z 11296.6 11332.6
```

❶ 连接到 Oanda API。

❷ 对给定的对象设定流化数据的固定刻度。

当然，打印出流数据字段只是为了说明。例如，某些金融应用可能需要对检索到的数据进行复杂的处理，并生成信号或进行统计。特别是在工作日和交易时段，金融工具的价格波动数量稳步增加，这要求金融机构具有强大的数据处理能力，因为它们需要实时或至少近实时（“近时间”）处理此类数据。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

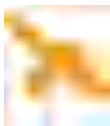
当观察 Apple 公司的股价时，这种观察的意义就变得很明显了。可以计算出，在 40 年的时间里，Apple 公司的股票大约有 252×40=10 080 个日终收盘报价。（Apple 公司于 1980 年 12 月 12 日上市。）下面的代码只检索一小时内 Apple 股票价格的**点数据**（tick data）。检索到的数据集（甚至可能在给定的时间段内都不完整）有 50 000 行，也就是在 40 年的交易中积累的日终数据的近 5 倍。

```
In [26]: data = ek.get_timeseries('AAPL.O',
                                  fields='*',
                                  start_date='2020-08-03 15:00:00',
                                  end_date='2020-08-03 16:00:00',
                                  interval='tick')  ❶

In [27]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 50000 entries, 2020-08-03 15:26:24.889000 to 2020-08-03
          15:59:59.762000
         Data columns (total 2 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   VALUE   49953 non-null  float64
          1   VOLUME  50000 non-null  float64
         dtypes: float64(2)
         memory usage: 1.1 MB

In [28]: data.head()
Out[28]: AAPL.O                    VALUE  VOLUME
         Date
         2020-08-03 15:26:24.889  439.06   175.0
         2020-08-03 15:26:24.889  439.08     3.0
         2020-08-03 15:26:24.890  439.08   100.0
         2020-08-03 15:26:24.890  439.08     5.0
         2020-08-03 15:26:24.899  439.10    35.0
```

❶ 检索 Apple 股票价格的点数据。



日终数据与点数据

时至今日仍在应用的大多数金融理论起源于日终数据是唯一可用的金融数据类型的时代。今天，金融机构，甚至是散户投资者，都面临着无穷无尽的实时数据流。Apple 股票的例子表明，对一只股票来说，在交易时间的一小时内，可能有 4 倍于过去 40 年积累的日终数据。这不仅对金融市场的参与者提出了挑战，也让人怀疑现有的金融理论能否适用于这样的数据环境。

4.3.4 非结构化历史数据

金融领域的许多重要数据源只提供非结构化数据，比如金融新闻或公司文件。毫无疑问，机器在处理大量结构化数字数据方面要比人类更好、更快。然而，**自然语言处理**（NLP）方面的最新进展也让机器在处理金融新闻方面变得更好、更快。2020 年，数据服务提供商每天要接收约 150 万篇新闻文章。很明显，人类无法正确处理如此庞大的基于文本的数据。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

幸运的是，现在非结构化数据在很大程度上也可以通过可编程 API 获得。下面的 Python 代码会从 Eikon 数据 API 中检索大量与特斯拉公司及其产品相关的新闻文章。选择一篇文章并完整展示。

```
In [29]: news = ek.get_news_headlines('R:TSLA.O PRODUCTION',
                                  date_from='2020-06-01',
                                  date_to='2020-08-01',
                                  count=7
                                 )  ❶

In [30]: news
Out[30]:                                           versionCreated  \
         2020-07-29 11:02:31.276 2020-07-29 11:02:31.276000+00:00
         2020-07-28 00:59:48.000        2020-07-28 00:59:48+00:00
         2020-07-23 21:20:36.090 2020-07-23 21:20:36.090000+00:00
         2020-07-23 08:22:17.000        2020-07-23 08:22:17+00:00
         2020-07-23 07:08:48.000        2020-07-23 07:46:56+00:00
         2020-07-23 00:55:54.000        2020-07-23 00:55:54+00:00
         2020-07-22 21:35:42.640 2020-07-22 22:13:26.597000+00:00

                                                                     text  \
         2020-07-29 11:02:31.276  Tesla Launches Hiring Spree in China as It Pre...
         2020-07-28 00:59:48.000    Tesla hiring in Shanghai as production ramps up
         2020-07-23 21:20:36.090     Tesla speeds up Model 3 production in Shanghai
         2020-07-23 08:22:17.000                               UPDATE 1-Musk urges...
         2020-07-23 07:08:48.000                               Musk urges as Tesla...
         2020-07-23 00:55:54.000  USA-Tesla choisit le Texas pour la production ...
         2020-07-22 21:35:42.640  TESLA INC - THE REAL LIMITATION ON TESLA GROWT...

                                                                  storyId  \
         2020-07-29 11:02:31.276  urn:newsml:reuters.com:20200729:nCXG3W8s9X:1
         2020-07-28 00:59:48.000  urn:newsml:reuters.com:20200728:nL3N2EY3PG:8
         2020-07-23 21:20:36.090  urn:newsml:reuters.com:20200723:nNRAcf1v8f:1
         2020-07-23 08:22:17.000  urn:newsml:reuters.com:20200723:nL3N2EU1P9:1
         2020-07-23 07:08:48.000  urn:newsml:reuters.com:20200723:nL3N2EU0HH:1
         2020-07-23 00:55:54.000  urn:newsml:reuters.com:20200723:nL5N2EU03M:1
         2020-07-22 21:35:42.640  urn:newsml:reuters.com:20200722:nFWN2ET120:2

                                 sourceCode
         2020-07-29 11:02:31.276  NS:CAIXIN
         2020-07-28 00:59:48.000    NS:RTRS
         2020-07-23 21:20:36.090  NS:SOUTHC
         2020-07-23 08:22:17.000    NS:RTRS
         2020-07-23 07:08:48.000    NS:RTRS
         2020-07-23 00:55:54.000    NS:RTRS
         2020-07-22 21:35:42.640    NS:RTRS

In [31]: storyId = news['storyId'][1]  ❷

In [32]: from IPython.display import HTML

In [33]: HTML(ek.get_news_story(storyId)[:1148])  ❸
Out[33]: <IPython.core.display.HTML object>
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
Jan 06, 2020

Tesla, Inc.TSLA registered record production and deliveries of 104,891 and
112,000 vehicles, respectively, in the fourth quarter of 2019.

Notably, the company's Model S/X and Model 3 reported record production and
deliveries in the fourth quarter. The Model S/X division recorded production
and delivery volume of 17,933 and 19,450 vehicles, respectively. The Model 3
division registered production of 86,958 vehicles, while 92,550 vehicles were
delivered.

In 2019, Tesla delivered 367,500 vehicles, reflecting an increase of 50%, year
over year, and nearly in line with the company's full-year guidance of 360,000
vehicles.
```

❶ 检索参数范围内的新闻文章的元数据。

❷ 选择一个要检索其全文的 `storyId`。

❸ 检索所选文章的全文并显示它。

4.3.5 非结构化流数据

与检索非结构化历史数据的方式相同，可编程 API 可以用于非结构化新闻数据流，比如实时或至少近实时传输。一个这样的 API 可用于道琼斯的 DNA 平台（Data, News and Analytics platform）。图 4-1 显示了一个 Web 应用的截图，该应用流式传输“商品和金融新闻”的文章，并使用 NLP 技术实时处理这些文章。

图 4-1：基于道琼斯 DNA 平台的新闻流应用

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

新闻流应用具有以下主要特点。

全文
　每篇文章的全文可通过点击文章标题获得。

关键字摘要
　创建关键字摘要并打印在屏幕上。

情绪分析
　情绪得分被计算出来并显示为彩色箭头。通过单击箭头可以看到详细信息。

词云
　可创建词云摘要位图，显示为缩略图，单击缩略图后可见（参见图 4-2）。

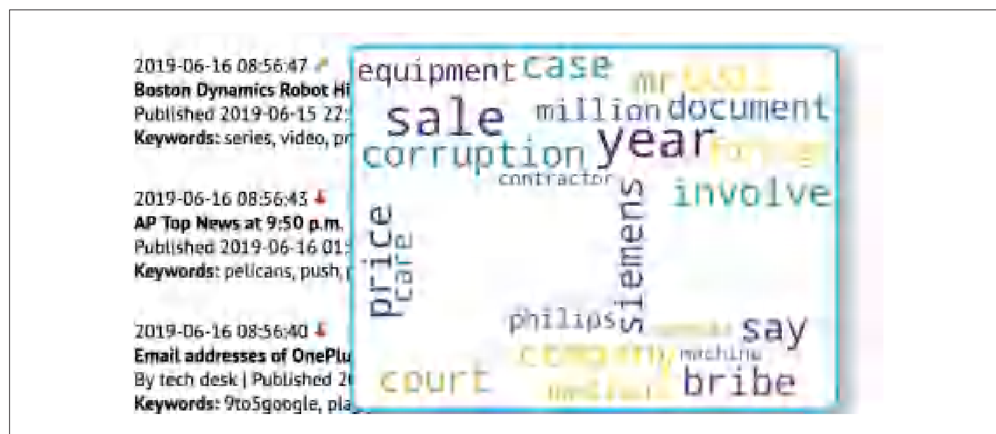


图 4-2：新闻流应用中显示的词云位图

4.3.6　非传统数据

如今，金融机构（特别是对冲基金）系统地挖掘了许多非传统的数据来源，以在交易和投资方面获得优势。彭博社最近的一篇文章列出了以下非传统的数据来源。

- 网页爬取数据
- 众包数据
- 信用卡和销售点（POS）系统
- 社交媒体情绪
- 搜索趋势
- Web 流量
- 供应链数据
- 能源生产数据
- 消费者资料
- 卫星图像 / 地球空间数据
- 应用安装

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

- 远洋船舶跟踪
- 可穿戴设备、无人机、物联网传感器

下面用两个例子说明非传统数据的用法。第一个例子以 HTML 页面的形式检索和处理 Apple 公司的新闻稿。下面的 Python 代码使用了一组辅助函数（参见 4.7 节）。这段代码定义了一个 URL 列表，每个 URL 代表 Apple 公司新闻稿的 HTML 页面。然后为每份新闻稿检索原始 HTML 代码段。最后清理原始代码段，并打印出一份新闻稿的摘录。

```
In [34]: import nlp  ❶
         import requests

In [35]: sources = [
             'https://exampleurl1',  # iPad Pro
             'https://exampleurl2',  # MacBook Air
             'https://exampleurl3',  # Mac Mini
         ]  ❷

In [36]: html = [requests.get(url).text for url in sources]  ❸

In [37]: data = [nlp.clean_up_text(t) for t in html]  ❹

In [38]: data[0][536:1001]  ❺
Out[38]: ' display, powerful a12x bionic chip and face id introducing the new ipad pro
          with all-screen design and next-generation performance. new york apple today
          introduced the new ipad pro with all-screen design and next-generation
          performance, marking the biggest change to ipad ever. the all-new design
          pushes 11-inch and 12.9-inch liquid retina displays to the edges of ipad pro
          and integrates face id to securely unlock ipad with just a glance.1 the a12x
          bionic chip w'
```

❶ 导入 NLP 辅助函数。

❷ 定义 3 份新闻稿的 URL。

❸ 检索 3 份新闻稿的原始 HTML 代码段。

❹ 清除原始 HTML 代码段（比如，删除 HTML 标记）。

❺ 打印一份新闻稿的摘录。

当然，像在本节中所做的那样广泛地定义非传统数据意味着人们可以检索和处理无限多的数据以满足财务目的。其核心是搜索引擎业务，比如谷歌的业务。在金融环境中，最重要的是明确指定要使用哪些非结构化且非传统的数据源。

第二个例子是关于从社交网络公司 Twitter 检索数据。Twitter 在其平台上提供对推文的 API 访问，前提是已经适当地设置了 Twitter 账户。下面的 Python 代码会连接到 Twitter 的 API，并分别从我的主时间线和用户时间线检索并打印最近的 5 条推文。

```
In [39]: from twitter import Twitter, OAuth

In [40]: t = Twitter(auth=OAuth(c['twitter']['access_token'],
                                c['twitter']['access_secret_token'],
                                c['twitter']['api_key'],
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
                              c['twitter']['api_secret_key']),
                    retry=True)  ❶

In [41]: l = t.statuses.home_timeline(count=5)  ❷

In [42]: for e in l:
             print(e['text'])  ❷
         The Bank of England is effectively subsidizing polluting industries in its
          pandemic rescue program, a think tank sa...
         Cool shared task: mining scientific contributions (by @SeeTedTalk @SoerenAuer
          and Jennifer D'Souza)
         Twelve people were hospitalized in Wyoming on Monday after a hot air balloon
          crash, officials said.

         Three hot air...
          Company announcement: Revolut launches Open Banking for its 400,000
           Italian...
          #fintech

In [43]: l = t.statuses.user_timeline(screen_name='dyjh', count=5)  ❸

In [44]: for e in l:
             print(e['text'])  ❸
         #Python for #AlgoTrading (focus on the process) &amp; #AI in #Finance (focus
          on prediction methods) will complement eac...
         Currently putting finishing touches on #AI in #Finance (@OReillyMedia). Book
          going into production shortly.
         Chinatown Is Coming Back, One Noodle at a Time
         Alt data industry balloons as hedge funds strive for edge via @FT |
         "We remain of the view that alternative d...
         He just follow me on Twitter (or LinkedIn). Then you will notice for
          sure when it is out.
```

❶ 连接到 Twitter API。

❷ 从我的主时间线检索并打印 5 条（最近的）推文。

❸ 从用户时间线检索并打印 5 条（最近的）推文。

Twitter API 还允许搜索，根据搜索可以检索和处理最近的推文。

```
In [45]: d = t.search.tweets(q='#Python', count=7)  ❶

In [46]: for e in d['statuses']:
             print(e['text'])  ❶
          RT @KirkDBorne: #AI is Reshaping Programming - Tips on How to Stay on Top:
          —
          Courses:
          1: #MachineLearning - Jupyte...
          RT @reuvenmlerner: Today, a #Python student's code didn't print:

          x = 5
          if x == 5:
              print: ('yes!')
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
There was a typo, namely : after pr...
RT @GavLaaaaaaaa: Javascript Does Not Need a StringBuilder
 #programming #softwareengineering #bigdata
 #datascience...
RT @CodeFlawCo: It is necessary to publish regular updates on Twitter
 #programmer #coder #developer #technology RT @pak_aims: Learning to C...
RT @GavLaaaaaaaa: Javascript Does Not Need a StringBuilder
 #programming #softwareengineering #bigdata
 #datascience...
```

❶ 搜索带有“Python”标签的推文，并打印出最近的 5 条。

还可以收集某个 Twitter 用户的大量推文，并以词云的形式创建摘要（参见图 4-3）。下面的 Python 代码再次使用了 NLP 辅助函数，如 4.7 节中的 Python 代码所示。

```
In [47]: l = t.statuses.user_timeline(screen_name='elonmusk', count=50)  ❶

In [48]: tl = [e['text'] for e in l]  ❷

In [49]: tl[:5]  ❸
Out[49]: ['@flcnhvy @Lindw0rm @cleantechnica True',
          '@Lindw0rm @cleantechnica Highly likely down the road',
          '@cleantechnica True fact',
         '@NASASpaceflight Scrubbed for the day. A Raptor turbopump spin start valve
          didn’t open, triggering an automatic abo...
         '@Erdayastronaut I’m in the Boca control room. Hop attempt in ~33 minutes.']

In [50]: wc = nlp.generate_word_cloud(' '.join(tl), 35,
                     name='../../images/ch04/musk_twitter_wc.png'
                     )  ❹
```

❶ 检索用户 `elonmusk` 的最近 50 条推文。

❷ 收集 `list` 对象中的文本。

❸ 显示最后 5 条推文的摘录。

❹ 生成词云摘要并显示它。

图 4-3：大量推文的词云摘要

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

一旦一个金融从业者定义了“相关金融数据”以超越结构化金融时间序列数据，数据来源在数量、多样性和速度方面看起来似乎就是无限的。从 Twitter API 检索推文的方式几乎是即时的，因为在示例中访问了最近的推文。因此，这些以及类似的基于 API 的数据源提供了无休止的非传统数据流。正如前面所指出的，重要的是要准确地指定所需的数据。否则，任何金融数据科学工作都可能很容易淹没在太多的数据或太嘈杂的数据中。

4.4　重新审视规范性理论

第 3 章介绍了规范性金融理论，比如 MVP 理论和 CAPM。在相当长的一段时间里，学生和学者或多或少地受限于学习和研究这些理论本身。如上一节所讨论和说明的，有了所有可用的金融数据，再加上强大的开源数据分析软件（比如 Python、NumPy、pandas 等），将金融理论用于现实世界的检验变得非常简单和直接。这样做不再需要小团队和大研究。一个典型的笔记本、互联网连接和标准的 Python 环境就足够了。这就是本节的内容。然而，在深入研究数据驱动的金融学之前，需要简要讨论 EUT 背景下的一些著名悖论，以及企业如何在实践中建模和预测个人行为。

4.4.1　预期效用与现实

在经济学中，**风险**描述了决策者提前知道未来可能出现的状态及其出现概率的一种情况。这是金融学和 EUT 的标准假设。相反，**模糊性**描述了决策者事先不知道的概率，甚至不知道可能的未来状态的情形。**不确定性**包含了这两种不同的决策情况。

在不确定性条件下，分析个体（“经济主体”）的具体决策行为有着悠久的传统。与 EUT 等理论所预测的不同，人们已经进行了无数的研究和实验来观察和分析经济主体在面对不确定性时的行为。几个世纪以来，**悖论**在决策理论和研究中发挥了重要作用。

其中一个悖论，即**圣彼得堡悖论**，催生了效用函数和 EUT 的诞生。Daniel Bernoulli 在 1738 年提出了这个悖论，并给出了解决方案。这个悖论基于下面的《抛硬币》游戏 G。经济主体面临这样一个游戏：一枚（完美的）硬币可能被抛无数次。如果第一次抛硬币后正面朝上，那么该主体将获得 1 货币单位的回报。只要看到正面朝上，该主体就会再抛一次硬币，否则游戏结束。如果第二次仍然正面朝上，那么该主体将额外获得 2 货币单位的回报。如果第三次还是正面朝上，那么额外回报是 4 货币单位，第四次是 8 货币单位，以此类推。这是一种风险情况，因为所有可能的未来状态及其相关的概率都是预先知道的。

这个游戏的期望回报是**无限**的。这可以从下面的无限和中看出，其中的每一个元素都是严格为正的：

$$\mathbf{E}(G)=\frac{1}{2}\times 1+\frac{1}{4}\times 2+\frac{1}{8}\times 4+\frac{1}{16}\times 8+\cdots=\sum_{k=1}^{\infty}\frac{1}{2^k}2^{k-1}=\sum_{k=1}^{\infty}\frac{1}{2}=\infty$$

然而，面对这样的游戏，决策者通常只愿意花**有限**的钱来玩。这样做的一个主要原因是，相对较大的回报只有在相对较小的概率下才会发生。考虑潜在回报 $W=511$：

$$W=1+2+4+8+16+32+64+128+256=511$$

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

赢得这种回报的概率很低。确切地说，这只有 $P(x=W)=\frac{1}{512}=0.001\,953\,125$ 。另外，获得这种回报或更小回报的概率却是相当高的：

$$P(x \leqslant W)=\sum_{k=1}^{9}\frac{1}{2^k}=0.998\,046\,875$$

换句话说，1000 场游戏中有 998 场的回报是 511 或更小。因此，一个经济主体可能不会下超过 511 的赌注来玩这个游戏。摆脱这个悖论的方法是引入一个**边际效用为正但递减**的效用函数。在圣彼得堡悖论中，这意味着存在一个函数 $u:\mathbb{R}_+ \to \mathbb{R}$，它赋予每个正的回报 x 一个实数值 $u(x)$。正但递减的边际效用为如下形式：

$$\frac{\partial u}{\partial x}>0$$
$$\frac{\partial^2 u}{\partial x^2}<0$$

如第 3 章所述，其中一个候选函数是 $u(x)=\ln(x)$，具有：

$$\frac{\partial u}{\partial x}=\frac{1}{x}$$
$$\frac{\partial^2 u}{\partial x^2}=-\frac{1}{x^2}$$

那么，预期效用是**有限**的，如下面的无限和的计算所示：

$$\mathbf{E}(u(G))=\sum_{k=1}^{\infty}\frac{1}{2^k}u(2^{k-1})=\sum_{k=1}^{\infty}\frac{\ln(2^{k-1})}{2^k}=\left(\sum_{k=1}^{\infty}\frac{(k-1)}{2^k}\right)\times\ln(2)=\ln(2)<\infty$$

与无穷大的预期回报相比，$\ln(2)=0.693\,147$ 这一预期效用显然是一个非常小的数字。伯努利效用函数和 EUT 解决了圣彼得堡悖论。

其他悖论，比如 Allais（1953）提出的**阿莱悖论**，解决了 EUT 本身。这个悖论基于有 4 个不同游戏的一个实验，在该实验中，应该对测试对象进行排名。表 4-2 显示了 4 个游戏（A、B、A'、B'）。对 (A, B) 和 (A', B') 进行排序。**独立性公理**假设表中的第一行对 (A', B') 的排序没有任何影响，因为两个博弈的回报是相同的。

表4-2：阿莱悖论中的博弈

概率	游戏A	游戏B	游戏A'	游戏B'
0.66	2400	2400	0	0
0.33	2500	2400	2500	2400
0.01	0	2400	0	2400

在实验中，大多数决策者对博弈的排序如下：$B \succ A$ 和 $A' \succ B'$。排序 $B \succ A$ 会导致以下不等式，其中 $u_1 \equiv u(2400), u_2 \equiv u(2500), u_3 \equiv u(0)$：

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

$$u_1 > 0.66 \times u_1 + 0.33 \times u_2 + 0.01 \times u_3$$
$$0.34 \times u_1 > 0.33 \times u_2 + 0.01 \times u_3$$

排序 $A' \succ B'$ 进而导致以下不等式：

$$0.33 \times u_2 + 0.01 \times u_3 > 0.33 \times u_1 + 0.01 \times u_1$$
$$0.34 \times u_1 < 0.33 \times u_2 + 0.01 \times u_3$$

这些不等式显然相互矛盾，并导致了阿莱悖论。一种可能的解释是，一般情形下决策者对价值确定性要求高于典型模型（如 EUT）的预测。虽然在 EUT 下有许多可用的效用函数会让决策者选择游戏而不是确定性回报，但大多数人可能宁愿选择确定获得 100 万美元，而不是玩一个可以赢得 1 亿美元的可能性为 5% 的游戏。

另一种解释在于决策**框架**和决策者的心理。众所周知，接受“95% 成功概率”手术的人比接受“5% 死亡概率”手术的人要多。简单地改变措辞可能会导致与 EUT 等决策理论不一致的行为。

根据 Savage（1954，1972）的说法，另一个弥补 EUT 主观形式缺陷的著名悖论是**埃尔斯伯格悖论**，这可以追溯到埃尔斯伯格（1961）的开创性论文。它强调了模糊性在许多现实世界决策场景中的重要性。这一悖论的标准设置包括两个瓮，其中都正好有 100 个球。对于瓮 1，已知它正好包含 50 个黑球和 50 个红球。而对于瓮 2，只知道它包含黑球和红球，但不知道比例。

测试对象可在以下游戏选项中选择。

- 第 1 场：红 1，黑 1，或无差别
- 第 2 场：红 2，黑 2，或无差别
- 第 3 场：红 1，红 2，或无差别
- 第 4 场：黑 1，黑 2，或无差别

例如，这里的“红 1”表示从瓮 1 中抽出一个红球。通常情况下，受试者应回答如下。

- 第 1 场：无差别
- 第 2 场：无差别
- 第 3 场：红 1
- 第 4 场：黑 1

这一系列的决定（这不是唯一被观察到的，却是常见的）例证了所谓的**模糊的厌恶**。由于瓮 2 中黑球和红球出现的概率未知，因此决策者更喜欢有**风险**的情况，而不是具有**模糊性**的情况。

阿莱悖论和埃尔斯伯格悖论表明，真实的测试对象行为经常与经济学中成熟的决策理论所预测的相反。换句话说，作为决策者的人类一般不能与那些仔细收集数据，然后在不确定的情况下（无论是以风险的形式还是以模糊的形式）对数据进行处理以做出决策的机器相比。作为决策者，人类的行为比目前大多数（如果不是全部的话）理论所说的要复杂得多。举例来说，如果读过 Sapolsky（2018）的 800 页的《行为》一书，就知道解释人类行

专为量化交易设计的实时行情数据API: www.alltick.co

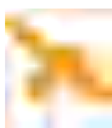
专为量化交易设计的实时行情数据API: www.alltick.co

为有多困难和复杂了。它以一种整合的方式涵盖了这个主题的多个方面，从生物化学过程到遗传学、人类进化、部落、语言、宗教等。

如果像 EUT 这样的标准经济决策范式不能很好地解释现实世界的决策过程，那么有什么替代方案呢？为阿莱悖论和埃尔斯伯格悖论奠定基础的经济实验，是学习决策者在特定、受控的情境下如何表现的良好起点。这样的实验及其有时令人惊讶又自相矛盾的结果确实激励了大量研究人员提出解决悖论的替代理论和模型。Fontaine 和 Leonard（2005）合著的 *The Experiment in the History of Economics* 一书讲述了实验在经济学中的历史作用。例如，一系列的文献都在讨论埃尔斯伯格悖论引发的问题。至于其他主题，该书涉及非可加性概率、Choquet 积分和决策启发法，比如**最大化最小回报**（max-min）或**最小化最大损失**（min-max）。至少在某些决策场景中，这些替代方法已被证明优于 EUT。但它们还远未成为金融领域的主流。

什么在实践中被证明是有用的？毫不奇怪，答案就在**数据和机器学习算法**中。拥有数十亿用户的互联网产生了描述人类真实行为的数据宝库，这有时也被称为“显示偏好”。互联网产生的大数据规模比单个实验所能产生的数据规模要大好几个数量级。像亚马逊、Facebook、谷歌和 Twitter 这样的公司能够通过记录用户行为（也就是他们显示的偏好）并利用基于这些数据训练的机器学习算法所产生的见解而赚取数十亿美元。

在这种情况下，默认的机器学习方法是监督学习。这些算法本身是**无理论和模型基础**的，神经网络的各种变体经常被应用。因此，如今公司在预测用户或客户行为时，通常会部署一个无模型的机器学习算法。EUT 或其后继理论等传统的决策理论一般都不起作用。不过让人有些惊讶的是，在 21 世纪 20 年代初，这些理论仍然是大多数应用于实践的经济理论和金融理论的基石，更不用说详细介绍传统决策理论的大量金融教科书了。如果金融理论的一个最基本的组成部分缺乏有意义的经验支持或实际的益处，那么建立在它之上的金融模型会如何呢？有关这方面的更多内容，请参见后续章节。



数据驱动的行为预测

一方面，标准的经济决策理论在智力上对许多人（包括那些在不确定性下具体决策的人）很有吸引力，但他们的行为与理论的预测相反。另一方面，大数据和无模型的监督学习方法在预测用户和客户行为方面被证明是既有效又成功的。在金融环境中，这可能意味着人们不应该真正担心金融主体为什么以及如何做出决策。人们更应该关注基于描述金融市场状态的特征数据（新信息）和反映金融主体决策影响的标签数据（结果）的间接显示偏好。这就使得金融市场决策由数据驱动，而不是由理论或模型驱动。金融主体成了数据处理的有机体，例如，与将简单的效用函数与假设概率分布相结合相比，通过复杂的神经网络可以更好地建模。

4.4.2　均值–方差投资组合理论

假设一位数据驱动型投资者希望将 MVP 理论应用于科技股投资组合，并希望增加一只与黄金相关的交易所交易基金（ETF）以实现多元化。或许，投资者可以通过 API 访问交易

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

平台或数据提供商的相关历史价格数据。为了使以下分析可重现，它依赖于远程存储的 CSV 数据文件。下面的 Python 代码会检索数据文件，根据投资者的目标选择一系列字段，并从价格时间序列数据中计算对数收益率。图 4-4 比较了所选字段的标准化价格时间序列。

```
In [51]: import numpy as np
         import pandas as pd
         from pylab import plt, mpl
         from scipy.optimize import minimize
         plt.style.use('seaborn')
         mpl.rcParams['savefig.dpi'] = 300
         mpl.rcParams['font.family'] = 'serif'
         np.set_printoptions(precision=5, suppress=True,
                             formatter={'float': lambda x: f'{x:6.3f}'})

In [52]: url = 'http://hilpisch.com/aiif_eikon_eod_data.csv'  ❶

In [53]: raw = pd.read_csv(url, index_col=0, parse_dates=True).dropna()  ❶

In [54]: raw.info()  ❶
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2516 entries, 2010-01-04 to 2019-12-31
         Data columns (total 12 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   AAPL.O  2516 non-null   float64
          1   MSFT.O  2516 non-null   float64
          2   INTC.O  2516 non-null   float64
          3   AMZN.O  2516 non-null   float64
          4   GS.N    2516 non-null   float64
          5   SPY     2516 non-null   float64
          6   .SPX    2516 non-null   float64
          7   .VIX    2516 non-null   float64
          8   EUR=    2516 non-null   float64
          9   XAU=    2516 non-null   float64
          10  GDX     2516 non-null   float64
          11  GLD     2516 non-null   float64
         dtypes: float64(12)
         memory usage: 255.5 KB

In [55]: symbols = ['AAPL.O', 'MSFT.O', 'INTC.O', 'AMZN.O', 'GLD']  ❷

In [56]: rets = np.log(raw[symbols] / raw[symbols].shift(1)).dropna()  ❸

In [57]: (raw[symbols] / raw[symbols].iloc[0]).plot(figsize=(10, 6));  ❹
```

❶ 从远程位置检索历史日终数据。

❷ 指定要投资的字段（RIC）。

❸ 计算所有时间序列的对数收益率。

❹ 绘制选定字段的标准化金融时间序列。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

图 4-4：标准化金融时间序列数据

数据驱动型投资者希望首先设定一条业绩基准线，即在整个可得数据时间段内，由一个同等权重的投资组合给出的业绩。为此，下面的 Python 代码定义了一些函数来计算投资组合收益率、投资组合波动率和给定一组权重的投资组合夏普比率。

```
In [58]: weights = len(rets.columns) * [1 / len(rets.columns)]  ❶

In [59]: def port_return(rets, weights):
             return np.dot(rets.mean(), weights) * 252  ❷

In [60]: port_return(rets, weights)  ❷
Out[60]: 0.15694764653018106

In [61]: def port_volatility(rets, weights):
             return np.dot(weights, np.dot(rets.cov() * 252 , weights)) ** 0.5  ❸

In [62]: port_volatility(rets, weights)  ❸
Out[62]: 0.16106507848480675

In [63]: def port_sharpe(rets, weights):
             return port_return(rets, weights) / port_volatility(rets, weights)  ❹

In [64]: port_sharpe(rets, weights)  ❹
Out[64]: 0.97443622172255
```

❶ 等权重投资组合。

❷ 投资组合收益率。

❸ 投资组合波动率。

❹ 投资组合夏普比率（短期利率为零）。

投资者还希望通过蒙特卡罗模拟来随机化投资组合的权重，以分析投资组合的风险和收益，从而分析夏普比率。排除卖空，并假设投资组合的权重之和为 100%。下面的 Python

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

代码实现了模拟并可视化了结果（参见图 4-5）。

```
In [65]: w = np.random.random((1000, len(symbols)))  ❶
         w = (w.T / w.sum(axis=1)).T  ❶

In [66]: w[:5]  ❶
Out[66]: array([[ 0.184, 0.157, 0.227, 0.353, 0.079],
                [ 0.207, 0.282, 0.258, 0.023, 0.230],
                [ 0.313, 0.284, 0.051, 0.340, 0.012],
                [ 0.238, 0.181, 0.145, 0.191, 0.245],
                [ 0.246, 0.256, 0.315, 0.181, 0.002]])

In [67]: pvr = [(port_volatility(rets[symbols], weights),
                 port_return(rets[symbols], weights))
                for weights in w]  ❷
         pvr = np.array(pvr)  ❷

In [68]: psr = pvr[:, 1] / pvr[:, 0]  ❸

In [69]: plt.figure(figsize=(10, 6))
         fig = plt.scatter(pvr[:, 0], pvr[:, 1],
                           c=psr, cmap='coolwarm')
         cb = plt.colorbar(fig)
         cb.set_label('Sharpe ratio')
         plt.xlabel('expected volatility')
         plt.ylabel('expected return')
         plt.title(' | '.join(symbols));
```

❶ 模拟投资组合权重之和为 100%。

❷ 得出投资组合波动率和收益率。

❸ 计算得到的夏普比率。

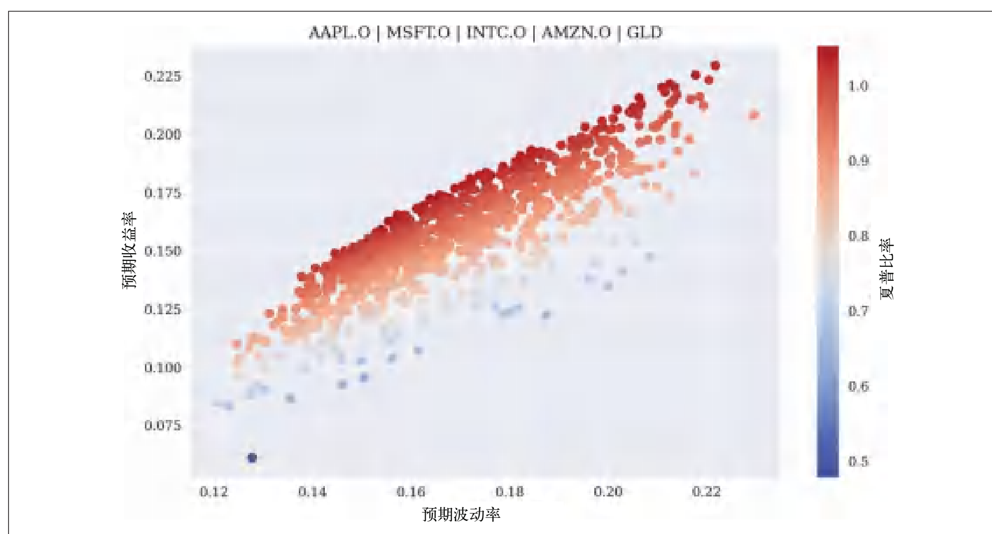


图 4-5：模拟投资组合波动率、收益率和夏普比率

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

数据驱动型投资者现在想对 2011 年年初创建的投资组合的业绩进行回测。最佳投资组合来源于 2010 年可用的金融时间序列数据。2012 年年初，根据 2011 年可用的数据调整投资组合构成，以此类推。为此，下面的 Python 代码导出了**使夏普比率最大化**的每一个相关年份的**投资组合权重**。

```
In [70]: bnds = len(symbols) * [(0, 1),]  ❶
         bnds  ❶
Out[70]: [(0, 1), (0, 1), (0, 1), (0, 1), (0, 1)]

In [71]: cons = {'type': 'eq', 'fun': lambda weights: weights.sum() - 1}  ❷

In [72]: opt_weights = {}
         for year in range(2010, 2019):
             rets_ = rets[symbols].loc[f'{year}-01-01':f'{year}-12-31']  ❸
             ow = minimize(lambda weights: -port_sharpe(rets_, weights),
                           len(symbols) * [1 / len(symbols)],
                           bounds=bnds,
                           constraints=cons)['x']  ❹
             opt_weights[year] = ow  ❺

In [73]: opt_weights  ❺
Out[73]: {2010: array([ 0.366,  0.000,  0.000,  0.056,  0.578]),
          2011: array([ 0.543,  0.000,  0.077,  0.000,  0.380]),
          2012: array([ 0.324,  0.000,  0.000,  0.471,  0.205]),
          2013: array([ 0.012,  0.305,  0.219,  0.464,  0.000]),
          2014: array([ 0.452,  0.115,  0.419,  0.000,  0.015]),
          2015: array([ 0.000,  0.000,  0.000,  1.000,  0.000]),
          2016: array([ 0.150,  0.260,  0.000,  0.058,  0.533]),
          2017: array([ 0.231,  0.203,  0.031,  0.109,  0.426]),
          2018: array([ 0.000,  0.295,  0.000,  0.705,  0.000])}
```

❶ 指定单个资产权重的边界。

❷ 指定所有权重之和为 100%。

❸ 选择给定年份的相关数据集。

❹ 得出使夏普比率最大化的投资组合权重。

❺ 将这些权重存储在 `dict` 对象中。

根据相关年份得出的最优投资组合构成表明，原始形式的 MVP 理论经常导致（相对）极端的情况，即要么一项或多项资产根本不包括在内，要么一项资产占投资组合的 100%。当然，这可以通过为每个资产设置最小权重来避免。结果还表明，这种方法会导致投资组合的重大再平衡，这是由上一年已经实现的统计数据和相关性所导致的。

为了完成回测，下面的代码会将预期的投资组合统计数据（应用于前一年的数据的最优组合）与当年已实现的投资组合统计数据（应用于当年数据的前一年的最优组合）进行比较。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [74]: res = pd.DataFrame()
         for year in range(2010, 2019):
             rets_ = rets[symbols].loc[f'{year}-01-01':f'{year}-12-31']
             epv = port_volatility(rets_, opt_weights[year])  ❶
             epr = port_return(rets_, opt_weights[year])  ❶
             esr = epr / epv  ❶
             rets_ = rets[symbols].loc[f'{year + 1}-01-01':f'{year + 1}-12-31']
             rpv = port_volatility(rets_, opt_weights[year])  ❷
             rpr = port_return(rets_, opt_weights[year])  ❷
             rsr = rpr / rpv  ❷
             res = res.append(pd.DataFrame({'epv': epv, 'epr': epr, 'esr': esr,
                                            'rpv': rpv, 'rpr': rpr, 'rsr': rsr},
                                          index=[year + 1]))

In [75]: res
Out[75]:            epv       epr       esr       rpv       rpr       rsr
         2011  0.157440  0.303003  1.924564  0.160622  0.133836  0.833235
         2012  0.173279  0.169321  0.977156  0.182292  0.161375  0.885256
         2013  0.202460  0.278459  1.375378  0.168714  0.166897  0.989228
         2014  0.181544  0.368961  2.032353  0.197798  0.026830  0.135645
         2015  0.160340  0.309486  1.930190  0.211368 -0.024560 -0.116194
         2016  0.326730  0.778330  2.382179  0.296565  0.103870  0.350242
         2017  0.106148  0.090933  0.856663  0.079521  0.230630  2.900235
         2018  0.086548  0.260702  3.012226  0.157337  0.038234  0.243004
         2019  0.323796  0.228008  0.704174  0.207672  0.275819  1.328147

In [76]: res.mean()
Out[76]: epv    0.190920
         epr    0.309689
         esr    1.688320
         rpv    0.184654
         rpr    0.123659
         rsr    0.838755
         dtype: float64
```

❶ 投资组合预期的统计值。

❷ 投资组合实际的统计值。

图 4-6 比较了单年的预期与实际的投资组合波动率。MVP 理论在预测投资组合波动率方面做得相当好。两个时间序列之间相对较高的相关性也支持了这一点。

```
In [77]: res[['epv', 'rpv']].corr()
Out[77]:           epv       rpv
         epv  1.000000  0.765733
         rpv  0.765733  1.000000

In [78]: res[['epv', 'rpv']].plot(kind='bar', figsize=(10, 6),
                 title='Expected vs. Realized Portfolio Volatility');
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

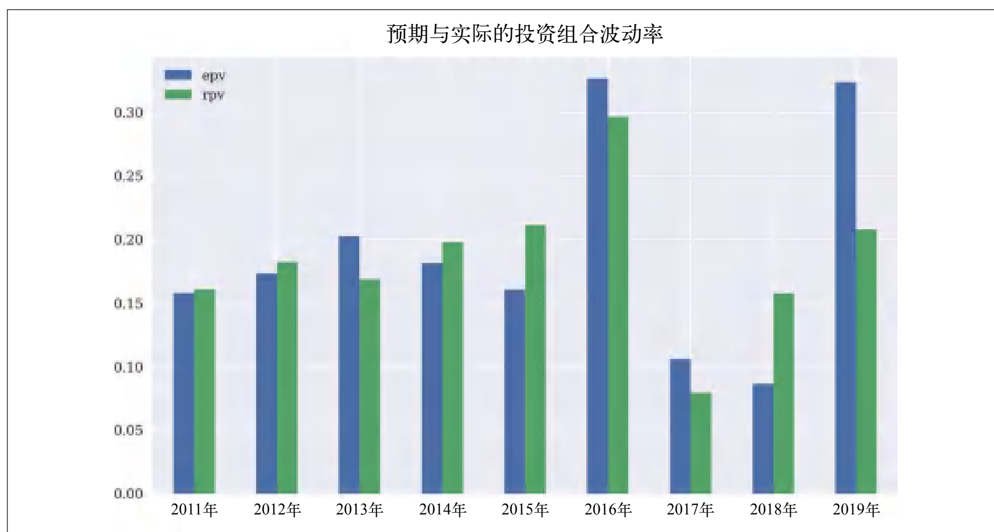


图 4-6：预期与实际的投资组合波动率

然而，当预期与实际的投资组合收益率进行比较时，结论则正好相反（参见图 4-7）。MVP 理论显然无法预测投资组合的收益率，两个时间序列之间的负相关性证实了这一点。

```
In [79]: res[['epr', 'rpr']].corr()
Out[79]:           epr       rpr
         epr  1.000000 -0.350437
         rpr -0.350437  1.000000

In [80]: res[['epr', 'rpr']].plot(kind='bar', figsize=(10, 6),
                 title='Expected vs. Realized Portfolio Return');
```

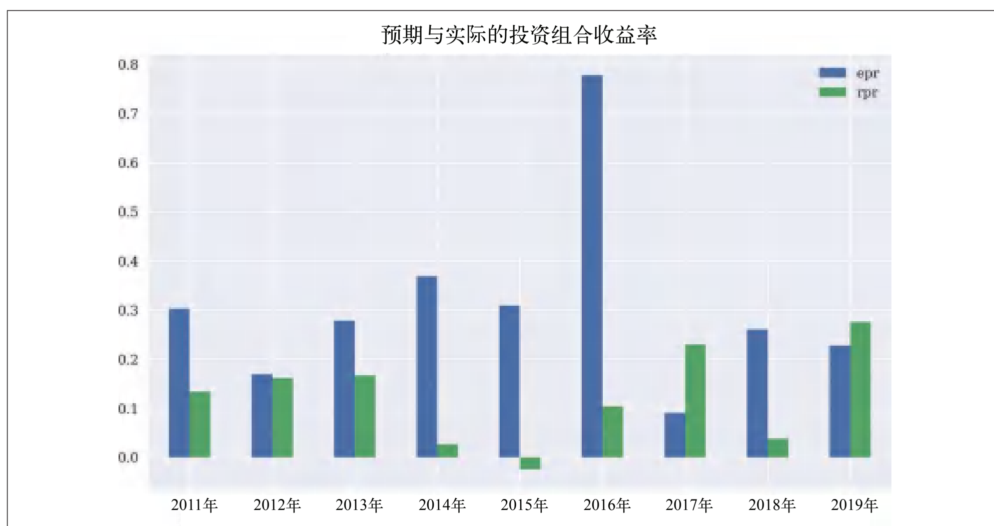


图 4-7：预期与实际的投资组合收益率

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

对夏普比率来说，我们会得出类似甚至更糟的结论（参见图 4-8）。对以投资组合夏普比率最大化为目标的数据驱动型投资者来说，该理论的预测值通常与实际值相差很大。两个时间序列之间的相关性甚至比收益率之间的相关性还要低。

```
In [81]: res[['esr', 'rsr']].corr()
Out[81]:           esr       rsr
         esr  1.000000 -0.698607
         rsr -0.698607  1.000000

In [82]: res[['esr', 'rsr']].plot(kind='bar', figsize=(10, 6),
                 title='Expected vs. Realized Sharpe Ratio');
```

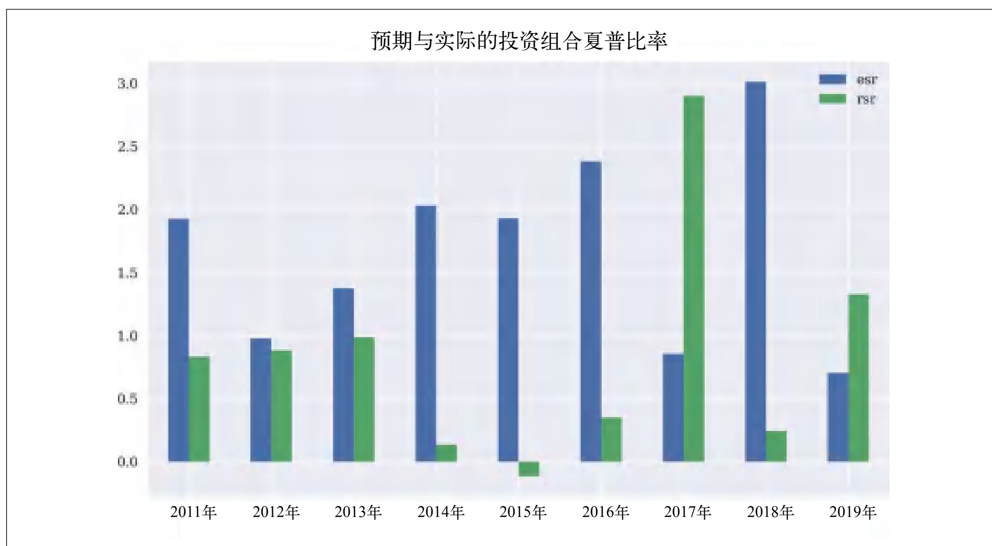
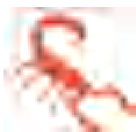


图 4-8：预期与实际的投资组合夏普比率



MVP 理论的预测能力

将 MVP 理论应用到真实世界的数据中揭示了其实际的不足。如果没有额外的约束条件，那么最优的投资组合的构成和再平衡可能是很极端的状态。在数值例子中，对投资组合收益率和夏普比率的预测能力是相当差的，而对投资组合风险的预测能力似乎是可以接受的。然而，投资者通常感兴趣的是风险调整后的业绩衡量指标，比如夏普比率，而在这个例子中，MVP 理论从统计上来说是最不适用的。

4.4.3 资本资产定价模型

类似的方法也可以应用于 CAPM 的实际测试。假设前面所讲的数据驱动型投资者希望应用 CAPM 来获得之前的 4 只科技股的预期收益率。下面的 Python 代码会首先求出给定年份的每只股票的 beta 系数，然后根据该股票的 beta 系数和市场投资组合的表现，计算其明年

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

的预期收益率。市场投资组合由标准普尔 500 指数近似计算。

```
In [83]: r = 0.005  ❶

In [84]: market = '.SPX'  ❷

In [85]: rets = np.log(raw / raw.shift(1)).dropna()

In [86]: res = pd.DataFrame()

In [87]: for sym in rets.columns[:4]:
             print('\n' + sym)
             print(54 * '=')
             for year in range(2010, 2019):
                 rets_ = rets.loc[f'{year}-01-01':f'{year}-12-31']
                 muM = rets_[market].mean() * 252
                 cov = rets_.cov().loc[sym, market]  ❸
                 var = rets_[market].var()  ❸
                 beta = cov / var  ❸
                 rets_ = rets.loc[f'{year + 1}-01-01':f'{year + 1}-12-31']
                 muM = rets_[market].mean() * 252
                 mu_capm = r + beta * (muM - r)  ❹
                 mu_real = rets_[sym].mean() * 252  ❺
                 res = res.append(pd.DataFrame({'symbol': sym,
                                                'mu_capm': mu_capm,
                                                'mu_real': mu_real},
                                               index=[year + 1]),
                                  sort=True)  ❻
                 print('{} | beta: {:.3f} | mu_capm: {:6.3f} | mu_real: {:6.3f}'
                       .format(year + 1, beta, mu_capm, mu_real))  ❻
```

❶ 指定无风险短期利率。

❷ 定义市场投资组合。

❸ 求出股票的 beta 系数。

❹ 根据上一年的 beta 系数和当年的市场投资组合表现计算预期收益率。

❺ 计算当年股票的实际业绩。

❻ 汇集并打印所有结果。

以上代码的输出如下。

```
AAPL.O
======================================================
2011 | beta: 1.052 | mu_capm: -0.000 | mu_real:  0.228
2012 | beta: 0.764 | mu_capm:  0.098 | mu_real:  0.275
2013 | beta: 1.266 | mu_capm:  0.327 | mu_real:  0.053
2014 | beta: 0.630 | mu_capm:  0.070 | mu_real:  0.320
2015 | beta: 0.833 | mu_capm: -0.005 | mu_real: -0.047
2016 | beta: 1.144 | mu_capm:  0.103 | mu_real:  0.096
2017 | beta: 1.009 | mu_capm:  0.180 | mu_real:  0.381
2018 | beta: 1.379 | mu_capm: -0.091 | mu_real: -0.071
2019 | beta: 1.252 | mu_capm:  0.316 | mu_real:  0.621
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
MSFT.O
=======================================================
2011 | beta: 0.890 | mu_capm:  0.001 | mu_real: -0.072
2012 | beta: 0.816 | mu_capm:  0.104 | mu_real:  0.029
2013 | beta: 1.109 | mu_capm:  0.287 | mu_real:  0.337
2014 | beta: 0.876 | mu_capm:  0.095 | mu_real:  0.216
2015 | beta: 0.955 | mu_capm: -0.007 | mu_real:  0.178
2016 | beta: 1.249 | mu_capm:  0.113 | mu_real:  0.113
2017 | beta: 1.224 | mu_capm:  0.217 | mu_real:  0.321
2018 | beta: 1.303 | mu_capm: -0.086 | mu_real:  0.172
2019 | beta: 1.442 | mu_capm:  0.364 | mu_real:  0.440

INTC.O
=======================================================
2011 | beta: 1.081 | mu_capm: -0.000 | mu_real:  0.142
2012 | beta: 0.842 | mu_capm:  0.108 | mu_real: -0.163
2013 | beta: 1.081 | mu_capm:  0.280 | mu_real:  0.230
2014 | beta: 0.883 | mu_capm:  0.096 | mu_real:  0.335
2015 | beta: 1.055 | mu_capm: -0.008 | mu_real: -0.052
2016 | beta: 1.009 | mu_capm:  0.092 | mu_real:  0.051
2017 | beta: 1.261 | mu_capm:  0.223 | mu_real:  0.242
2018 | beta: 1.163 | mu_capm: -0.076 | mu_real:  0.017
2019 | beta: 1.376 | mu_capm:  0.347 | mu_real:  0.243

AMZN.O
=======================================================
2011 | beta: 1.102 | mu_capm: -0.001 | mu_real: -0.039
2012 | beta: 0.958 | mu_capm:  0.122 | mu_real:  0.374
2013 | beta: 1.116 | mu_capm:  0.289 | mu_real:  0.464
2014 | beta: 1.262 | mu_capm:  0.135 | mu_real: -0.251
2015 | beta: 1.473 | mu_capm: -0.013 | mu_real:  0.778
2016 | beta: 1.122 | mu_capm:  0.102 | mu_real:  0.104
2017 | beta: 1.118 | mu_capm:  0.199 | mu_real:  0.446
2018 | beta: 1.300 | mu_capm: -0.086 | mu_real:  0.251
2019 | beta: 1.619 | mu_capm:  0.408 | mu_real:  0.207
```

图 4-9 比较了在考虑到上一年的 beta 系数和当年的市场投资组合表现时，单只股票的预期（期望）收益率与该股票本年的实际收益率。显然，CAPM 的原始形式在仅基于 beta 系数预测股票表现时并没有真正发挥作用。

```
In [88]: sym = 'AMZN.O'

In [89]: res[res['symbol'] == sym].corr()
Out[89]:            mu_capm   mu_real
         mu_capm  1.000000 -0.004826
         mu_real -0.004826  1.000000

In [90]: res[res['symbol'] == sym].plot(kind='bar',
                         figsize=(10, 6), title=sym);
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

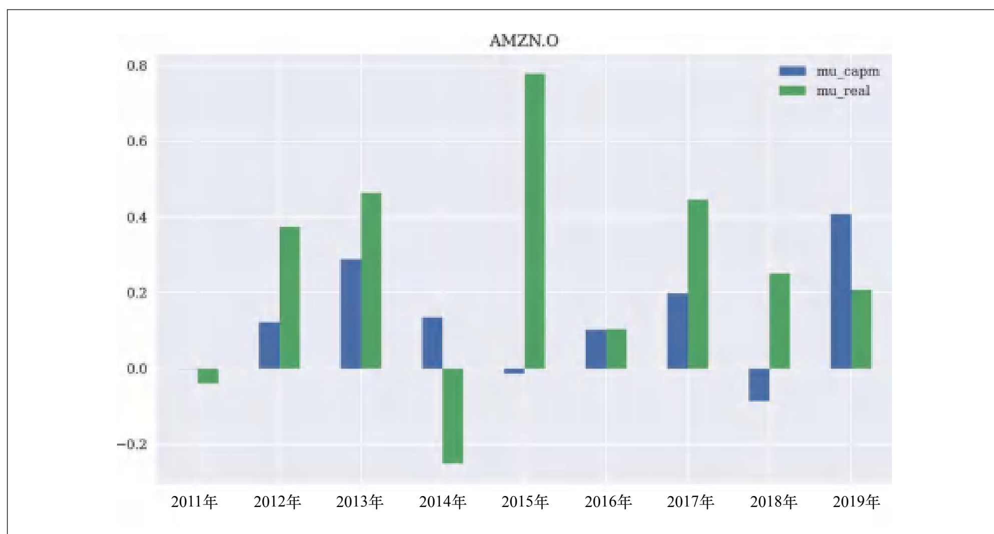


图 4-9：单只股票的 CAPM 预期收益率与实际股票收益率

图 4-10 比较了 CAPM 预期股票收益率的平均值与实际收益率的平均值。在这里，CAPM 同样表现不佳。

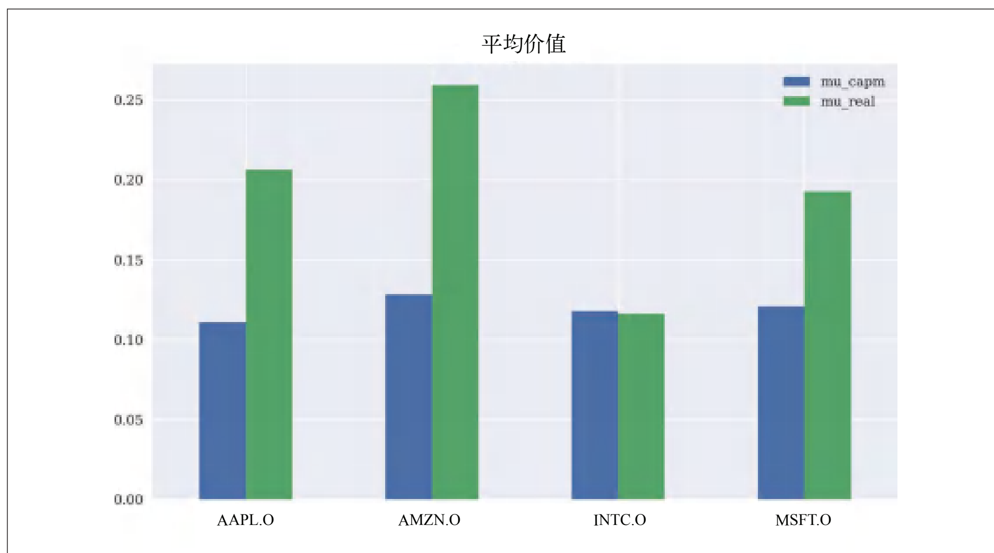


图 4-10：多只股票的 CAPM 预期平均收益率与实际股票平均收益率

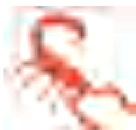
很容易看到，CAPM 对所分析股票的预期平均收益率影响不大，在 12.2% 和 14.4% 之间。然而，股票的实际平均收益率表现出了很高的变异性，这一比例在 9.4% 和 29.2% 之间。仅凭市场投资组合表现和 beta 系数的关系显然不能解释（科技）股票已观察到的收益率。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [91]: grouped = res.groupby('symbol').mean()
         grouped
Out[91]:          mu_capm   mu_real
         symbol
         AAPL.O  0.110855  0.206158
         AMZN.O  0.128223  0.259395
         INTC.O  0.117929  0.116180
         MSFT.O  0.120844  0.192655

In [92]: grouped.plot(kind='bar', figsize=(10, 6), title='Average Values');
```



CAPM 的预测能力

相对于市场投资组合而言，CAPM 对股票未来表现的预测能力非常低，甚至对某些股票不具备预测能力。其中一个原因可能是 CAPM 基于与 MVP 理论相同的核心假设，即投资者只关心投资组合和 / 或股票的（预期）收益率和（预期）波动率。从建模的角度来看，一个假设性的问题是，单个风险因子是否足以解释股票收益率的变化？或者股票收益率和市场投资组合表现之间是否可能存在非线性关系？

4.4.4 套利定价理论

从前面的数值例子的结果来看，CAPM 的预测能力似乎相当有限。这里值得质疑的是，仅凭市场投资组合的表现是否足以解释股票收益率的变化。APT 的答案是**否定的**，因为可以有多个（甚至非常多个）因子来共同解释股票收益率的可变性。3.5 节正式描述过 APT 的框架，但它也依赖于这些因子和股票收益率之间的线性关系。

数据驱动型投资者认识到，CAPM 不足以可靠地预测股票相对于市场投资组合的表现。因此，投资者决定在市场投资组合中增加 3 个可能推动股票表现的额外因子。

- 市场波动率（由 VIX 指数表示，`.VIX`）
- 汇率（以欧元 / 美元汇率表示，`EUR=`）
- 商品价格（以黄金价格表示，`XAU=`）

下面的 Python 代码实现了一种简单的 APT 方法，它使用 4 个因子结合多元回归方法来解释股票未来的表现。

```
In [93]: factors = ['.SPX', '.VIX', 'EUR=', 'XAU=']  ❶

In [94]: res = pd.DataFrame()

In [95]: np.set_printoptions(formatter={'float': lambda x: f'{x:5.2f}'})

In [96]: for sym in rets.columns[:4]:
             print('\n' + sym)
             print(71 * '=')
             for year in range(2010, 2019):
                 rets_ = rets.loc[f'{year}-01-01':f'{year}-12-31']
                 reg = np.linalg.lstsq(rets_[factors],
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
                                  rets_[sym], rcond=-1)[0]  ❷
            rets_ = rets.loc[f'{year + 1}-01-01':f'{year + 1}-12-31']
            mu_apt = np.dot(rets_[factors].mean() * 252, reg)  ❸
            mu_real = rets_[sym].mean() * 252  ❹
            res = res.append(pd.DataFrame({'symbol': sym,
                             'mu_apt': mu_apt, 'mu_real': mu_real},
                              index=[year + 1]))
            print('{} | fl: {} | mu_apt: {:6.3f} | mu_real: {:6.3f}'
                  .format(year + 1, reg.round(2), mu_apt, mu_real))
```

❶ 4 个因子。

❷ 多元回归。

❸ APT 预测的股票收益率。

❹ 股票的实际收益率。

以上代码的输出如下。

```
AAPL.O
=======================================================================
2011 | fl: [ 0.91 -0.04  0.35  0.12] | mu_apt:  0.011 | mu_real:  0.228
2012 | fl: [ 0.76 -0.02 -0.24  0.05] | mu_apt:  0.099 | mu_real:  0.275
2013 | fl: [ 1.67  0.04 -0.56  0.10] | mu_apt:  0.366 | mu_real:  0.053
2014 | fl: [ 0.53 -0.00  0.02  0.16] | mu_apt:  0.050 | mu_real:  0.320
2015 | fl: [ 1.07  0.02  0.25  0.01] | mu_apt: -0.038 | mu_real: -0.047
2016 | fl: [ 1.21  0.01 -0.14 -0.02] | mu_apt:  0.110 | mu_real:  0.096
2017 | fl: [ 1.10  0.01 -0.15 -0.02] | mu_apt:  0.170 | mu_real:  0.381
2018 | fl: [ 1.06 -0.03 -0.15  0.12] | mu_apt: -0.088 | mu_real: -0.071
2019 | fl: [ 1.37  0.01 -0.20  0.13] | mu_apt:  0.364 | mu_real:  0.621

MSFT.O
=======================================================================
2011 | fl: [ 0.98  0.01  0.02 -0.11] | mu_apt: -0.008 | mu_real: -0.072
2012 | fl: [ 0.82  0.00 -0.03 -0.01] | mu_apt:  0.103 | mu_real:  0.029
2013 | fl: [ 1.14  0.00 -0.07 -0.01] | mu_apt:  0.294 | mu_real:  0.337
2014 | fl: [ 1.28  0.05  0.04  0.07] | mu_apt:  0.149 | mu_real:  0.216
2015 | fl: [ 1.20  0.03  0.05  0.01] | mu_apt: -0.016 | mu_real:  0.178
2016 | fl: [ 1.44  0.03 -0.17 -0.02] | mu_apt:  0.127 | mu_real:  0.113
2017 | fl: [ 1.33  0.01 -0.14  0.00] | mu_apt:  0.216 | mu_real:  0.321
2018 | fl: [ 1.10 -0.02 -0.14  0.22] | mu_apt: -0.087 | mu_real:  0.172
2019 | fl: [ 1.51  0.01 -0.16 -0.02] | mu_apt:  0.378 | mu_real:  0.440

INTC.O
=======================================================================
2011 | fl: [ 1.17  0.01  0.05 -0.13] | mu_apt: -0.010 | mu_real:  0.142
2012 | fl: [ 1.03  0.04  0.01  0.03] | mu_apt:  0.122 | mu_real: -0.163
2013 | fl: [ 1.06 -0.01 -0.10  0.01] | mu_apt:  0.267 | mu_real:  0.230
2014 | fl: [ 0.96  0.02  0.36 -0.02] | mu_apt:  0.063 | mu_real:  0.335
2015 | fl: [ 0.93 -0.01 -0.09  0.02] | mu_apt:  0.001 | mu_real: -0.052
2016 | fl: [ 1.02  0.00 -0.05  0.06] | mu_apt:  0.099 | mu_real:  0.051
2017 | fl: [ 1.41  0.02 -0.18  0.03] | mu_apt:  0.226 | mu_real:  0.242
2018 | fl: [ 1.12 -0.01 -0.11  0.17] | mu_apt: -0.076 | mu_real:  0.017
2019 | fl: [ 1.50  0.01 -0.34  0.30] | mu_apt:  0.431 | mu_real:  0.243
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
AMZN.O
==========================================================================
2011 | fl: [ 1.02 -0.03 -0.18 -0.14] | mu_apt: -0.016 | mu_real: -0.039
2012 | fl: [ 0.98 -0.01 -0.17 -0.09] | mu_apt:  0.117 | mu_real:  0.374
2013 | fl: [ 1.07 -0.00  0.09  0.00] | mu_apt:  0.282 | mu_real:  0.464
2014 | fl: [ 1.54  0.03  0.01 -0.08] | mu_apt:  0.176 | mu_real: -0.251
2015 | fl: [ 1.26 -0.02  0.45 -0.11] | mu_apt: -0.044 | mu_real:  0.778
2016 | fl: [ 1.06 -0.00 -0.15 -0.04] | mu_apt:  0.099 | mu_real:  0.104
2017 | fl: [ 0.94 -0.02  0.12 -0.03] | mu_apt:  0.185 | mu_real:  0.446
2018 | fl: [ 0.90 -0.04 -0.25  0.28] | mu_apt: -0.085 | mu_real:  0.251
2019 | fl: [ 1.99  0.05 -0.37  0.12] | mu_apt:  0.506 | mu_real:  0.207
```

图 4-11 比较了随时间变化的 APT 预期股票收益率和实际股票收益率。与单因子 CAPM 相比，似乎没有任何改善。

```
In [97]: sym = 'AMZN.O'

In [98]: res[res['symbol'] == sym].corr()
Out[98]:            mu_apt   mu_real
         mu_apt    1.000000 -0.098281
         mu_real  -0.098281  1.000000

In [99]: res[res['symbol'] == sym].plot(kind='bar',
                         figsize=(10, 6), title=sym);
```

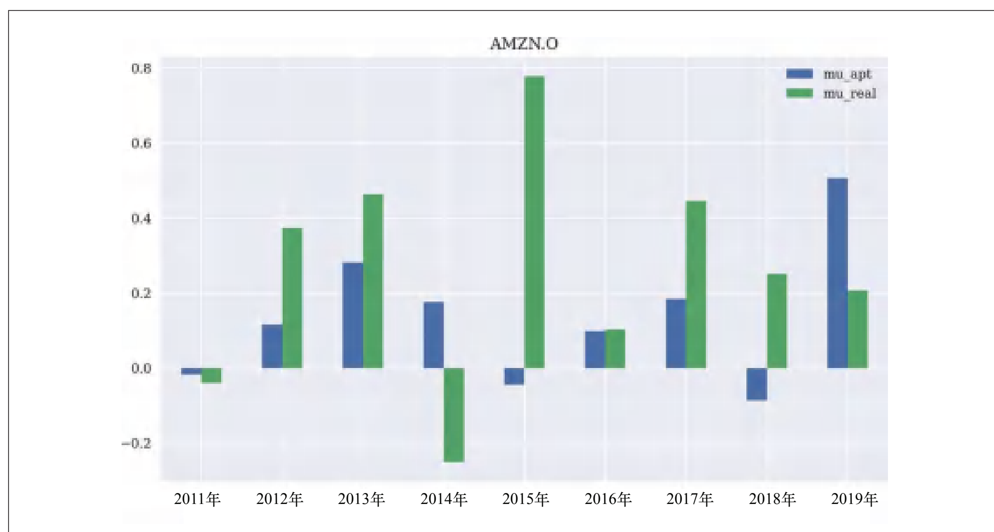


图 4-11：单只股票的 APT 预期收益率与实际股票收益率

同样的情况出现在了图 4-12 中，下面的代码比较了多只股票的平均值。APT 预测的平均值几乎没有什么变化，但实际收益率的平均值差异是很大的。

```
In [100]: grouped = res.groupby('symbol').mean()
          grouped
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
Out[100]:           mu_apt   mu_real
          symbol
          AAPL.O  0.116116  0.206158
          AMZN.O  0.135528  0.259395
          INTC.O  0.124811  0.116180
          MSFT.O  0.128441  0.192655

In [101]: grouped.plot(kind='bar', figsize=(10, 6), title='Average Values');
```

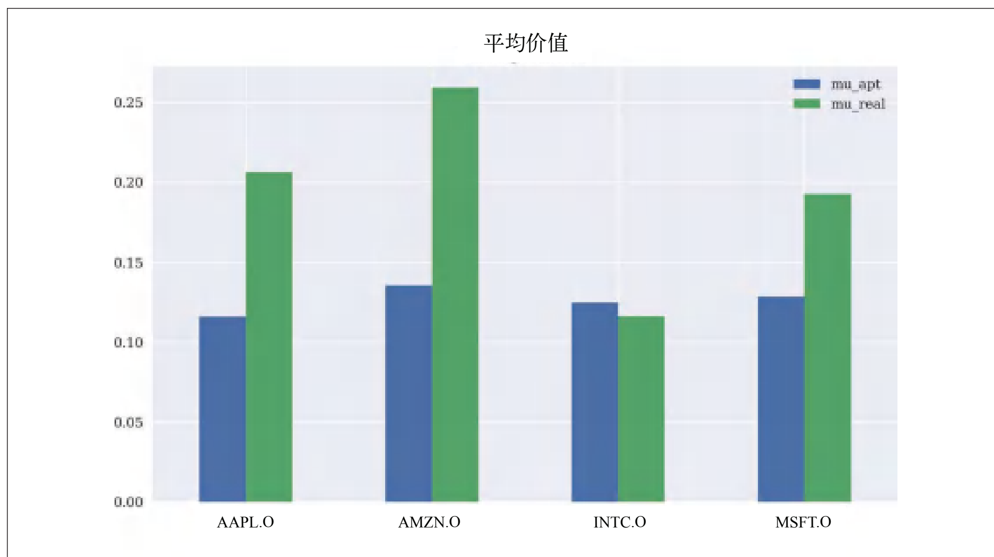


图 4-12：多只股票的 APT 预期平均收益率与实际股票平均收益率

当然，风险因子的选择在这种情况下至关重要。数据驱动型投资者决定找出那些通常被认为与股票相关的风险因子。通过研究 Bender 等（2013）的论文，投资者用一组新的风险因子替换了原来的风险因子，如表 4-3 所示。

表4-3：APT的风险因子

因子	描述	RIC
市场	MSCI 全球每日美元总收益率（PUS= 价格收益率）	.dMIWO00000GUS
规模	MSCI 全球同等权重价格净指数日终数据	.dMIWO0000ENUS
波动率	MSCI 全球最小波动率净收益率	.dMIWO0000YNUS
价值	MSCI 全球价值加权总值	.dMIWO000PkGUS
风险	MSCI 全球风险加权总美元日终数据	.dMIWO000PlGUS
增长	MSCI 全球质量美元净收益率	.MIWO0000vNUS
动量	MSCI 全球动量总指数美元日终数据	.dMIWO0000NGUS

下面的 Python 代码会从远程检索相应的数据集，并将标准化的时间序列数据进行可视化（参见图 4-13）。简单一看就可以发现，时间序列是高度正相关的。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [102]: factors = pd.read_csv('http://hilpisch.com/aiif_eikon_eod_factors.csv',
                                index_col=0, parse_dates=True)  ❶

In [103]: (factors / factors.iloc[0]).plot(figsize=(10, 6));  ❷
```

❶ 检索各因子时间序列数据。

❷ 标准化数据并绘制数据图。

图 4-13：标准化后的各因子时间序列数据

时间序列高度相关的效果可以通过以下计算和因子收益率的相关矩阵得到证实，所有相关系数约为 0.75 或更高。

```
In [104]: start = '2017-01-01'  ❶
          end = '2020-01-01'  ❶

In [105]: retsd = rets.loc[start:end].copy()  ❷
          retsd.dropna(inplace=True)  ❷

In [106]: retsf = np.log(factors / factors.shift(1))  ❸
          retsf = retsf.loc[start:end]  ❸
          retsf.dropna(inplace=True)  ❸
          retsf = retsf.loc[retsd.index].dropna()  ❸

In [107]: retsf.corr()  ❹
Out[107]:               market      size  volatility     value      risk    growth \
          market      1.000000  0.935867    0.845010  0.964124  0.947150  0.959038
          size        0.935867  1.000000    0.791767  0.965739  0.983238  0.835477
          volatility  0.845010  0.791767    1.000000  0.778294  0.865467  0.818280
          value       0.964124  0.965739    0.778294  1.000000  0.958359  0.864222
          risk        0.947150  0.983238    0.865467  0.958359  1.000000  0.858546
          growth      0.959038  0.835477    0.818280  0.864222  0.858546  1.000000
          momentum    0.928705  0.796420    0.819585  0.818796  0.825563  0.952956
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
             momentum
market       0.928705
size         0.796420
volatility   0.819585
value        0.818796
risk         0.825563
growth       0.952956
momentum     1.000000
```

❶ 定义数据选择的开始日期和结束日期。

❷ 选择相关的收益率数据子集。

❸ 计算并处理因子的对数收益率。

❹ 显示因子的相关矩阵。

下面的 Python 代码会使用新的因子求出原始股票的因子载荷。它们是从数据集的前半部分求出的，并用于预测后半部分数据在给定单因子下的股票收益率。实际收益率也是通过计算得到的。图 4-14 比较了两个时间序列。正如预期的那样，鉴于各因子间的高度相关性，APT 方法的解释能力并不比 CAPM 高多少。

```
In [108]: res = pd.DataFrame()

In [109]: np.set_printoptions(formatter={'float': lambda x: f'{x:5.2f}'})

In [110]: split = int(len(retsf) * 0.5)
          for sym in rets.columns[:4]:
              print('\n' + sym)
              print(74 * '=')
              retsf_, retsd_ = retsf.iloc[:split], retsd.iloc[:split]
              reg = np.linalg.lstsq(retsf_, retsd_[sym], rcond=-1)[0]
              retsf_, retsd_ = retsf.iloc[split:], retsd.iloc[split:]
              mu_apt = np.dot(retsf_.mean() * 252, reg)
              mu_real = retsd_[sym].mean() * 252
              res = res.append(pd.DataFrame({'mu_apt': mu_apt,
                                'mu_real': mu_real}, index=[sym,]),
                                sort=True)
              print('fl: {} | apt: {:.3f} | real: {:.3f}'
                    .format(reg.round(1), mu_apt, mu_real))


          AAPL.O
          ==========================================================================
          fl: [ 2.30  2.80 -0.70 -1.40 -4.20  2.00 -0.20] | apt: 0.115 | real: 0.301

          MSFT.O
          ==========================================================================
          fl: [ 1.50  0.00  0.10 -1.30 -1.40  0.80  1.00] | apt: 0.181 | real: 0.304

          INTC.O
          ==========================================================================
          fl: [-3.10  1.60  0.40  1.30 -2.60  2.50  1.10] | apt: 0.186 | real: 0.118
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        AMZN.O
        ==========================================================================
        fl: [ 9.10  3.30 -1.00 -7.10 -3.10 -1.80  1.20] | apt: 0.019 | real: 0.050

In [111]: res.plot(kind='bar', figsize=(10, 6));
```

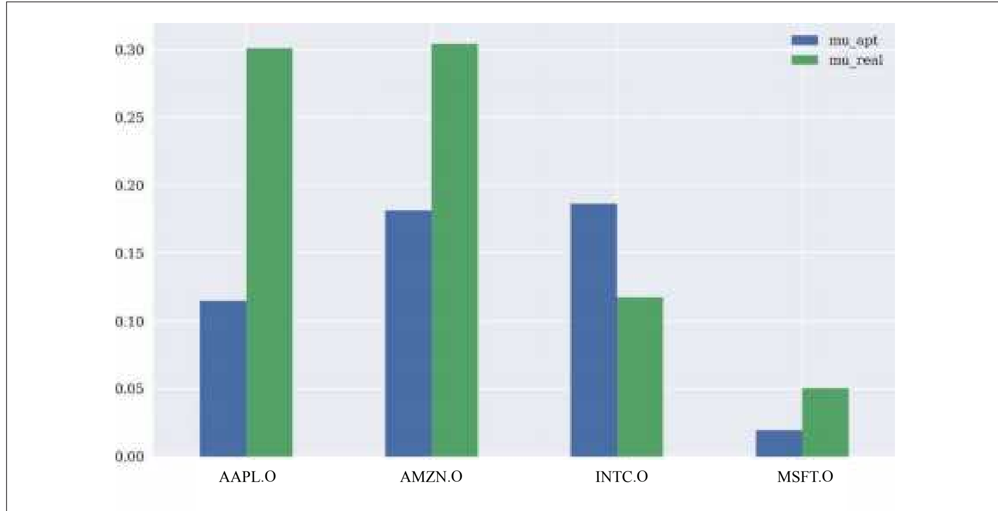


图 4-14：基于典型因子的 APT 预期收益率与实际收益率

数据驱动型投资者不愿完全忽视 APT，因此，可以通过一个额外的检验来进一步说明 APT 的解释能力。我们使用因子载荷来检验 APT 能否正确地解释股价随时间的变化。的确，尽管 APT 不能正确预测股价的明确表现（偏离了 10 个百分点），但在大多数情况下，它能正确预测股价的走势（参见图 4-15）。预期收益率和实际收益率之间的相关性也很高，约为 85%。然而，该分析使用的是已实现的因子收益率来产生 APT 预测——当然这种预测在相关交易日的前一天实际是无法实现的。

```
In [112]: sym
Out[112]: 'AMZN.O'

In [113]: rets_sym = np.dot(retsf_, reg)  ❶

In [114]: rets_sym = pd.DataFrame(rets_sym,
                                  columns=[sym + '_apt'],
                                  index=retsf_.index)  ❷

In [115]: rets_sym[sym + '_real'] = retsd_[sym]  ❸

In [116]: rets_sym.mean() * 252  ❹
Out[116]: AMZN.O_apt     0.019401
          AMZN.O_real    0.050344
          dtype: float64
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [117]: rets_sym.std() * 252 ** 0.5  ❺
Out[117]: AMZN.O_apt     0.270995
          AMZN.O_real    0.307653
          dtype: float64

In [118]: rets_sym.corr()  ❻
Out[118]:              AMZN.O_apt  AMZN.O_real
          AMZN.O_apt     1.000000     0.832218
          AMZN.O_real    0.832218     1.000000

In [119]: rets_sym.cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ 根据实际因子收益率预测每日股价收益率。

❷ 将结果存储在 DataFrame 对象中，并添加列和索引数据。

❸ 将实际股票价格收益率添加到 DataFrame 对象中。

❹ 计算年化收益率。

❺ 计算年化波动率。

❻ 计算相关因子。

图 4-15：随时间变化的 APT 预期表现和实际表现（总量）

在给定实际因子收益率的情况下，APT 如何准确地预测股票价格的走势呢？下面的 Python 代码显示，准确率得分略高于 75%。

```
In [120]: rets_sym['same'] = (np.sign(rets_sym[sym + '_apt']) ==
                              np.sign(rets_sym[sym + '_real']))

In [121]: rets_sym['same'].value_counts()
Out[121]: True     288
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
         False     89
         Name: same, dtype: int64

In [122]: rets_sym['same'].value_counts()[True] / len(rets_sym)
Out[122]: 0.7639257294429708
```

4.5 揭示中心假设

上一节提供了一些真实的数值示例，表明流行的规范性金融理论可能在实践中会失效。本节认为，主要原因之一是这些流行的金融理论的中心假设是无效的，也就是说，它们根本无法描述金融市场的现实。这里要分析的两个假设是**正态分布收益率**和**线性关系**。

4.5.1 正态分布收益率

事实上，只有正态分布是可以通过其一阶矩（期望值）和二阶矩（标准差）完全确定的。

1. 样本数据集

为了进行说明，考虑由以下 Python 代码随机生成的一组标准正态分布数字。[4] 如图 4-16 所示，结果直方图是典型的钟形。

```
In [1]: import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        np.random.seed(100)
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'

In [2]: N = 10000

In [3]: snrn = np.random.standard_normal(N)  ❶
        snrn -= snrn.mean()  ❷
        snrn /= snrn.std()  ❸

In [4]: round(snrn.mean(), 4)  ❷
Out[4]: -0.0

In [5]: round(snrn.std(), 4)  ❸
Out[5]: 1.0

In [6]: plt.figure(figsize=(10, 6))
        plt.hist(snrn, bins=35);
```

❶ 绘制标准正态分布随机数。

❷ 将一阶矩（期望值）修正为 0.0。

❸ 将二阶矩（标准差）修正为 1.0。

注 4：NumPy 的随机数生成器生成的数字是**伪随机数**，不过它们在本书中被称为**随机数**。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

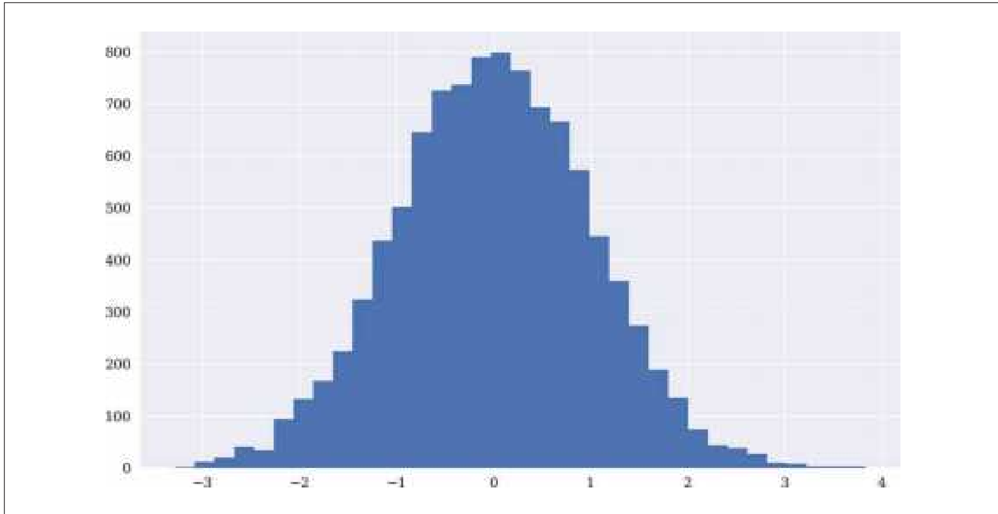


图 4-16：标准正态分布随机数

现在考虑一组随机数，它们具有相同的一阶矩值和二阶矩值，但分布与图 4-16 完全不同。如图 4-17 所示，虽然矩是相同的，但是这个分布只包含 3 个离散值。

```
In [7]: numbers = np.ones(N) * 1.5  ❶
        split = int(0.25 * N)  ❶
        numbers[split:3 * split] = -1  ❶
        numbers[3 * split:4 * split] = 0  ❶

In [8]: numbers -= numbers.mean()  ❷
        numbers /= numbers.std()  ❸

In [9]: round(numbers.mean(), 4)  ❷
Out[9]: 0.0

In [10]: round(numbers.std(), 4)  ❸
Out[10]: 1.0

In [11]: plt.figure(figsize=(10, 6))
         plt.hist(numbers, bins=35);
```

❶ 只有 3 个离散值的一组数字。

❷ 将一阶矩（期望值）修正为 0.0。

❸ 将二阶矩（标准差）修正为 1.0。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

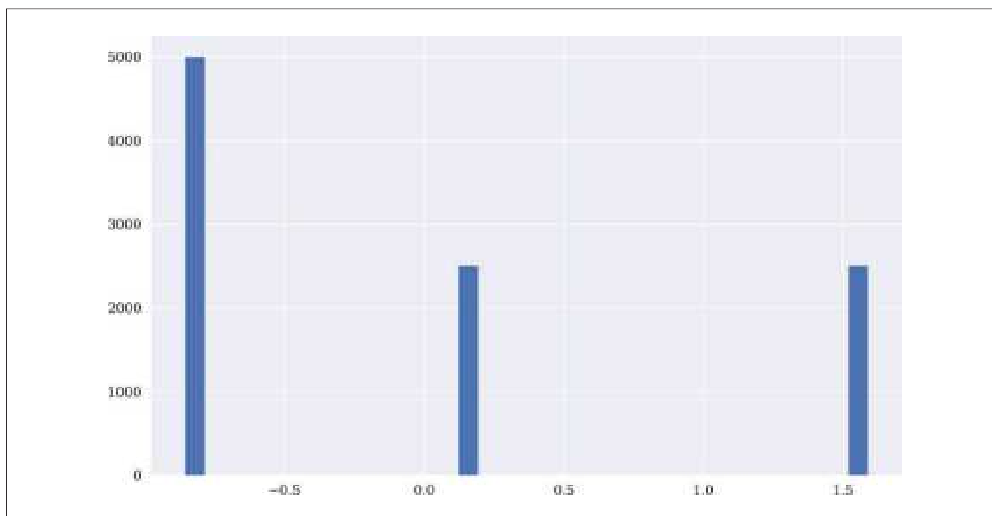


图 4-17：一阶矩和二阶矩分别为 0.0 和 1.0 的分布

一阶矩和二阶矩

概率分布的一阶矩和二阶矩只能完整描述一个正态分布。有无穷多的其他分布可能与正态分布共享前两个矩，但它们完全不同。

在准备测试真实的金融收益率时，考虑以下 Python 函数，它可以将数据可视化为直方图，并在数据的前两个矩上添加一个正态分布的概率密度函数（PDF）。

```
In [12]: import math
         import scipy.stats as scs
         import statsmodels.api as sm

In [13]: def dN(x, mu, sigma):
             ''' 正态随机变量x的概率密度函数
             '''
             z = (x - mu) / sigma
             pdf = np.exp(-0.5 * z ** 2) / math.sqrt(2 * math.pi * sigma ** 2)
             return pdf

In [14]: def return_histogram(rets, title=''):
             ''' 绘制收益率直方图
             '''
             plt.figure(figsize=(10, 6))
             x = np.linspace(min(rets), max(rets), 100)
             plt.hist(np.array(rets), bins=50,
                      density=True, label='frequency')  ❶
             y = dN(x, np.mean(rets), np.std(rets))  ❷
             plt.plot(x, y, linewidth=2, label='PDF')  ❷
             plt.xlabel('log returns')
             plt.ylabel('frequency/probability')
             plt.title(title)
             plt.legend()
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❶ 绘制数据的直方图。

❷ 绘制相应正态分布的 PDF。

图 4-18 显示了直方图与标准正态分布随机数的 PDF 的近似效果。

```
In [15]: return_histogram(snrn)
```

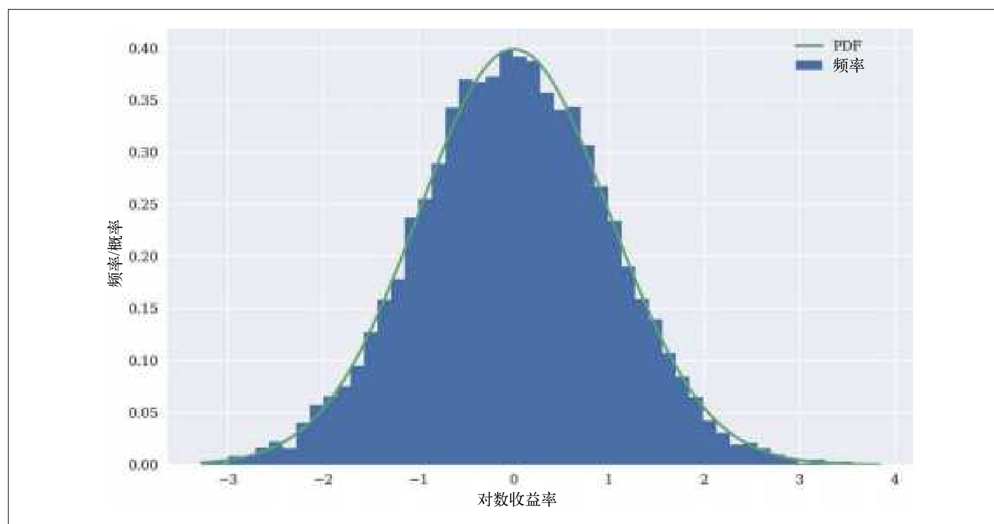


图 4-18：标准正态分布随机数的直方图和 PDF

相比之下，图 4-19 说明了正态分布的 PDF 与直方图显示的数据无关。

```
In [16]: return_histogram(numbers)
```

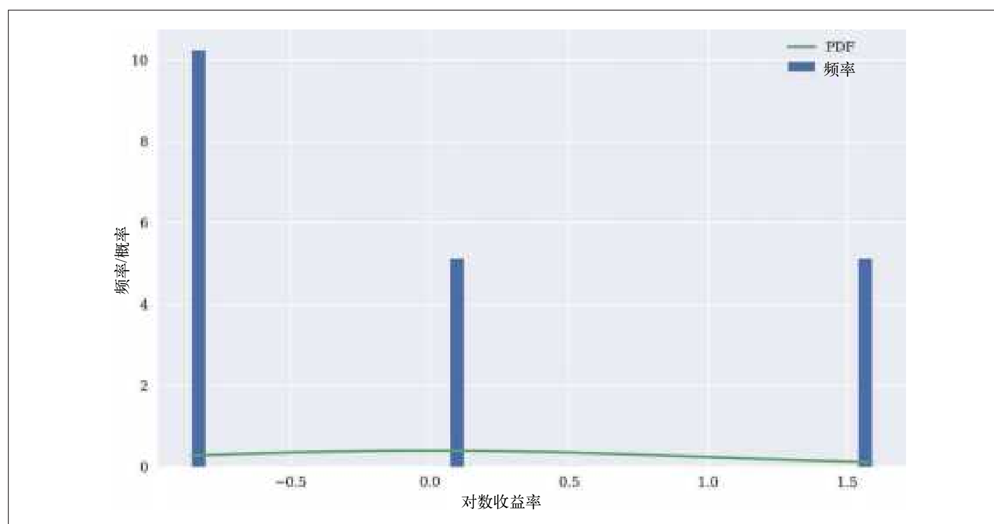


图 4-19：离散数字的直方图和正态 PDF

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

将正态分布与数据进行比较的另一种方法是绘制分位数 – 分位数（Q – Q）图，如图 4-20 所示。对正态分布的数字来说，数字本身（大部分）会位于 Q – Q 平面的一条直线上。

```
In [17]: def return_qqplot(rets, title=''):
             ''' 生成收益率的Q-Q图
             '''
             fig = sm.qqplot(rets, line='s', alpha=0.5)
             fig.set_size_inches(10, 6)
             plt.title(title)
             plt.xlabel('theoretical quantiles')
             plt.ylabel('sample quantiles')

In [18]: return_qqplot(snrn)
```

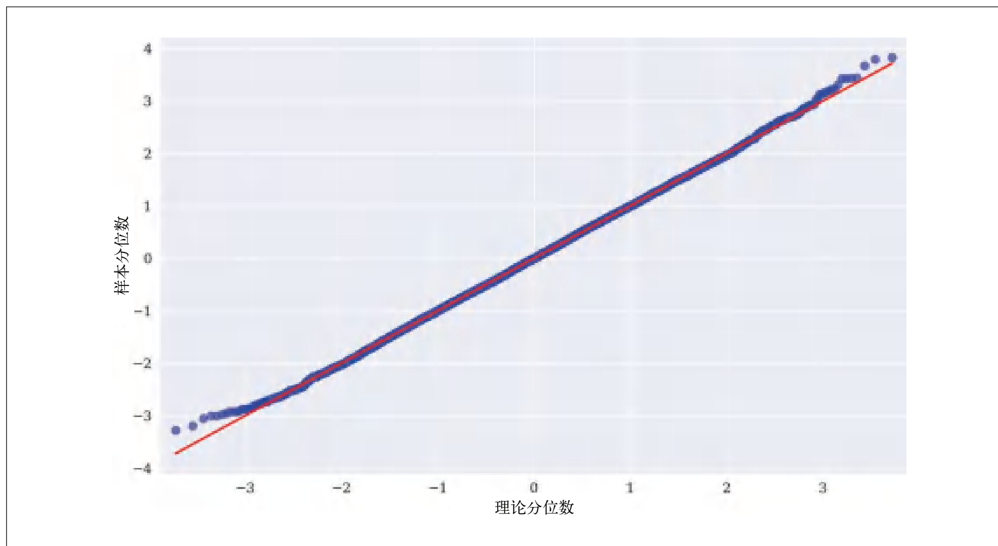


图 4-20：标准正态分布数的 Q – Q 图

图 4-21 所示的离散数的 Q – Q 图与图 4-20 所示的完全不同。

```
In [19]: return_qqplot(numbers)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

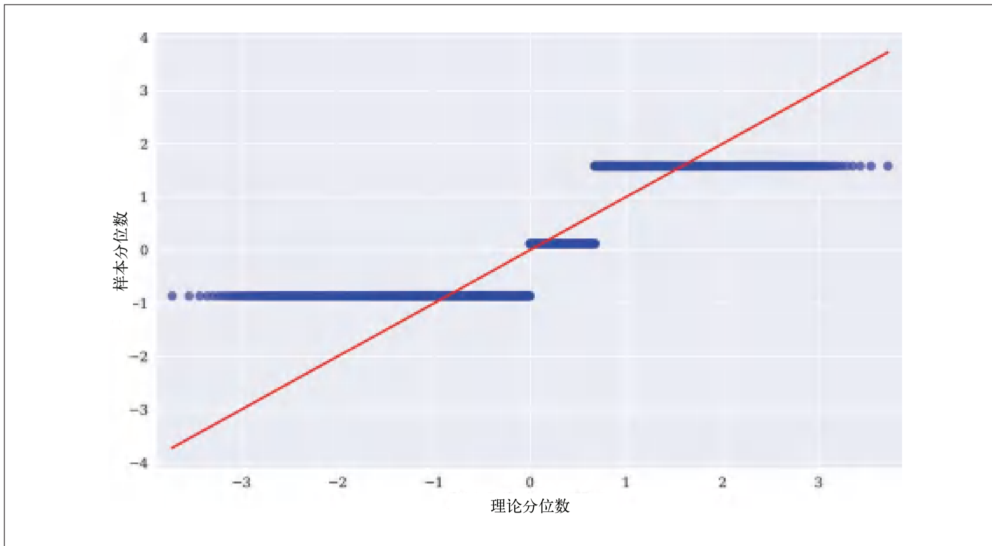


图 4-21：离散数的 Q－Q 图

最后，还可以使用统计方法测试来检查一组数字是否符合正态分布。

以下 Python 函数实现了 3 个测试。

- 正态偏度测试
- 正态峰度测试
- 正态偏度和峰度组合测试

p 值低于 0.05 通常被认为是符合正态的反指标，也就是说，数字分布是正态分布的假设被拒绝了。从这个意义上说，如图 4-21 所示，两个数据集的 p 值说明了问题。

```
In [20]: def print_statistics(rets):
             print('RETURN SAMPLE STATISTICS')
             print('---------------------------------------------')
             print('Skew of Sample Log Returns {:9.6f}'.format(
                         scs.skew(rets)))
             print('Skew Normal Test p-value   {:9.6f}'.format(
                         scs.skewtest(rets)[1]))
             print('---------------------------------------------')
             print('Kurt of Sample Log Returns {:9.6f}'.format(
                         scs.kurtosis(rets)))
             print('Kurt Normal Test p-value   {:9.6f}'.format(
                         scs.kurtosistest(rets)[1]))
             print('---------------------------------------------')
             print('Normal Test p-value        {:9.6f}'.format(
                         scs.normaltest(rets)[1]))
             print('---------------------------------------------')
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [21]: print_statistics(snrn)
         RETURN SAMPLE STATISTICS
         ---------------------------------------------
         Skew of Sample Log Returns  0.016793
         Skew Normal Test p-value    0.492685
         ---------------------------------------------
         Kurt of Sample Log Returns -0.024540
         Kurt Normal Test p-value    0.637637
         ---------------------------------------------
         Normal Test p-value         0.707334
         ---------------------------------------------

In [22]: print_statistics(numbers)
         RETURN SAMPLE STATISTICS
         ---------------------------------------------
         Skew of Sample Log Returns  0.689254
         Skew Normal Test p-value    0.000000
         ---------------------------------------------
         Kurt of Sample Log Returns -1.141902
         Kurt Normal Test p-value    0.000000
         ---------------------------------------------
         Normal Test p-value         0.000000
         ---------------------------------------------
```

2. 实际的金融收益率

下面的 Python 代码会从远程数据源检索日终数据（如本章前面所述），并计算数据集中包含的所有金融时间序列的对数收益率。从图 4-22 中可以看出，标准普尔 500 指数的对数收益率以柱状图表示，与具有相同样本期望值和标准差的正态 PDF 相比，显示出了更高的峰值和更肥的尾部。这两个特征是**典型的事实**，因为在不同的金融工具中它们可以一直被观察到。

```
In [23]: raw = pd.read_csv('http://hilpisch.com/aiif_eikon_eod_data.csv',
                           index_col=0, parse_dates=True).dropna()

In [24]: rets = np.log(raw / raw.shift(1)).dropna()

In [25]: symbol = '.SPX'

In [26]: return_histogram(rets[symbol].values, symbol)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

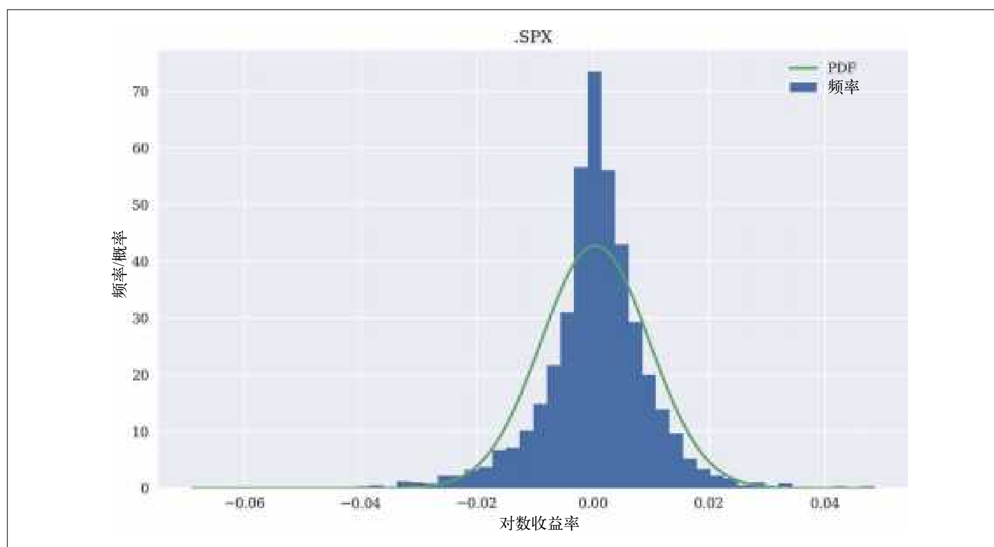


图 4-22：标准普尔 500 指数对数收益率的频率分布和正态 PDF

当考虑图 4-23 中标准普尔 500 指数对数收益率的 Q – Q 图时，也可以得到类似的结论，尤其是 Q – Q 图很好地可视化了肥尾（点位于左侧的直线下方与点位于右侧的直线上方）。

```
In [27]: return_qqplot(rets[symbol].values, symbol)
```

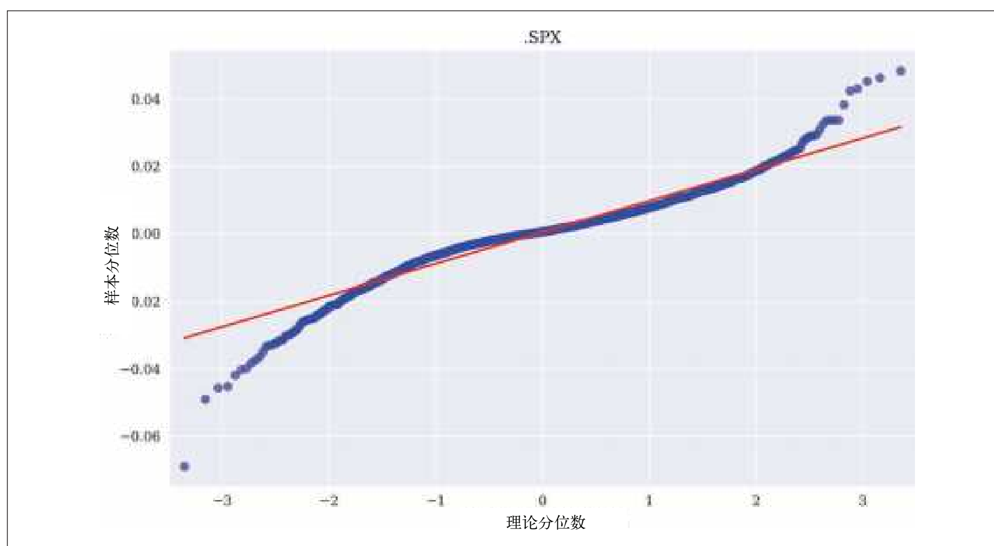


图 4-23：标准普尔 500 指数对数收益率的 Q – Q 图

下面的 Python 代码针对从数据集中选择的金融时间序列，对实际金融收益率的正态性进行了统计检验。实际金融收益率通常无法通过这些测试。因此，可以肯定的是，如果要描述

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

金融现实，则很难对金融收益率进行正态性假设。

```
In [28]: symbols = ['.SPX', 'AMZN.O', 'EUR=', 'GLD']

In [29]: for sym in symbols:
             print('\n{}'.format(sym))
             print(45 * '=')
             print_statistics(rets[sym].values)

         .SPX
         =============================================
         RETURN SAMPLE STATISTICS
         ---------------------------------------------
         Skew of Sample Log Returns -0.497160
         Skew Normal Test p-value    0.000000
         ---------------------------------------------
         Kurt of Sample Log Returns  4.598167
         Kurt Normal Test p-value    0.000000
         ---------------------------------------------
         Normal Test p-value         0.000000
         ---------------------------------------------

         AMZN.O
         =============================================
         RETURN SAMPLE STATISTICS
         ---------------------------------------------
         Skew of Sample Log Returns  0.135268
         Skew Normal Test p-value    0.005689
         ---------------------------------------------
         Kurt of Sample Log Returns  7.344837
         Kurt Normal Test p-value    0.000000
         ---------------------------------------------
         Normal Test p-value         0.000000
         ---------------------------------------------

         EUR=
         =============================================
         RETURN SAMPLE STATISTICS
         ---------------------------------------------
         Skew of Sample Log Returns -0.053959
         Skew Normal Test p-value    0.268203
         ---------------------------------------------
         Kurt of Sample Log Returns  1.780899
         Kurt Normal Test p-value    0.000000
         ---------------------------------------------
         Normal Test p-value         0.000000
         ---------------------------------------------

         GLD
         =============================================
         RETURN SAMPLE STATISTICS
         ---------------------------------------------
         Skew of Sample Log Returns -0.581025
         Skew Normal Test p-value    0.000000
         ---------------------------------------------
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
Kurt of Sample Log Returns   5.899701
Kurt Normal Test p-value     0.000000
---------------------------------------------
Normal Test p-value          0.000000
---------------------------------------------
```

正态性假设

尽管正态性假设可以很好地近似现实世界中许多现象（比如物理学中的问题），但它并不适合用在金融收益率上，因为这样做可能会带来风险。几乎没有金融收益率样本数据集能够通过统计正态性检验。除了在一些领域被证明是有用的以外，这一假设在如此多的金融模型中被发现的一个主要原因是它导致了优雅且相对简单的数学模型、计算和证明。

4.5.2　线性关系

与金融模型和理论中“无处不在”的正态性假设类似，变量之间的**线性关系**似乎是另一个广泛存在的基准。本节讨论了一个重要的问题，即 CAPM 中假设的股票 beta 系数与其预期（实际）收益率之间是线性关系。一般来说，beta 系数越高，在市场表现良好的情况下，预期收益率就越高，这是由 beta 系数本身给定的固定比例所决定的。

回想一下上一节中对所选择的科技股的 beta 系数、CAPM 预期收益率和实际收益率的计算，为方便起见，下面的 Python 代码重复了这些内容。这一次，beta 系数也被添加到了结果的 DataFrame 对象中。

```
In [30]: r = 0.005

In [31]: market = '.SPX'

In [32]: res = pd.DataFrame()

In [33]: for sym in rets.columns[:4]:
             for year in range(2010, 2019):
                 rets_ = rets.loc[f'{year}-01-01':f'{year}-12-31']
                 muM = rets_[market].mean() * 252
                 cov = rets_.cov().loc[sym, market]
                 var = rets_[market].var()
                 beta = cov / var
                 rets_ = rets.loc[f'{year + 1}-01-01':f'{year + 1}-12-31']
                 muM = rets_[market].mean() * 252
                 mu_capm = r + beta * (muM - r)
                 mu_real = rets_[sym].mean() * 252
                 res = res.append(pd.DataFrame({'symbol': sym,
                                                'beta': beta,
                                                'mu_capm': mu_capm,
                                                'mu_real': mu_real},
                                               index=[year + 1]),
                                  sort=True)
```

下面计算线性回归的 R^2 值，其中以 β 为自变量，以给定市场投资组合表现的 CAPM 预期

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

收益率为因变量。R^2 指的是**决定系数**，它会以简单平均值的形式衡量模型相对于基线预测器的表现。线性回归只能解释 CAPM 预期收益率中约 10% 的变异性，这是一个很低的值，图 4-24 也证实了这一点。

```
In [34]: from sklearn.metrics import r2_score

In [35]: reg = np.polyfit(res['beta'], res['mu_capm'], deg=1)
         res['mu_capm_ols'] = np.polyval(reg, res['beta'])

In [36]: r2_score(res['mu_capm'], res['mu_capm_ols'])
Out[36]: 0.09272355783573516

In [37]: res.plot(kind='scatter', x='beta', y='mu_capm', figsize=(10, 6))
         x = np.linspace(res['beta'].min(), res['beta'].max())
         plt.plot(x, np.polyval(reg, x), 'g--', label='regression')
         plt.legend();
```

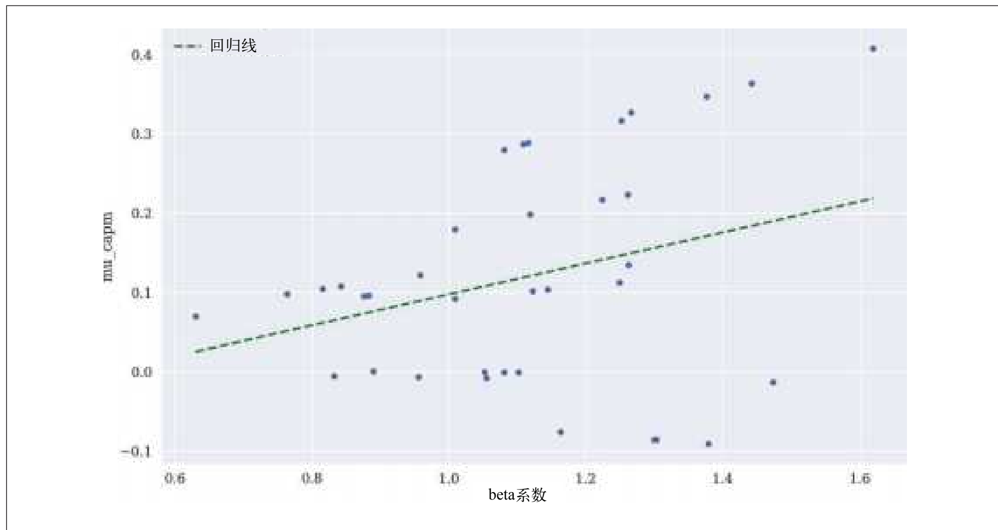


图 4-24：预期 CAPM 收益率与 beta 系数（包括线性回归）

对于**现实收益率**，线性回归的解释能力更低，约为 4.5%（参见图 4-25）。线性回归恢复了 beta 系数和股票收益率之间的正相关关系，即“beta 系数越高，在市场投资组合表现良好的情况下，收益率就越高”，如回归线的正斜率所示。然而，它们只能解释观察到的股票收益率总体变化的一小部分。

```
In [38]: reg = np.polyfit(res['beta'], res['mu_real'], deg=1)
         res['mu_real_ols'] = np.polyval(reg, res['beta'])

In [39]: r2_score(res['mu_real'], res['mu_real_ols'])
Out[39]: 0.04466919444752959

In [40]: res.plot(kind='scatter', x='beta', y='mu_real', figsize=(10, 6))
         x = np.linspace(res['beta'].min(), res['beta'].max())
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
plt.plot(x, np.polyval(reg, x), 'g--', label='regression')
plt.legend();
```

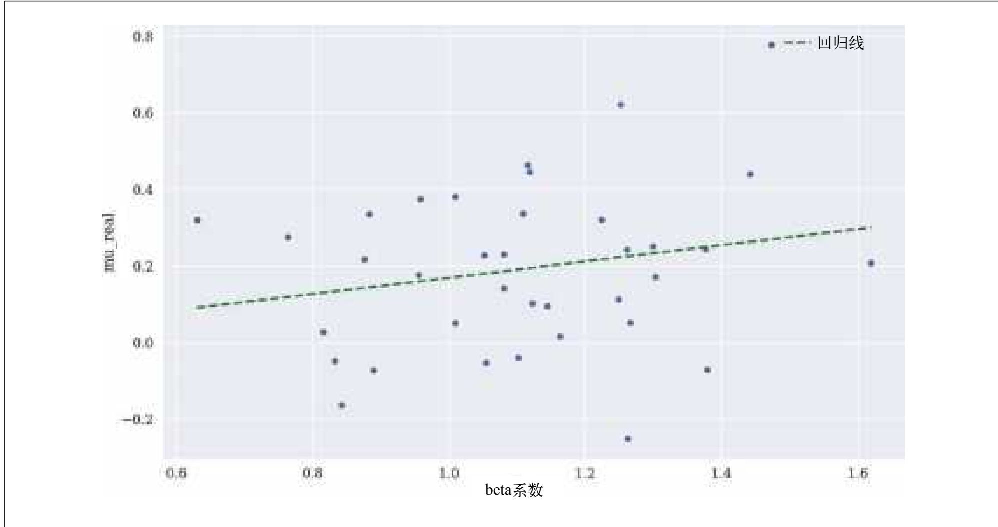


图 4-25：现实 CAPM 收益率与 beta 系数（包括线性回归）

线性关系

与正态性假设一样，在物理世界中通常可以观察到线性关系。然而，在金融领域中，几乎没有任何一种情况是变量之间以明显的线性方式相互依赖。从建模的角度来看，和正态性假设一样，线性关系会导致优雅且相对简单的数学模型、计算和证明。此外，金融计量经济学中的标准工具（OLS 回归）非常适合处理数据中的线性关系，这也是人们常常有意选择正态性和线性以方便对金融模型和理论进行构建的主要原因。

4.6　结论

几个世纪以来，严谨的数据生成和分析推动了科学的发展。然而，过去，金融学是以简化的金融市场数学模型为基础的规范性理论，这些理论依赖于收益率正态性和线性关系等假设。（金融）数据的普遍且全面的可用性导致了焦点从**理论优先**的方式转向**数据驱动型**金融。基于真实金融数据的例子表明，许多流行的金融模型和理论在面对金融市场现实时无法发挥作用。尽管它们很优雅，但在捕捉金融市场的复杂性、变化的本质和非线性方面可能过于简单。

4.7　Python代码段

下面的 Python 文件包含了一些辅助函数，以用于简化 NLP 中的某些任务。

专为量化交易设计的实时行情数据API: www.alltick.co

AllTick

实时行情数据接口

专为量化交易打造

全方位的市场行情数据接口

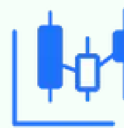
包含实时和历史行情

外汇API

来自世界领先银行机构的 100 多种货币对的逐笔更新。

商品API

所有贵金属（如黄金、白银）和所有能源类别的实时和历史商品数据 API。



股票API

适用于 170,000+ 美国和香港股票的实时和历史股票数据 API。

加密货币API

来自所有主要加密货币交易所的实时和历史加密货币数据，统一在一个易于使用的 API 中。

量化交易神器
回测必备！！

可靠的行情源

我们的系统具有 99.95% 的 SLA。AllTick 倾向于将质量和正常运行时间的标准提升到下一个级别。

低延时接口实时推送

通过 WebSocket 进行的实时数据流具有超低延迟，平均仅约 170 毫秒。

逐笔更新的高频数据

每个交易的实时推送，每个都可追踪，并且与交易所的实时交易行情完全同步。

马上登录ALLTICK.CO

免费试用!

全美股财报免费送!

官方网站：https://www.alltick.co | Telegram：https://t.me/alltick001

专为量化交易设计的实时行情数据API: www.alltick.co

```
#
# NLP辅助函数
#
# Artificial Intelligence in Finance
# (c) Dr Yves J Hilpisch
# The Python Quants GmbH
#
import re
import nltk
import string
import pandas as pd
from pylab import plt
from wordcloud import WordCloud
from nltk.corpus import stopwords
from nltk.corpus import wordnet as wn
from lxml.html.clean import Cleaner
from sklearn.feature_extraction.text import TfidfVectorizer
plt.style.use('seaborn')

cleaner = Cleaner(style=True, links=True, allow_tags=[''],
                  remove_unknown_tags=False)

stop_words = stopwords.words('english')
stop_words.extend(['new', 'old', 'pro', 'open', 'menu', 'close'])


def remove_non_ascii(s):
    ''' 移除所有非ascii字符
    '''
    return ''.join(i for i in s if ord(i) < 128)

def clean_up_html(t):
    t = cleaner.clean_html(t)
    t = re.sub('[\n\t\r]', ' ', t)
    t = re.sub(' +', ' ', t)
    t = re.sub('<.*?>', '', t)
    t = remove_non_ascii(t)
    return t

def clean_up_text(t, numbers=False, punctuation=False):
    ''' 清理HTML标签和文本中的缩写
    '''
    try:
        t = clean_up_html(t)
    except:
        pass
    t = t.lower()
    t = re.sub(r"what's", "what is ", t)
    t = t.replace('(ap)', '')
    t = re.sub(r"\'ve", " have ", t)
    t = re.sub(r"can't", "cannot ", t)
    t = re.sub(r"n't", " not ", t)
    t = re.sub(r"i'm", "i am ", t)
    t = re.sub(r"\'s", "", t)
    t = re.sub(r"\'re", " are ", t)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
    t = re.sub(r"\'d", " would ", t)
    t = re.sub(r"\'ll", " will ", t)
    t = re.sub(r'\s+', ' ', t)
    t = re.sub(r"\\", "", t)
    t = re.sub(r"\'", "", t)
    t = re.sub(r"\"", "", t)
    if numbers:
        t = re.sub('[^a-zA-Z ?!]+', '', t)
    if punctuation:
        t = re.sub(r'\W+', ' ', t)
    t = remove_non_ascii(t)
    t = t.strip()
    return t

def nltk_lemma(word):
    ''' 返回单词的词源，即字典中的拼写形式
    '''
    lemma = wn.morphy(word)
    if lemma is None:
        return word
    else:
        return lemma

def tokenize(text, min_char=3, lemma=True, stop=True,
             numbers=False):
    ''' 分词并进行转换
    '''
    tokens = nltk.word_tokenize(text)
    tokens = [t for t in tokens if len(t) >= min_char]
    if numbers:
        tokens = [t for t in tokens if t[0].lower()
                  in string.ascii_lowercase]
    if stop:
        tokens = [t for t in tokens if t not in stop_words]
    if lemma:
        tokens = [nltk_lemma(t) for t in tokens]
    return tokens

def generate_word_cloud(text, no, name=None, show=True):
    ''' 生成给定文档的词云位图数据，使用TF/IDF特征矩阵计算词
        的重要性，在词云中表示为词的大小

    参数
    ==========
    text: str
        文本
    no: int
    词云中的词数
    name: str
        图片保存路径
    show: bool
        是否展示所生成的图片
    '''
    tokens = tokenize(text)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
    vec = TfidfVectorizer(min_df=2,
                          analyzer='word',
                          ngram_range=(1, 2),
                          stop_words='english'
                         )
    vec.fit_transform(tokens)
    wc = pd.DataFrame({'words': vec.get_feature_names(),
                       'tfidf': vec.idf_})
    words = ' '.join(wc.sort_values('tfidf', ascending=True)['words'].head(no))
    wordcloud = WordCloud(max_font_size=110,
                          background_color='white',
                          width=1024, height=768,
                          margin=10, max_words=150).generate(words)
    if show:
        plt.figure(figsize=(10, 10))
        plt.imshow(wordcloud, interpolation='bilinear')
        plt.axis('off')
        plt.show()
    if name is not None:
        wordcloud.to_file(name)

def generate_key_words(text, no):
    try:
        tokens = tokenize(text)
        vec = TfidfVectorizer(min_df=2,
                          analyzer='word',
                          ngram_range=(1, 2),
                          stop_words='english'
                         )

        vec.fit_transform(tokens)
        wc = pd.DataFrame({'words': vec.get_feature_names(),
                       'tfidf': vec.idf_})
        words = wc.sort_values('tfidf', ascending=False)['words'].values
        words = [ a for a in words if not a.isnumeric()][:no]
    except:
        words = list()
    return words
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 5 章

机器学习

> 数据主义认为，宇宙是由数据流组成的，任何现象或实体的价值都取决于其对数据处理的贡献……因此，数据主义打破了动物（人类）和机器之间的界限，期望电子化算法最终能破译生化算法并超越它。
>
> ——Yuval Noah Harari，2015 年

> 机器学习是一种科学方法，它遵循生成、测试、丢弃或完善假设等一系列相同的流程。但是一位科学家可能会花费一生来提出和测试几百个假设，而机器学习系统可以在一秒内完成同样的事情。机器学习能使科学发现过程自动化。因此，毫不奇怪，它在给科学带来革命性变化的同时，也给商业带来了革命性的变化。
>
> ——Pedro Domingos，2015 年

本章会**将机器学习作为一个过程**，虽然它使用特定的算法和特定的数据进行说明，但本章讨论的概念和方法本质上是通用的。本章的目标是以一种易于理解和可视化的方式集中展示机器学习的最重要元素。本章的方法本质上具有实用性和说明性，所以省略了大部分技术细节。从这个意义上说，本章为以后更现实的机器学习应用提供了一种蓝图。

5.1 节简要讨论了学习的机器的概念。5.2 节导入并预处理了后面章节中使用的样本数据。样本数据是欧元 / 美元汇率的时间序列。5.3 节在给定样本数据的情况下，实现了 OLS 回归和神经网络估计，并使用 MSE 作为成功的衡量标准。5.4 节讨论了模型容量使模型在评估问题中更成功的作用。5.5 节解释了模型评估（通常基于验证数据子集）在机器学习过程中所起的作用。5.6 节讨论了**高偏差**模型和**高方差**模型的概念以及它们在估计问题中的典型特征。5.7 节解释了交叉验证的概念，以避免由于模型容量过大而导致的过拟合。

VanderPlas（2017）的第 5 章主要使用 `scikit-learn` 这个 Python 包讨论了与本章所涉及的主题类似的主题。Chollet（2017）的第 4 章也有与本章类似的概述，但主要使用了 `Keras`

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

深度学习包。Goodfellow 等（2016）的第 5 章对机器学习和相关重要概念进行了技术和数学方面的概述。

5.1 学习

在正式且更抽象的层面上，通过算法或计算机程序进行**学习**可以像 Mitchell（1997）中那样定义：

> 如果一个计算机程序在 T 类任务中的性能（用 P 衡量）随着经验 E 的增加而提高，则该程序可以从经验 E 中学习关于 T 类任务和回测性能 P 的知识。

如果有一类任务需要执行（比如**评估**或**分类**）且有一个回测性能，比如**均方差**（MSE）或**准确率**，则可以根据任务中的算法经验，通过性能的改善来衡量**学习**。通常根据给定的数据集来描述手头的任务类别，在监督学习的情况下包括特征数据和标签数据，在无监督学习的情况下只包括特征数据。

学习任务与待学习的任务

在通过算法或计算机程序进行学习的定义中，重要的是要注意学习任务和待学习的任务之间的区别。**学习**是指学会如何（最好地）执行某项任务，比如估计或分类。

5.2 数据

本节会介绍将在随后章节中使用的样本数据集。样本数据是基于欧元 / 美元汇率的真实金融时间序列创建的。首先，从 CSV 文件中导入数据，然后将数据重新采样为月度数据，并存储在 `Series` 对象中。

```
In [1]: import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        np.random.seed(100)
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'

In [2]: url = 'http://hilpisch.com/aiif_eikon_eod_data.csv'  ❶

In [3]: raw = pd.read_csv(url, index_col=0, parse_dates=True)['EUR=']  ❶

In [4]: raw.head()
Out[4]: Date
        2010-01-01    1.4323
        2010-01-04    1.4411
        2010-01-05    1.4368
        2010-01-06    1.4412
        2010-01-07    1.4318
        Name: EUR=, dtype: float64
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [5]: raw.tail()
Out[5]: Date
        2019-12-26    1.1096
        2019-12-27    1.1175
        2019-12-30    1.1197
        2019-12-31    1.1210
        2020-01-01    1.1210
        Name: EUR=, dtype: float64

In [6]: l = raw.resample('1M').last()  ❷

In [7]: l.plot(figsize=(10, 6), title='EUR/USD monthly');
```

❶ 导入金融时间序列数据。

❷ 以月为周期对数据重新采样。

图 5-1 显示了金融时间序列。

图 5-1：欧元 / 美元汇率的时间序列（以月为单位）

为了只有单一特征，下面的 Python 代码创建了一个合成特征向量，用来在图表中进行简单的可视化。当然，这一合成特征（自变量）对欧元 / 美元汇率（标签数据，因变量）没有任何解释力。在接下来的内容中，它会把标签数据是连续且瞬时的本质属性抽象出来。本章会将样本数据集视为由一维特征向量和一维标签向量组成的通用数据集。图 5-2 对样本数据集进行了可视化，以表明该样本数据集可以被当作一个**估计问题**。

```
In [8]: l = l.values  ❶
        l -= l.mean()  ❷
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [9]: f = np.linspace(-2, 2, len(l))  ❸

In [10]: plt.figure(figsize=(10, 6))
         plt.plot(f, l, 'ro')
         plt.title('Sample Data Set')
         plt.xlabel('features')
         plt.ylabel('labels');
```

❶ 将标签数据转换为 ndarray 对象。

❷ 从数据元素中减去平均值。

❸ 创建一个合成特征作为 ndarray 对象。

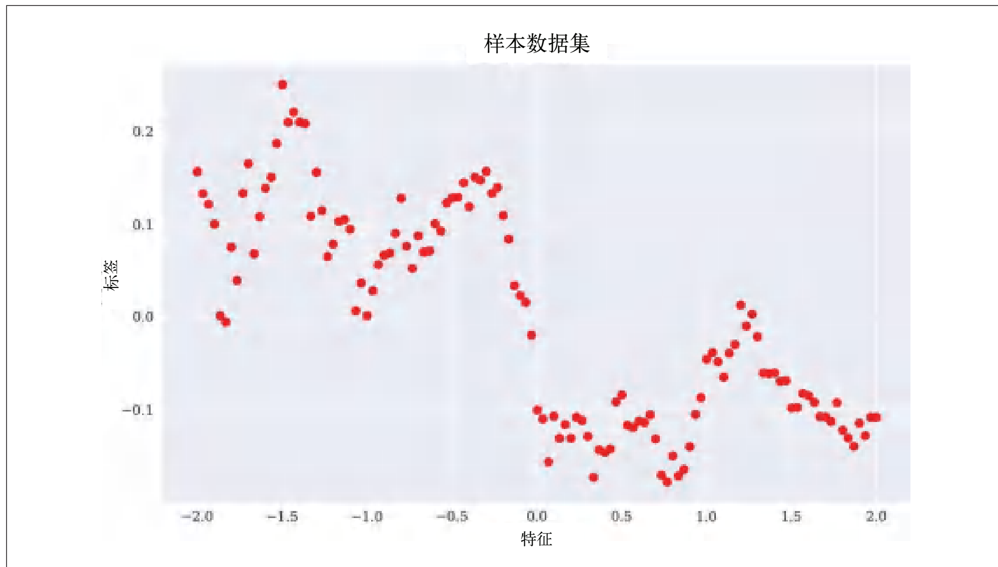


图 5-2：样本数据集

5.3 成功

一般来说，估计问题的成功度量是 MSE，就如第 1 章中所使用的那样。基于 MSE，以标签数据作为相关基准，并以算法在接触到数据集（或部分数据集）后的预测值来判断其是否成功。如第 1 章所述，本节和随后内容将分别考虑两种算法：OLS 回归和神经网络。

首先是 OLS 回归，这个应用非常简单，如下面的 Python 代码所示。如图 5-3 所示，回归结果是包含五阶单项式的回归，并计算了结果的 MSE。

```
In [11]: def MSE(l, p):
             return np.mean((l - p) ** 2)  ❶

In [12]: reg = np.polyfit(f, l, deg=5)  ❷
         reg  ❷
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
Out[12]: array([-0.01910626, -0.0147182 ,  0.10990388,  0.06007211, -0.20833598,
                -0.03275423])

In [13]: p = np.polyval(reg, f)  ❸

In [14]: MSE(l, p)  ❹
Out[14]: 0.0034166422957371025

In [15]: plt.figure(figsize=(10, 6))
         plt.plot(f, l, 'ro', label='sample data')
         plt.plot(f, p, '--', label='regression')
         plt.legend();
```

❶ 函数 MSE 会计算 MSE。

❷ OLS 回归模型的拟合，包括五阶单项式。

❸ 通过 OLS 回归模型的预测给出了最优参数。

❹ 给定预测值的 MSE 值。

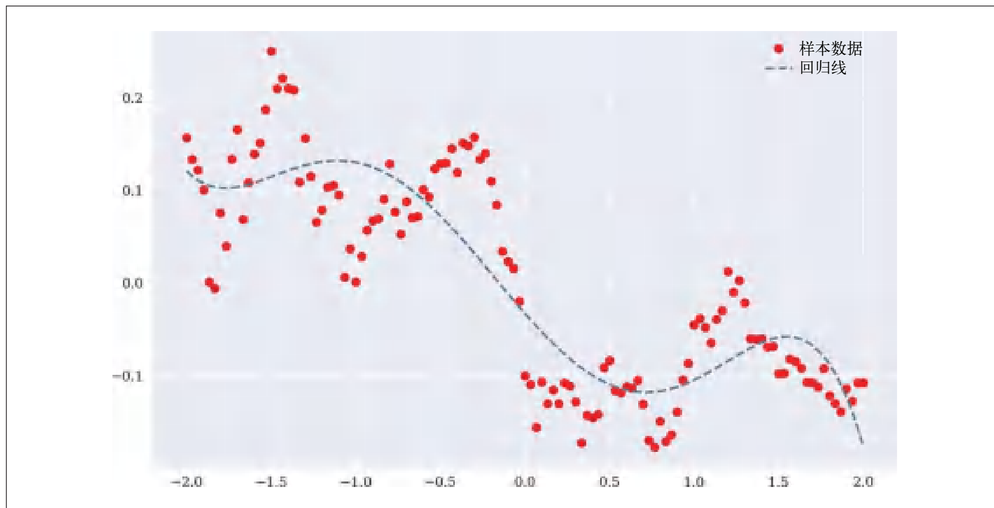


图 5-3：样本数据与三次回归线

OLS 回归一般采用解析解，因此没有迭代学习发生，然而可以通过逐步将更多数据赋给算法来模拟学习过程。下面的 Python 代码实现了 OLS 回归和预测，开始时只有几个样本，然后逐渐增加样本的数量，最终达到数据集的完整长度。回归步骤是基于较小的子集实现的，而预测步骤是基于每一种情况下的全部特征数据实现的。一般来说，随着训练数据集的增加，MSE 会明显下降。

```
In [16]: for i in range(10, len(f) + 1, 20):
             reg = np.polyfit(f[:i], l[:i], deg=3)  ❶
             p = np.polyval(reg, f)  ❷
             mse = MSE(l, p)  ❸
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
    print(f'{i:3d} | MSE={mse}')
 10 | MSE=248628.10681642237
 30 | MSE=731.9382249304651
 50 | MSE=12.236088505004465
 70 | MSE=0.7410590619743301
 90 | MSE=0.0057430617304093275
110 | MSE=0.006492800939555582
```

❶ 基于数据子集的回归步骤。

❷ 基于完整数据集的预测步骤。

❸ 产生的 MSE 值。

神经网络在样本数据上的应用同样非常简单，类似于第 1 章中的情况。图 5-4 显示了神经网络是如何逼近样本数据的。

```
In [17]: import tensorflow as tf
         tf.random.set_seed(100)

In [18]: from keras.layers import Dense
         from keras.models import Sequential
         Using TensorFlow backend.

In [19]: model = Sequential()
         model.add(Dense(256, activation='relu', input_dim=1))  ❶
         model.add(Dense(1, activation='linear'))  ❶
         model.compile(loss='mse', optimizer='rmsprop')

In [20]: model.summary()
         Model: "sequential_1"
         _________________________________________________________________
         Layer (type)                 Output Shape              Param #
         =================================================================
         dense_1 (Dense)              (None, 256)               512
         _________________________________________________________________
         dense_2 (Dense)              (None, 1)                 257
         =================================================================
         Total params: 769
         Trainable params: 769
         Non-trainable params: 0
         _________________________________________________________________

In [21]: %time model.fit(f, l, epochs=1500, verbose=False)  ❷
         CPU times: user 5.89 s, sys: 761 ms, total: 6.66 s
         Wall time: 4.43 s

Out[21]: <keras.callbacks.callbacks.History at 0x7fc05d599d90>

In [22]: p = model.predict(f).flatten()  ❸

In [23]: MSE(l, p)  ❹
Out[23]: 0.0020217512014360102
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [24]: plt.figure(figsize=(10, 6))
         plt.plot(f, l, 'ro', label='sample data')
         plt.plot(f, p, '--', label='DNN approximation')
         plt.legend();
```

❶ 此神经网络是一个只有单个隐藏层的浅层网络。

❷ 设定具有相对较高训练次数的拟合步骤。

❸ 在预测步骤中展平 `ndarray` 对象。

❹ DNN 预测结果的 MSE 值。

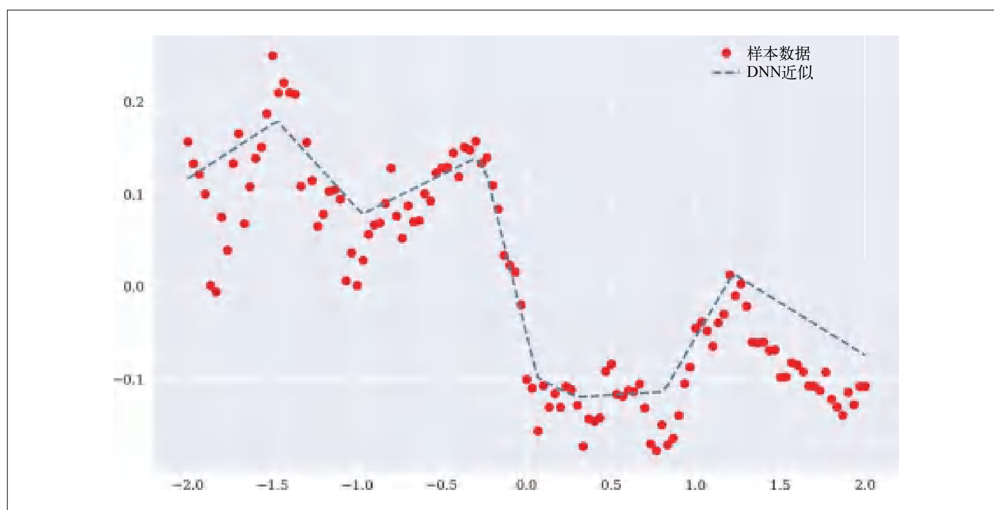


图 5-4：样本数据和 DNN 近似

使用 Keras 软件包，每个学习步骤之后都会存储 MSE 值。图 5-5 显示了 MSE 值（“损失”）是如何随着神经网络训练周期的增加而平均下降的（从图中可以看出）。

```
In [25]: import pandas as pd

In [26]: res = pd.DataFrame(model.history.history)

In [27]: res.tail()
Out[27]:           loss
         1495  0.001547
         1496  0.001520
         1497  0.001456
         1498  0.001356
         1499  0.001325

In [28]: res.iloc[100:].plot(figsize=(10, 6))
         plt.ylabel('MSE')
         plt.xlabel('epochs');
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

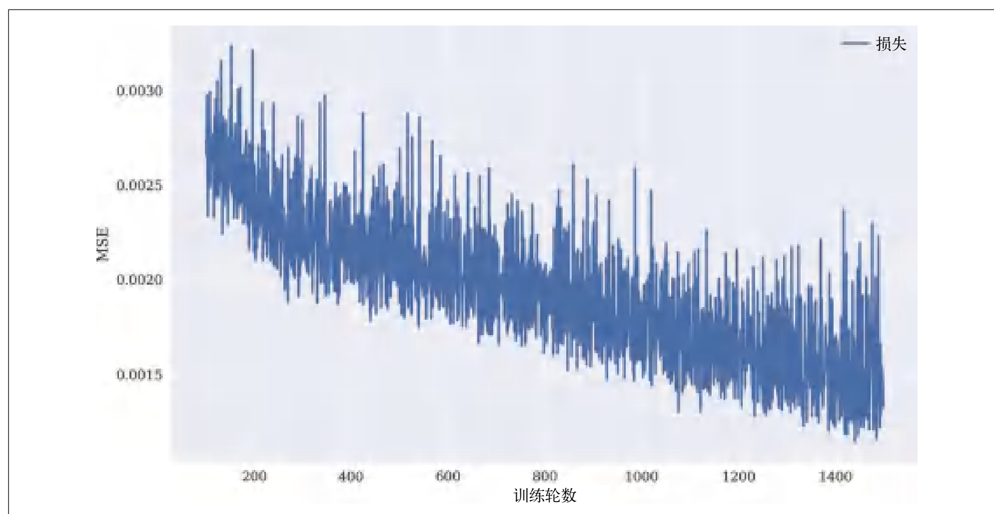


图 5-5：MSE 值与训练轮数的关系

5.4 容量

模型或算法的**容量**定义了模型或算法基本上可以学习什么类型的函数或关系。对于仅基于单项式的 OLS 回归，只有一个参数定义了模型的容量：要使用的最高阶单项式的次数。如果将该阶数参数设置为 deg=3，则 OLS 回归模型可以学习常数型、线性型、二次型或三次型的函数关系。参数 deg 越大，OLS 回归模型的容量越大。

下面的 Python 代码从 deg=1 开始，以 2 为增量增大阶数，MSE 值随着阶数参数的增大而单调递减。图 5-6 显示了所有考虑阶数的回归线。

```
In [29]: reg = {}
         for d in range(1, 12, 2):
             reg[d] = np.polyfit(f, l, deg=d)  ❶
             p = np.polyval(reg[d], f)
             mse = MSE(l, p)
             print(f'{d:2d} | MSE={mse}')
          1 | MSE=0.005322474034260403
          3 | MSE=0.004353110724143185
          5 | MSE=0.0034166422957371025
          7 | MSE=0.0027389501772354025
          9 | MSE=0.001411961626330845
         11 | MSE=0.0012651237868752322

In [30]: plt.figure(figsize=(10, 6))
         plt.plot(f, l, 'ro', label='sample data')
         for d in reg:
             p = np.polyval(reg[d], f)
             plt.plot(f, p, '--', label=f'deg={d}')
         plt.legend();
```

❶ deg 不同值的回归步骤。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

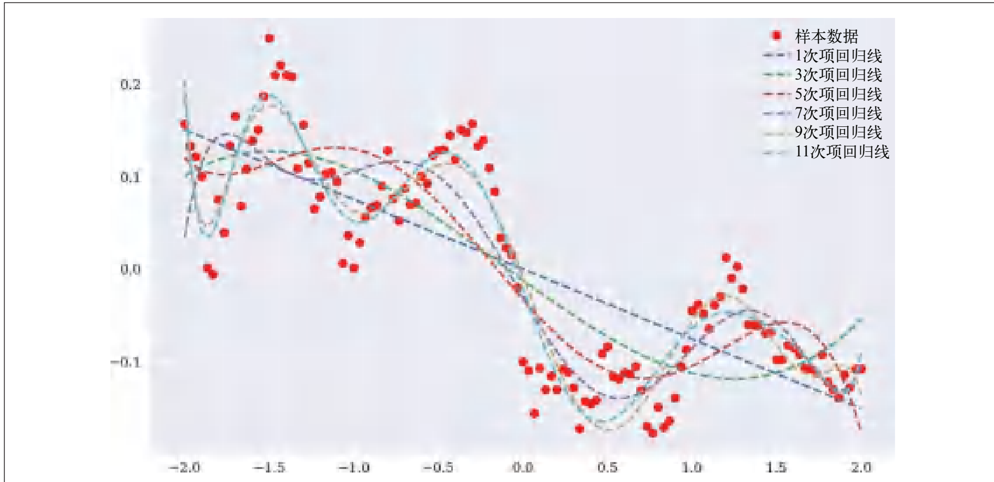


图 5-6：不同最高阶数的回归线

神经网络的容量取决于许多**超参数**，一般而言包括：

- 隐藏层的数量；
- 每个隐藏层的隐藏单元数量。

这两个超参数共同定义了神经网络中可训练参数（权重）的数量。上一节中的神经网络模型的可训练参数相对较少。例如，只要再增加一个相同大小的层，就可以显著增加可训练参数的数量。尽管可能需要增加训练轮数，但是容量较大的神经网络的 MSE 值会显著降低，并且拟合在视觉上也会更好，如图 5-7 所示。

```
In [31]: def create_dnn_model(hl=1, hu=256):
             ''' 创建Keras DNN模型的函数

             参数
             ==========
             hl: int
                 隐藏层数量
             hu: int
                 每个隐藏层的单元数量
             '''
             model = Sequential()
             for _ in range(hl):
                 model.add(Dense(hu, activation='relu', input_dim=1))  ❶
             model.add(Dense(1, activation='linear'))
             model.compile(loss='mse', optimizer='rmsprop')
             return model

In [32]: model = create_dnn_model(3)  ❷

In [33]: model.summary()  ❸
         Model: "sequential_2"
         _________________________________________________________________
         Layer (type)                 Output Shape              Param #
         =================================================================
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
dense_3 (Dense)              (None, 256)               512
_________________________________________________________________
dense_4 (Dense)              (None, 256)               65792
_________________________________________________________________
dense_5 (Dense)              (None, 256)               65792
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 257
=================================================================
Total params: 132,353
Trainable params: 132,353
Non-trainable params: 0
_________________________________________________________________

In [34]: %time model.fit(f, l, epochs=2500, verbose=False)
         CPU times: user 34.9 s, sys: 5.91 s, total: 40.8 s
         Wall time: 15.5 s

Out[34]: <keras.callbacks.callbacks.History at 0x7fc03fc18890>

In [35]: p = model.predict(f).flatten()

In [36]: MSE(l, p)
Out[36]: 0.00046612284916401614

In [37]: plt.figure(figsize=(10, 6))
         plt.plot(f, l, 'ro', label='sample data')
         plt.plot(f, p, '--', label='DNN approximation')
         plt.legend();
```

❶ 为神经网络添加潜在的多个隐藏层。

❷ 具有 3 个隐藏层的 DNN。

❸ 这个总结展示了增加的可训练参数数量（增加的容量）。

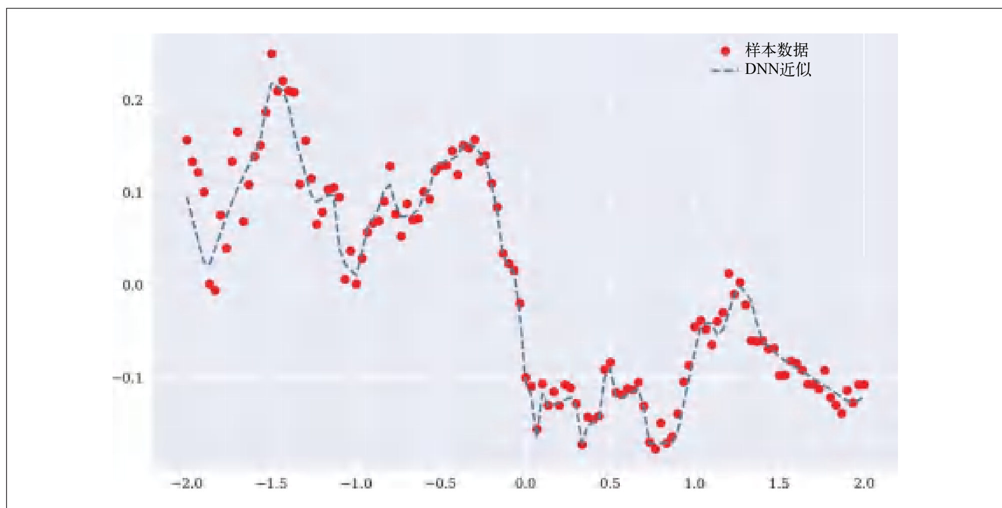


图 5-7：样本数据和 DNN 近似（更大容量）

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

5.5 评估

前面章节分析的重点是估计算法在整个样本数据集上的性能。一般来说，当在同一数据集上对模型或算法进行训练和评估时，模型或算法的容量会直接影响其性能。然而，这是机器学习中“简单易用的情况”，更复杂且有趣的是，应使用经过训练的模型或算法对其从未见过的数据进行泛化。例如，这种泛化可以是根据股票价格的历史预测（估计）未来股票价格，或者根据现有债务人的数据将潜在债务人分为“有信用”或“无信用”。

尽管术语**预测**通常在估计的上下文中可自由使用，但考虑到用于训练的特征数据集，真正的预测可能需要预测一些事先未知且从未见过的东西。同样，对未来股票价格的预测是时间意义上的真实预测的一个很好的例子。

通常，给定的数据集会被分为多个子集，每个子集有不同的用途。

训练数据集

  这是用于算法训练的子集。

验证数据集

  这是用于在训练期间验证算法性能的子集，该数据集与训练数据集不同。

测试数据集

  这是训练完成后仅在其上测试训练算法的子集。

通过对验证数据集应用一个（当前）经过训练的算法获得的结果，可能会反映在训练本身上（比如，对模型超参数进行调整会影响训练结果）。另外，在测试数据集上测试训练算法的结果不应反映在训练本身上或超参数中。

下面的 Python 代码任意选择了 25% 的样本数据进行测试，在训练（学习）完成之前，模型或算法不会“看到”这些数据。同样，另外 25% 的样本数据被保留以用于验证，该数据用于在训练步骤和许多学习迭代期间监控性能。剩余的 50% 用于训练（学习）本身。[1] 给定样本数据集，打乱顺序随机填充所有样本数据子集是有意义的。

```
In [38]: te = int(0.25 * len(f))  ❶
         va = int(0.25 * len(f))  ❷

In [39]: np.random.seed(100)
         ind = np.arange(len(f))  ❸
         np.random.shuffle(ind)  ❸

In [40]: ind_te = np.sort(ind[:te])  ❹
         ind_va = np.sort(ind[te:te + va])  ❹
         ind_tr = np.sort(ind[te + va:])  ❹

In [41]: f_te = f[ind_te]  ❺
         f_va = f[ind_va]  ❺
         f_tr = f[ind_tr]  ❺
```

注 1：通常，在此上下文中提到的经验法则是“60%、20%、20%”，用于将给定数据集拆分为训练数据集、验证数据集和测试数据集。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [42]: l_te = l[ind_te]  ❻
         l_va = l[ind_va]  ❻
         l_tr = l[ind_tr]  ❻
```

❶ 测试数据集样本的数量。

❷ 验证数据集样本的数量。

❸ 完整数据集的随机索引。

❹ 为数据子集生成的排序索引。

❺ 生成数据子集的特征。

❻ 生成数据子集的标签。

随机采样

训练数据集、验证数据集和测试数据集的随机分组对于既非序列也非时间性质的数据集是一种常见且有用的技术。然而，当处理金融时间序列时，通常要避免对数据进行洗牌，因为它会破坏时间结构，并使用较晚的样本进行训练，对较早的样本进行测试，将预见偏差偷偷带入这个过程。

基于训练数据子集和验证数据子集，下面的 Python 代码实现了不同 deg 参数值的回归，并计算了两个数据子集上的预测的 MSE 值。虽然训练数据集上的 MSE 值单调减小，但验证数据集上的 MSE 值往往在某一参数值达到最小值后又再次增大。这个现象就是所谓的**过拟合**。图 5-8 显示了不同 deg 值的回归拟合情况，并比较了训练数据集和验证数据集的拟合情况。

```
In [43]: reg = {}
         mse = {}
         for d in range(1, 22, 4):
             reg[d] = np.polyfit(f_tr, l_tr, deg=d)
             p = np.polyval(reg[d], f_tr)
             mse_tr = MSE(l_tr, p)  ❶
             p = np.polyval(reg[d], f_va)
             mse_va = MSE(l_va, p)  ❷
             mse[d] = (mse_tr, mse_va)
             print(f'{d:2d} | MSE_tr={mse_tr:7.5f} | MSE_va={mse_va:7.5f}')
          1 | MSE_tr=0.00574 | MSE_va=0.00492
          5 | MSE_tr=0.00375 | MSE_va=0.00273
          9 | MSE_tr=0.00132 | MSE_va=0.00243
         13 | MSE_tr=0.00094 | MSE_va=0.00183
         17 | MSE_tr=0.00060 | MSE_va=0.00153
         21 | MSE_tr=0.00046 | MSE_va=0.00837

In [44]: fig, ax = plt.subplots(2, 1, figsize=(10, 8), sharex=True)
         ax[0].plot(f_tr, l_tr, 'ro', label='training data')
         ax[1].plot(f_va, l_va, 'go', label='validation data')
         for d in reg:
             p = np.polyval(reg[d], f_tr)
             ax[0].plot(f_tr, p, '--', label=f'deg={d} (tr)')
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        p = np.polyval(reg[d], f_va)
        plt.plot(f_va, p, '--', label=f'deg={d} (va)')
        ax[0].legend()
        ax[1].legend();
```

❶ 训练数据集的 MSE 值。

❷ 验证数据集的 MSE 值。

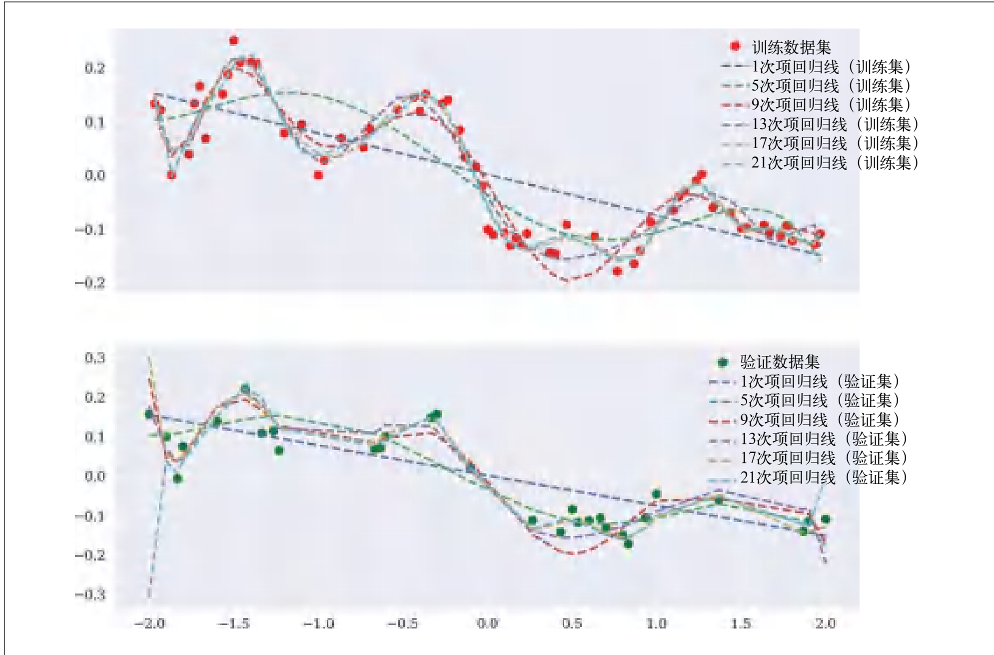


图 5-8：包括回归拟合的训练数据集和验证数据集

通过 Keras 和神经网络模型，可以针对每个学习步骤监控验证数据集的性能。当没有观察到进一步的改进时（比如，在训练数据集上的性能方面），也可以使用回调函数提前停止模型训练。下面的 Python 代码使用了这样的回调函数。图 5-9 显示了神经网络对训练数据集和验证数据集的预测。

```
In [45]: from keras.callbacks import EarlyStopping

In [46]: model = create_dnn_model(2, 256)

In [47]: callbacks = [EarlyStopping(monitor='loss',  ❶
                                    patience=100,  ❷
                                    restore_best_weights=True)]  ❸

In [48]: %%time
         model.fit(f_tr, l_tr, epochs=3000, verbose=False,
                   validation_data=(f_va, l_va),  ❹
                   callbacks=callbacks)  ❺
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        CPU times: user 8.07 s, sys: 1.33 s, total: 9.4 s
        Wall time: 4.81 s

Out[48]: <keras.callbacks.callbacks.History at 0x7fc0438b47d0>

In [49]: fig, ax = plt.subplots(2, 1, sharex=True, figsize=(10, 8))
         ax[0].plot(f_tr, l_tr, 'ro', label='training data')
         p = model.predict(f_tr)
         ax[0].plot(f_tr, p, '--', label=f'DNN (tr)')
         ax[0].legend()
         ax[1].plot(f_va, l_va, 'go', label='validation data')
         p = model.predict(f_va)
         ax[1].plot(f_va, p, '--', label=f'DNN (va)')
         ax[1].legend();
```

❶ 根据训练数据集的 MSE 值停止学习。

❷ 在一定的训练轮数后没有显示出改善就停止学习。

❸ 当学习停止时保存最好的权重。

❹ 指定验证数据子集。

❺ 将回调函数传给 `fit()` 方法。

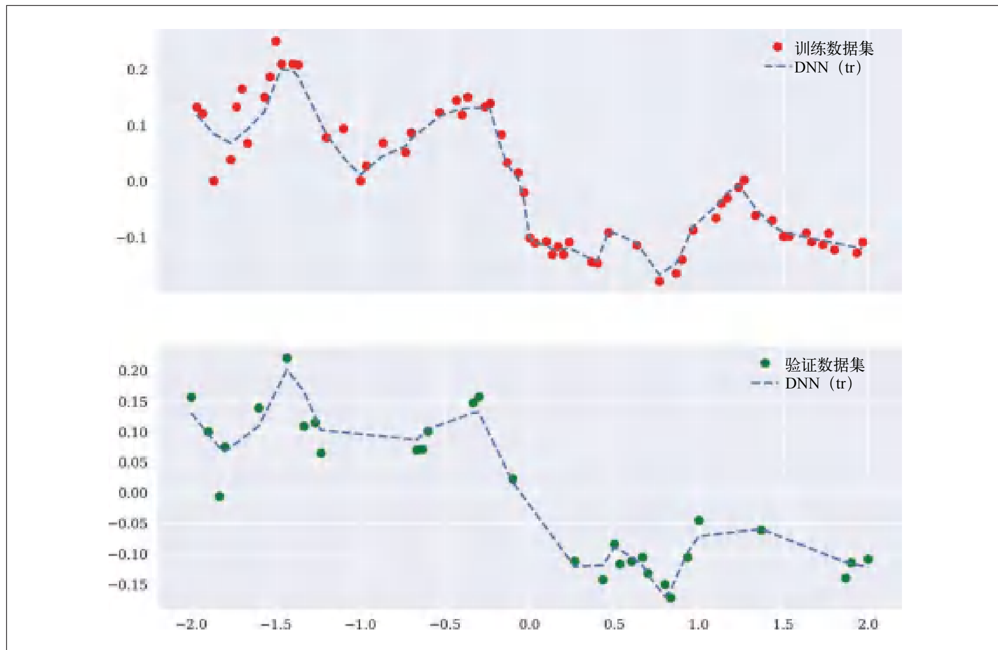


图 5-9：包括 DNN 预测的训练数据集和验证数据集

Keras 允许对模型每一轮训练的两个数据集上 MSE 值的变化进行分析。从图 5-10 中可以看出，MSE 值随着训练轮数的增加而减小，但只是平均减小，而不是单调减小。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [50]: res = pd.DataFrame(model.history.history)

In [51]: res.tail()
Out[51]:       val_loss      loss
         1375  0.000854  0.000544
         1376  0.000685  0.000473
         1377  0.001326  0.000942
         1378  0.001026  0.000867
         1379  0.000710  0.000500

In [52]: res.iloc[35::25].plot(figsize=(10, 6))
         plt.ylabel('MSE')
         plt.xlabel('epochs');
```

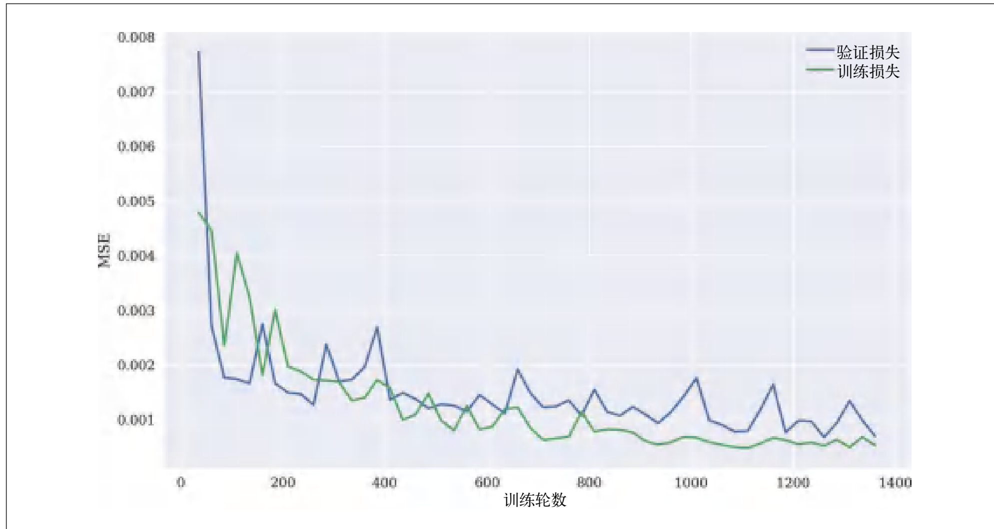


图 5-10：DNN 模型在训练数据集和验证数据集上的 MSE 值

在 OLS 回归的情况下，可能会为阶数参数选择一个高但不太高的值，比如 deg=9。神经网络模型的参数化会在训练结束时自动给出最佳模型配置。图 5-11 比较了两个模型之间的预测以及与测试数据集的预测。鉴于样本数据的性质，神经网络的测试数据集性能更好一点儿并不令人惊讶。

```
In [53]: p_ols = np.polyval(reg[5], f_te)
         p_dnn = model.predict(f_te).flatten()

In [54]: MSE(l_te, p_ols)
Out[54]: 0.0038960346771028356

In [55]: MSE(l_te, p_dnn)
Out[55]: 0.000705705678438721

In [56]: plt.figure(figsize=(10, 6))
         plt.plot(f_te, l_te, 'ro', label='test data')
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
plt.plot(f_te, p_ols, '--', label='OLS prediction')
plt.plot(f_te, p_dnn, '-.', label='DNN prediction');
plt.legend();
```

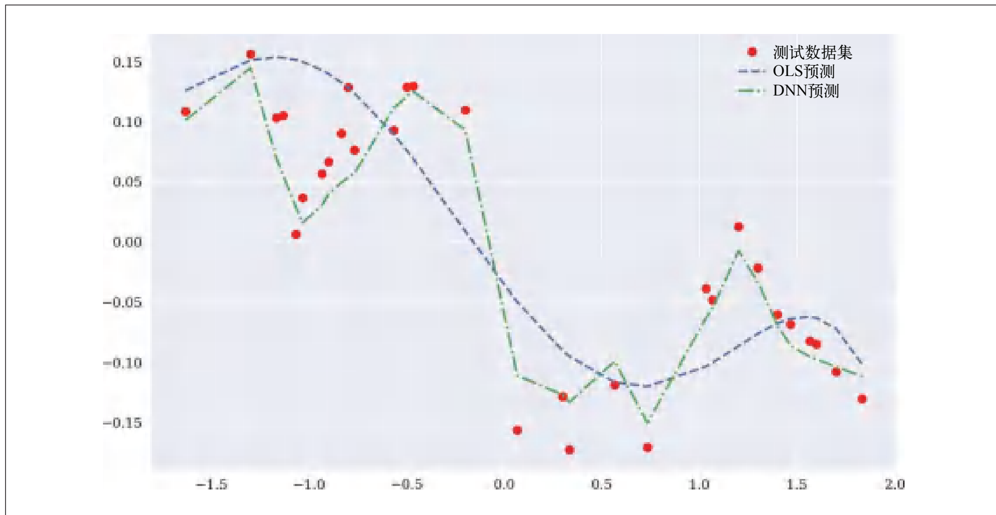


图 5-11：OLS 回归和 DNN 模型的测试数据集和预测

5.6 偏差和方差

一般来说，机器学习中的一个主要问题是**过拟合**，特别是当机器学习算法应用于金融数据时。当验证数据集和测试数据集的性能比训练数据集的性能差时，就是模型过度拟合了其训练数据，使用 OLS 回归的示例可以在视觉上和数字上说明这个问题。

下面的 Python 代码使用较小的子集进行训练和验证，并实现了线性回归和高阶回归。如图 5-12 所示，线性回归拟合在训练数据集上有**较高的偏差**，预测数据和标签数据之间的绝对差异相对较大。高阶拟合显示出了**较高的方差**，它精确地命中所有训练数据点，但为实现完美拟合，拟合本身变化会很大。

```
In [57]: f_tr = f[:20:2]  ❶
         l_tr = l[:20:2]  ❶

In [58]: f_va = f[1:20:2]  ❷
         l_va = l[1:20:2]  ❷

In [59]: reg_b = np.polyfit(f_tr, l_tr, deg=1)  ❸

In [60]: reg_v = np.polyfit(f_tr, l_tr, deg=9, full=True)[0]  ❹

In [61]: f_ = np.linspace(f_tr.min(), f_va.max(), 75)  ❺

In [62]: plt.figure(figsize=(10, 6))
         plt.plot(f_tr, l_tr, 'ro', label='training data')
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
plt.plot(f_va, l_va, 'go', label='validation data')
plt.plot(f_, np.polyval(reg_b, f_), '--', label='high bias')
plt.plot(f_, np.polyval(reg_v, f_), '--', label='high variance')
plt.ylim(-0.2)
plt.legend(loc=2);
```

❶ 较少特征的数据子集。

❷ 较少标签的数据子集。

❸ 高偏差 OLS 回归（线性）。

❹ 高方差 OLS 回归（高阶）。

❺ 用于绘图的放大的特征数据集。

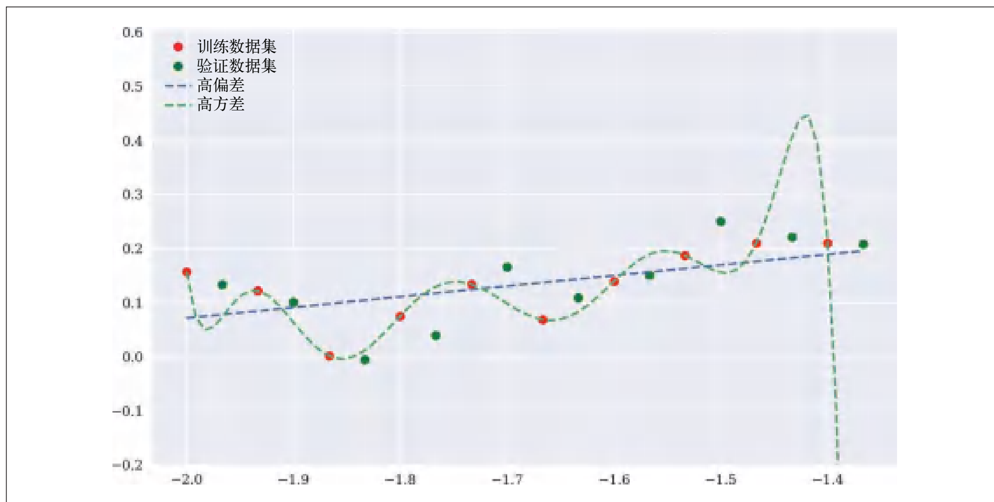


图 5-12：高偏差和高方差的 OLS 回归拟合

图 5-12 显示，在本例中，高偏差拟合比高方差拟合在训练数据集上的表现更差。但高方差拟合在很大程度上是过拟合，在验证数据集上表现得更差。这可以通过比较所有情况下的回测性能来说明。以下 Python 代码不仅计算 MSE 值，还计算 R^2 值。

```
In [63]: from sklearn.metrics import r2_score

In [64]: def evaluate(reg, f, l):
             p = np.polyval(reg, f)
             bias = np.abs(l - p).mean()  ❶
             var = p.var()  ❷
             msg = f'MSE={MSE(l, p):.4f} | R2={r2_score(l, p):9.4f} | '
             msg += f'bias={bias:.4f} | var={var:.4f}'
             print(msg)

In [65]: evaluate(reg_b, f_tr, l_tr)  ❸
         MSE=0.0026 | R2=   0.3484 | bias=0.0423 | var=0.0014
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [66]: evaluate(reg_b, f_va, l_va)  ❹
         MSE=0.0032 | R2=   0.4498 | bias=0.0460 | var=0.0014

In [67]: evaluate(reg_v, f_tr, l_tr)  ❺
         MSE=0.0000 | R2=   1.0000 | bias=0.0000 | var=0.0040

In [68]: evaluate(reg_v, f_va, l_va)  ❻
         MSE=0.8752 | R2=-149.2658 | bias=0.3565 | var=0.7539
```

❶ 平均差异绝对值作为模型偏差。

❷ 模型预测的方差作为模型方差。

❸ **高偏差**模型在**训练数据集**上的性能。

❹ **高偏差**模型在**验证数据集**上的性能。

❺ **高方差**模型在**训练数据集**上的性能。

❻ **高方差**模型在**验证数据集**上的性能。

结果表明，在训练数据集和验证数据集上，高偏差模型的性能大致相当。相比之下，高方差模型在训练数据集上的性能很好，在验证数据集上的性能则相当差。

5.7 交叉验证

避免过拟合的标准方法是**交叉验证**，在此过程中对多个训练数据集和验证数据集进行测试。`scikit-learn` 包提供了以标准化方式实现交叉验证的功能，函数 `cross_val_score` 可以用于任何 `scikit-learn` 模型对象。

下面的代码使用 `scikit-learn` 的多项式 OLS 回归模型在完整的样本数据集上实现 OLS 回归方法，对最高多项式进行了不同阶数的五折交叉验证。平均而言，回归中最高阶数越高，交叉验证得分就越差。当使用样本数据集中前 20% 的数据（图 5-3 中左侧的数据）或最后 20% 的数据（图 5-3 中右侧的数据）进行验证时，观察到的结果尤为糟糕。但是，在样本数据集中间 20% 的数据处，能观察到最佳验证分数。

```
In [69]: from sklearn.model_selection import cross_val_score
         from sklearn.preprocessing import PolynomialFeatures
         from sklearn.linear_model import LinearRegression
         from sklearn.pipeline import make_pipeline

In [70]: def PolynomialRegression(degree=None, **kwargs):
             return make_pipeline(PolynomialFeatures(degree),
                                  LinearRegression(**kwargs))  ❶

In [71]: np.set_printoptions(suppress=True,
                 formatter={'float': lambda x: f'{x:12.2f}'})  ❷

In [72]: print('\nCross-validation scores')
         print(74 * '=')
         for deg in range(0, 10, 1):
             model = PolynomialRegression(deg)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
    cvs = cross_val_score(model, f.reshape(-1, 1), l, cv=5)  ❸
    print(f'deg={deg} | ' + str(cvs.round(2)))

Cross-validation scores
==========================================================================
deg=0 | [       -6.07       -7.34      -0.09       -6.32            -8.69]
deg=1 | [       -0.28       -1.40       0.16       -1.66            -4.62]
deg=2 | [       -3.48       -2.45       0.19       -1.57           -12.94]
deg=3 | [       -0.00       -1.24       0.32       -0.48           -43.62]
deg=4 | [     -222.81       -2.88       0.37       -0.32          -496.61]
deg=5 | [     -143.67       -5.85       0.49        0.12         -1241.04]
deg=6 | [    -4038.96      -14.71       0.49       -0.33          -317.32]
deg=7 | [    -9937.83      -13.98       0.64        0.22        -18725.61]
deg=8 | [    -3514.36      -11.22      -0.15       -6.29       -298744.18]
deg=9 | [    -7454.15       -0.91       0.15       -0.41        -13580.75]
```

❶ 创建多项式回归模型类。

❷ 调整 numpy 的默认打印设置。

❸ 实施五折交叉验证。

Keras 提供了包装类来使用带有 scikit-learn 接口的 Keras 模型对象，比如 cross_val_score 函数。下面的例子使用了 KerasRegressor 类来包装神经网络模型，并对其应用交叉验证。与 OLS 回归交叉验证分数相比，测试的两个网络的交叉验证分数始终更高。在这个例子中，神经网络容量并没有发挥太大作用。

```
In [73]: np.random.seed(100)
         tf.random.set_seed(100)
         from keras.wrappers.scikit_learn import KerasRegressor

In [74]: model = KerasRegressor(build_fn=create_dnn_model,
                                verbose=False, epochs=1000,
                                hl=1, hu=36)  ❶

In [75]: %time cross_val_score(model, f, l, cv=5)  ❷
         CPU times: user 18.6 s, sys: 2.17 s, total: 20.8 s
         Wall time: 14.6 s

Out[75]: array([       -0.02,       -0.01,         -0.00,      -0.00,
                -0.01])

In [76]: model = KerasRegressor(build_fn=create_dnn_model,
                                verbose=False, epochs=1000,
                                hl=3, hu=256)  ❸

In [77]: %time cross_val_score(model, f, l, cv=5)  ❹
         CPU times: user 1min 5s, sys: 11.6 s, total: 1min 16s
         Wall time: 30.1 s

Out[77]: array([       -0.08,       -0.00,         -0.00,      -0.00,
                -0.05])
```

❶ 低容量神经网络的包装类。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❷ **低**容量神经网络的交叉验证。

❸ **高**容量神经网络的包装类。

❹ **高**容量神经网络的交叉验证。

避免过拟合

一般而言，过拟合，即模型在训练数据集上比在验证数据集和测试数据集上表现得更好，在机器学习中是要避免的，尤其是在金融中。适当的评估程序和分析（比如交叉验证）有助于防止过拟合并能找到足够的模型容量。

5.8 结论

本章介绍了机器学习过程的蓝图，主要内容如下。

学习

机器学习到底意味着什么？

数据

使用哪些原始数据以及哪些（预处理）特征和标签数据？

成功

考虑到依据数据来间接定义的问题（估计、分类等），什么是衡量成功的适当标准？

容量

模型容量扮演着什么角色？基于当前的问题，多大的容量才算足够？

评估

根据所训练模型的目标，如何评估模型的性能？

偏差和方差

哪种模型更适合解决当前的问题：是有相当大偏差的模型还是有相当大方差的模型？

交叉验证

对非序列数据集来说，当对训练数据子集和验证数据子集的不同配置进行交叉验证时，模型如何执行？

该蓝图在后续章节中会被应用到许多现实世界的金融用例中。有关机器学习过程的更多背景信息和细节，请参阅本书参考文献。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 6 章

人工智能优先的金融

> 计算是获取信息并对其进行转换的过程，也就是数学家定义的函数……如果你有一个函数，这个函数能够输入世界上所有的金融数据，并且能够输出最应该购买的股票，那么很快你就会变得非常富有。
>
> ——Max Tegmark，2017 年

本章将具体介绍数据驱动的金融如何与第 5 章介绍的机器学习方法相结合。本章的工作仅仅是一个开端，因为这是第一次通过神经网络去揭示统计失效。6.1 节通过 OLS 回归模型，基于金融时间序列对“有效市场假说”进行阐释。6.2 节首次使用神经网络模型和 OLS 回归通过历史收益数据来预测未来金融产品价格走势（“市场方向”）。6.3 节主要介绍“基于更多特征的市场预测”，其中融入了更多特征，比如传统的财务指标数据。通过对上述内容的讨论，最初的结果表明，统计失效的问题有可能真实存在。6.4 节讨论相较于使用日终数据，利用日内数据对日内市场方向进行预测，能够证明统计失效。6.5 节阐释大数据与人工智能相结合在特定领域的有效性，并且认为由 AI 引领，摒弃理论的金融也许可以摆脱传统金融理论中的谬误。

6.1 有效市场

有效市场假说（efficient market hypothesis，EMH）是有强大经验支持的假说之一，它也被称为“随机游走假说”（random walk hypothesis，RWH）[1]。简单来说，EMH 认为金融产品在某一时间点的价格反映了当前时刻的所有可获得的信息。如果 EMH 成立，那么讨论某只股票在某一时间点价格太高或太低是没有意义的。根据 EMH，股票的价格任何时刻都

注 1：就本章和本书的目的而言，这两个假说应被同等对待，尽管 RWH 比 EMH 的假设更强一些，详情可参见 Copeland 等（2005）的第 10 章。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

处于合理的水平，能够准确反应所有已知信息。

自 20 世纪 60 年代 EMH 开始形成并被提出以来，人们投入了大量的精力来完善和规范其概念。EMH 的定义是 Jensen 在 1978 年提出的，一直沿用至今。Jensen 的定义如下：

> 如果市场不可能通过信息集 θ_t 获得经济收益，那么市场对于信息集 θ_t 是有效的。这里的经济收益是指扣除所有成本后的净收益。

在此基础上，Jensen 区分了 3 种形式的市场有效性。

弱式 EMH
: 在这个场景中，信息集 θ_t 包含过去的价格和历史收益。

半强式 EMH
: 在这个场景中，信息集 θ_t 被视为所有公开可获得的信息，不仅包括过去的价格和历史收益，还包括财务报告、新闻资讯，天气数据，等等。

强式 EMH
: 在这个场景中，信息集 θ_t 包含所有人的所有可用信息（甚至包含隐私信息）。

无论采用哪种形式，EMH 的影响都是深远的。Fama（1965）在他关于 EMH 的文章中总结到：

> 许多年来，经济学家、统计学家和金融领域的教师一直对开发和测试股票价格行为模型感兴趣。对价格行为模型的研究衍生出了一个重要理论模型，即随机游走理论。这一理论对许多试图描述和预测股票价格行为的方法提出了严重质疑，而这类方法在学术界之外相当受欢迎。例如，我们稍后会看到，如果随机游走理论对现实有准确的描述，那么预测股票价格的各种“技术”或“图表”程序就完全没有价值。

换言之，如果 EMH 有效，那么在实践中，任何以获得超出市场收益为目的的研究或者数据分析都应该是无效的。从另一个角度来看，管理规模达到万亿美元的资管行业已经发展成为可以获得超出市场收益的行业，而其依靠的就是严谨的研究和主动的资产管理方式。特别地，对冲基金承诺提供的收益为 alpha 收益。alpha 收益是指投资收益率超出市场收益率的部分，其收益很大一部分甚至独立于市场收益，与市场无关。Preqin 咨询公司最近的研究报告显示，实现这样的承诺非常难。该报告显示 Preqin 全球对冲基金指数在 2018 年下跌了 3.42%。而在这一年所报告研究的全部对冲基金中有近 4 成的损失超过 5% 或者更多。

如果股票价格（或者其他任何金融资产的价格）符合随机游走理论，那么收益率应该满足零均值的正态分布。股票价格上涨和下跌的概率均应为 50%。在这种前提下，从最小均方误差的角度，对于明日股票价格的最好预测应该是当日的价格。这是因为随机游走具有的马尔可夫性，即未来股票的价格分布与历史价格走势无关，仅取决于当前价格水平。因此，在随机游走理论的前提下，对历史价格（或收益率）的分析对于预测未来价格是没有意义的。

在此背景下，可以对 EMH 进行如下半正式测试。[2] 取一段金融时间序列，多次滞后价格数据，并以滞后的价格数据作为特征，以当前价格作为标签，输入 OLS 回归模型。这本质上

注 2：另见 Hilpisch（2018）的第 15 章。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

类似于依赖过去价格预测未来价格的图表技术。

以下 Python 代码对许多金融资产（可交易和不可交易）的滞后价格数据进行了上述分析。

首先，引入数据并将数据结构可视化（参见图 6-1）。

```
In [1]: import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        pd.set_option('precision', 4)
        np.set_printoptions(suppress=True, precision=4)

In [2]: url = 'http://hilpisch.com/aiif_eikon_eod_data.csv'  ❶

In [3]: data = pd.read_csv(url, index_col=0, parse_dates=True).dropna()  ❶

In [4]: (data / data.iloc[0]).plot(figsize=(10, 6), cmap='coolwarm');  ❷
```

❶ 读取数据，存储为 `DataFrame` 类型。

❷ 绘制归一化时间序列数据。

图 6-1：归一化时间序列数据（收盘价）

然后，滞后所有金融时间序列价格数据，并以 `DataFrame` 格式存储。

```
In [5]: lags = 7  ❶

In [6]: def add_lags(data, ric, lags):
            cols = []
            df = pd.DataFrame(data[ric])
            for lag in range(1, lags + 1):
                col = 'lag_{}'.format(lag)  ❷
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            df[col] = df[ric].shift(lag)  ❸
            cols.append(col)  ❹
        df.dropna(inplace=True)  ❺
        return df, cols

In [7]: dfs = {}
        for sym in data.columns:
            df, cols = add_lags(data, sym, lags)  ❻
            dfs[sym] = df  ❼

In [8]: dfs[sym].head(7)  ❽
Out[8]:                GLD   lag_1   lag_2   lag_3   lag_4   lag_5   lag_6   lag_7

        Date
        2010-01-13  111.54  110.49  112.85  111.37  110.82  111.51  109.70  109.80
        2010-01-14  112.03  111.54  110.49  112.85  111.37  110.82  111.51  109.70
        2010-01-15  110.86  112.03  111.54  110.49  112.85  111.37  110.82  111.51
        2010-01-19  111.52  110.86  112.03  111.54  110.49  112.85  111.37  110.82
        2010-01-20  108.94  111.52  110.86  112.03  111.54  110.49  112.85  111.37
        2010-01-21  107.37  108.94  111.52  110.86  112.03  111.54  110.49  112.85
        2010-01-22  107.17  107.37  108.94  111.52  110.86  112.03  111.54  110.49
```

❶ 定义参数滞后时间（对于交易日而言）。

❷ 创建列名。

❸ 滞后价格序列。

❹ 以 list 对象存储列名。

❺ 删除所有不完整的数据行。

❻ 滞后每一个金融时间序列。

❼ 以 dict 对象存储结果。

❽ 展示一个滞后价格数据的样例。

最后，数据准备好后，进行 OLS 回归分析就变得很容易了。图 6-2 展示了平均最优的回归结果。毫无疑问，滞后 1 天的价格数据具有最高的可解释性权重。它的权重接近于 1，恰恰证实了对明日金融资产价格的最好预测就是当日价格的观点。这个结论对于单一金融时间序列的回归分析同样成立。

```
In [9]: regs = {}
        for sym in data.columns:
            df = dfs[sym]  ❶
            reg = np.linalg.lstsq(df[cols], df[sym], rcond=-1)[0]  ❷
            regs[sym] = reg  ❸

In [10]: rega = np.stack(tuple(regs.values()))  ❹

In [11]: regd = pd.DataFrame(rega, columns=cols, index=data.columns)  ❺

In [12]: regd  ❺
Out[12]:          lag_1   lag_2   lag_3   lag_4   lag_5   lag_6   lag_7
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
AAPL.O  1.0106 -0.0592  0.0258  0.0535 -0.0172  0.0060 -0.0184
MSFT.O  0.8928  0.0112  0.1175 -0.0832 -0.0258  0.0567  0.0323
INTC.O  0.9519  0.0579  0.0490 -0.0772 -0.0373  0.0449  0.0112
AMZN.O  0.9799 -0.0134  0.0206  0.0007  0.0525 -0.0452  0.0056
GS.N    0.9806  0.0342 -0.0172  0.0042 -0.0387  0.0585 -0.0215
SPY     0.9692  0.0067  0.0228 -0.0244 -0.0237  0.0379  0.0121
.SPX    0.9672  0.0106  0.0219 -0.0252 -0.0318  0.0515  0.0063
.VIX    0.8823  0.0591 -0.0289  0.0284 -0.0256  0.0511  0.0306
EUR=    0.9859  0.0239 -0.0484  0.0508 -0.0217  0.0149 -0.0055
XAU=    0.9864  0.0069  0.0166 -0.0215  0.0044  0.0198 -0.0125
GDX     0.9765  0.0096 -0.0039  0.0223 -0.0364  0.0379 -0.0065
GLD     0.9766  0.0246  0.0060 -0.0142 -0.0047  0.0223 -0.0106

In [13]: regd.mean().plot(kind='bar', figsize=(10, 6));  ❻
```

❶ 获取当前时间序列的数据。

❷ 进行回归分析。

❸ 以 `dict` 对象存储最优回归参数。

❹ 拼接多时间序列的最优结果，以单个 `ndarray` 对象存储。

❺ 将结果存入 `DataFrame` 对象中并进行展示。

❻ 可视化每一个滞后时间的多个最优参数（权重）结果的平均值。

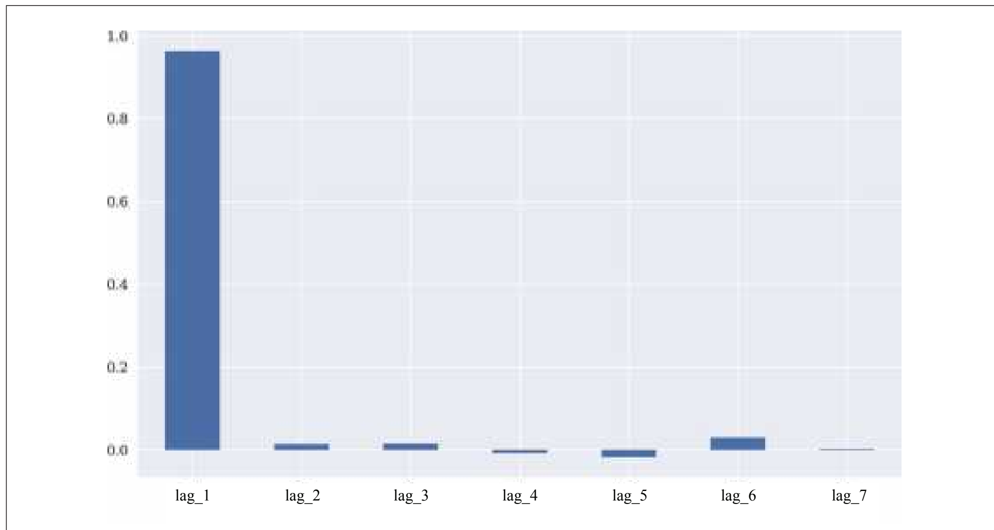


图 6-2：滞后价格的平均最优回归参数

上述半正式分析至少可以说是弱式 EMH 的强力证据。但值得注意的是，这里实现的 OLS 回归分析违反了几个假设，其中，OLS 假设特征之间相互独立，而此处特征与标签数据高度相关，并且，滞后的价格数据也使得特征之间高度相关。以下 Python 代码展示了数据之间高度的相关性。这也解释了为什么只用一个特征（“滞后 1 天”）就足以完成基于 OLS 回

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

归方法的近似和预测。增加更多高度相关的特性并不会对模型结果产生任何改进。另一个被违反的基本假设是时间序列数据的**平稳性**，下面的代码也对此进行了测试。[3]

```
In [14]: dfs[sym].corr()  ❶
Out[14]:           GLD   lag_1   lag_2   lag_3   lag_4   lag_5   lag_6   lag_7
         GLD    1.0000  0.9972  0.9946  0.9920  0.9893  0.9867  0.9841  0.9815
         lag_1  0.9972  1.0000  0.9972  0.9946  0.9920  0.9893  0.9867  0.9842
         lag_2  0.9946  0.9972  1.0000  0.9972  0.9946  0.9920  0.9893  0.9867
         lag_3  0.9920  0.9946  0.9972  1.0000  0.9972  0.9946  0.9920  0.9893
         lag_4  0.9893  0.9920  0.9946  0.9972  1.0000  0.9972  0.9946  0.9920
         lag_5  0.9867  0.9893  0.9920  0.9946  0.9972  1.0000  0.9972  0.9946
         lag_6  0.9841  0.9867  0.9893  0.9920  0.9946  0.9972  1.0000  0.9972
         lag_7  0.9815  0.9842  0.9867  0.9893  0.9920  0.9946  0.9972  1.0000

In [15]: from statsmodels.tsa.stattools import adfuller  ❷

In [16]: adfuller(data[sym].dropna())  ❷
Out[16]: (-1.9488969577009954,
          0.3094193074034718,
          0,
          2515,
          {'1%': -3.4329527780962255,
           '5%': -2.8626898965523724,
           '10%': -2.567382133955709},
          8446.683102944744)
```

❶ 展示滞后时间序列的相关性。

❷ 使用迪基 – 福勒检验（Dickey-Fuller test）进行平稳性测试。

综上所述，如果 EMH 成立，那么主动投资组合应该没有任何经济学意义。简单地投资基于最小方差获得的投资组合或者股票，并且被动地长期持有该投资，无须付出任何努力，至少可以获得相同甚至更高的回报。根据 CAPM 和 MVP，投资者愿意承担的风险越高，预期收益就应该越高。事实上，正如 Copeland 等（2005）的第 10 章指出，CAPM 和 EMH 形成了一个关于金融市场的联合假设：如果 EMH 不成立，那么 CAPM 也必然不成立，因为后者的理论推导是建立在前者成立的基础上的。

6.2 基于收益数据的市场预测

正如第 2 章所述，近些年来，机器学习，特别是深度学习算法已被证明能够在统计方法或数学计算长期陷入瓶颈的领域取得突破。那么在金融市场领域呢？ 机器学习和深度学习算法是否能够解决传统金融计量经济学方法（如 OLS 回归）不能解决的问题？ 当然，这些问题目前还没有简单且一目了然的答案。

但是，一些具体的例子也许会揭示可能的答案。为此，使用与上一节相同的数据并获取其对数收益率。此做法的目的是对比 OLS 回归分析和神经网络在预测不同时间序列下一个交

注 3：有关时间序列数据的**平稳性**的详细信息，请参阅 Tsay（2005）的 2.1 节。Tsay 指出：“平稳性是时间序列数据分析的基础。”

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

易日多空方向的表现。相对于**经济失效**，本阶段的目标主要是发现**统计失效**。统计失效是指一个模型能够以一定的优势预测未来价格的走势（例如，在 55% 或 60% 的情况下预测结果是正确的）。而经济失效是指在考虑交易成本的前提下，可以通过制定交易策略利用统计失效来获利。

分析的第一步是创建具有滞后对数收益率数据的数据集。归一化滞后对数收益率数据也进行平稳性测试，并对特征进行相关性测试。由于以下分析仅依赖于时间序列相关的数据，因此它们处理的是**弱式有效市场**。

```
In [17]: rets = np.log(data / data.shift(1))  ❶

In [18]: rets.dropna(inplace=True)

In [19]: dfs = {}
         for sym in data:
             df, cols = add_lags(rets, sym, lags)  ❷
             mu, std = df[cols].mean(), df[cols].std()  ❸
             df[cols] = (df[cols] - mu) / std  ❸
             dfs[sym] = df

In [20]: dfs[sym].head()  ❹
Out[20]:                GLD   lag_1   lag_2   lag_3   lag_4   lag_5   lag_6   lag_7
         Date
         2010-01-14  0.0044  0.9570 -2.1692  1.3386  0.4959 -0.6434  1.6613 -0.1028
         2010-01-15 -0.0105  0.4379  0.9571 -2.1689  1.3388  0.4966 -0.6436  1.6614
         2010-01-19  0.0059 -1.0842  0.4385  0.9562 -2.1690  1.3395  0.4958 -0.6435
         2010-01-20 -0.0234  0.5967 -1.0823  0.4378  0.9564 -2.1686  1.3383  0.4958
         2010-01-21 -0.0145 -2.4045  0.5971 -1.0825  0.4379  0.9571 -2.1680  1.3384

In [21]: adfuller(dfs[sym]['lag_1'])  ❺
Out[21]: (-51.568251505825536,
          0.0,
          0,
          2507,
          {'1%': -3.4329610922579095,
           '5%': -2.8626935681060375,
           '10%': -2.567384088736619},
          7017.165474260225)

In [22]: dfs[sym].corr()  ❻
Out[22]:          GLD   lag_1   lag_2       lag_3   lag_4       lag_5   lag_6   lag_7
         GLD    1.0000 -0.0297  0.0003  1.2635e-02 -0.0026 -5.9392e-03  0.0099 -0.0013
         lag_1 -0.0297  1.0000 -0.0305  8.1418e-04  0.0128 -2.8765e-03 -0.0053  0.0098
         lag_2  0.0003 -0.0305  1.0000 -3.1617e-02  0.0003  1.3234e-02 -0.0043 -0.0052
         lag_3  0.0126  0.0008 -0.0316  1.0000e+00 -0.0313 -6.8542e-06  0.0141 -0.0044
         lag_4 -0.0026  0.0128  0.0003 -3.1329e-02  1.0000 -3.1761e-02  0.0002  0.0141
         lag_5 -0.0059 -0.0029  0.0132 -6.8542e-06 -0.0318  1.0000e+00 -0.0323  0.0002
         lag_6  0.0099 -0.0053 -0.0043  1.4115e-02  0.0002 -3.2289e-02  1.0000 -0.0324
         lag_7 -0.0013  0.0098 -0.0052 -4.3869e-03  0.0141  2.1707e-04 -0.0324  1.0000
```

❶ 计算价格数据的对数收益率。

❷ 滞后对数收益率数据。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❸ 对特征数据进行**高斯归一化**（Gaussian normalization）。[4]

❹ 展示一个滞后对数收益率实例。

❺ 测试时间序列数据的稳定性。

❻ 展示数据特征的相关性。

首先，实现 OLS 回归并获得回归的预测结果。此分析是在完整的数据集上进行的，它将显示算法在样本内的表现。除了一个特例，通过 OLS 回归的方法预测明日市场走向的准确率均能略高于 50%。

```
In [23]: from sklearn.metrics import accuracy_score

In [24]: %%time
         for sym in data:
             df = dfs[sym]
             reg = np.linalg.lstsq(df[cols], df[sym], rcond=-1)[0]  ❶
             pred = np.dot(df[cols], reg)  ❷
             acc = accuracy_score(np.sign(df[sym]), np.sign(pred))  ❸
             print(f'OLS | {sym:10s} | acc={acc:.4f}')
         OLS | AAPL.O     | acc=0.5056
         OLS | MSFT.O     | acc=0.5088
         OLS | INTC.O     | acc=0.5040
         OLS | AMZN.O     | acc=0.5048
         OLS | GS.N       | acc=0.5080
         OLS | SPY        | acc=0.5080
         OLS | .SPX       | acc=0.5167
         OLS | .VIX       | acc=0.5291
         OLS | EUR=       | acc=0.4984
         OLS | XAU=       | acc=0.5207
         OLS | GDX        | acc=0.5307
         OLS | GLD        | acc=0.5072
         CPU times: user 201 ms, sys: 65.8 ms, total: 267 ms
         Wall time: 60.8 ms
```

❶ 模型回归。

❷ 模型预测。

❸ 准确率分析。

然后，使用基于 `scikit-learn` 包构建的神经网络模型进行学习和预测，并对模型结果进行同样的分析。在测试样本内，预测结果准确率明显超过了 50%，并且有一些结果超过了 60%。

```
In [25]: from sklearn.neural_network import MLPRegressor

In [26]: %%time
         for sym in data.columns:
             df = dfs[sym]
             model = MLPRegressor(hidden_layer_sizes=[512],
                                  random_state=100,
```

注 4：该方法也被称为 z-score 归一化。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
                              max_iter=1000,
                              early_stopping=True,
                              validation_fraction=0.15,
                              shuffle=False)  ❶
            model.fit(df[cols], df[sym])  ❷
            pred = model.predict(df[cols])  ❸
            acc = accuracy_score(np.sign(df[sym]), np.sign(pred))  ❹
            print(f'MLP | {sym:10s} | acc={acc:.4f}')
        MLP | AAPL.O     | acc=0.6005
        MLP | MSFT.O     | acc=0.5853
        MLP | INTC.O     | acc=0.5766
        MLP | AMZN.O     | acc=0.5510
        MLP | GS.N       | acc=0.6527
        MLP | SPY        | acc=0.5419
        MLP | .SPX       | acc=0.5399
        MLP | .VIX       | acc=0.6579
        MLP | EUR=       | acc=0.5642
        MLP | XAU=       | acc=0.5522
        MLP | GDX        | acc=0.6029
        MLP | GLD        | acc=0.5259
        CPU times: user 1min 37s, sys: 6.74 s, total: 1min 44s
        Wall time: 14 s
```

❶ 模型实例化。

❷ 模型拟合。

❸ 模型预测。

❹ 准确率计算。

最后，使用基于 keras 包构建的神经网络再次进行同样的分析。此方法的准确率与 `MLPRegressor` 近似，但是平均准确率更高。

```
In [27]: import tensorflow as tf
         from keras.layers import Dense
         from keras.models import Sequential
         Using TensorFlow backend.

In [28]: np.random.seed(100)
         tf.random.set_seed(100)

In [29]: def create_model(problem='regression'):  ❶
             model = Sequential()
             model.add(Dense(512, input_dim=len(cols),
                             activation='relu'))
             if problem == 'regression':
                 model.add(Dense(1, activation='linear'))
                 model.compile(loss='mse', optimizer='adam')
             else:
                 model.add(Dense(1, activation='sigmoid'))
                 model.compile(loss='binary_crossentropy', optimizer='adam')
             return model

In [30]: %%time
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        for sym in data.columns[:]:
            df = dfs[sym]
            model = create_model()  ❷
            model.fit(df[cols], df[sym], epochs=25, verbose=False)  ❸
            pred = model.predict(df[cols])  ❹
            acc = accuracy_score(np.sign(df[sym]), np.sign(pred))  ❺
            print(f'DNN | {sym:10s} | acc={acc:.4f}')
        DNN | AAPL.O     | acc=0.6292
        DNN | MSFT.O     | acc=0.5981
        DNN | INTC.O     | acc=0.6073
        DNN | AMZN.O     | acc=0.5781
        DNN | GS.N       | acc=0.6196
        DNN | SPY        | acc=0.5829
        DNN | .SPX       | acc=0.6077
        DNN | .VIX       | acc=0.6392
        DNN | EUR=       | acc=0.5845
        DNN | XAU=       | acc=0.5881
        DNN | GDX        | acc=0.5829
        DNN | GLD        | acc=0.5666
        CPU times: user 34.3 s, sys: 5.34 s, total: 39.6 s
        Wall time: 23.1 s
```

❶ 模型创建函数。

❷ 模型实例化。

❸ 模型拟合。

❹ 模型预测。

❺ 准确率分析。

这个简单的例子表明，在**样本内**，神经网络在预测明日价格走势上可以显著优于 OSL 回归。那么测试两类模型在数据**样本外**的表现，又会有怎样的变化呢？

为此，继续重复上述分析过程，只不过使用全部数据的 80% 进行模型训练，剩下的 20% 进行效果测试。首先，分析 OSL 回归模型。OSL 回归模型在样本外的结果与样本内准确率近似，都在 50% 左右。

```
In [31]: split = int(len(dfs[sym]) * 0.8)

In [32]: %%time
         for sym in data.columns:
             df = dfs[sym]
             train = df.iloc[:split]  ❶
             reg = np.linalg.lstsq(train[cols], train[sym], rcond=-1)[0]
             test = df.iloc[split:]  ❷
             pred = np.dot(test[cols], reg)
             acc = accuracy_score(np.sign(test[sym]), np.sign(pred))
             print(f'OLS | {sym:10s} | acc={acc:.4f}')
         OLS | AAPL.O     | acc=0.5219
         OLS | MSFT.O     | acc=0.4960
         OLS | INTC.O     | acc=0.5418
         OLS | AMZN.O     | acc=0.4841
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
OLS | GS.N       | acc=0.4980
OLS | SPY        | acc=0.5020
OLS | .SPX       | acc=0.5120
OLS | .VIX       | acc=0.5458
OLS | EUR=       | acc=0.4482
OLS | XAU=       | acc=0.5299
OLS | GDX        | acc=0.5159
OLS | GLD        | acc=0.5100
CPU times: user 200 ms, sys: 60.6 ms, total: 261 ms
Wall time: 61.7 ms
```

❶ 创建训练数据子集。

❷ 创建测试数据子集。

`MLPRegressor` 在样本外的表现相比于样本内要差得多，其结果接近于 OSL 回归。

```
In [34]: %%time
         for sym in data.columns:
             df = dfs[sym]
             train = df.iloc[:split]
             model = MLPRegressor(hidden_layer_sizes=[512],
                                  random_state=100,
                                  max_iter=1000,
                                  early_stopping=True,
                                  validation_fraction=0.15,
                                  shuffle=False)
             model.fit(train[cols], train[sym])
             test = df.iloc[split:]
             pred = model.predict(test[cols])
             acc = accuracy_score(np.sign(test[sym]), np.sign(pred))
             print(f'MLP | {sym:10s} | acc={acc:.4f}')
         MLP | AAPL.O     | acc=0.4920
         MLP | MSFT.O     | acc=0.5279
         MLP | INTC.O     | acc=0.5279
         MLP | AMZN.O     | acc=0.4641
         MLP | GS.N       | acc=0.5040
         MLP | SPY        | acc=0.5259
         MLP | .SPX       | acc=0.5478
         MLP | .VIX       | acc=0.5279
         MLP | EUR=       | acc=0.4980
         MLP | XAU=       | acc=0.5239
         MLP | GDX        | acc=0.4880
         MLP | GLD        | acc=0.5000
         CPU times: user 1min 39s, sys: 4.98 s, total: 1min 44s
         Wall time: 13.7 s
```

对于基于 `Keras` 构建的 `Sequential` 模型同样成立，在样本外数据准确率都在 50% 上下波动。

```
In [35]: %%time
         for sym in data.columns:
             df = dfs[sym]
             train = df.iloc[:split]
             model = create_model()
             model.fit(train[cols], train[sym], epochs=50, verbose=False)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
    test = df.iloc[split:]
    pred = model.predict(test[cols])
    acc = accuracy_score(np.sign(test[sym]), np.sign(pred))
    print(f'DNN | {sym:10s} | acc={acc:.4f}')
DNN | AAPL.O     | acc=0.5179
DNN | MSFT.O     | acc=0.5598
DNN | INTC.O     | acc=0.4821
DNN | AMZN.O     | acc=0.4920
DNN | GS.N       | acc=0.5179
DNN | SPY        | acc=0.4861
DNN | .SPX       | acc=0.5100
DNN | .VIX       | acc=0.5378
DNN | EUR=       | acc=0.4661
DNN | XAU=       | acc=0.4602
DNN | GDX        | acc=0.4841
DNN | GLD        | acc=0.5378
CPU times: user 50.4 s, sys: 7.52 s, total: 57.9 s
Wall time: 32.9 s
```

弱式有效市场

尽管根据**弱式**有效市场的标签数据并没有证实统计失效，但仅依赖于时间序列数据去证明统计失效本身就是十分困难的。在半强式有效市场的形式下，可以加入其他公开市场数据以提升预测准确率。

实验结果似乎证明市场应该是弱有效的，但仅仅通过 OLS 回归或神经网络来分析历史收益的模式不足以证明统计失效。

为进一步提升预测效果，在本节的方法中，可以对以下两个主要元素进行调整。

特征

除了原始的价格和收益率数据，也可以将其他特征加入数据中，比如简单移动平均线（simple moving average，SMA）等技术指标。这些技术指标有可能提升预测准确率。

交易时段

不使用日终数据，而改用日内数据，也许会获得更高的预测准确率。所有市场参与者会高度关注当日的最后交易，使用各种公开数据信息帮助决策，这就导致日终数据往往比日内其他时刻的数据更难证明统计失效。

接下来的两节将具体实现上述讨论。

6.3 基于更多特征的市场预测

传统的交易方式是根据观测技术指标生成买入卖出信号。任意一个这样的技术指标都可以用作训练神经网络的特征。

以下 Python 代码使用 SMA、滚动最小值与滚动最大值、动量以及滚动波动率作为输入特征。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [36]: url = 'http://hilpisch.com/aiif_eikon_eod_data.csv'

In [37]: data = pd.read_csv(url, index_col=0, parse_dates=True).dropna()

In [38]: def add_lags(data, ric, lags, window=50):
             cols = []
             df = pd.DataFrame(data[ric])
             df.dropna(inplace=True)
             df['r'] = np.log(df / df.shift())
             df['sma'] = df[ric].rolling(window).mean()  ❶
             df['min'] = df[ric].rolling(window).min()  ❷
             df['max'] = df[ric].rolling(window).max()  ❸
             df['mom'] = df['r'].rolling(window).mean()  ❹
             df['vol'] = df['r'].rolling(window).std()  ❺
             df.dropna(inplace=True)
             df['d'] = np.where(df['r'] > 0, 1, 0)  ❻
             features = [ric, 'r', 'd', 'sma', 'min', 'max', 'mom', 'vol']
             for f in features:
                 for lag in range(1, lags + 1):
                     col = f'{f}_lag_{lag}'
                     df[col] = df[f].shift(lag)
                     cols.append(col)
             df.dropna(inplace=True)
             return df, cols

In [39]: lags = 5

In [40]: dfs = {}
         for ric in data:
             df, cols = add_lags(data, ric, lags)
             dfs[ric] = df.dropna(), cols
```

❶ SMA。

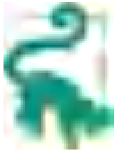
❷ 滚动最小值。

❸ 滚动最大值。

❹ 动量 – 对数收益率滑动均值。

❺ 滚动波动率。

❻ 当日收益率方向（二进制 0, 1 特征）。



作为特征的技术指标

正如前面例子所示，基本上任意一个用于投资和日内交易的技术指标都可以作为神经网络模型的输入特征。从这个意义上说，这些经典的特征大大丰富了人工智能和机器学习的交易策略，也因此不会过时。

通过增加新特征，并将其归一化训练，`MLPClassifier` 模型在样本内效果大大提升。基于 `Keras` 构建的 `Sequential` 模型在当前训练轮数下能够达到 70% 的准确率。根据经验，如果进一步增加模型训练轮数或者神经网络参数量，则可以轻松提升模型效果。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [41]: from sklearn.neural_network import MLPClassifier

In [42]: %%time
         for ric in data:
             model = MLPClassifier(hidden_layer_sizes=[512],
                                   random_state=100,
                                   max_iter=1000,
                                   early_stopping=True,
                                   validation_fraction=0.15,
                                   shuffle=False)
             df, cols = dfs[ric]
             df[cols] = (df[cols] - df[cols].mean()) / df[cols].std()  ❶
             model.fit(df[cols], df['d'])
             pred = model.predict(df[cols])
             acc = accuracy_score(df['d'], pred)
             print(f'IN-SAMPLE | {ric:7s} | acc={acc:.4f}')
         IN-SAMPLE | AAPL.O  | acc=0.5510
         IN-SAMPLE | MSFT.O  | acc=0.5376
         IN-SAMPLE | INTC.O  | acc=0.5607
         IN-SAMPLE | AMZN.O  | acc=0.5559
         IN-SAMPLE | GS.N    | acc=0.5794
         IN-SAMPLE | SPY     | acc=0.5729
         IN-SAMPLE | .SPX    | acc=0.5941
         IN-SAMPLE | .VIX    | acc=0.6940
         IN-SAMPLE | EUR=    | acc=0.5766
         IN-SAMPLE | XAU=    | acc=0.5672
         IN-SAMPLE | GDX     | acc=0.5847
         IN-SAMPLE | GLD     | acc=0.5567
         CPU times: user 1min 1s, sys: 4.5 s, total: 1min 6s
         Wall time: 9.05 s

In [43]: %%time
         for ric in data:
             model = create_model('classification')
             df, cols = dfs[ric]
             df[cols] = (df[cols] - df[cols].mean()) / df[cols].std()  ❶
             model.fit(df[cols], df['d'], epochs=50, verbose=False)
             pred = np.where(model.predict(df[cols]) > 0.5, 1, 0)
             acc = accuracy_score(df['d'], pred)
             print(f'IN-SAMPLE | {ric:7s} | acc={acc:.4f}')
         IN-SAMPLE | AAPL.O  | acc=0.7156
         IN-SAMPLE | MSFT.O  | acc=0.7156
         IN-SAMPLE | INTC.O  | acc=0.7046
         IN-SAMPLE | AMZN.O  | acc=0.6640
         IN-SAMPLE | GS.N    | acc=0.6855
         IN-SAMPLE | SPY     | acc=0.6696
         IN-SAMPLE | .SPX    | acc=0.6579
         IN-SAMPLE | .VIX    | acc=0.7489
         IN-SAMPLE | EUR=    | acc=0.6737
         IN-SAMPLE | XAU=    | acc=0.7143
         IN-SAMPLE | GDX     | acc=0.6826
         IN-SAMPLE | GLD     | acc=0.7078
         CPU times: user 1min 5s, sys: 7.06 s, total: 1min 12s
         Wall time: 44.3 s
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❶ 归一化特征数据。

这些改进是否也会提升样本外的预测准确率？跟上一节一样，以下 Python 代码会对数据进行训练集和测试集拆分，分析模型在样本外测试集的表现。不幸的是，实验结果喜忧参半。与只使用滞后的收益率数据作为特征训练的模型相比，引入更多的技术指标特征并没有带来真实预测效果的提升。如 MLPClassifier 模型结果所示，对参与实验的金融标的，有些品种标的预测准确率似乎高于 50% 的基准，而对于其他的一些品种，准确率则仍然低于 50%。

```
In [44]: def train_test_model(model):
             for ric in data:
                 df, cols = dfs[ric]
                 split = int(len(df) * 0.85)
                 train = df.iloc[:split].copy()
                 mu, std = train[cols].mean(), train[cols].std()  ❶
                 train[cols] = (train[cols] - mu) / std
                 model.fit(train[cols], train['d'])
                 test = df.iloc[split:].copy()
                 test[cols] = (test[cols] - mu) / std
                 pred = model.predict(test[cols])
                 acc = accuracy_score(test['d'], pred)
                 print(f'OUT-OF-SAMPLE | {ric:7s} | acc={acc:.4f}')

In [45]: model_mlp = MLPClassifier(hidden_layer_sizes=[512],
                                   random_state=100,
                                   max_iter=1000,
                                   early_stopping=True,
                                   validation_fraction=0.15,
                                   shuffle=False)

In [46]: %time train_test_model(model_mlp)
         OUT-OF-SAMPLE | AAPL.O  | acc=0.4432
         OUT-OF-SAMPLE | MSFT.O  | acc=0.4595
         OUT-OF-SAMPLE | INTC.O  | acc=0.5000
         OUT-OF-SAMPLE | AMZN.O  | acc=0.5270
         OUT-OF-SAMPLE | GS.N    | acc=0.4838
         OUT-OF-SAMPLE | SPY     | acc=0.4811
         OUT-OF-SAMPLE | .SPX    | acc=0.5027
         OUT-OF-SAMPLE | .VIX    | acc=0.5676
         OUT-OF-SAMPLE | EUR=    | acc=0.4649
         OUT-OF-SAMPLE | XAU=    | acc=0.5514
         OUT-OF-SAMPLE | GDX     | acc=0.5162
         OUT-OF-SAMPLE | GLD     | acc=0.4946
         CPU times: user 44.9 s, sys: 2.64 s, total: 47.5 s
         Wall time: 6.37 s
```

❶ 用于归一化的训练数据集统计信息。

在样本内表现好而在样本外表现差的现象意味着神经网络模型也许存在过拟合的情况。一种避免过拟合的方法是模型集成，即将训练好的多个同类型模型集合在一起，以提升元模型的健壮性并获得更好的预测结果。有一种这样的方法叫作**装袋**（bagging），可以通过 scikit-learn 包的 BaggingClassifier 类来实现。使用多个评估模型，并且每个模型只使

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

用部分训练数据或者部分特征训练。这样应该有助于避免过拟合。

以下 Python 代码基于多个相同的 `MLPClassifier` 模型实现了装袋算法逻辑。预测结果准确率现在可以稳定在 50% 以上。一些品种的预测准确率已经高于 55%，在这种情况下可以被认为是相当高的。总的来说，装袋的方法似乎从某种程度上避免了模型过拟合，并且显著提升了预测的准确率。

```
In [47]: from sklearn.ensemble import BaggingClassifier

In [48]: base_estimator = MLPClassifier(hidden_layer_sizes=[256],
                                  random_state=100,
                                  max_iter=1000,
                                  early_stopping=True,
                                  validation_fraction=0.15,
                                  shuffle=False)  ❶

In [49]: model_bag = BaggingClassifier(base_estimator=base_estimator,  ❶
                                  n_estimators=35,  ❷
                                  max_samples=0.25,  ❸
                                  max_features=0.5,  ❹
                                  bootstrap=False,  ❺
                                  bootstrap_features=True,  ❻
                                  n_jobs=8,  ❼
                                  random_state=100
                                 )

In [50]: %time train_test_model(model_bag)
         OUT-OF-SAMPLE | AAPL.O  | acc=0.5243
         OUT-OF-SAMPLE | MSFT.O  | acc=0.5703
         OUT-OF-SAMPLE | INTC.O  | acc=0.5027
         OUT-OF-SAMPLE | AMZN.O  | acc=0.5270
         OUT-OF-SAMPLE | GS.N    | acc=0.5243
         OUT-OF-SAMPLE | SPY     | acc=0.5595
         OUT-OF-SAMPLE | .SPX    | acc=0.5514
         OUT-OF-SAMPLE | .VIX    | acc=0.5649
         OUT-OF-SAMPLE | EUR=    | acc=0.5108
         OUT-OF-SAMPLE | XAU=    | acc=0.5378
         OUT-OF-SAMPLE | GDX     | acc=0.5162
         OUT-OF-SAMPLE | GLD     | acc=0.5432
         CPU times: user 2.55 s, sys: 494 ms, total: 3.05 s
         Wall time: 11.1 s
```

❶ 基础模型。

❷ 使用模型量。

❸ 每个模型使用的最大训练数据量。

❹ 每个模型使用的最大特征数。

❺ 是否可以重用数据。

❻ 是否可以重用特征。

❼ 并行线程数。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

日终交易的有效市场

EMH 可以追溯到 20 世纪 60 年代到 70 年代，在此期间，日终数据基本上是唯一可以获取的时间序列数据。回到那段时间（直到现在），能够想象每一位市场玩家都会格外关注他们在临近盘终的头寸与交易。对股票来说更是如此，而对原则上可以全天候交易的货币来说，情况相对会好一些。

6.4 日内市场预测

到目前为止，本章还没有产生确定性的结论，但是从当前的实验分析结果来看，对于日终交易价格的预测倾向于市场是弱有效的。那么对于日内交易情况如何呢？是否存在更多一致的统计失效的情况？为了回答以上问题，需要再准备一份日内时间序列数据集。下述 Python 代码使用了与之前日终数据一样的商品标的，但不同的是，其包含这些标的的小时级别收盘价。由于对于不同标的的交易时间段是不同的，因此这个数据集并不完整。但这并不是问题，因为分析是基于时间序列片段的。

小时级别的数据分析与之前的日终分析本质上是一样的，会使用与日终分析同样的代码。

```
In [51]: url = 'http://hilpisch.com/aiif_eikon_id_data.csv'

In [52]: data = pd.read_csv(url, index_col=0, parse_dates=True)

In [53]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 5529 entries, 2019-03-01 00:00:00 to 2020-01-01 00:00:00
         Data columns (total 12 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   AAPL.O  3384 non-null   float64
          1   MSFT.O  3378 non-null   float64
          2   INTC.O  3275 non-null   float64
          3   AMZN.O  3381 non-null   float64
          4   GS.N    1686 non-null   float64
          5   SPY     3388 non-null   float64
          6   .SPX    1802 non-null   float64
          7   .VIX    2959 non-null   float64
          8   EUR=    5429 non-null   float64
          9   XAU=    5149 non-null   float64
         10   GDX     3173 non-null   float64
         11   GLD     3351 non-null   float64
         dtypes: float64(12)
         memory usage: 561.5 KB

In [54]: lags = 5

In [55]: dfs = {}
         for ric in data:
             df, cols = add_lags(data, ric, lags)
             dfs[ric] = df, cols
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

对单一神经网络日内交易的预测结果准确率依然在 50% 左右宽幅波动。好的方面是，一些标的的准确率可以在 55% 以上。元模型通过装袋的方法表现出了在样本外更加一致的预测效果，尽管很多观测到的准确率只略高于基准值 50%。

```
In [56]: %time train_test_model(model_mlp)
         OUT-OF-SAMPLE | AAPL.O  | acc=0.5420
         OUT-OF-SAMPLE | MSFT.O  | acc=0.4930
         OUT-OF-SAMPLE | INTC.O  | acc=0.5549
         OUT-OF-SAMPLE | AMZN.O  | acc=0.4709
         OUT-OF-SAMPLE | GS.N    | acc=0.5184
         OUT-OF-SAMPLE | SPY     | acc=0.4860
         OUT-OF-SAMPLE | .SPX    | acc=0.5019
         OUT-OF-SAMPLE | .VIX    | acc=0.4885
         OUT-OF-SAMPLE | EUR=    | acc=0.5130
         OUT-OF-SAMPLE | XAU=    | acc=0.4824
         OUT-OF-SAMPLE | GDX     | acc=0.4765
         OUT-OF-SAMPLE | GLD     | acc=0.5455
         CPU times: user 1min 4s, sys: 5.05 s, total: 1min 9s
         Wall time: 9.56 s

In [57]: %time train_test_model(model_bag)
         OUT-OF-SAMPLE | AAPL.O  | acc=0.5660
         OUT-OF-SAMPLE | MSFT.O  | acc=0.5431
         OUT-OF-SAMPLE | INTC.O  | acc=0.5072
         OUT-OF-SAMPLE | AMZN.O  | acc=0.5110
         OUT-OF-SAMPLE | GS.N    | acc=0.5020
         OUT-OF-SAMPLE | SPY     | acc=0.5120
         OUT-OF-SAMPLE | .SPX    | acc=0.4677
         OUT-OF-SAMPLE | .VIX    | acc=0.5092
         OUT-OF-SAMPLE | EUR=    | acc=0.5242
         OUT-OF-SAMPLE | XAU=    | acc=0.5255
         OUT-OF-SAMPLE | GDX     | acc=0.5085
         OUT-OF-SAMPLE | GLD     | acc=0.5374
         CPU times: user 2.64 s, sys: 439 ms, total: 3.08 s
         Wall time: 12.4 s
```

日内交易的有效市场

尽管对日终交易来说市场是弱有效的，但是对日内交易而言，市场绝不是弱有效的。上述统计失效的结果可能是由于短暂性的不平衡、买入或卖出压力、市场过度反应、技术性的买卖指令触发等原因造成的。关键的问题是，统计失效一旦出现，能否通过特定的交易策略获得收益。

6.5 结论

Halevy 等（2009）在其撰写且被广泛引用的“The Unreasonable Effectiveness of Data”一文中指出，经济学家往往存在所谓的“物理妒忌”。也就是说，经济学家十分羡慕物理学家可以通过优雅的数学推导来解释复杂的现实世界现象，而他们却没有办法用同样的方式来解释人类行为。最为人熟知的类似例子是爱因斯坦的 $E = mc^2$，它将复杂的能量概念等同于质量与速度平方的乘积。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

数十年来，在经济学和金融学领域，研究人员模仿物理学的分析方法，通过简单且优雅的数学公式来解释经济学和金融学现象并对其进行证明。但是结合本书第 3 章和第 4 章的内容来看，大多数优雅的金融理论在现实的金融场景中很难找到强有力的证据支持，因为这些理论简化了假设条件，而现实场景并不支持诸如数据正态分布或者线性关系的假设。

正如 Halevy 等（2009）在其文章中指出的那样，也许在自然语言处理等领域，这些规则是无法通过简单且优雅的理论推导形成的。研究人员也许只能依赖由数据驱动的复杂理论和模型。尤其对语言模型来说，万维网代表着大数据语料宝库。而像自然语言处理，或者能够与人工翻译媲美的机器翻译等类似的特定任务，恰恰需要用大数据来训练机器学习算法和深度学习算法。

相较于物理学而言，金融学可能是一门与自然语言更相似的学科。毕竟，也许还没有可以描述货币汇率价格波动或股票价格变化等重要金融现象的简单且优雅的公式。[5] 也许真正的规则只能通过大数据找到，而如今这些大数据正以程序化的方式提供给金融研究人员和学者。

本章介绍了寻求市场真相、发现金融学圣杯的开端：证明市场并非那么有效。本章使用的神经网络方法相对简单，仅通过时间序列相关特征训练模型。它的训练标签也十分简单直接：就是市场（或金融产品）价格上涨或下跌。而最终目标是为了发现在预测未来市场方向上存在**统计失效**。发现统计失效也恰恰是利用可实施的交易策略获利的第一步。

Agrawal 等（2018）使用大量例子详细解释了预测自身走势只是硬币的一面。如何对每一个具体预测结果进行决策并实施规则同样重要。在算法交易流程中也是如此：信号的预测仅仅是开始。最难的部分是如何最优地执行恰当的交易逻辑，监控活跃交易，实施适当的风控措施，比如设置止损单和止盈单等。

在寻求统计失效的过程中，本章仅依赖数据和神经网络，没有涉及任何其他背景理论，也没有对市场参与者的行为或其他关联概念进行假设。建模的主要工作是准备特征，这无疑是开发人员最重视的环节。在所采用的方法中存在一个隐含假设，即认为仅仅依赖时间序列数据就可以发现统计失效。这就意味着需要证明市场甚至都不满足弱有效性，即便弱有效性是市场的 3 种形式中最难证明的。

仅仅通过金融数据并使用传统的机器学习算法和深度学习算法对其建模就是本书定义的人工智能优先的金融。不需要理论，不需要人为建模，也不需要关于数据分布或相互关系的假设，仅仅依赖数据和算法。从这个意义上说，人工智能优先的金融不依赖金融理论或金融模型。

注 5：当然，更多的金融概念可以通过简单的公式来建模，比如说期限为两年（$T = 2$）、相对对数收益率为 0.01（$r = 0.01$）的连续贴现因子 D 的推导可以直接表达为 $D(r, T) = \exp(-rT) = \exp(-0.01 \times 2) = 0.9802$。在这个问题里并不能通过人工智能和机器学习获得任何利益。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第三部分

统计失效

“市场是有模式的，”Simons 告诉一位同事，“我知道我们可以找到这些模式。”[1]

——Gregory Zuckerman，2019 年

第三部分的主要目标是应用神经网络和强化学习来发现金融市场（数据）中的统计失效。就本书而言，当**预测器**（泛指模型或算法，或专指神经网络）比随机预测器（为向上和向下运动分配相等概率）对市场的预测明显更好时，就会发现**统计失效**。在算法交易环境中，拥有这样一个可用的预测器是产生 alpha 收益或高于市场收益率的先决条件。

第三部分由 3 章组成，介绍了与密集神经网络（DNN）、循环神经网络（RNN）和强化学习（RL）相关的更多背景、细节和示例。

- 第 7 章更详细地介绍了 DNN，并将它们应用于预测金融市场走势的问题。用历史数据生成滞后特征数据和二进制标签数据，然后使用这些数据集通过监督学习来训练 DNN，重点在于识别金融市场中的统计失效。在一些示例中，DNN 实现了超过 60% 的样本外预测准确率。
- 第 8 章是关于 RNN 的，其旨在适应像文本数据或时间序列数据这样的序列数据的特定性质。这样做的目的是向网络添加某种形式的记忆，通过网络（层）携带以前（历史）的信息。这一章所采用的方法与第 7 章中的方法接近，目的是发现金融市场数据中的统计失效。正如数值例子所示，RNN 也可以达到 60% 以上的样本外预测准确率。
- 第 9 章将强化学习作为人工智能的主要成功案例之一进行了讨论，其中包括应用于 OpenAI Gym 模拟物理环境和这一章开发的金融市场环境的不同 RL 智能体。强化学习中选择的算法通常是 Q 学习，这一章对此进行了详细讨论并将其应用于训练交易机器人。交易机器人显示出了可观的样本外财务表现，这通常是比单独的预测准确率更重要的衡

注 1：*The Man Who Solved the Market*，Gregory Zuckerman，2019 年。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

量标准。从这个意义上说，这一章为第四部分建立了一个自然的桥梁，第四部分主要关注经济地利用统计失效。

尽管**卷积神经网络**（CNN）是一种非常重要的神经网络类型，但这一部分不详细讨论，附录 C 简要说明了 CNN 的应用。在很多情况下，CNN 也可以应用于 DNN 和 RNN 在本书的这一部分中所应用的问题。

这一部分的方法是一种实用的方法，省略了许多关于所应用的算法和技术的重要细节。之所以如此，是因为我们可以通过图书以及其他形式的可用资源获取技术细节和背景信息。接下来的几章将在适当的时候提供部分的资源参考。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 7 章

密集神经网络

> 如果你试图根据股票的历史价格预测股票市场的走势，那么你不太可能成功，因为历史价格并未包含太多预测性信息。
>
> ——François Chollet，2017 年

本章是对**密集神经网络**（DNN）的重要方面的介绍，前面的章节已经使用过这种类型的神经网络。特别是，基于 `scikit-learn` 的 `MLPClassifier` 模型和 `MLPRegressor` 模型，以及基于 `Keras` 的用于分类和估计的 `Sequential` 模型，它们都是 DNN。本章专门介绍了 `Keras`，因为它为 DNN 建模提供了更大的自由度和灵活性。[1]

7.1 节介绍了本章其他部分将要使用的外汇（FX）数据集。7.2 节在新数据集上生成了基线样本内预测。训练和测试数据的归一化在 7.3 节中做了介绍。7.4 节和 7.5 节讨论了“暂退”和“正则化”这两种流行的避免过拟合的方法。装袋是另一种避免过拟合的方法，已在第 6 章中使用过，7.6 节会重新讨论。7.7 节比较了可与 `Keras` DNN 模型一起使用的不同优化器的性能。

尽管章首引言可能没有给出让我们抱有希望的理由，但本章以及整个第三部分的主要目标是通过应用神经网络来发现金融市场（时间序列）中的统计失效。本章中给出的数值结果，比如在某些情况下预测准确率为 60% 甚至更高，表明至少有一些希望是合理的。

7.1　数据

第 6 章发现了欧元 / 美元货币对的日内价格序列等时间序列统计失效的线索，本章和随后几章会重点介绍作为资产类别的外汇，特别是欧元 / 美元货币对。其中一个原因是，在经

注 1：有关 `Keras` 包的详细信息和背景知识，请参阅 Chollet（2017），或参阅 Goodfellow 等（2016）对神经网络和相关方法的综述。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

济上利用外汇统计失效通常不像其他资产类别那样复杂，比如 VIX 波动率指数等波动率产品。外汇也经常提供免费和全面的数据。以下数据集来自 Refinitiv Eikon Data API。数据集已通过 API 读取，包含开盘价、最高价、最低价和收盘价，图 7-1 显示了收盘价。

```
In [1]: import os
        import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        pd.set_option('precision', 4)
        np.set_printoptions(suppress=True, precision=4)
        os.environ['PYTHONHASHSEED'] = '0'

In [2]: url = 'http://hilpisch.com/aiif_eikon_id_eur_usd.csv'  ❶

In [3]: symbol = 'EUR_USD'

In [4]: raw = pd.read_csv(url, index_col=0, parse_dates=True)  ❶

In [5]: raw.head()
Out[5]:                        HIGH     LOW    OPEN   CLOSE
        Date
        2019-10-01 00:00:00  1.0899  1.0897  1.0897  1.0899
        2019-10-01 00:01:00  1.0899  1.0896  1.0899  1.0898
        2019-10-01 00:02:00  1.0898  1.0896  1.0898  1.0896
        2019-10-01 00:03:00  1.0898  1.0896  1.0897  1.0898
        2019-10-01 00:04:00  1.0898  1.0896  1.0897  1.0898

In [6]: raw.info()
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 96526 entries, 2019-10-01 00:00:00 to 2019-12-31 23:06:00
        Data columns (total 4 columns):
         #   Column  Non-Null Count  Dtype
        ---  ------  --------------  -----
         0   HIGH    96526 non-null  float64
         1   LOW     96526 non-null  float64
         2   OPEN    96526 non-null  float64
         3   CLOSE   96526 non-null  float64
        dtypes: float64(4)
        memory usage: 3.7 MB

In [7]: data = pd.DataFrame(raw['CLOSE'].loc[:])  ❷
        data.columns = [symbol]  ❷

In [8]: data = data.resample('1h', label='right').last().ffill()  ❷

In [9]: data.info()
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 2208 entries, 2019-10-01 01:00:00 to 2020-01-01 00:00:00
        Freq: H
        Data columns (total 1 columns):
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   EUR_USD  2208 non-null   float64
dtypes: float64(1)
memory usage: 34.5 KB

In [10]: data.plot(figsize=(10, 6));  ❷
```

❶ 将数据读入 DataFrame 对象。

❷ 选择、重新采样并绘制收盘价。

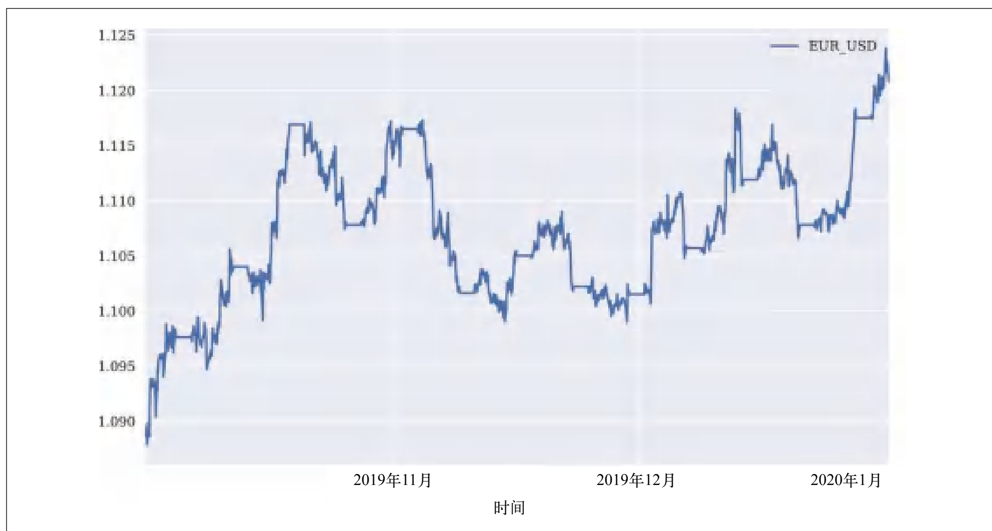


图 7-1：欧元 / 美元货币对的中间收盘价（日内）

7.2　基线预测

基于新的数据集，重复使用第 6 章的预测方法。首先，创建滞后特征。

```
In [11]: lags = 5

In [12]: def add_lags(data, symbol, lags, window=20):  ❶
             cols = []
             df = data.copy()
             df.dropna(inplace=True)
             df['r'] = np.log(df / df.shift())
             df['sma'] = df[symbol].rolling(window).mean()
             df['min'] = df[symbol].rolling(window).min()
             df['max'] = df[symbol].rolling(window).max()
             df['mom'] = df['r'].rolling(window).mean()
             df['vol'] = df['r'].rolling(window).std()
             df.dropna(inplace=True)
             df['d'] = np.where(df['r'] > 0, 1, 0)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            features = [symbol, 'r', 'd', 'sma', 'min', 'max', 'mom', 'vol']
            for f in features:
                for lag in range(1, lags + 1):
                    col = f'{f}_lag_{lag}'
                    df[col] = df[f].shift(lag)
                    cols.append(col)
            df.dropna(inplace=True)
            return df, cols

In [13]: data, cols = add_lags(data, symbol, lags)
```

❶ 基于第 6 章的函数进行微调。

其次，看一下标签数据。取决于可用的数据集，分类中可能出现的一个主要问题是**类别不平衡**。这意味着在二分类标签中一个特定类别的频率可能高于另一个类别。这可能会导致神经网络简单地以更高的频率预测类别，因为这已经可以导致低损失和高准确率值。应用适当的权重，可以确保两个类在 DNN 训练步骤中获得同等重要性。[2]

```
In [14]: len(data)
Out[14]: 2183

In [15]: c = data['d'].value_counts()  ❶
         c  ❶
Out[15]: 0    1445
         1     738
         Name: d, dtype: int64

In [16]: def cw(df):  ❷
             c0, c1 = np.bincount(df['d'])
             w0 = (1 / c0) * (len(df)) / 2
             w1 = (1 / c1) * (len(df)) / 2
             return {0: w0, 1: w1}

In [17]: class_weight = cw(data)  ❷

In [18]: class_weight  ❷
Out[18]: {0: 0.755363321799308, 1: 1.4789972899728998}

In [19]: class_weight[0] * c[0]  ❸
Out[19]: 1091.5

In [20]: class_weight[1] * c[1]  ❸
Out[20]: 1091.5
```

❶ 显示两个类的频率。

❷ 计算适当的权重以达到相等的权重。

❸ 根据计算出的权重，两个类的权重相等。

注 2：请参阅博客文章“Practical Guide to Handling Imbalanced Datasets”，其中讨论了使用 Keras 解决类别不平衡问题。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

最后，使用 Keras 创建 DNN 模型并在完整数据集上训练模型。样本内预测的基线性能约为 60%。

```
In [21]: import random
         import tensorflow as tf
         from keras.layers import Dense
         from keras.models import Sequential
         from keras.optimizers import Adam
         from sklearn.metrics import accuracy_score
         Using TensorFlow backend.

In [22]: def set_seeds(seed=100):
             random.seed(seed)  ❶
             np.random.seed(seed)  ❷
             tf.random.set_seed(seed)  ❸

In [23]: optimizer = Adam(lr=0.001)  ❹

In [24]: def create_model(hl=1, hu=128, optimizer=optimizer):
             model = Sequential()
             model.add(Dense(hu, input_dim=len(cols),
                             activation='relu'))  ❺
             for _ in range(hl):
                 model.add(Dense(hu, activation='relu'))  ❻
             model.add(Dense(1, activation='sigmoid'))  ❼
             model.compile(loss='binary_crossentropy',  ❽
                           optimizer=optimizer,  ❾
                           metrics=['accuracy'])  ❿
             return model

In [25]: set_seeds()
         model = create_model(hl=1, hu=128)

In [26]: %%time
         model.fit(data[cols], data['d'], epochs=50,
                   verbose=False, class_weight=cw(data))
         CPU times: user 6.44 s, sys: 939 ms, total: 7.38 s
         Wall time: 4.07 s

Out[26]: <keras.callbacks.callbacks.History at 0x7fbfc2ee6690>

In [27]: model.evaluate(data[cols], data['d'])
         2183/2183 [==============================] - 0s 24us/step

Out[27]: [0.582192026280068, 0.6087952256202698]

In [28]: data['p'] = np.where(model.predict(data[cols]) > 0.5, 1, 0)

In [29]: data['p'].value_counts()
Out[29]: 1    1340
         0     843
         Name: p, dtype: int64
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❶ 设定 Python 随机数种子。

❷ 设定 NumPy 随机数种子。

❸ 设定 TensorFlow 随机数种子。

❹ 默认优化器。

❺ 第 1 层。

❻ 中间层。

❼ 输出层。

❽ 损失函数。

❾ 使用的优化器。

❿ 要收集的其他指标。

样本外数据模型的性能也是如此，仍然远远高于 60%，这已经算是相当不错了。

```
In [30]: split = int(len(data) * 0.8)  ❶

In [31]: train = data.iloc[:split].copy()  ❷

In [32]: test = data.iloc[split:].copy()  ❸

In [33]: set_seeds()
         model = create_model(hl=1, hu=128)

In [34]: %%time
         model.fit(train[cols], train['d'],
                   epochs=50, verbose=False,
                   validation_split=0.2, shuffle=False,
                   class_weight=cw(train))
         CPU times: user 4.72 s, sys: 686 ms, total: 5.41 s
         Wall time: 3.14 s

Out[34]: <keras.callbacks.callbacks.History at 0x7fbfc3231250>

In [35]: model.evaluate(train[cols], train['d'])  ❹
         1746/1746 [==============================] - 0s 13us/step

Out[35]: [0.612861613500842, 0.5853379368782043]

In [36]: model.evaluate(test[cols], test['d'])  ❺
         437/437 [==============================] - 0s 16us/step

Out[36]: [0.5946959675858714, 0.6247139573097229]

In [37]: test['p'] = np.where(model.predict(test[cols]) > 0.5, 1, 0)

In [38]: test['p'].value_counts()
Out[38]: 1    291
         0    146
         Name: p, dtype: int64
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❶ 拆分整个数据集为……

❷ ……训练集……

❸ ……和测试集。

❹ 评估**样本内**性能。

❺ 评估**样本外**性能。

图 7-2 显示了训练集准确率和验证集准确率如何在训练轮数内变化。

```
In [39]: res = pd.DataFrame(model.history.history)

In [40]: res[['accuracy', 'val_accuracy']].plot(figsize=(10, 6), style='--');
```

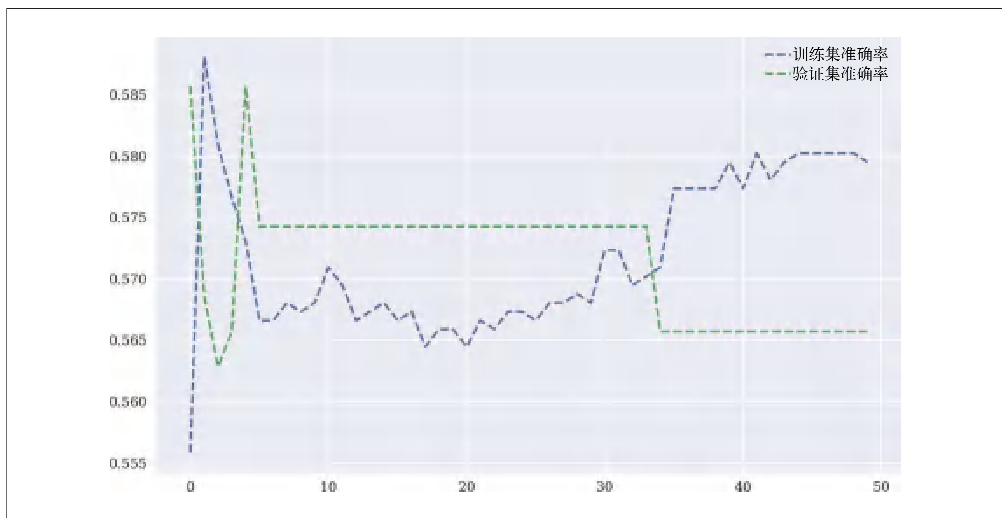


图 7-2：训练集准确率值和验证集准确率值

本节中的分析为更精细地使用 Keras 和 DNN 奠定了基础，它提出了一种市场预测的基线方法。以下各节添加了不同的元素，主要是为了提高样本外模型的性能并避免模型过拟合训练数据。

7.3 归一化

7.2 节中的基线预测采用了滞后特征。在第 6 章中，特征数据是通过减去每个特征的训练数据的均值并除以训练数据的标准差来归一化的，这种归一化技术称为**高斯归一化**，并且已被证明通常是训练神经网络时的一个重要方面。正如以下 Python 代码及其结果所示，在处理归一化特征数据时，样本内性能显著提高，样本外性能也略有提高，但是不能保证通过特征归一化总能提高样本外性能。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [41]: mu, std = train.mean(), train.std()  ❶

In [42]: train_ = (train - mu) / std  ❷

In [43]: set_seeds()
         model = create_model(hl=2, hu=128)

In [44]: %%time
         model.fit(train_[cols], train['d'],
                   epochs=50, verbose=False,
                   validation_split=0.2, shuffle=False,
                   class_weight=cw(train))
         CPU times: user 5.81 s, sys: 879 ms, total: 6.69 s
         Wall time: 3.53 s

Out[44]: <keras.callbacks.callbacks.History at 0x7fbfa51353d0>

In [45]: model.evaluate(train_[cols], train['d'])  ❸
         1746/1746 [==============================] - 0s 14us/step

Out[45]: [0.4253406366728084, 0.887170672416687]

In [46]: test_ = (test - mu) / std  ❹

In [47]: model.evaluate(test_[cols], test['d'])  ❺
         437/437 [==============================] - 0s 24us/step

Out[47]: [1.1377735263422917, 0.681922197341919]

In [48]: test['p'] = np.where(model.predict(test_[cols]) > 0.5, 1, 0)

In [49]: test['p'].value_counts()
Out[49]: 0    281
         1    156
         Name: p, dtype: int64
```

❶ 计算所有**训练特征**的均值和标准差。

❷ 基于高斯归一化对**训练数据集**进行归一化。

❸ 评估**样本内**性能。

❹ 基于高斯归一化对**测试数据集**进行归一化。

❺ 评估**样本外**性能。

经常出现的一个主要问题是**过拟合**。从图 7-3 中可以看出，训练集准确率稳步提高，而验证集准确率缓慢下降。

```
In [50]: res = pd.DataFrame(model.history.history)

In [51]: res[['accuracy', 'val_accuracy']].plot(figsize=(10, 6), style='--');
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

图 7-3：训练集准确率值和验证集准确率值（归一化特征数据）

3 种避免过拟合的方法是暂退（dropout）、正则化（regularization）和装袋，接下来会分别讨论这些方法，之后还会讨论所选优化器的影响。

7.4 暂退

暂退是指神经网络在训练阶段不应该使用所有隐藏单元。人类经常忘记先前学到的信息，这使人的大脑保持“思想开放”。理想情况下，神经网络的行为应该类似：DNN 中的连接不应变得太强，以避免过拟合训练数据。

从技术上讲，`Keras` 模型在隐藏层间有额外的层管理暂退，主要参数是层的隐藏单元被丢弃的速率。这些丢弃通常以随机方式发生，可以通过固定**种子**参数来避免这种随机性。在样本内性能下降的同时，样本外性能也略有下降。但是，两种性能指标之间的差异较小，这通常是一种理想的情况。

```
In [52]: from keras.layers import Dropout

In [53]: def create_model(hl=1, hu=128, dropout=True, rate=0.3,
                          optimizer=optimizer):
             model = Sequential()
             model.add(Dense(hu, input_dim=len(cols),
                             activation='relu'))
             if dropout:
                 model.add(Dropout(rate, seed=100))  ❶
             for _ in range(hl):
                 model.add(Dense(hu, activation='relu'))
                 if dropout:
                     model.add(Dropout(rate, seed=100))  ❶
             model.add(Dense(1, activation='sigmoid'))
             model.compile(loss='binary_crossentropy', optimizer=optimizer,
                           metrics=['accuracy'])
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            return model

In [54]: set_seeds()
         model = create_model(hl=1, hu=128, rate=0.3)

In [55]: %%time
         model.fit(train_[cols], train['d'],
                   epochs=50, verbose=False,
                   validation_split=0.15, shuffle=False,
                   class_weight=cw(train))
         CPU times: user 5.46 s, sys: 758 ms, total: 6.21 s
         Wall time: 3.53 s

Out[55]: <keras.callbacks.callbacks.History at 0x7fbfa6386550>

In [56]: model.evaluate(train_[cols], train['d'])
         1746/1746 [==============================] - 0s 20us/step

Out[56]: [0.4423361133190911, 0.7840778827667236]

In [57]: model.evaluate(test_[cols], test['d'])
         437/437 [==============================] - 0s 34us/step

Out[57]: [0.5875822428434883, 0.6430205702781677]
```

❶ 在每一层之后加入暂退。

```
In [58]: res = pd.DataFrame(model.history.history)

In [59]: res[['accuracy', 'val_accuracy']].plot(figsize=(10, 6), style='--');
```

如图 7-4 所示，训练集准确率和验证集准确率没有像以前那样快速分开。

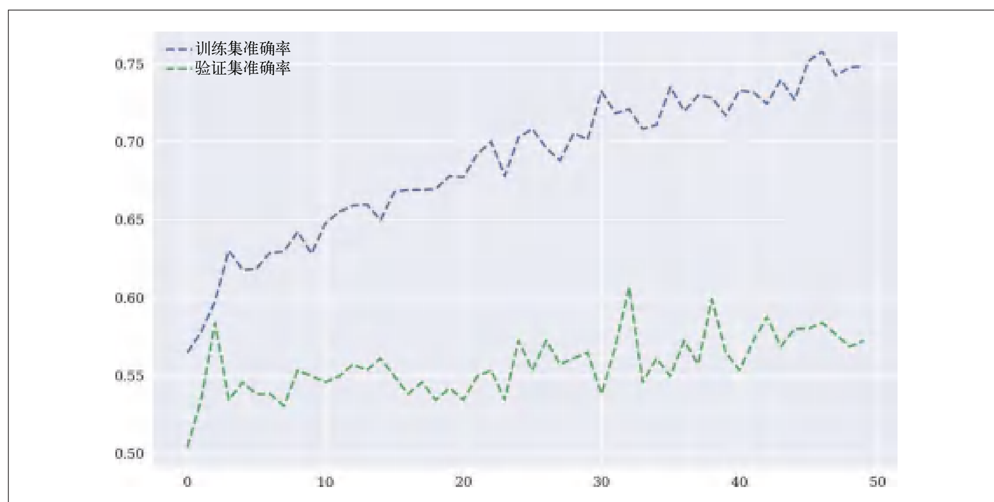


图 7-4：训练集准确率值和验证集准确率值（使用暂退）

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

刻意遗忘

Keras 的 Sequential 模型中的暂退模拟了人类的经历：忘记以前记忆的信息。这是通过在训练期间停用隐藏层的某些隐藏单元来实现的。实际上，这通常在更大程度上避免了神经网络对训练数据的过拟合。

7.5 正则化

另一种避免过拟合的方法是**正则化**。通过正则化，神经网络中数值大的权重在计算损失（函数）时会受到惩罚，这避免了 DNN 中某些连接变得过于强大和占主导地位的情况。可以通过 Dense 层中的参数在 Keras 的 DNN 中引入正则化。根据选择的正则化参数，训练集准确率和测试集准确率可以非常接近。一般使用两种正则化器，一种基于线性范数 l1，一种基于欧几里得范数 l2。以下 Python 代码为模型创建函数添加了正则化。

```
In [60]: from keras.regularizers import l1, l2

In [61]: def create_model(hl=1, hu=128, dropout=False, rate=0.3,
                          regularize=False, reg=l1(0.0005),
                          optimizer=optimizer, input_dim=len(cols)):
             if not regularize:
                 reg = None
             model = Sequential()
             model.add(Dense(hu, input_dim=input_dim,
                             activity_regularizer=reg,  ❶
                             activation='relu'))
             if dropout:
                 model.add(Dropout(rate, seed=100))
             for _ in range(hl):
                 model.add(Dense(hu, activation='relu',
                                 activity_regularizer=reg))  ❶
                 if dropout:
                     model.add(Dropout(rate, seed=100))
             model.add(Dense(1, activation='sigmoid'))
             model.compile(loss='binary_crossentropy', optimizer=optimizer,
                           metrics=['accuracy'])
             return model

In [62]: set_seeds()
         model = create_model(hl=1, hu=128, regularize=True)

In [63]: %%time
         model.fit(train_[cols], train['d'],
                   epochs=50, verbose=False,
                   validation_split=0.2, shuffle=False,
                   class_weight=cw(train))
         CPU times: user 5.49 s, sys: 1.05 s, total: 6.54 s
         Wall time: 3.15 s

Out[63]: <keras.callbacks.callbacks.History at 0x7fbfa6b8e110>

In [64]: model.evaluate(train_[cols], train['d'])
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        1746/1746 [==============================] - 0s 15us/step

Out[64]: [0.5307255412568205, 0.7691867351531982]

In [65]: model.evaluate(test_[cols], test['d'])
         437/437 [==============================] - 0s 22us/step

Out[65]: [0.8428352184644826, 0.6590389013290405]
```

❶ 正则化被添加到每一层。

图 7-5 显示了正则化下的训练集准确率和验证集准确率，这两个性能指标比以前看到的要紧密得多。

```
In [66]: res = pd.DataFrame(model.history.history)

In [67]: res[['accuracy', 'val_accuracy']].plot(figsize=(10, 6), style='--');
```

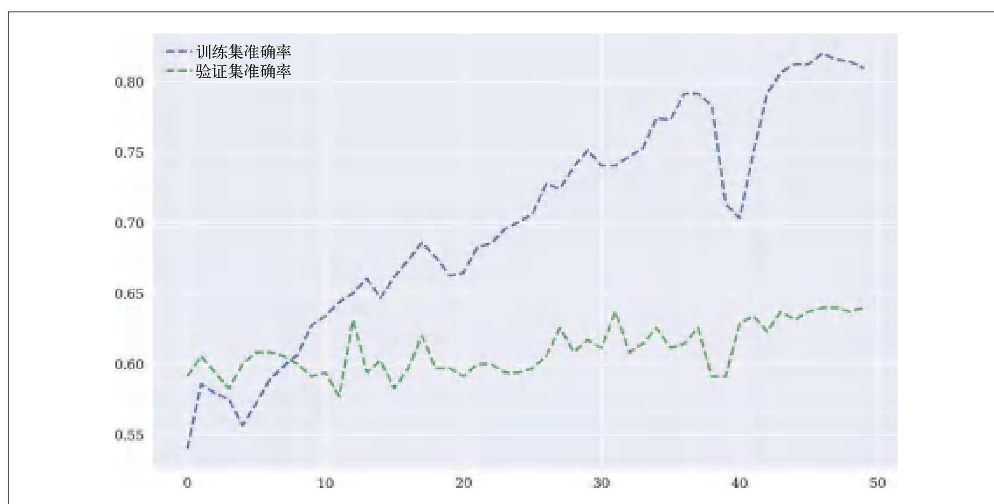


图 7-5：训练集准确率值和验证集准确率值（使用正则化）

当然，暂退和正则化可以一起使用。将两个度量结合起来甚至可以更好地避免过拟合，并使样本内和样本外的准确率值更接近。在这种情况下，训练集准确率和验证集准确率之间的差异确实是最小的。

```
In [68]: set_seeds()
         model = create_model(hl=2, hu=128,
                              dropout=True, rate=0.3,  ❶
                              regularize=True, reg=l2(0.001),  ❷
                             )

In [69]: %%time
         model.fit(train_[cols], train['d'],
                   epochs=50, verbose=False,
                   validation_split=0.2, shuffle=False,
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
                class_weight=cw(train))
        CPU times: user 7.06 s, sys: 958 ms, total: 8.01 s
        Wall time: 4.28 s

Out[69]: <keras.callbacks.callbacks.History at 0x7fbfa701cb50>

In [70]: model.evaluate(train_[cols], train['d'])
        1746/1746 [==============================] - 0s 18us/step

Out[70]: [0.5007762827004764, 0.7691867351531982]

In [71]: model.evaluate(test_[cols], test['d'])
        437/437 [==============================] - 0s 23us/step

Out[71]: [0.6191965124699835, 0.6864988803863525]
```

❶ 在模型创建中加入暂退。

❷ 在模型创建中加入正则化。

图 7-6 显示了结合暂退和正则化时的训练集准确率和验证集准确率。在整个训练期间，训练集准确率和验证集准确率之间的差异平均仅为大约 4 个百分点。

```
In [72]: res = pd.DataFrame(model.history.history)

In [73]: res[['accuracy', 'val_accuracy']].plot(figsize=(10, 6), style='--');
```

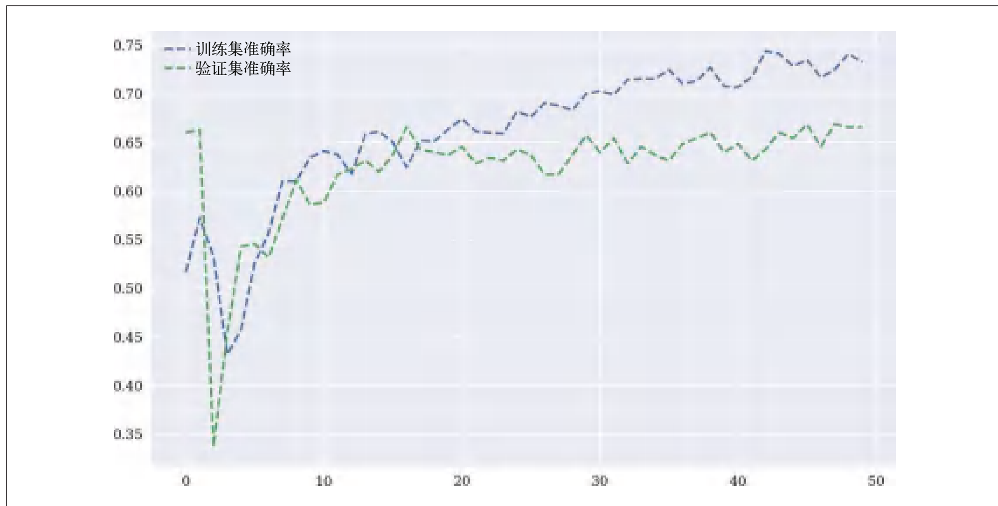


图 7-6：训练集准确率值和验证集准确率值（同时使用暂退和正则化）

惩罚大权重

正则化通过惩罚神经网络中的大权重来避免过拟合。单个权重不能大到足以支配神经网络，从而使权重整体保持在可比水平。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

7.6 装袋

避免过拟合的装袋方法已经在第 6 章中使用过，尽管仅用于 scikit-learn 的 MLPClassifier 模型。另外，Keras DNN 分类模型的包装器也会以 scikit-learn 的方式使用装袋，即 KerasClassifier 类。以下 Python 代码将基于包装器的 Keras DNN 建模与来自 scikit-learn 的 BaggingClassifier 结合了起来。样本内和样本外模型性能相对较高，约为 70%。然而，如前所述，结果是由类不平衡驱动的，正如这里反映的类别 0 的预测频率较高。

```
In [75]: from sklearn.ensemble import BaggingClassifier
         from keras.wrappers.scikit_learn import KerasClassifier

In [76]: max_features = 0.75

In [77]: set_seeds()
         base_estimator = KerasClassifier(build_fn=create_model,
                                verbose=False, epochs=20, hl=1, hu=128,
                                dropout=True, regularize=False,
                                input_dim=int(len(cols) * max_features))  ❶

In [78]: model_bag = BaggingClassifier(base_estimator=base_estimator,
                                 n_estimators=15,
                                 max_samples=0.75,
                                 max_features=max_features,
                                 bootstrap=True,
                                 bootstrap_features=True,
                                 n_jobs=1,
                                 random_state=100,
                                )  ❷

In [79]: %time model_bag.fit(train_[cols], train['d'])
         CPU times: user 40 s, sys: 5.23 s, total: 45.3 s
         Wall time: 26.3 s

Out[79]: BaggingClassifier(base_estimator=<keras.wrappers.scikit_learn.KerasClassifier
          object at 0x7fbfa7cc7b90>,
         bootstrap_features=True, max_features=0.75, max_samples=0.75,
                           n_estimators=15, n_jobs=1, random_state=100)

In [80]: model_bag.score(train_[cols], train['d'])
Out[80]: 0.720504009163803

In [81]: model_bag.score(test_[cols], test['d'])
Out[81]: 0.6704805491990846

In [82]: test['p'] = model_bag.predict(test_[cols])

In [83]: test['p'].value_counts()
Out[83]: 0    408
         1     29
         Name: p, dtype: int64
```

❶ 基础估计器，这里是 Keras 的 Sequential 模型的实例。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❷ BaggingClassifier 模型被实例化为许多相同的基础估计器。

分布式学习

从某种意义上说，装袋在许多神经网络（或其他模型）之间分配学习任务，比如，每个神经网络只能看到训练集的某些数据和部分特征。这避免了单个神经网络过拟合完整训练集的风险。预测结果基于所有进行选择性训练的神经网络的输出综合得出。

7.7　优化器

Keras 包提供了一系列可与 Sequential 模型结合使用的优化器（optimizer）。不同的优化器可能会表现出不同的性能，包括训练时间和预测准确率。以下 Python 代码使用不同的优化器并对其性能进行了基准测试。在所有情况下，都应该使用 Keras 的默认参数化。样本外性能变化不大。然而，不同优化器的样本内性能差异很大。

```
In [84]: import time

In [85]: optimizers = ['sgd', 'rmsprop', 'adagrad', 'adadelta',
                       'adam', 'adamax', 'nadam']

In [86]: %%time
         for optimizer in optimizers:
             set_seeds()
             model = create_model(hl=1, hu=128,
                                  dropout=True, rate=0.3,
                                  regularize=False, reg=l2(0.001),
                                  optimizer=optimizer
                                 )  ❶
             t0 = time.time()
             model.fit(train_[cols], train['d'],
                       epochs=50, verbose=False,
                       validation_split=0.2, shuffle=False,
                       class_weight=cw(train))  ❷
             t1 = time.time()
             t = t1 - t0
             acc_tr = model.evaluate(train_[cols], train['d'], verbose=False)[1]  ❸
             acc_te = model.evaluate(test_[cols], test['d'], verbose=False)[1]  ❹
             out = f'{optimizer:10s} | time[s]: {t:.4f} | in-sample={acc_tr:.4f}'
             out += f' | out-of-sample={acc_te:.4f}'
             print(out)
         sgd        | time[s]: 2.8092 | in-sample=0.6363 | out-of-sample=0.6568
         rmsprop    | time[s]: 2.9480 | in-sample=0.7600 | out-of-sample=0.6613
         adagrad    | time[s]: 2.8472 | in-sample=0.6747 | out-of-sample=0.6499
         adadelta   | time[s]: 3.2068 | in-sample=0.7279 | out-of-sample=0.6522
         adam       | time[s]: 3.2364 | in-sample=0.7365 | out-of-sample=0.6545
         adamax     | time[s]: 3.2465 | in-sample=0.6982 | out-of-sample=0.6476
         nadam      | time[s]: 4.1275 | in-sample=0.7944 | out-of-sample=0.6590
         CPU times: user 35.9 s, sys: 4.55 s, total: 40.4 s
         Wall time: 23.1 s
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❶ 为给定的优化器实例化 DNN 模型。

❷ 使用给定的优化器拟合模型。

❸ 评估**样本内**性能。

❹ 评估**样本外**性能。

7.8 结论

本章深入探讨了 DNN，并使用 Keras 作为主要库。Keras 在构造 DNN 时提供了高度的灵活性。本章中的结果很不错，因为就预测准确率而言，样本内和样本外的性能始终保持在 60% 或更高。然而，预测准确率只是一个方面，为了从预测或“信号”中获利，必须有适当的交易策略可用且可实施，第四部分详细讨论了这个在算法交易背景下最重要的主题。接下来的两章首先说明了不同神经网络架构（循环神经网络和卷积神经网络）和学习技术（强化学习）的使用。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 8 章

循环神经网络

历史从不重演，但会模仿。

——Mark Twain（可能）

我的生活似乎是一连串的事件和意外，然而，当我回头看时，在其中看到了模式。

——Bernoît Mandelbrot

本章将介绍**循环神经网络**（recurrent neural network，RNN），这种类型的网络专门用于学习序列数据，比如文本或时间序列数据。和以前一样，本章的讨论采用实证的方法，主要依赖于基于 `Keras` 库的 Python 示例。[1]

8.1 节和 8.2 节中的两个示例在两个带有样本数值数据的简单示例的基础上介绍了 RNN，说明了 RNN 在预测序列数据中的应用。8.3 节介绍了“金融价格序列”，然后使用金融价格序列数据并应用 RNN 方法通过估计来直接预测此类序列。8.4 节介绍了“金融收益率序列”，然后使用收益率数据，通过估计方法来预测金融工具价格的未来方向。除了价格和收益率数据，8.5 节还在组合中添加了金融特征来预测市场方向。这一节说明了如下方法：一是通过用于估计和分类的浅层 RNN 模型进行预测，二是通过用于分类的深度 RNN 模型进行预测。

本章表明，将 RNN 应用于金融时间序列数据可以在定向市场预测的背景下实现远高于样本外 60% 的预测准确率。然而，得到的结果不能完全与第 7 章中的结果相比。这可能会让人感到意外，因为 RNN 旨在很好地处理金融时间序列数据，而这也是本书的重点。

注 1：有关 RNN 的技术细节，请参阅 Goodfellow 等（2016）的第 10 章。对于实际实现，请参阅 Chollet 等（2017）的第 6 章。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

8.1 第一个示例

为了演示 RNN 的训练和使用，考虑一个基于整数序列的简单示例。首先，代码中的引用和配置如下。

```
In [1]: import os
        import random
        import numpy as np
        import pandas as pd
        import tensorflow as tf
        from pprint import pprint
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        pd.set_option('precision', 4)
        np.set_printoptions(suppress=True, precision=4)
        os.environ['PYTHONHASHSEED'] = '0'

In [2]: def set_seeds(seed=100):  ❶
            random.seed(seed)
            np.random.seed(seed)
            tf.random.set_seed(seed)
        set_seeds()  ❶
```

❶ 设置随机数种子。

然后，转换成合适形状的简单数据集。

```
In [3]: a = np.arange(100)  ❶
        a
Out[3]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
               17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
               34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
               51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
               68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
               85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99])

In [4]: a = a.reshape((len(a), -1))  ❷

In [5]: a.shape  ❷
Out[5]: (100, 1)

In [6]: a[:5]  ❷
Out[6]: array([[0],
               [1],
               [2],
               [3],
               [4]])
```

❶ 样本数据。

❷ 转换为二维数据。

可以使用 `TimeseriesGenerator` 将原始数据转换为适合 RNN 训练的对象。这样做的目的是利用原始数据中一些滞后数据来训练模型以预测序列中的下一个值，例如，`0`、`1` 和 `2` 是 `3`

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

个滞后值（特征），用来预测值 3（标签）。同理，用 1、2 和 3 来预测 4。

```
In [7]: from keras.preprocessing.sequence import TimeseriesGenerator
        Using TensorFlow backend.

In [8]: lags = 3

In [9]: g = TimeseriesGenerator(a, a, length=lags, batch_size=5)  ❶

In [10]: pprint(list(g)[0])  ❶
         (array([[[0],
                  [1],
                  [2]],

                 [[1],
                  [2],
                  [3]],

                 [[2],
                  [3],
                  [4]],

                 [[3],
                  [4],
                  [5]],

                 [[4],
                  [5],
                  [6]]]),
          array([[3],
                 [4],
                 [5],
                 [6],
                 [7]]))
```

❶ 通过 `TimeseriesGenerator` 创建一批滞后数据。

RNN 模型的创建类似于 DNN，以下 Python 代码使用了 `SimpleRNN` 类的单个隐藏层。[参阅 Chollet 等（2017）的第 6 章以及 `Keras` 中循环层相关文档。] 即使考虑到隐藏单元相对较少，可训练参数的数量仍然相当大。`.fit_generator()` 方法会将使用 `TimeseriesGenerator` 创建的对象作为输入生成器对象。

```
In [11]: from keras.models import Sequential
         from keras.layers import SimpleRNN, LSTM, Dense

In [12]: model = Sequential()
         model.add(SimpleRNN(100, activation='relu',
                             input_shape=(lags, 1)))  ❶
         model.add(Dense(1, activation='linear'))
         model.compile(optimizer='adagrad', loss='mse',
                       metrics=['mae'])

In [13]: model.summary()  ❷
         Model: "sequential_1"
         _________________________________________________________________
         Layer (type)                 Output Shape              Param #
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
=================================================================
simple_rnn_1 (SimpleRNN)    (None, 100)               10200
_________________________________________________________________
dense_1 (Dense)             (None, 1)                 101
=================================================================
Total params: 10,301
Trainable params: 10,301
Non-trainable params: 0
_________________________________________________________________

In [14]: %%time
         model.fit_generator(g, epochs=1000, steps_per_epoch=5,
                             verbose=False)  ❸
         CPU times: user 17.4 s, sys: 3.9 s, total: 21.3 s
         Wall time: 30.8 s

Out[14]: <keras.callbacks.callbacks.History at 0x7f7f079058d0>
```

❶ 用 SimpleRNN 作为单一隐藏层。

❷ 浅层 RNN 模型描述信息。

❸ 基于生成器对象对 RNN 进行拟合。

在训练 RNN 时，模型性能指标可能会表现出相对不稳定的行为（参见图 8-1）。

```
In [15]: res = pd.DataFrame(model.history.history)

In [16]: res.tail(3)
Out[16]:        loss     mae
         997  0.0001  0.0109
         998  0.0007  0.0211
         999  0.0001  0.0101

In [17]: res.iloc[10:].plot(figsize=(10, 6), style=['--', '--']);
```

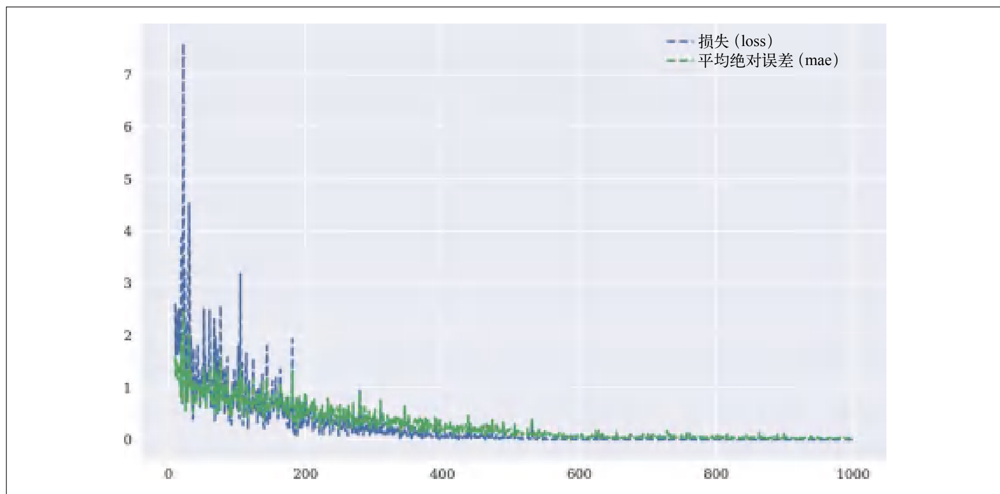


图 8-1：RNN 训练期间的性能指标

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

基于训练好的 RNN，以下 Python 代码生成了样本内预测和样本外预测。

```
In [18]: x = np.array([21, 22, 23]).reshape((1, lags, 1))
         y = model.predict(x, verbose=False)  ❶
         int(round(y[0, 0]))
Out[18]: 24

In [19]: x = np.array([87, 88, 89]).reshape((1, lags, 1))
         y = model.predict(x, verbose=False)  ❶
         int(round(y[0, 0]))
Out[19]: 90

In [20]: x = np.array([187, 188, 189]).reshape((1, lags, 1))
         y = model.predict(x, verbose=False)  ❷
         int(round(y[0, 0]))
Out[20]: 190

In [21]: x = np.array([1187, 1188, 1189]).reshape((1, lags, 1))
         y = model.predict(x, verbose=False)  ❸
         int(round(y[0, 0]))
Out[21]: 1194
```

❶ 样本内预测。

❷ 样本外预测。

❸ 远离样本的预测。

即使对于远离样本的预测，在这种简单的情形下，结果通常也很好。然而，同样的问题通过 OLS 回归也可以完美解决。因此，考虑到 RNN 的性能，针对此类问题训练 RNN 所涉及的工作量非常大。

8.2 第二个示例

第一个示例说明了针对简单问题的 RNN 训练，该问题不仅可以通过 OLS 回归轻松解决，也可以通过人工检查数据轻松解决。第二个示例更具挑战性，输入数据通过二次项和三角项进行转换，并添加了白噪声。图 8-2 显示了区间 $[-2\pi, 2\pi]$ 的序列值。

```
In [22]: def transform(x):
             y = 0.05 * x ** 2 + 0.2 * x + np.sin(x) + 5  ❶
             y += np.random.standard_normal(len(x)) * 0.2  ❷
             return y

In [23]: x = np.linspace(-2 * np.pi, 2 * np.pi, 500)
         a = transform(x)

In [24]: plt.figure(figsize=(10, 6))
         plt.plot(x, a);
```

❶ 确定性变换。

❷ 随机变换。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

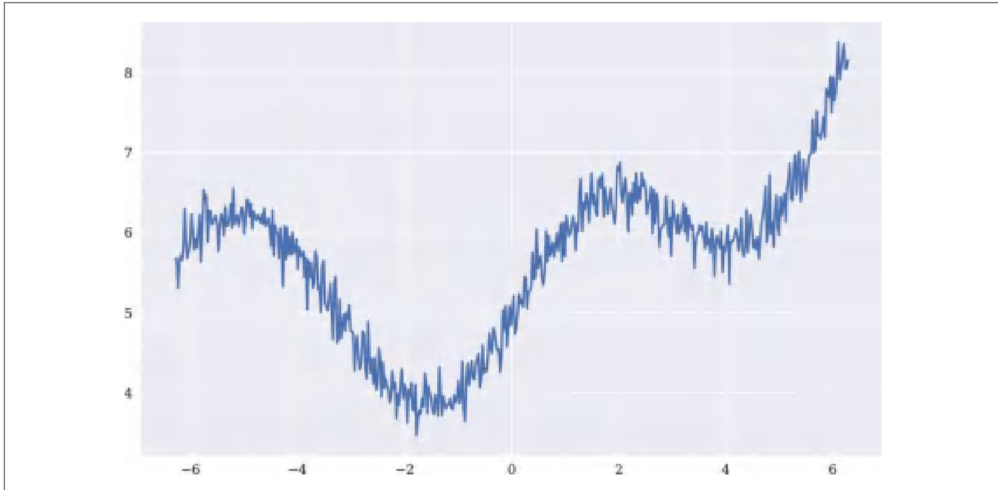


图 8-2：样本序列数据

和之前一样，应用 TimeseriesGenerator 对原始数据进行变换，并训练具有单个隐藏层的 RNN。

```
In [25]: a = a.reshape((len(a), -1))

In [26]: a[:5]
Out[26]: array([[5.6736],
                [5.68  ],
                [5.3127],
                [5.645 ],
                [5.7118]])

In [27]: lags = 5

In [28]: g = TimeseriesGenerator(a, a, length=lags, batch_size=5)

In [29]: model = Sequential()
         model.add(SimpleRNN(500, activation='relu', input_shape=(lags, 1)))
         model.add(Dense(1, activation='linear'))
         model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

In [30]: model.summary()
         Model: "sequential_2"
         _________________________________________________________________
         Layer (type)                 Output Shape              Param #
         =================================================================
         simple_rnn_2 (SimpleRNN)     (None, 500)               251000
         _________________________________________________________________
         dense_2 (Dense)              (None, 1)                 501
         =================================================================
         Total params: 251,501
         Trainable params: 251,501
         Non-trainable params: 0
         _________________________________________________________________
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [31]: %%time
         model.fit_generator(g, epochs=500,
                             steps_per_epoch=10,
                             verbose=False)
         CPU times: user 1min 6s, sys: 14.6 s, total: 1min 20s
         Wall time: 23.1 s

Out[31]: <keras.callbacks.callbacks.History at 0x7f7f09c11810>
```

以下 Python 代码会预测区间 [−6π, 6π] 的序列值，这个区间是训练区间的 3 倍，并且在训练区间的左侧和右侧都包含样本外预测。图 8-3 显示了即使在样本外该模型表现也非常好。

```
In [32]: x = np.linspace(-6 * np.pi, 6 * np.pi, 1000)  ❶
         d = transform(x)

In [33]: g_ = TimeseriesGenerator(d, d, length=lags, batch_size=len(d))  ❶

In [34]: f = list(g_)[0][0].reshape((len(d) - lags, lags, 1))  ❶

In [35]: y = model.predict(f, verbose=False)  ❷

In [36]: plt.figure(figsize=(10, 6))
         plt.plot(x[lags:], d[lags:], label='data', alpha=0.75)
         plt.plot(x[lags:], y, 'r.', label='pred', ms=3)
         plt.axvline(-2 * np.pi, c='g', ls='--')
         plt.axvline(2 * np.pi, c='g', ls='--')
         plt.text(-15, 22, 'out-of-sample')
         plt.text(-2, 22, 'in-sample')
         plt.text(10, 22, 'out-of-sample')
         plt.legend();
```

❶ 扩大样本数据集。

❷ 样本内预测和样本外预测。

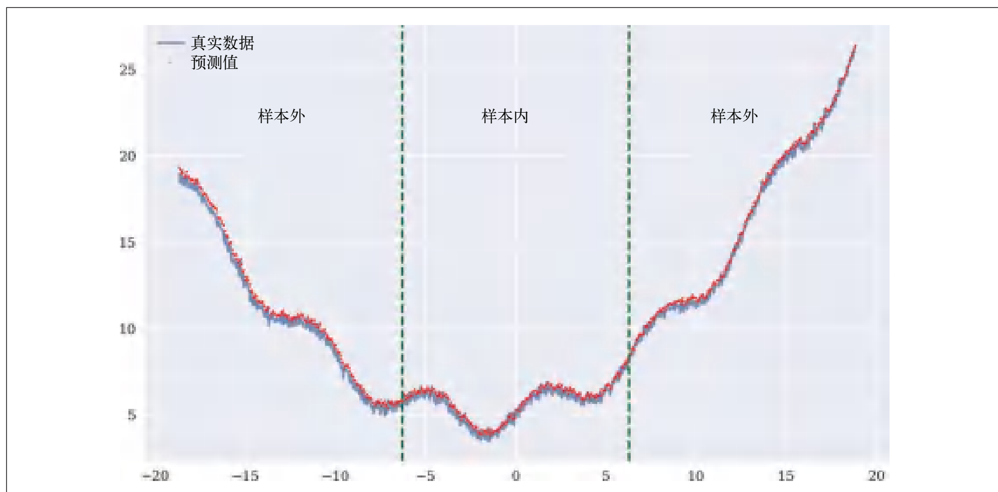


图 8-3：RNN 的样本内预测和样本外预测

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

示例的简化

上述两个示例是有意简化过的，示例中提出的两个问题都可以使用 OLS 回归更有效地解决，例如，通过允许在第二个示例中使用三角函数来解决。然而，RNN 对非平凡序列数据（如金融时间序列数据）的训练基本相同，在这种情况下，OLS 回归的性能通常无法与 RNN 相比。

8.3 金融价格序列

作为 RNN 首次应用于金融时间序列数据的尝试，这里考虑日内欧元 / 美元报价。使用前两节介绍的方法，RNN 在金融时间序列上的训练很简单。首先，导入数据并重新采样。也可以将数据规范化并转换为适当的 `ndarray` 对象。

```
In [37]: url = 'http://hilpisch.com/aiif_eikon_id_eur_usd.csv'

In [38]: symbol = 'EUR_USD'

In [39]: raw = pd.read_csv(url, index_col=0, parse_dates=True)

In [40]: def generate_data():
             data = pd.DataFrame(raw['CLOSE'])  ❶
             data.columns = [symbol]  ❷
             data = data.resample('30min', label='right').last().ffill()  ❸
             return data

In [41]: data = generate_data()

In [42]: data = (data - data.mean()) / data.std()  ❹

In [43]: p = data[symbol].values  ❺

In [44]: p = p.reshape((len(p), -1))  ❺
```

❶ 选择一列。

❷ 将该列重命名。

❸ 重新采样数据。

❹ 应用高斯归一化。

❺ 将数据集重塑为二维。

其次，基于生成器对象对 RNN 模型进行训练。函数 `create_rnn_model()` 允许使用 `SimpleRNN` 层或 `LSTM`（长短期记忆）层创建 RNN。[参阅 Chollet 等（2017）的第 6 章以及 Keras 中循环层相关文档。]

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [45]: lags = 5

In [46]: g = TimeseriesGenerator(p, p, length=lags, batch_size=5)

In [47]: def create_rnn_model(hu=100, lags=lags, layer='SimpleRNN',
                                    features=1, algorithm='estimation'):
             model = Sequential()
             if layer is 'SimpleRNN':
                 model.add(SimpleRNN(hu, activation='relu',
                                      input_shape=(lags, features)))  ❶
             else:
                 model.add(LSTM(hu, activation='relu',
                                 input_shape=(lags, features)))  ❶
             if algorithm == 'estimation':
                 model.add(Dense(1, activation='linear'))  ❷
                 model.compile(optimizer='adam', loss='mse', metrics=['mae'])
             else:
                 model.add(Dense(1, activation='sigmoid'))  ❷
                 model.compile(optimizer='adam', loss='mse', metrics=['accuracy'])
             return model

In [48]: model = create_rnn_model()

In [49]: %%time
         model.fit_generator(g, epochs=500, steps_per_epoch=10,
                             verbose=False)
         CPU times: user 20.8 s, sys: 4.66 s, total: 25.5 s
         Wall time: 11.2 s

Out[49]: <keras.callbacks.callbacks.History at 0x7f7ef6716590>
```

❶ 增加一个 SimpleRNN 层或 LSTM 层。

❷ 增加一个用于**估计**或**分类**的输出层。

最后，生成样本内预测。如图 8-4 所示，RNN 能够捕获归一化金融时间序列数据的结构。从可视化结果可以看出，预测准确率似乎相当不错。

```
In [50]: y = model.predict(g, verbose=False)

In [51]: data['pred'] = np.nan
         data['pred'].iloc[lags:] = y.flatten()

In [52]: data[[symbol, 'pred']].plot(
                     figsize=(10, 6), style=['b', 'r-.'],
                     alpha=0.75);
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

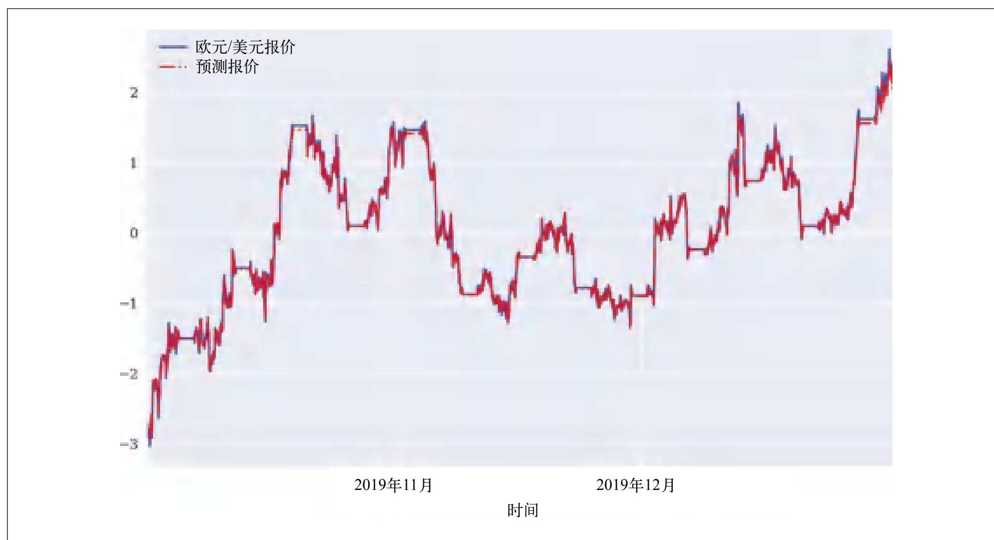


图 8-4：RNN 对金融价格序列的样本内预测（数据全集）

然而，可视化显示的结果经不起仔细检查，将图 8-5 放大，仅显示来自原始数据集和预测的 50 个数据点。很明显，来自 RNN 的预测值基本上只是最近的滞后值，并移动了一个时间间隔。直观上看，预测的价格序列是金融时间序列本身，向右移动了一个时间间隔。

```
In [53]: data[[symbol, 'pred']].iloc[50:100].plot(
                     figsize=(10, 6), style=['b', 'r-.'],
                     alpha=0.75);
```

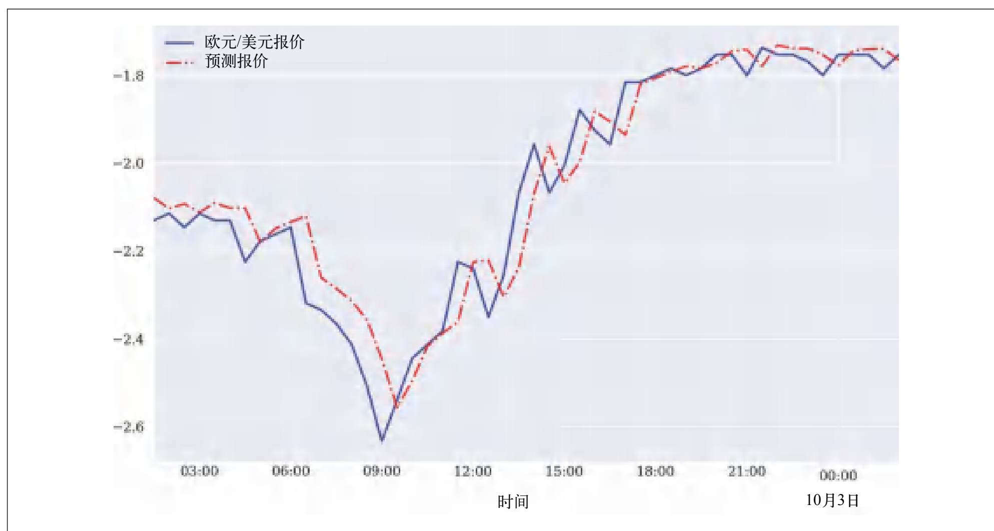


图 8-5：RNN 对金融价格序列的样本内预测（数据子集）

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

RNN 和有效市场

基于 RNN 的金融价格序列的预测结果与第 6 章中用来说明 EMH 的 OLS 回归方法的预测结果一致。第 6 章演示了在最小二乘意义上，当日价格是明日价格的最佳预测指标。在本章，我们可以看到将 RNN 应用于价格数据不会产生任何其他见解。

8.4 金融收益率序列

正如之前的分析所表明的那样，预测收益率可能比预测价格更容易。因此，下面的 Python 代码根据对数收益率重复了前面的分析。

```
In [54]: data = generate_data()

In [55]: data['r'] = np.log(data / data.shift(1))

In [56]: data.dropna(inplace=True)

In [57]: data = (data - data.mean()) / data.std()

In [58]: r = data['r'].values

In [59]: r = r.reshape((len(r), -1))

In [60]: g = TimeseriesGenerator(r, r, length=lags, batch_size=5)

In [61]: model = create_rnn_model()

In [62]: %%time
         model.fit_generator(g, epochs=500, steps_per_epoch=10,
                             verbose=False)
         CPU times: user 20.4 s, sys: 4.2 s, total: 24.6 s
         Wall time: 11.3 s

Out[62]: <keras.callbacks.callbacks.History at 0x7f7ef47a8dd0>
```

如图 8-6 所示，RNN 的预测绝对值不太好。然而，它们似乎以某种方式正确地了解了市场方向（收益率的迹象）。

```
In [63]: y = model.predict(g, verbose=False)

In [64]: data['pred'] = np.nan
         data['pred'].iloc[lags:] = y.flatten()
         data.dropna(inplace=True)

In [65]: data[['r', 'pred']].iloc[50:100].plot(
                     figsize=(10, 6), style=['b', 'r-.'],
                     alpha=0.75);
         plt.axhline(0, c='grey', ls='--')
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

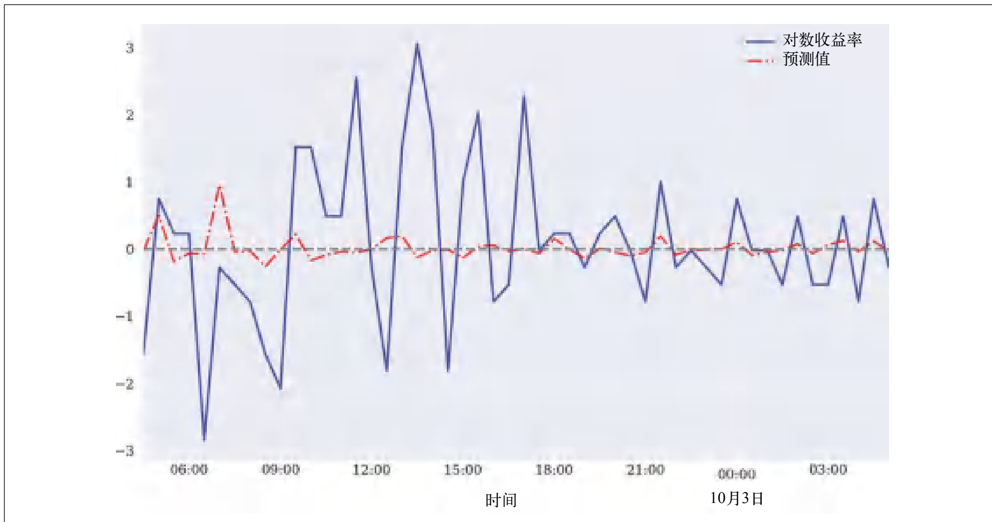


图 8-6：RNN 对金融收益率序列的样本内预测（数据子集）

虽然图 8-6 只提供了一个指示信息，但以相对较高的准确率得分验证了 RNN 在收益率序列上的预测表现可能比在价格序列上的预测表现效果更好的假设。

```
In [66]: from sklearn.metrics import accuracy_score

In [67]: accuracy_score(np.sign(data['r']), np.sign(data['pred']))
Out[67]: 0.6806532093445226
```

然而，为了获得真实的效果，需要对数据进行训练 – 测试拆分。样本外的准确率得分没有样本内的准确率得分高，但对我们关心的问题来说这个准确率仍然很高。

```
In [68]: split = int(len(r) * 0.8)  ❶

In [69]: train = r[:split]  ❶

In [70]: test = r[split:]  ❶

In [71]: g = TimeseriesGenerator(train, train, length=lags, batch_size=5)  ❷

In [72]: set_seeds()
         model = create_rnn_model(hu=100)

In [73]: %%time
         model.fit_generator(g, epochs=100, steps_per_epoch=10, verbose=False)  ❷
         CPU times: user 5.67 s, sys: 1.09 s, total: 6.75 s
         Wall time: 2.95 s

Out[73]: <keras.callbacks.callbacks.History at 0x7f7ef5482dd0>

In [74]: g_ = TimeseriesGenerator(test, test, length=lags, batch_size=5)  ❸

In [75]: y = model.predict(g_)  ❸
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [76]: accuracy_score(np.sign(test[lags:]), np.sign(y))  ❸
Out[76]: 0.6708428246013668
```

❶ 将数据拆分为训练数据子集和测试数据子集。

❷ 在训练数据上拟合模型。

❸ 在测试数据上测试模型。

8.5 金融特征

RNN 的应用不仅限于原始价格或收益率数据，还可以包括附加特征以改进它的预测性能。以下 Python 代码向数据集中添加了典型的金融特征。

```
In [77]: data = generate_data()

In [78]: data['r'] = np.log(data / data.shift(1))

In [79]: window = 20
         data['mom'] = data['r'].rolling(window).mean()  ❶
         data['vol'] = data['r'].rolling(window).std()  ❷

In [80]: data.dropna(inplace=True)
```

❶ 增加时间序列**动量**特征。

❷ 增加滚动**波动率**特征。

8.5.1 估计

在估计任务中，样本外准确率可能会显著下降，这有些出人意料。换句话说，在这种特殊情况下添加金融特征并没有观察到任何改进。

```
In [81]: split = int(len(data) * 0.8)

In [82]: train = data.iloc[:split].copy()

In [83]: mu, std = train.mean(), train.std()  ❶

In [84]: train = (train - mu) / std  ❷

In [85]: test = data.iloc[split:].copy()

In [86]: test = (test - mu) / std  ❸

In [87]: g = TimeseriesGenerator(train.values, train['r'].values,
                                 length=lags, batch_size=5)  ❹

In [88]: set_seeds()
         model = create_rnn_model(hu=100, features=len(data.columns),
                                  layer='SimpleRNN')
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [89]: %%time
         model.fit_generator(g, epochs=100, steps_per_epoch=10,
                             verbose=False)  ❹
         CPU times: user 5.24 s, sys: 1.08 s, total: 6.32 s
         Wall time: 2.73 s

Out[89]: <keras.callbacks.callbacks.History at 0x7f7ef313c950>

In [90]: g_ = TimeseriesGenerator(test.values, test['r'].values,
                                  length=lags, batch_size=5)  ❺

In [91]: y = model.predict(g_).flatten()  ❺

In [92]: accuracy_score(np.sign(test['r'].iloc[lags:]), np.sign(y))  ❺
Out[92]: 0.37299771167048057
```

❶ 计算训练数据的一阶矩和二阶矩。

❷ 对训练数据应用高斯归一化。

❸ 对测试数据应用高斯归一化（基于来自训练数据的统计数据）。

❹ 在训练数据上拟合模型。

❺ 在测试数据上测试模型。

8.5.2 分类

迄今为止的分析都在使用 Keras 中的 RNN 模型进行**估计**，以预测金融工具价格的未来方向。我们所关心的问题可能会更好地直接转换为**分类**问题。以下 Python 代码会处理二进制标签数据并直接预测价格变动的方向。这次我们使用 LSTM 层，即使对于相对少量的隐藏单元和有限的几个训练轮数，样本外的准确率也相当高。该方法通过适当调整类权重来解决类别不平衡的问题。在这种情况下，预测准确率非常高，约为 65%。

```
In [93]: set_seeds()
         model = create_rnn_model(hu=50,
                     features=len(data.columns),
                     layer='LSTM',
                     algorithm='classification')  ❶

In [94]: train_y = np.where(train['r'] > 0, 1, 0)  ❷

In [95]: np.bincount(train_y)  ❸
Out[95]: array([2374, 1142])

In [96]: def cw(a):
             c0, c1 = np.bincount(a)
             w0 = (1 / c0) * (len(a)) / 2
             w1 = (1 / c1) * (len(a)) / 2
             return {0: w0, 1: w1}

In [97]: g = TimeseriesGenerator(train.values, train_y,
                                 length=lags, batch_size=5)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [98]: %%time
         model.fit_generator(g, epochs=5, steps_per_epoch=10,
                             verbose=False, class_weight=cw(train_y))
         CPU times: user 1.25 s, sys: 159 ms, total: 1.41 s
         Wall time: 947 ms

Out[98]: <keras.callbacks.callbacks.History at 0x7f7ef43baf90>

In [99]: test_y = np.where(test['r'] > 0, 1, 0)  ❹

In [100]: g_ = TimeseriesGenerator(test.values, test_y,
                                   length=lags, batch_size=5)

In [101]: y = np.where(model.predict(g_, batch_size=None) > 0.5, 1, 0).flatten()

In [102]: np.bincount(y)
Out[102]: array([492, 382])

In [103]: accuracy_score(test_y[lags:], y)
Out[103]: 0.6498855835240275
```

❶ 用于分类的 RNN 模型。

❷ 二元训练标签。

❸ 训练标签的组频率。

❹ 二进制测试标签。

8.5.3 深度RNN

最后，本章将演示深度 RNN，它是具有多个隐藏层的 RNN。如同创造深度 DNN 一样简单，唯一的要求是对于非最终隐藏层，参数 `return_sequences` 需要被设置为 `True`。以下用于创建深度 RNN 的 Python 函数还添加了 `Dropout` 层以避免潜在的过拟合，预测准确率与上一节中的预测准确率相当。

```
In [104]: from keras.layers import Dropout

In [105]: def create_deep_rnn_model(hl=2, hu=100, layer='SimpleRNN',
                                    optimizer='rmsprop', features=1,
                                    dropout=False, rate=0.3, seed=100):
              if hl <= 2: hl = 2  ❶
              if layer == 'SimpleRNN':
                  layer = SimpleRNN
              else:
                  layer = LSTM
              model = Sequential()
              model.add(layer(hu, input_shape=(lags, features),
                              return_sequences=True,
                             ))  ❷
              if dropout:
                  model.add(Dropout(rate, seed=seed))  ❸
              for _ in range(2, hl):
                  model.add(layer(hu, return_sequences=True))
                  if dropout:
                      model.add(Dropout(rate, seed=seed))  ❸
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            model.add(layer(hu))  ❹
            model.add(Dense(1, activation='sigmoid'))  ❺
            model.compile(optimizer=optimizer,
                          loss='binary_crossentropy',
                          metrics=['accuracy'])
            return model

In [106]: set_seeds()
          model = create_deep_rnn_model(
                      hl=2, hu=50, layer='SimpleRNN',
                      features=len(data.columns),
                      dropout=True, rate=0.3)  ❶

In [107]: %%time
          model.fit_generator(g, epochs=200, steps_per_epoch=10,
                              verbose=False, class_weight=cw(train_y))
          CPU times: user 14.2 s, sys: 2.85 s, total: 17.1 s
          Wall time: 7.09 s

Out[107]: <keras.callbacks.callbacks.History at 0x7f7ef6428790>

In [108]: y = np.where(model.predict(g_, batch_size=None) > 0.5, 1, 0).flatten()

In [109]: np.bincount(y)
Out[109]: array([550, 324])

In [110]: accuracy_score(test_y[lags:], y)
Out[110]: 0.6430205949656751
```

❶ 保证最少有两个隐藏层。

❷ 第一个隐藏层。

❸ Dropout 层。

❹ 最终隐藏层。

❺ 建立分类模型。

8.6 结论

本章基于 Keras 库介绍了 RNN 模型，并说明了这种神经网络在金融时间序列数据中的应用。从 Python 语言的角度看，使用 RNN 与使用 DNN 并无太大区别。一个主要区别是训练数据和测试数据必须以序列形式呈现给各自的方法，但是通过应用 TimeseriesGenerator 函数可以轻松实现这一点，该函数使用的生成器对象会将序列数据转换为 Keras 中的 RNN 能处理的数据。

本章中的示例适用于金融价格序列和金融收益率序列。此外，我们还可以轻松添加金融特征，比如时间序列动量。为模型创建提供的函数可以使用 SimpleRNN 层或 LSTM 层以及不同的优化器等，还可以在浅层神经网络和深层神经网络的背景下对估计和分类问题进行建模。

在预测市场方向时，分类示例得到的样本外预测准确率相对较高，但对估计示例而言并没有那么高，甚至可能很低。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 9 章
强化学习

像人类一样，我们的智能体能够自己学习以实现成功的策略，从而获得最大的长期回报。这种通过完全基于自奖励或惩罚的试错学习的范式被称为强化学习。[1]

——DeepMind，2016 年

第 7 章和第 8 章中应用的学习算法属于**监督学习**的范畴，这些方法要求有一个包含特征和标签的数据集，此数据集能够支持这些算法学习特征和标签之间的关系，从而在估计或分类任务中取得成功。正如第 1 章中的简单示例所示，**强化学习**（reinforcement learning，RL）的工作方式与此不同。它不需要预先给出一个全面的特征和标签数据集，数据是由学习智能体在与感兴趣的环境交互时生成的。本章详细介绍了强化学习，说明了其基本概念，以及该领域最流行的算法之一：**Q 学习**。神经网络不会被强化学习算法取代，在本章的讨论范畴内它们通常也发挥着重要作用。

9.1 节解释了强化学习中的基本概念，比如环境、状态和智能体。9.2 节介绍了强化学习环境的 OpenAI Gym 套件，其中是以 `CartPole` 环境作为示例的。第 2 章简要介绍和讨论过在这种环境中，智能体必须学习如何通过将推车向左或向右移动来平衡推车上的杆子。9.3 节展示了如何使用降维和蒙特卡罗模拟来解决 `CartPole` 问题。DNN 等标准监督学习算法通常不适合解决像 `CartPole` 这样的问题，因为它们缺乏延迟奖励的概念，9.4 节会对此进行说明。9.5 节讨论了 DQL 智能体，该智能体明确地考虑了延迟奖励并能够解决 `CartPole` 问题。9.6 节将相同的智能体应用于简单的金融市场环境，尽管智能体在这一场景中表现不佳，但该示例表明 DQL 智能体也可以学习交易并成为通常所说的**交易机器人**。为了改进 DQL 智能体的学习，9.7 节提出了一个改进的金融市场环境，除其他好处外，它允许使用一种以上的特征来描述环境状态。基于这种改进的环境，9.8 节引入并应用了一种改进的 Q 学习金融智能体，作为交易机器人，它的表现更加出色。

注 1：参见“Deep Reinforcement Learning”。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

9.1 基本概念

本节简要概述了强化学习中的基本概念，包括以下几种。

环境

环境定义了当前的问题，可以是要玩的计算机游戏或要进行交易的金融市场。

状态

状态包含描述环境当前状态的所有相关参数。在计算机游戏中，这可能是整个屏幕及其像素。在金融市场中，这可能包括当前和历史价格水平或金融指标，比如移动平均线、宏观经济变量等。

智能体

智能体这个词包含了与环境交互并从这些交互中学习的强化学习算法的所有元素。在游戏环境中，智能体可能代表玩游戏的玩家。在金融环境中，智能体可以代表在市场中进行交易的交易者。

动作

智能体可以从一组（有限的）被允许的动作中选择一个**动作**。在计算机游戏中，被允许的动作可能是向左或向右移动，而在金融市场中，被允许的动作可能是做多或做空。

步骤

给定智能体的动作，环境状态会被更新，这样的更新通常被称为一个**步骤**。步骤的概念可以包含两个步骤之间的相同或者不同的时间间隔。虽然在计算机游戏中，与游戏环境的实时交互是通过相当短且相同的时间间隔（“游戏时钟”）来模拟的，但诸如与金融市场环境交互的交易机器人则可以在更长且不同的时间间隔内采取动作。

奖励

根据智能体选择的动作，对其实行**奖励**（或惩罚）。对于计算机游戏，积分是一种典型的奖励。在金融环境中，利润（或亏损）是一种标准的奖励（或惩罚）。

目标

目标是指智能体试图最大化的内容。在计算机游戏中，这通常是智能体达到的分数。对于金融交易机器人，这可能是累积的交易利润。

策略

策略定义了智能体在给定环境状态下所采取的动作。给定计算机游戏的特定状态（由构成当前场景的所有像素表示），策略可能会指定智能体选择“向右移动”作为动作。观察到连续 3 个价格上涨的交易机器人可能会根据其策略决定做空市场。

回合

回合是从环境的初始状态到成功或可预见的失败的一组步骤。在游戏中，这是从游戏开始到输或赢为止。以金融界为例，这就是从年初到年底，或者到破产。

Sutton 和 Barto（2018）介绍了强化学习这一领域，该书详细讨论了上述概念，并在大量具体示例的基础上对它们进行了说明。接下来的几节再次选择了一种实用的、以应用为导向的强化学习方法，所讨论的示例会基于 Python 代码进一步说明上述所有概念。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

9.2 OpenAI Gym

在第 2 章介绍的大多数成功案例中，强化学习起着主导作用，这激发了人们对强化学习算法的广泛兴趣。OpenAI 是一个致力于促进人工智能研究，特别是强化学习研究的组织。OpenAI 开发并开源了一套环境，称为 `OpenAI Gym`，允许通过标准化 API 训练 RL 智能体。

在众多环境中，有模拟经典强化学习问题的 `CartPole` 环境（或游戏），即把一根杆子直立在推车上，目的是通过左右移动推车来学习平衡杆子的策略。环境状态由 4 个参数表示，包括以下物理测量值：推车位置、推车速度、极角和极角速度（尖端）。图 9-1 描述了一个可视化的环境。

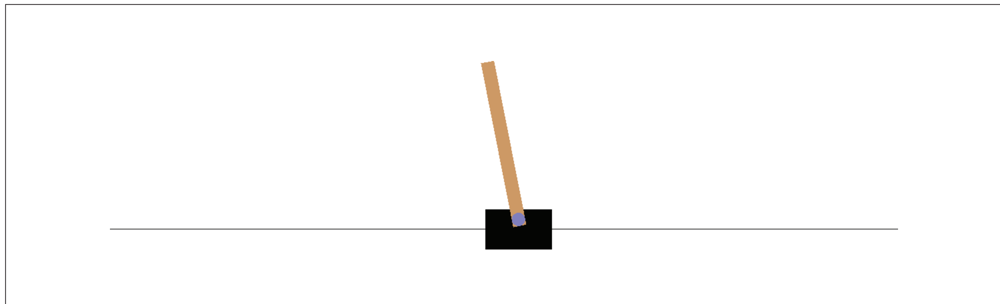


图 9-1：OpenAI Gym 的 `CartPole` 环境

考虑以下 Python 代码，该代码实例化了 `CartPole` 的环境对象并检查了**观察空间**。观察空间是环境状态的模型。

```
In [1]: import os
        import math
        import random
        import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        np.set_printoptions(precision=4, suppress=True)
        os.environ['PYTHONHASHSEED'] = '0'

In [2]: import gym

In [3]: env = gym.make('CartPole-v0')  ❶

In [4]: env.seed(100)  ❶
        env.action_space.seed(100)  ❶
Out[4]: [100]

In [5]: env.observation_space  ❷
Out[5]: Box(4,)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [6]: env.observation_space.low.astype(np.float16)  ❷
Out[6]: array([-4.8  ,   -inf, -0.419,   -inf], dtype=float16)

In [7]: env.observation_space.high.astype(np.float16)  ❷
Out[7]: array([4.8  ,   inf, 0.419,   inf], dtype=float16)

In [8]: state = env.reset()  ❸

In [9]: state  ❹
Out[9]: array([-0.0163, 0.0238, -0.0392, -0.0148])
```

❶ 具有固定种子值的环境对象。

❷ 具有最小值和最大值的观察空间。

❸ 环境重置。

❹ 初始状态：推车位置、推车速度、极角和极角速度。

在以下环境中，允许的动作由**动作空间**来描述，在这种情况下有两个**动作空间**，分别用 `0`（向左推车）和 `1`（向右推车）来表示。

```
In [10]: env.action_space  ❶
Out[10]: Discrete(2)

In [11]: env.action_space.n  ❶
Out[11]: 2

In [12]: env.action_space.sample()  ❷
Out[12]: 1

In [13]: env.action_space.sample()  ❷
Out[13]: 0

In [14]: a = env.action_space.sample()  ❷
         a  ❷
Out[14]: 1

In [15]: state, reward, done, info = env.step(a)  ❸
         state, reward, done, info  ❹
Out[15]: (array([-0.0158, 0.2195, -0.0395, -0.3196]), 1.0, False, {})
```

❶ 动作空间。

❷ 从动作空间中采样的随机动作。

❸ 基于随机动作向前迈进。

❹ 环境的新状态、奖励、成功 / 失败以及附加信息。

只要 `done=False`，智能体就还在游戏中，并且可以选择另一个动作。当智能体达到总计 200 步或总奖励达到 200（每步奖励 1.0）时，即取得成功。当推车上的杆子到达一定角度导致杆子从推车上掉下来时，故障被观测到，在这种情况下，返回 `done=True`。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

简单智能体是遵循完全随机策略的智能体：无论观察到什么状态，智能体都会选择一个随机动作，如下面的代码实现所示。在这种情况下，智能体可以走的步数仅取决于它的幸运程度，其并未以更新策略的形式进行学习。

```
In [16]: env.reset()
         for e in range(1, 200):
             a = env.action_space.sample()  ❶
             state, reward, done, info = env.step(a)  ❷
             print(f'step={e:2d} | state={state} | action={a} | reward={reward}')
             if done and (e + 1) < 200:  ❸
                 print('*** FAILED ***')  ❸
                 break
         step= 1 | state=[-0.0423  0.1982  0.0256 -0.2476] | action=1 | reward=1.0
         step= 2 | state=[-0.0383  0.0028  0.0206  0.0531] | action=0 | reward=1.0
         step= 3 | state=[-0.0383  0.1976  0.0217 -0.2331] | action=1 | reward=1.0
         step= 4 | state=[-0.0343  0.0022  0.017   0.0664] | action=0 | reward=1.0
         step= 5 | state=[-0.0343  0.197   0.0184 -0.2209] | action=1 | reward=1.0
         step= 6 | state=[-0.0304  0.0016  0.0139  0.0775] | action=0 | reward=1.0
         step= 7 | state=[-0.0303  0.1966  0.0155 -0.2107] | action=1 | reward=1.0
         step= 8 | state=[-0.0264  0.0012  0.0113  0.0868] | action=0 | reward=1.0
         step= 9 | state=[-0.0264  0.1962  0.013  -0.2023] | action=1 | reward=1.0
         step=10 | state=[-0.0224  0.3911  0.009  -0.4908] | action=1 | reward=1.0
         step=11 | state=[-0.0146  0.5861 -0.0009 -0.7807] | action=1 | reward=1.0
         step=12 | state=[-0.0029  0.7812 -0.0165 -1.0736] | action=1 | reward=1.0
         step=13 | state=[ 0.0127  0.9766 -0.0379 -1.3714] | action=1 | reward=1.0
         step=14 | state=[ 0.0323  1.1722 -0.0654 -1.6758] | action=1 | reward=1.0
         step=15 | state=[ 0.0557  0.9779 -0.0989 -1.4041] | action=0 | reward=1.0
         step=16 | state=[ 0.0753  0.7841 -0.127  -1.1439] | action=0 | reward=1.0
         step=17 | state=[ 0.0909  0.5908 -0.1498 -0.8936] | action=0 | reward=1.0
         step=18 | state=[ 0.1028  0.7876 -0.1677 -1.2294] | action=1 | reward=1.0
         step=19 | state=[ 0.1185  0.9845 -0.1923 -1.5696] | action=1 | reward=1.0
         step=20 | state=[ 0.1382  0.7921 -0.2237 -1.3425] | action=0 | reward=1.0
         *** FAILED ***

In [17]: done
Out[17]: True
```

❶ 随机动作策略。

❷ 向前一步。

❸ 少于 200 步则失败。

通过交互获得数据

在监督学习中，假设训练数据集、验证数据集和测试数据集在训练开始之前已经存在，而在强化学习中，智能体通过与环境交互来生成自己的数据。在许多情况下（比如在游戏中），这是一个巨大的简化。考虑一下国际象棋游戏：一个 RL 智能体可以通过与另一个国际象棋引擎或另一个版本的自身对战，而不是将数千个人类历史上所生成的国际象棋棋谱加载到计算机中。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

9.3 蒙特卡罗智能体

CartPole 问题不一定非要使用成熟的强化学习方法和一些神经网络来解决，本节介绍了基于蒙特卡罗模拟的问题的简单解决方案，并使用了**降维**的特定策略。在这种情况下，定义环境状态的 4 个参数通过线性组合被压缩为了单个实值参数。[2] 以下 Python 代码实现了这个想法。

```
In [18]: np.random.seed(100)  ❶

In [19]: weights = np.random.random(4) * 2 - 1  ❶

In [20]: weights  ❶
Out[20]: array([ 0.0868, -0.4433, -0.151 ,  0.6896])

In [21]: state = env.reset()  ❷

In [22]: state  ❷
Out[22]: array([-0.0347, -0.0103,  0.047 , -0.0315])

In [23]: s = np.dot(state, weights)  ❸
         s  ❸
Out[23]: -0.02725361929630797
```

❶ 固定种子值的随机权重。

❷ 环境的初始状态。

❸ 状态和权重的点积。

然后根据单个状态参数 s 的符号来定义策略。

```
In [24]: if s < 0:
             a = 0
         else:
             a = 1

In [25]: a
Out[25]: 0
```

接下来可以使用此策略玩一回合 CartPole 游戏。鉴于所应用的权重的随机性，通常结果并不比上一节的随机动作策略的结果好。

```
In [26]: def run_episode(env, weights):
             state = env.reset()
             treward = 0
             for _ in range(200):
                 s = np.dot(state, weights)
                 a = 0 if s < 0 else 1
                 state, reward, done, info = env.step(a)
                 treward += reward
```

注 2：例如，参见博客文章“Simple reinforcement learning methods to learn CartPole”。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            if done:
                break
        return treward

In [27]: run_episode(env, weights)
Out[27]: 41.0
```

因此，可以应用蒙特卡罗模拟来测试大量不同的权重。下面的代码模拟了大量的权重，检查它们是成功还是失败，然后选择产生成功的权重。

```
In [28]: def set_seeds(seed=100):
             random.seed(seed)
             np.random.seed(seed)
             env.seed(seed)

In [29]: set_seeds()
         num_episodes = 1000

In [30]: besttreward = 0
         for e in range(1, num_episodes + 1):
             weights = np.random.rand(4) * 2 - 1  ❶
             treward = run_episode(env, weights)  ❷
             if treward > besttreward:  ❸
                 besttreward = treward  ❹
                 bestweights = weights  ❺
                 if treward == 200:
                     print(f'SUCCESS | episode={e}')
                     break
                 print(f'UPDATE  | episode={e}')
         UPDATE  | episode=1
         UPDATE  | episode=2
         SUCCESS | episode=13

In [31]: weights
Out[31]: array([-0.4282,  0.7048,  0.95  ,  0.7697])
```

❶ 随机权重。

❷ 这些权重的总奖励。

❸ 观察是否有改善。

❹ 替换最佳总奖励。

❺ 替换最佳权重。

如果连续 100 回合的平均总奖励为 195 或更高，则认为 `CartPole` 问题已被智能体解决，如以下代码所示。

```
In [32]: res = []
         for _ in range(100):
             treward = run_episode(env, weights)
             res.append(treward)
         res[:10]
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
Out[32]: [200.0, 200.0, 200.0, 200.0, 200.0, 200.0, 200.0, 200.0, 200.0, 200.0]

In [33]: sum(res) / len(res)
Out[33]: 200.0
```

这是一个用来与其他更复杂的方法做比较的强有力的基准。

9.4　神经网络智能体

CartPole 游戏也可以被转换为一个分类问题：环境状态由 4 个特征值组成，对应每组给定特征值的正确操作是需要预测的标签。通过与环境交互，神经网络智能体可以收集由特征值和标签组合组成的数据集。给定这个不断增长的数据集，可以训练神经网络来学习给定环境状态的正确动作。在这种情况下，神经网络代表策略，智能体会根据新体验更新策略。

首先是一些库的引用。

```
In [34]: import tensorflow as tf
         from keras.layers import Dense, Dropout
         from keras.models import Sequential
         from keras.optimizers import Adam, RMSprop
         from sklearn.metrics import accuracy_score
         Using TensorFlow backend.

In [35]: def set_seeds(seed=100):
             random.seed(seed)
             np.random.seed(seed)
             tf.random.set_seed(seed)
             env.seed(seed)
             env.action_space.seed(100)
```

然后是一个 NNAgent 类，它结合了智能体的主要元素：策略的神经网络模型、根据策略选择动作、更新策略（训练神经网络），以及多个回合的学习过程本身。智能体同时使用**探索**和**利用**来选择一个动作。探索是指随机动作，独立于当前策略。利用是指从当前策略派生的动作。这么做是因为某种程度的探索可以确保获得更丰富的经验，从而使智能体的学习得到改善。

```
In [36]: class NNAgent:
             def __init__(self):
                 self.max = 0  ❶
                 self.scores = list()
                 self.memory = list()
                 self.model = self._build_model()

             def _build_model(self):  ❷
                 model = Sequential()
                 model.add(Dense(24, input_dim=4,
                                 activation='relu'))
                 model.add(Dense(1, activation='sigmoid'))
                 model.compile(loss='binary_crossentropy',
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
                      optimizer=RMSprop(lr=0.001))
        return model

    def act(self, state):  ❸
        if random.random() <= 0.5:
            return env.action_space.sample()
        action = np.where(self.model.predict(
            state, batch_size=None)[0, 0] > 0.5, 1, 0)
        return action

    def train_model(self, state, action):  ❹
        self.model.fit(state, np.array([action,]),
                       epochs=1, verbose=False)

    def learn(self, episodes):  ❺
        for e in range(1, episodes + 1):
            state = env.reset()
            for _ in range(201):
                state = np.reshape(state, [1, 4])
                action = self.act(state)
                next_state, reward, done, info = env.step(action)
                if done:
                    score = _ + 1
                    self.scores.append(score)
                    self.max = max(score, self.max)  ❶
                    print('episode: {:4d}/{} | score: {:3d} | max: {:3d}'
                          .format(e, episodes, score, self.max), end='\r')
                    break
                self.memory.append((state, action))
                self.train_model(state, action)  ❹
                state = next_state
```

❶ 最高总奖励。

❷ 策略的 DNN 分类模型。

❸ 选择动作的方法（探索和利用）。

❹ 更新策略的方法（训练神经网络）。

❺ 从与环境的交互中学习的方法。

神经网络智能体没有解决上述配置下的问题，最大总奖励甚至一次也没有达到 200。

```
In [37]: set_seeds(100)
         agent = NNAgent()

In [38]: episodes = 500

In [39]: agent.learn(episodes)
         episode:  500/500 | score:  11 | max:  44
In [40]: sum(agent.scores) / len(agent.scores)  ❶
Out[40]: 13.682
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❶ 所有回合的平均总奖励。

这种方法似乎缺少一些东西，而其中一个主要的缺失元素是超越当前状态和要选择的动作的想法。目前为止所实现的方法无论如何都没有考虑到只有当智能体连续存活 200 步时才能取得成功。简单地说，智能体会避免采取错误的动作，但没有学会赢得比赛。

分析收集到的状态（特征）和动作（标签）的历史数据表明，神经网络的准确率达到了 75% 左右。

然而，这并没有转化为之前看到的获胜策略。

```
In [41]: f = np.array([m[0][0] for m in agent.memory])  ❶
         f  ❶
Out[41]: array([[-0.0163,  0.0238, -0.0392, -0.0148],
                [-0.0158,  0.2195, -0.0395, -0.3196],
                [-0.0114,  0.0249, -0.0459, -0.0396],
                ...,
                [ 0.0603,  0.9682, -0.0852, -1.4595],
                [ 0.0797,  1.1642, -0.1144, -1.7776],
                [ 0.103 ,  1.3604, -0.15  , -2.1035]])

In [42]: l = np.array([m[1] for m in agent.memory])  ❷
         l  ❷
Out[42]: array([1, 0, 1, ..., 1, 1, 1])

In [43]: accuracy_score(np.where(agent.model.predict(f) > 0.5, 1, 0), l)
Out[43]: 0.7525626872733008
```

❶ 所有回合的特征（状态）。

❷ 所有回合的标签（动作）。

9.5 DQL智能体

Q 学习是一种强化学习算法，除了会考虑来自动作的即时奖励，还会考虑延迟奖励。该算法归功于 Watkins（1989）以及 Watkins 和 Dayan（1992），并在 Sutton 和 Barto（2018）的第 6 章中有详细解释。Q 学习解决了神经网络智能体遇到的超越下一个立即奖励的问题。

该算法的工作原理大致如下：有一个**动作 – 价值**策略 Q，它为每个状态和动作的组合分配一个值。值越高，从智能体的角度来看动作越好。如果智能体使用策略 Q 选择一个动作，则它会选择具有最高值的动作。

那么一个动作的价值是如何得出的呢？一个动作的价值由它的**直接奖励**和下一状态下最优动作的**折现值**组成，以下是正式表达。

$$Q(S_t, A_t) = R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$$

这里，S_t 是步骤（时间）t 的状态，A_t 是状态 S_t 采取的动作，R_{t+1} 是动作 A_t 的直接奖励，$0 < \gamma < 1$ 是折扣因子，$\max_a Q(S_{t+1}, a)$ 是给定当前策略 Q 的最优动作下的最大延迟奖励。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

在一个只有有限数量的可能状态的简单环境中，Q 可以用一张**表格**表示，其中列出了每个状态 – 动作组合的相应值。然而，在更有趣或更复杂的设置中（如 CartPole 环境），状态数量过多，无法通过穷尽的方式写出 Q，因此，Q 通常被理解为一个**函数**。

这就是神经网络发挥作用的地方。在现实的设置和环境中，函数 Q 的封闭形式可能不存在，或者利用动态规划方法很难推导。因此 Q 学习算法通常仅以**近似值**为目标。神经网络具有通用逼近能力，是完成 Q 函数逼近的自然选择。

Q 学习的另一个关键要素是**重放**。DQL 智能体会重放许多经验（状态 – 动作组合）以定期更新策略函数 Q，这可以大大提高学习效果。此外，下面介绍的 DQL 智能体（DQLAgent）于学习过程中在探索和利用之间交替。交替会以系统的方式进行，因为智能体仅从探索开始（一开始它不可能学到任何东西），然后会缓慢但稳定地降低探索率 ϵ 直到达到最低水平。[3]

```
In [44]: from collections import deque
         from keras.optimizers import Adam, RMSprop

In [45]: class DQLAgent:
             def __init__(self, gamma=0.95, hu=24, opt=Adam,
                    lr=0.001, finish=False):
                 self.finish = finish
                 self.epsilon = 1.0  ❶
                 self.epsilon_min = 0.01  ❷
                 self.epsilon_decay = 0.995  ❸
                 self.gamma = gamma  ❹
                 self.batch_size = 32  ❺
                 self.max_treward = 0
                 self.averages = list()
                 self.memory = deque(maxlen=2000)  ❻
                 self.osn = env.observation_space.shape[0]
                 self.model = self._build_model(hu, opt, lr)

             def _build_model(self, hu, opt, lr):
                 model = Sequential()
                 model.add(Dense(hu, input_dim=self.osn,
                                 activation='relu'))
                 model.add(Dense(hu, activation='relu'))
                 model.add(Dense(env.action_space.n, activation='linear'))
                 model.compile(loss='mse', optimizer=opt(lr=lr))
                 return model

             def act(self, state):
                 if random.random() <= self.epsilon:
                     return env.action_space.sample()
                 action = self.model.predict(state)[0]
                 return np.argmax(action)
```

注 3：实现类似于博客文章“Deep Q-Learning with Keras and Gym”中的发现。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
    def replay(self):
        batch = random.sample(self.memory, self.batch_size)  ❼
        for state, action, reward, next_state, done in batch:
            if not done:
                reward += self.gamma * np.amax(
                    self.model.predict(next_state)[0])  ❽
            target = self.model.predict(state)
            target[0, action] = reward
            self.model.fit(state, target, epochs=1,
                           verbose=False)  ❾
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay  ❿

    def learn(self, episodes):
        trewards = []
        for e in range(1, episodes + 1):
            state = env.reset()
            state = np.reshape(state, [1, self.osn])
            for _ in range(5000):
                action = self.act(state)
                next_state, reward, done, info = env.step(action)
                next_state = np.reshape(next_state,
                                        [1, self.osn])
                self.memory.append([state, action, reward,
                                    next_state, done])  ⓫
                state = next_state
                if done:
                    treward = _ + 1
                    trewards.append(treward)
                    av = sum(trewards[-25:]) / 25
                    self.averages.append(av)
                    self.max_treward = max(self.max_treward, treward)
                    templ = 'episode: {:4d}/{} | treward: {:4d} | '
                    templ += 'av: {:6.1f} | max: {:4d}'
                    print(templ.format(e, episodes, treward, av,
                                       self.max_treward), end='\r')
                    break
            if av > 195 and self.finish:
                break
            if len(self.memory) > self.batch_size:
                self.replay()  ⓬
    def test(self, episodes):
        trewards = []
        for e in range(1, episodes + 1):
            state = env.reset()
        for _ in range(5001):
            state = np.reshape(state, [1, self.osn])
            action = np.argmax(self.model.predict(state)[0])
            next_state, reward, done, info = env.step(action)
            state = next_state
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
                if done:
                    treward = _ + 1
                    trewards.append(treward)
                    print('episode: {:4d}/{} | treward: {:4d}'
                          .format(e, episodes, treward), end='\r')
                    break
        return trewards
```

❶ 初始探索率。

❷ 最小探索率。

❸ 探索率的衰减率。

❹ 延迟奖励的折扣因子。

❺ 重放的批次大小。

❻ 有限历史的双端队列 deque 集合。

❼ 随机选择历史批次进行回放。

❽ 状态 – 动作对的 Q 值。

❾ 为新的动作 – 价值对更新神经网络。

❿ 更新探索率。

⓫ 存储新数据。

⓬ 根据过去的经验重放以更新策略。

DQL 智能体如何执行？如以下代码所示，它达到了 CartPole 的获胜状态，总奖励为 200。图 9-2 显示了分数的移动平均线以及它是如何随时间增加的，尽管不是单调增加。相反，智能体的性能有时会显著下降。除此之外，一直在进行的探索会导致随机操作，这不一定会导致总奖励方面的良好结果，但可能会为更新策略网络带来有益的经验。

```
In [46]: episodes = 1000

In [47]: set_seeds(100)
         agent = DQLAgent(finish=True)

In [48]: agent.learn(episodes)
         episode:  400/1000 | treward:  200 | av:  195.4 | max:  200

In [49]: plt.figure(figsize=(10, 6))
         x = range(len(agent.averages))
         y = np.polyval(np.polyfit(x, agent.averages, deg=3), x)
         plt.plot(agent.averages, label='moving average')
         plt.plot(x, y, 'r--', label='trend')
         plt.xlabel('episodes')
         plt.ylabel('total reward')
         plt.legend();
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

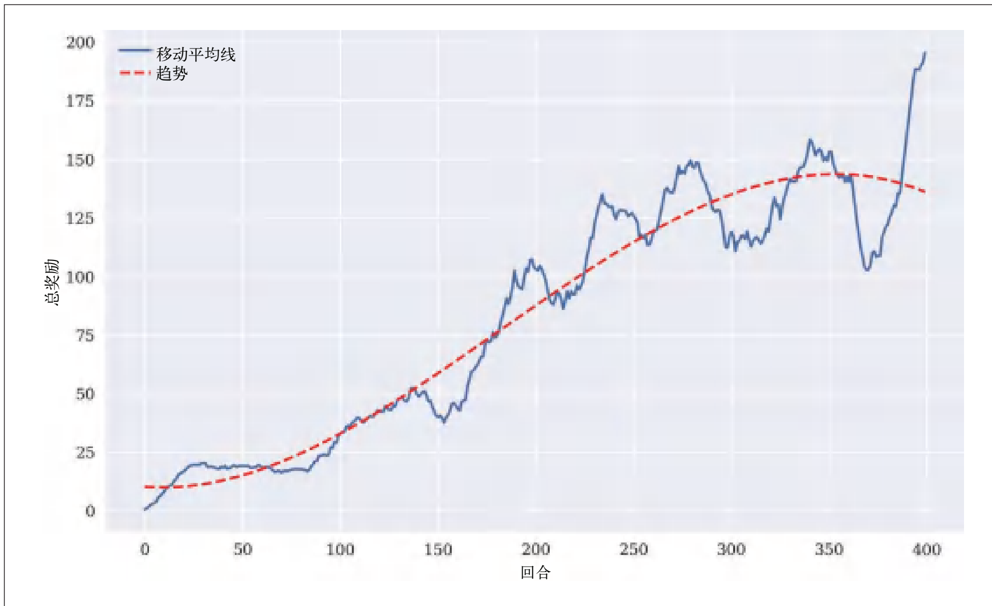


图 9-2：CartPole 的 DQLAgent 平均总奖励

DQL 智能体是否解决了 CartPole 问题？在现有的游戏设置下，考虑到 OpenAI Gym 对成功的定义，确实解决了。

```
In [50]: trewards = agent.test(100)
         episode:  100/100 | treward:  200

In [51]: sum(trewards) / len(trewards)
Out[51]: 200.0
```

9.6　简单的金融沙箱

为了将 Q 学习方法迁移至金融领域，本节展示了一个模拟 OpenAI Gym 环境的类，但其仅适用于以金融时间序列数据为代表的金融市场。这里的思路是，类似于 CartPole 环境，4 个历史价格代表金融市场的状态，当给定一个状态时，智能体可以决定是做多还是做空。在这种情况下，两个环境是可比较的，因为每一个状态都是由 4 个参数给出，并且智能体可以采取两种不同的动作。

为了模仿 OpenAI Gym API，需要两个辅助类，一个用于观察空间，一个用于动作空间。

```
In [52]: class observation_space:
             def __init__(self, n):
                 self.shape = (n,)

In [53]: class action_space:
             def __init__(self, n):
                 self.n = n
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            def seed(self, seed):
                pass
            def sample(self):
                return random.randint(0, self.n - 1)
```

以下 Python 代码定义了 Finance 类，该类提取了许多交易品种的日终历史价格。该类的主要方法是 .reset() 和 .step()，其中，.step() 方法会检查是否采取了正确的动作，相应地定义奖励，并检查成功或失败。当智能体能够通过整个数据集正确交易时，就取得了成功。当然，这可以有不同的定义（例如，定义当智能体成功交易 1000 步时就取得了成功）。失败被定义为准确率小于 50%（总奖励除以总步数）。然而，这仅在一定数量的步骤之后进行检查，以避免该指标的初始方差过高。

```
In [54]: class Finance:
             url = 'http://hilpisch.com/aiif_eikon_eod_data.csv'
             def __init__(self, symbol, features):
                 self.symbol = symbol
                 self.features = features
                 self.observation_space = observation_space(4)
                 self.osn = self.observation_space.shape[0]
                 self.action_space = action_space(2)
                 self.min_accuracy = 0.475  ❶
                 self._get_data()
                 self._prepare_data()
             def _get_data(self):
                 self.raw = pd.read_csv(self.url, index_col=0,
                                        parse_dates=True).dropna()
             def _prepare_data(self):
                 self.data = pd.DataFrame(self.raw[self.symbol])
                 self.data['r'] = np.log(self.data / self.data.shift(1))
                 self.data.dropna(inplace=True)
                 self.data = (self.data - self.data.mean()) / self.data.std()
                 self.data['d'] = np.where(self.data['r'] > 0, 1, 0)
             def _get_state(self):
                 return self.data[self.features].iloc[
                     self.bar - self.osn:self.bar].values  ❷
             def seed(self, seed=None):
                 pass
             def reset(self):  ❸
                 self.treward = 0
                 self.accuracy = 0
                 self.bar = self.osn
                 state = self.data[self.features].iloc[
                     self.bar - self.osn:self.bar]
                 return state.values
             def step(self, action):
                 correct = action == self.data['d'].iloc[self.bar]  ❹
                 reward = 1 if correct else 0  ❺
                 self.treward += reward  ❻
                 self.bar += 1  ❼
                 self.accuracy = self.treward / (self.bar - self.osn)  ❽
                 if self.bar >= len(self.data):  ❾
                     done = True
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            elif reward == 1:  ❿
                done = False
            elif (self.accuracy < self.min_accuracy and
                  self.bar > self.osn + 10):  ⓫
                done = True
            else:  ⓬
                done = False
            state = self._get_state()
            info = {}
            return state, reward, done, info
```

❶ 定义所需的最低准确率。

❷ 选择用于定义金融市场状态的数据。

❸ 将环境重置为其初始值。

❹ 检查智能体是否选择了正确的动作（成功交易）。

❺ 定义智能体收到的奖励。

❻ 将奖励添加到总奖励中。

❼ 使环境向前运行一步。

❽ 计算给定所有步骤（交易）的成功的动作（交易）的准确率。

❾ 如果智能体到达数据集的末尾，则成功。

❿ 如果智能体采取了正确的动作，那么它可以继续前进。

⓫ 如果在一些初始步骤之后，准确率下降到最低水平以下，则该回合结束（失败）。

⓬ 对于其余情况，智能体可以继续前进。

`Finance` 类的实例表现得很像 OpenAI Gym 的环境。特别是，在这种基本情况下，该实例表现得与 `CartPole` 环境完全一样。

```
In [55]: env = Finance('EUR=', 'EUR=')  ❶

In [56]: env.reset()
Out[56]: array([1.819 , 1.8579, 1.7749, 1.8579])

In [57]: a = env.action_space.sample()
         a
Out[57]: 0

In [58]: env.step(a)
Out[58]: (array([1.8579, 1.7749, 1.8579, 1.947 ]), 0, False, {})
```

❶ 指定用于定义代表状态数据的交易标的代号和特征类型（交易标的代号或对数收益率）。

为 `CartPole` 游戏开发的 `DQLAgent` 能否学习在金融市场中进行交易？是的，它能，如下面的代码所示。然而，尽管智能体在训练回合中提高了其交易技能（平均而言），但结果并不“惊艳”（参见图 9-3）。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [59]: set_seeds(100)
         agent = DQLAgent(gamma=0.5, opt=RMSprop)

In [60]: episodes = 1000

In [61]: agent.learn(episodes)
         episode: 1000/1000 | treward: 2511 | av: 1012.7 | max: 2511
In [62]: agent.test(3)
         episode:    3/3 | treward: 2511
Out[62]: [2511, 2511, 2511]

In [63]: plt.figure(figsize=(10, 6))
         x = range(len(agent.averages))
         y = np.polyval(np.polyfit(x, agent.averages, deg=3), x)
         plt.plot(agent.averages, label='moving average')
         plt.plot(x, y, 'r--', label='regression')
         plt.xlabel('episodes')
         plt.ylabel('total reward')
         plt.legend();
```

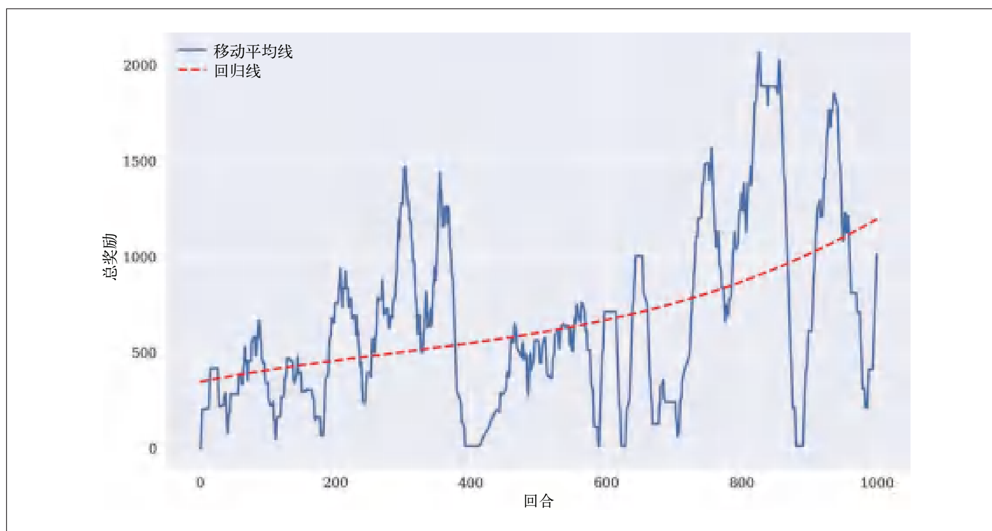


图 9-3：运行于 Finance 环境的 DQLAgent 的平均总奖励

通用 RL 智能体

本节为金融市场环境提供了一个类，该类模拟了 OpenAI Gym 环境的 API。它还可以将 DQL 智能体应用于新的金融市场环境，无须对智能体本身进行任何更改。尽管智能体在这种新环境中的表现可能并不“惊艳”，但它说明了本章介绍的强化学习方法是相当通用的。RL 智能体通常可以从与它们交互的不同环境中学习，这在一定程度上解释了为什么第 2 章所述的 DeepMind 的 AlphaZero 不仅能够掌握围棋，还能够掌握国际象棋和将棋。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

9.7 更好的金融沙箱

上一节的目的是开发一个简单的类，该类允许在金融市场环境中进行强化学习。本节的主要目标是复制 OpenAI Gym 环境的 API。然而，没有必要将这样的环境限制为以单一类型的特征来描述金融市场的状态，也没有必要只使用 4 个滞后项。本节介绍了改进的 `Finance` 类，该类允许使用多个特征和灵活长度的滞后项，以及为所用基本数据集指定起点和终点。这将允许将数据集的一部分用于学习，另一部分用于验证或测试。下面提供的 Python 代码也允许使用杠杆，当日内数据被认为具有相对较小的绝对收益率时，这可能会有所帮助。

```
In [64]: class Finance:
             url = 'http://hilpisch.com/aiif_eikon_eod_data.csv'
             def __init__(self, symbol, features, window, lags,
                          leverage=1, min_performance=0.85,
                          start=0, end=None, mu=None, std=None):
                 self.symbol = symbol
                 self.features = features  ❶
                 self.n_features = len(features)
                 self.window = window
                 self.lags = lags  ❶
                 self.leverage = leverage  ❷
                 self.min_performance = min_performance  ❸
                 self.start = start
                 self.end = end
                 self.mu = mu
                 self.std = std
                 self.observation_space = observation_space(self.lags)
                 self.action_space = action_space(2)
                 self._get_data()
                 self._prepare_data()
             def _get_data(self):
                 self.raw = pd.read_csv(self.url, index_col=0,
                                        parse_dates=True).dropna()
             def _prepare_data(self):
                 self.data = pd.DataFrame(self.raw[self.symbol])
                 self.data = self.data.iloc[self.start:]
                 self.data['r'] = np.log(self.data / self.data.shift(1))
                 self.data.dropna(inplace=True)
                 self.data['s'] = self.data[self.symbol].rolling(
                                                       self.window).mean()  ❹
                 self.data['m'] = self.data['r'].rolling(self.window).mean()  ❹
                 self.data['v'] = self.data['r'].rolling(self.window).std()  ❹
                 self.data.dropna(inplace=True)
                 if self.mu is None:
                     self.mu = self.data.mean()  ❺
                     self.std = self.data.std()  ❺
                 self.data_ = (self.data - self.mu) / self.std  ❺
                 self.data_['d'] = np.where(self.data['r'] > 0, 1, 0)
                 self.data_['d'] = self.data_['d'].astype(int)
                 if self.end is not None:
                     self.data = self.data.iloc[:self.end - self.start]
                     self.data_ = self.data_.iloc[:self.end - self.start]
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
    def _get_state(self):
        return self.data_[self.features].iloc[self.bar -
                                  self.lags:self.bar]
    def seed(self, seed):
        random.seed(seed)
        np.random.seed(seed)
    def reset(self):
        self.treward = 0
        self.accuracy = 0
        self.performance = 1
        self.bar = self.lags
        state = self.data_[self.features].iloc[self.bar-
                        self.lags:self.bar]
        return state.values
    def step(self, action):
        correct = action == self.data_['d'].iloc[self.bar]
        ret = self.data['r'].iloc[self.bar] * self.leverage  ❻
        reward_1 = 1 if correct else 0
        reward_2 = abs(ret) if correct else -abs(ret)  ❼
        factor = 1 if correct else -1
        self.treward += reward_1
        self.bar += 1
        self.accuracy = self.treward / (self.bar - self.lags)
        self.performance *= math.exp(reward_2)  ❽
        if self.bar >= len(self.data):
            done = True
        elif reward_1 == 1:
            done = False
        elif (self.performance < self.min_performance and
              self.bar > self.lags + 5):
            done = True
        else:
            done = False
        state = self._get_state()
        info = {}
        return state.values, reward_1 + reward_2 * 5, done, info
```

❶ 定义状态的特征。

❷ 要使用的滞后项长度。

❸ 所需的最低总收益。

❹ 添加金融特征变量（简单移动平均线、动量、滚动波动率）。

❺ 数据的高斯归一化。

❻ 该步的杠杆收益率。

❼ 该步的基于收益率的奖励。

❽ 该步结束后的总收益。

新的 Finance 类为金融市场环境的建模提供了更大的灵活性。以下代码是一个有两个特征和 5 个滞后项的示例。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [65]: env = Finance('EUR=', ['EUR=', 'r'], 10, 5)

In [66]: a = env.action_space.sample()
         a
Out[66]: 0

In [67]: env.reset()
Out[67]: array([[ 1.7721, -1.0214],
                [ 1.5973, -2.4432],
                [ 1.5876, -0.1208],
                [ 1.6292,  0.6083],
                [ 1.6408,  0.1807]])

In [68]: env.step(a)
Out[68]: (array([[ 1.5973, -2.4432],
                 [ 1.5876, -0.1208],
                 [ 1.6292,  0.6083],
                 [ 1.6408,  0.1807],
                 [ 1.5725, -0.9502]]),
          1.0272827803740798,
          False,
          {})
```

不同类型的环境和数据

需要注意的是，CartPole 环境和 Finance 环境的两个版本之间存在根本区别。在 CartPole 环境中，没有可用的数据，仅选择具有某种程度随机性的初始状态。给定此状态和智能体采取的操作，可以应用确定性转换来生成新状态（数据）。之所以这是可能的，是因为模拟了遵循物理定律的物理系统。

另外，Finance 环境是从真实的历史市场数据开始的，并且仅以与 CartPole 环境类似的方式（逐步和逐个状态地）将可用数据呈现给智能体。在这种情况下，智能体的行为并没有真正影响环境。相反，环境演化是具有确定性的，智能体只是学习如何在该环境中以最佳方式进行有利可图的交易。

从这个意义上说，Finance 环境更像是在迷宫中寻找最短路径的问题。在这种情况下，代表迷宫的数据是预先给出的，并且当智能体在迷宫中移动时，它只会看到数据的相关子集（当前状态）。

9.8 FQL智能体

本节依托新的 Finance 环境，对简单的 DQL 智能体进行了改进，以提升其在金融市场环境下的性能。FQLAgent 类能够处理多个特征和灵活长度的滞后项。它还可以将学习环境（learn_env）与验证环境（valid_env）区分开来，从而使我们能够在训练期间获得智能体的样本外性能的更真实表现。DQLAgent 类与 FQLAgent 类的类和 RL/QL 学习方法拥有相同的基本结构。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [69]: class FQLAgent:
             def __init__(self, hidden_units, learning_rate, learn_env, valid_env):
                 self.learn_env = learn_env
                 self.valid_env = valid_env
                 self.epsilon = 1.0
                 self.epsilon_min = 0.1
                 self.epsilon_decay = 0.98
                 self.learning_rate = learning_rate
                 self.gamma = 0.95
                 self.batch_size = 128
                 self.max_treward = 0
                 self.trewards = list()
                 self.averages = list()
                 self.performances = list()
                 self.aperformances = list()
                 self.vperformances = list()
                 self.memory = deque(maxlen=2000)
                 self.model = self._build_model(hidden_units, learning_rate)

             def _build_model(self, hu, lr):
                 model = Sequential()
                 model.add(Dense(hu, input_shape=(
                     self.learn_env.lags, self.learn_env.n_features),
                                 activation='relu'))
                 model.add(Dropout(0.3, seed=100))
                 model.add(Dense(hu, activation='relu'))
                 model.add(Dropout(0.3, seed=100))
                 model.add(Dense(2, activation='linear'))
                 model.compile(
                     loss='mse',
                     optimizer=RMSprop(lr=lr)
                 )
                 return model

             def act(self, state):
                 if random.random() <= self.epsilon:
                     return self.learn_env.action_space.sample()
                 action = self.model.predict(state)[0, 0]
                 return np.argmax(action)

             def replay(self):
                 batch = random.sample(self.memory, self.batch_size)
                 for state, action, reward, next_state, done in batch:
                     if not done:
                         reward += self.gamma * np.amax(
                             self.model.predict(next_state)[0, 0])
                     target = self.model.predict(state)
                     target[0, 0, action] = reward
                     self.model.fit(state, target, epochs=1,
                                    verbose=False)
                 if self.epsilon > self.epsilon_min:
                     self.epsilon *= self.epsilon_decay
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
    def learn(self, episodes):
        for e in range(1, episodes + 1):
            state = self.learn_env.reset()
            state = np.reshape(state, [1, self.learn_env.lags,
                                       self.learn_env.n_features])
            for _ in range(10000):
                action = self.act(state)
                next_state, reward, done, info = \
                                self.learn_env.step(action)
                next_state = np.reshape(next_state,
                                [1, self.learn_env.lags,
                                 self.learn_env.n_features])
                self.memory.append([state, action, reward,
                                     next_state, done])
                state = next_state
                if done:
                    treward = _ + 1
                    self.trewards.append(treward)
                    av = sum(self.trewards[-25:]) / 25
                    perf = self.learn_env.performance
                    self.averages.append(av)
                    self.performances.append(perf)
                    self.aperformances.append(
                        sum(self.performances[-25:]) / 25)
                    self.max_treward = max(self.max_treward, treward)
                    templ = 'episode: {:2d}/{} | treward: {:4d} | '
                    templ += 'perf: {:5.3f} | av: {:5.1f} | max: {:4d}'
                    print(templ.format(e, episodes, treward, perf,
                                  av, self.max_treward), end='\r')
                    break
            self.validate(e, episodes)
            if len(self.memory) > self.batch_size:
                self.replay()
    def validate(self, e, episodes):
        state = self.valid_env.reset()
        state = np.reshape(state, [1, self.valid_env.lags,
                                   self.valid_env.n_features])
        for _ in range(10000):
            action = np.argmax(self.model.predict(state)[0, 0])
            next_state, reward, done, info = self.valid_env.step(action)
            state = np.reshape(next_state, [1, self.valid_env.lags,
                                   self.valid_env.n_features])
            if done:
                treward = _ + 1
                perf = self.valid_env.performance
                self.vperformances.append(perf)
                if e % 20 == 0:
                    templ = 71 * '='
                    templ += '\nepisode: {:2d}/{} | VALIDATION | '
                    templ += 'treward: {:4d} | perf: {:5.3f} | '
                    templ += 'eps: {:.2f}\n'
                    templ += 71 * '='
                    print(templ.format(e, episodes, treward,
                                       perf, self.epsilon))
                break
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

以下 Python 代码表明，FQLAgent 的性能明显优于解决 CartPole 问题的简单的 DQLAgent，这个交易机器人似乎会通过与金融市场环境的互动来相当有效地学习交易（参见图 9-4）。

```
In [70]: symbol = 'EUR='
         features = [symbol, 'r', 's', 'm', 'v']

In [71]: a = 0
         b = 2000
         c = 500

In [72]: learn_env = Finance(symbol, features, window=10, lags=6,
                             leverage=1, min_performance=0.85,
                             start=a, end=a + b, mu=None, std=None)

In [73]: learn_env.data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2000 entries, 2010-01-19 to 2017-12-26
         Data columns (total 5 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   EUR=    2000 non-null   float64
          1   r       2000 non-null   float64
          2   s       2000 non-null   float64
          3   m       2000 non-null   float64
          4   v       2000 non-null   float64
         dtypes: float64(5)
         memory usage: 93.8 KB

In [74]: valid_env = Finance(symbol, features, window=learn_env.window,
                             lags=learn_env.lags, leverage=learn_env.leverage,
                             min_performance=learn_env.min_performance,
                             start=a + b, end=a + b + c,
                             mu=learn_env.mu, std=learn_env.std)

In [75]: valid_env.data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 500 entries, 2017-12-27 to 2019-12-20
         Data columns (total 5 columns):
          #   Column  Non-Null Count  Dtype
        ---   ------  --------------  -----
         0    EUR=    500 non-null    float64
         1    r       500 non-null    float64
         2    s       500 non-null    float64
         3    m       500 non-null    float64
         4    v       500 non-null    float64
        dtypes: float64(5)
        memory usage: 23.4 KB

In [76]: set_seeds(100)
         agent = FQLAgent(24, 0.0001, learn_env, valid_env)

In [77]: episodes = 61

In [78]: agent.learn(episodes)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
=======================================================================
episode: 20/61 | VALIDATION | treward: 494 | perf: 1.169 | eps: 0.68
=======================================================================
=======================================================================
episode: 40/61 | VALIDATION | treward: 494 | perf: 1.111 | eps: 0.45
=======================================================================
=======================================================================
episode: 60/61 | VALIDATION | treward: 494 | perf: 1.089 | eps: 0.30
=======================================================================
episode: 61/61 | treward: 1994 | perf: 1.268 | av: 1615.1 | max: 1994

In [79]: agent.epsilon
Out[79]: 0.291602079838278

In [80]: plt.figure(figsize=(10, 6))
         x = range(1, len(agent.averages) + 1)
         y = np.polyval(np.polyfit(x, agent.averages, deg=3), x)
         plt.plot(agent.averages, label='moving average')
         plt.plot(x, y, 'r--', label='regression')
         plt.xlabel('episodes')
         plt.ylabel('total reward')
         plt.legend();
```

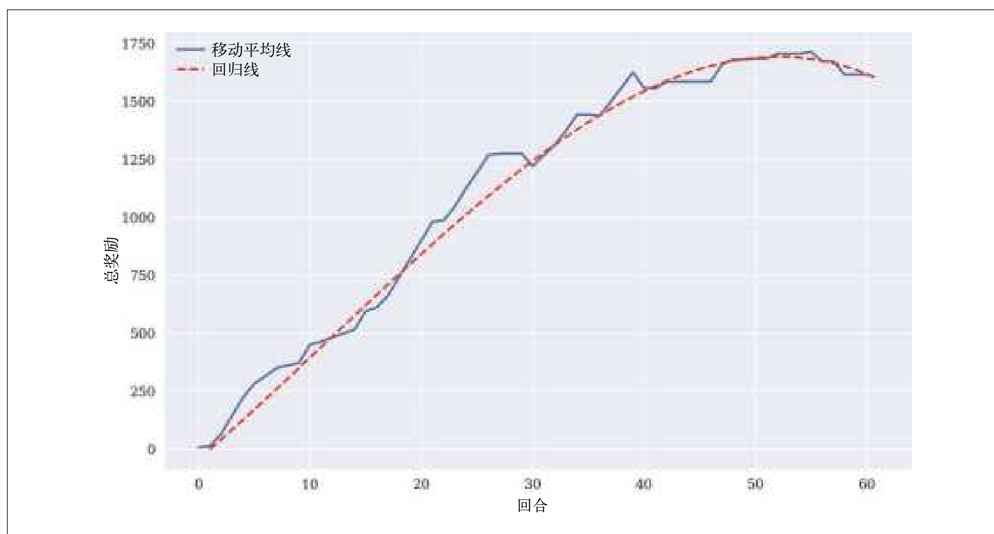


图 9-4：运行于 Finance 环境的 FQLAgent 的平均总奖励

训练集和验证集的性能也出现了一个有趣的现象，如图 9-5 所示。训练集性能显示出了很大的方差，这是由于除了**利用**当前最佳策略之外，还要利用正在进行的**探索**。相比之下，验证集性能的方差要小得多，因为它仅依赖于对当前最优策略的利用。

```
In [81]: plt.figure(figsize=(10, 6))
         x = range(1, len(agent.performances) + 1)
         y = np.polyval(np.polyfit(x, agent.performances, deg=3), x)
         y_ = np.polyval(np.polyfit(x, agent.vperformances, deg=3), x)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
plt.plot(agent.performances[:], label='training')
plt.plot(agent.vperformances[:], label='validation')
plt.plot(x, y, 'r--', label='regression (train)')
plt.plot(x, y_, 'r-.', label='regression (valid)')
plt.xlabel('episodes')
plt.ylabel('gross performance')
plt.legend();
```

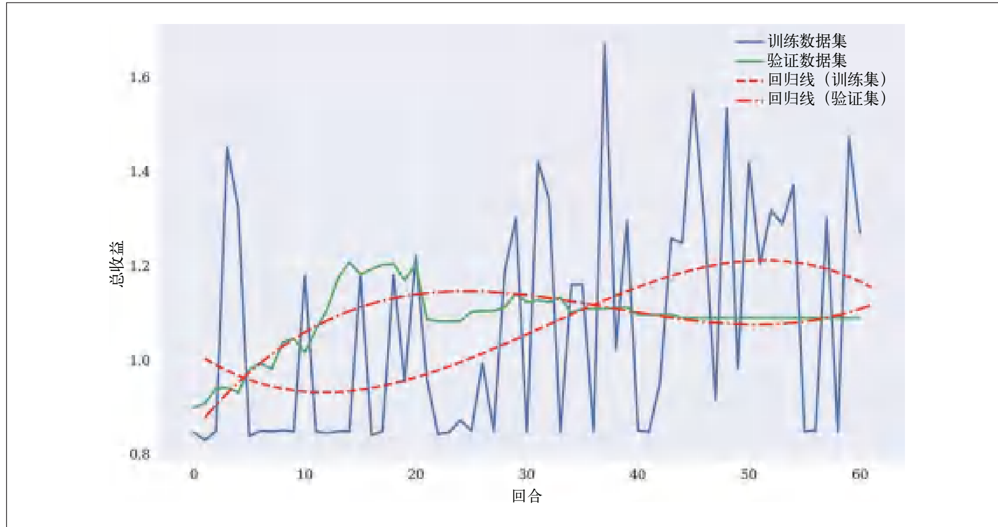


图 9-5：FQLAgent 的每回合的训练集性能和验证集性能

9.9 结论

本章讨论了人工智能中最成功的一个算法类——强化学习。第 2 章中讨论的大多数进展和成功案例源于强化学习领域的改进。在这种情况下，神经网络并没有变得毫无用处。相反，神经网络在逼近最优动作策略方面发挥着重要作用，通常以策略 Q 的形式存在，即给定某种状态，为每个动作分配一个值。如果考虑即时奖励和延迟奖励，则值越高，动作越好。

当然，延迟奖励是与许多重要环境相关的信息。在游戏环境中，通常有多种操作可用，最好选择能获得最高总奖励而不仅仅是最高即时奖励的操作，因为最终的总分是要最大化的分数。在金融环境中也是如此，通常，交易和投资的适当目标是看长期表现，而不是可能会增加破产风险的快速短期利润。

本章中的示例还表明强化学习方法相当灵活且通用，因为它同样可以应用于不同的设置。解决 CartPole 问题的 DQL 智能体也可以学习如何在金融市场中进行交易，尽管表现得不是很好。基于 Finance 环境和 FQL 智能体的改进，FQL 交易机器人在样本内（在训练集数据上）和样本外（在验证集数据上）都表现出了可观的性能。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第四部分

算法交易

成功意味着制造盈利，避免亏损。

——Martin Zweig

第三部分涉及使用深度学习和强化学习技术发现金融市场中**统计**失效的问题。相比之下，第四部分关注的是识别和利用**经济**失效，而统计失效通常是其先决条件。利用经济失效的首选工具是**算法交易**，即基于交易机器人生成的预测自动执行交易策略。

表 IV-1 以简化的方式比较了训练和部署交易机器人与构建和部署自动驾驶汽车的问题。

表IV-1：自动驾驶汽车与交易机器人的比较

步骤	自动驾驶汽车	交易机器人
训练	在虚拟和记录环境中训练人工智能	使用模拟和真实历史数据训练人工智能
风险管理	添加规则以避免碰撞、车祸等	添加规则以避免大额损失、尽早获利等
部署	人工智能与汽车硬件相结合，在街道上部署汽车并进行监控	人工智能与交易平台相结合，部署交易机器人进行真实交易，并进行监控

第四部分由 3 章组成，按照表 IV-1 所示的 3 个步骤的结构，通过交易机器人来利用经济失效。也就是说，从交易策略的向量化回测开始，涵盖基于事件回测的风险管理措施分析，并在策略执行和部署的背景下讨论技术细节。

- 第 10 章是关于算法交易策略的**向量化回测**，比如基于 DNN 进行市场预测的策略。这种方法对交易策略的经济潜力的初步判断既有效又富有洞察力，它还允许人们评估交易成本对经济绩效的影响。
- 第 11 章涵盖了管理算法交易策略风险的核心方面，比如使用止损单或止盈单。除了向量化回测，这一章还介绍了基于事件的回测，以作为一种更灵活的方法来判断交易策略的经济潜力。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

- 第 12 章主要是关于交易策略的执行，主题是历史数据的提取、基于这些数据的交易机器人的训练、实时数据的流传输以及订单的放置。这一章不仅介绍了适合算法交易的交易平台 Oanda 及其 API，还涵盖了以自动方式部署人工智能算法交易策略的基本方法。

算法交易策略

算法交易是一个广阔的领域，包含不同类型的交易策略。例如，有些人试图在执行大订单期间最小化市场影响（流动性算法），有些人则试图尽可能地复制衍生工具的回报（动态对冲 / 复制）。这些例子表明，并非所有算法交易策略都以利用经济失效为目标。就本书而言，关注由（以 DNN 智能体或 RL 智能体形式出现的）交易机器人做出的预测所产生的算法交易策略似乎是合适且有用的。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第10章

向量化回测

特斯拉（Tesla）首席执行官、连续创业的科技公司创始人 Elon Musk 表示，在未来两年内，特斯拉汽车在美国各地能被车主自己召唤并自动驾驶接送车主。

——Samuel Gibbs，2016 年

在股市中，只要在重大变动中站在正确的一边，就能赚大钱。

——Martin Zweig

术语**向量化回测**指的是对算法交易策略［比如基于“密集神经网络”（DNN）进行市场预测策略］进行回测的技术方法。Hilpisch（2018，第 15 章；2020，第 4 章）涵盖了基于一些具体向量化回测的例子。在这种情况下，**量化**是指一种编程范式，它在很大程度上甚至完全依赖于向量化代码（在 Python 中没有任何循环的代码）。向量化代码通常对于 `Numpy` 或 `pandas` 这样的包是一种很好的实践，在前面的章节中已进行了广泛的应用。向量化代码的好处是代码更简洁且易于阅读，而且在许多重要场景中执行更快。不过，在回测交易策略方面，它可能没有第 11 章中介绍和使用的基于事件的反向测试灵活。

拥有一个比简单的基准预测工具更好的人工智能预测器很重要，但通常不足以产生 alpha 收益（高于市场收益率，可能经过了风险调整）。例如，对基于预测的交易策略来说，正确预测大的市场波动很重要，而不仅仅是正确预测大多数（可能相当小的）市场波动。向量化回测是一种简单而快速地指出交易策略经济价值潜力的方式。

与自动驾驶汽车相比，向量化回测就像在虚拟环境中测试自动驾驶汽车的人工智能，只是为了看看它在无风险环境中“总体”表现如何。然而，对自动驾驶汽车的人工智能来说，虽然**平均**表现良好很重要，但最重要的是看它如何掌控危急甚至极端的情况。这种人工智能应该平均造成“零伤亡”，而不是存在 0.1 或 0.5 的“伤亡”概率。对金融领域的人工智能来说，虽然与自动驾驶汽车的人工智能不完全相同，但也是类似的，重要的是正确把握

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

大的市场波动。本章关注的是金融人工智能体（交易机器人）的纯性能，而第 11 章更深入地探讨了风险评估和标准风控措施的回测。

10.1 节介绍了向量化回测的一个简单示例，该示例使用简单移动平均值作为技术指标和日终数据。这让你在开始的时候就有了一个深刻的可视化且更容易理解的方法。10.2 节利用日终数据对 DNN 进行了训练，并基于预测的策略对其经济绩效进行了回测。10.3 节则对日内数据进行了相同的操作。在所有的例子中，按比例的交易成本都以假定的买卖价差的形式包括在内。

10.1 基于SMA策略的回测

本节介绍了使用简单移动平均线（SMA）作为技术指标的经典交易策略的向量化回测。以下代码实现了必要的导入和配置，并检索了欧元 / 美元货币对的日终数据。

```
In [1]: import os
        import math
        import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        pd.set_option('mode.chained_assignment', None)
        pd.set_option('display.float_format', '{:.4f}'.format)
        np.set_printoptions(suppress=True, precision=4)
        os.environ['PYTHONHASHSEED'] = '0'

In [2]: url = 'http://hilpisch.com/aiif_eikon_eod_data.csv'  ❶

In [3]: symbol = 'EUR='  ❶

In [4]: data = pd.DataFrame(pd.read_csv(url, index_col=0,
                                        parse_dates=True).dropna()[symbol])  ❶

In [5]: data.info()  ❶
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 2516 entries, 2010-01-04 to 2019-12-31
        Data columns (total 1 columns):
         #   Column  Non-Null Count  Dtype
        ---  ------  --------------  -----
         0   EUR=    2516 non-null   float64
        dtypes: float64(1)
        memory usage: 39.3 KB
```

❶ 检索欧元 / 美元的日终数据。

该策略的理念如下：计算一个较短的 `SMA1`（如 42 天）和一个较长的 `SMA2`（如 258 天）。当 `SMA1` 高于 `SMA2` 时，就做多该金融工具。当 `SMA1` 低于 `SMA2` 时，就做空该金融工具。因为这个例子是以欧元 / 美元为基础的，所以做多或做空都很容易完成。

下面的 Python 代码以向量化的方式计算 SMA 值，并将得到的时间序列与原始时间序列一

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

起进行可视化（参见图 10-1）。

```
In [6]: data['SMA1'] = data[symbol].rolling(42).mean()  ❶

In [7]: data['SMA2'] = data[symbol].rolling(258).mean()  ❷

In [8]: data.plot(figsize=(10, 6));  ❸
```

❶ 计算较短的 SMA1。

❷ 计算较长的 SMA2。

❸ 可视化 3 个时间序列。

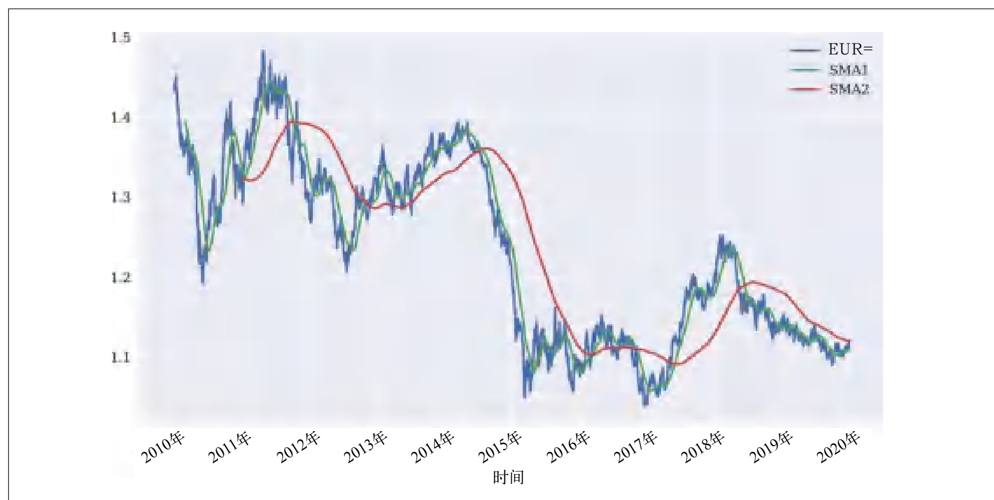


图 10-1：欧元 / 美元及其 SMA 的时间序列数据

利用 SMA 时间序列数据，可以再次以向量化方式导出最终头寸。注意，由此产生的最终头寸值的时间序列要移动一天，以避免数据中的预测偏差。这种移动是必要的，因为 SMA 的计算包括当天的收盘值。因此，从某一天的 SMA 值得到的头寸对应着整个时间序列的第二天。

图 10-2 会将最终头寸可视化为对其他时间序列的叠加。

```
In [9]: data.dropna(inplace=True)  ❶

In [10]: data['p'] = np.where(data['SMA1'] > data['SMA2'], 1, -1)  ❷

In [11]: data['p'] = data['p'].shift(1)  ❸

In [12]: data.dropna(inplace=True)  ❶

In [13]: data.plot(figsize=(10, 6), secondary_y='p');  ❹
```

❶ 删除包含 NaN 值的行。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❷ 基于当天 SMA 值导出头寸值。

❸ 将头寸值移动一天以避免预测偏差。

❹ 可视化从 SMA 导出的位置值。

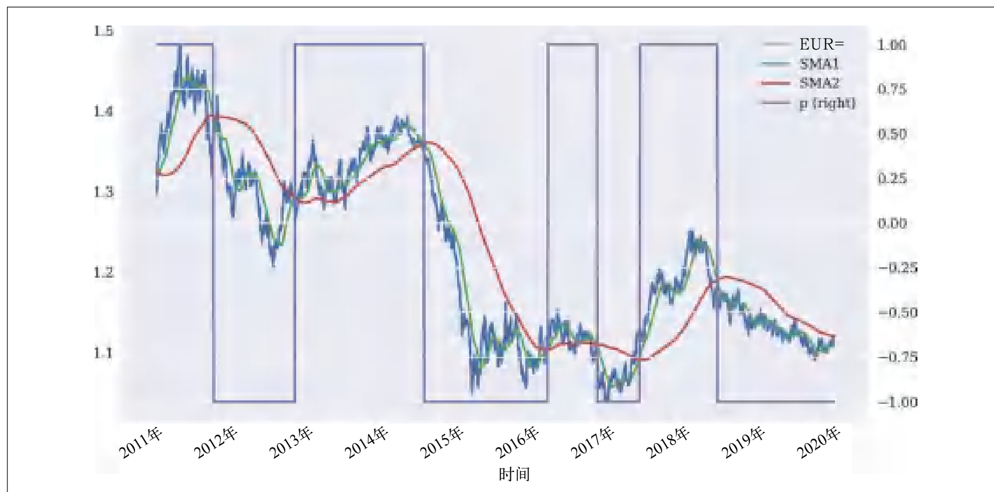


图 10-2：欧元 / 美元、SMA 和最终头寸的时间序列数据

这里缺失了一个关键步骤，即将头寸与金融工具的收益率结合起来。由于可以很方便地用 a+1 表示多头头寸，用 a-1 表示空头头寸，因此这一步可将 DataFrame 对象的两列以向量化的方式相乘。如图 10-3 所展示的，基于 SMA 的交易策略的表现远远好于被动基准投资。

```
In [14]: data['r'] = np.log(data[symbol] / data[symbol].shift(1))  ❶

In [15]: data.dropna(inplace=True)

In [16]: data['s'] = data['p'] * data['r']  ❷

In [17]: data[['r', 's']].sum().apply(np.exp)  ❸
Out[17]: r   0.8640
         s   1.3773
         dtype: float64

In [18]: data[['r', 's']].sum().apply(np.exp) - 1  ❹
Out[18]: r   -0.1360
         s    0.3773
         dtype: float64

In [19]: data[['r', 's']].cumsum().apply(np.exp).plot(figsize=(10, 6));  ❺
```

❶ 计算对数收益率。

❷ 计算策略收益率。

❸ 计算总收益。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❹ 计算净收益。

❺ 可视化总收益随时间的变化。

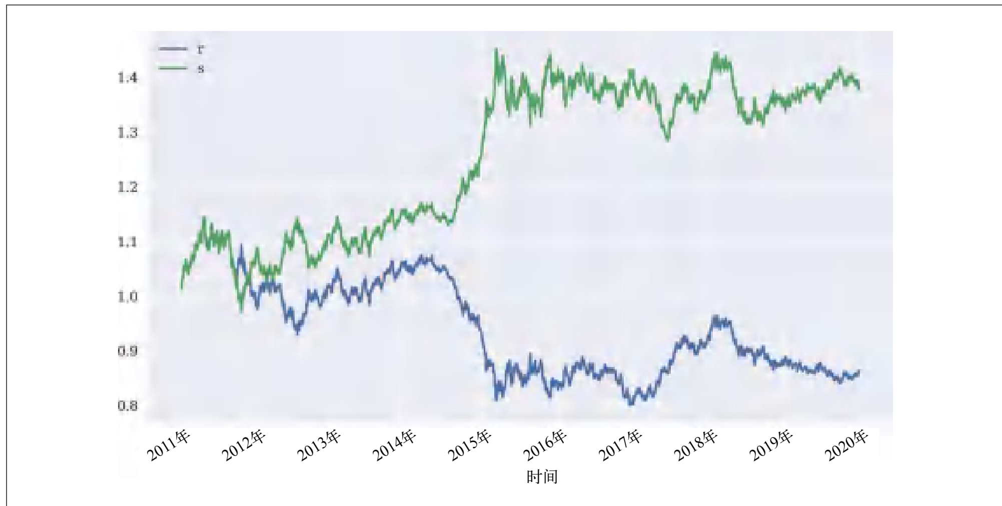


图 10-3：被动基准投资和 SMA 策略的总体表现

到目前为止，收益数据并没有考虑交易成本。当然，在判断一项交易策略的经济价值潜力时，这些因素是至关重要的。在当前设置中，成比例交易成本可以轻松地包含在计算中。它的目的是确定交易何时发生，并将交易策略的收益降低一定的价值，以计入相关的买卖价差。如以下计算及图 10-2 所示，交易策略不会太频繁地改变头寸。因此，为了对交易成本产生一些有意义的影响，假设它们比通常看到的欧元 / 美元要高得多。在给定假设下，减去交易成本的净影响为几个百分点（参见图 10-4）。

```
In [20]: sum(data['p'].diff() != 0) + 2  ❶
Out[20]: 10

In [21]: pc = 0.005  ❷

In [22]: data['s_'] = np.where(data['p'].diff() != 0,
                               data['s'] - pc, data['s'])  ❸

In [23]: data['s_'].iloc[0] -= pc  ❹

In [24]: data['s_'].iloc[-1] -= pc  ❺

In [25]: data[['r', 's', 's_']][data['p'].diff() != 0]  ❻
Out[25]:                  r       s      s_
         Date
         2011-01-12  0.0123  0.0123  0.0023
         2011-10-10  0.0198 -0.0198 -0.0248
         2012-11-07 -0.0034 -0.0034 -0.0084
         2014-07-24 -0.0001  0.0001 -0.0049
         2016-03-16  0.0102  0.0102  0.0052
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
           2016-11-10 -0.0018  0.0018 -0.0032
           2017-06-05 -0.0025 -0.0025 -0.0075
           2018-06-15  0.0035 -0.0035 -0.0085

In [26]: data[['r', 's', 's_']].sum().apply(np.exp)
Out[26]: r     0.8640
         s     1.3773
         s_    1.3102
         dtype: float64

In [27]: data[['r', 's', 's_']].sum().apply(np.exp) - 1
Out[27]: r     -0.1360
         s      0.3773
         s_     0.3102
         dtype: float64

In [28]: data[['r', 's', 's_']].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ 计算交易数量，包括进入交易和退出交易。

❷ 固定交易成本比例（故意设置得很高）。

❸ 因交易成本调整策略收益。

❹ 为**进入**交易调整策略收益。

❺ 为**退出**交易调整策略收益。

❻ 显示常规交易在调整后的收益值。

图 10-4：设置交易成本前后 SMA 策略的总体表现

交易策略带来的风险是什么？对一种基于方向预测且只采取多头头寸或空头头寸的交易策略来说，其风险以波动率（对数收益率的标准差）表示，与被动基准投资完全相同。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [29]: data[['r', 's', 's_']].std()  ❶
Out[29]: r    0.0054
         s    0.0054
         s_   0.0054
         dtype: float64

In [30]: data[['r', 's', 's_']].std() * math.sqrt(252)  ❷
Out[30]: r    0.0853
         s    0.0853
         s_   0.0855
         dtype: float64
```

❶ 日波动率。

❷ 年化波动率。

向量化回测

向量化回测是一种强大而有效的方法，可以回测基于预测的交易策略的“纯”收益。它还可以考虑成比例交易成本的影响。然而，它并不太适合分析典型的风险管理措施，比如（跟踪）止损单或获利单。第 11 章会对此进行论述。

10.2 基于DNN的每日策略的回测

上一节在一个简单且易于可视化的交易策略的基础上列出了向量化回测方案。同样的方案也可以应用于基于 DNN 的交易策略，只需进行最小的技术调整。下面训练一个 Keras 的 DNN 模型，其所使用的数据与上一个例子中的数据相同。但是，如第 7 章所述，需要向 DataFrame 对象添加不同的特征和滞后项。

```
In [31]: data = pd.DataFrame(pd.read_csv(url, index_col=0,
                                         parse_dates=True).dropna()[symbol])

In [32]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2516 entries, 2010-01-04 to 2019-12-31
         Data columns (total 1 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
         0    EUR=    2516 non-null   float64
         dtypes: float64(1)
         memory usage: 39.3 KB

In [33]: lags = 5

In [34]: def add_lags(data, symbol, lags, window=20):
             cols = []
             df = data.copy()
             df.dropna(inplace=True)
             df['r'] = np.log(df / df.shift(1))
             df['sma'] = df[symbol].rolling(window).mean()
             df['min'] = df[symbol].rolling(window).min()
             df['max'] = df[symbol].rolling(window).max()
             df['mom'] = df['r'].rolling(window).mean()
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        df['vol'] = df['r'].rolling(window).std()
        df.dropna(inplace=True)
        df['d'] = np.where(df['r'] > 0, 1, 0)
        features = [symbol, 'r', 'd', 'sma', 'min', 'max', 'mom', 'vol']
        for f in features:
            for lag in range(1, lags + 1):
                col = f'{f}_lag_{lag}'
                df[col] = df[f].shift(lag)
                cols.append(col)
        df.dropna(inplace=True)
        return df, cols

In [35]: data, cols = add_lags(data, symbol, lags, window=20)
```

下面的 Python 代码会完成其他导入并定义 `set_seeds()` 函数和 `create_ model()` 函数。

```
In [36]: import random
         import tensorflow as tf
         from keras.layers import Dense, Dropout
         from keras.models import Sequential
         from keras.regularizers import l1
         from keras.optimizers import Adam
         from sklearn.metrics import accuracy_score
         Using TensorFlow backend.

In [37]: def set_seeds(seed=100):
             random.seed(seed)
             np.random.seed(seed)
             tf.random.set_seed(seed)
         set_seeds()

In [38]: optimizer = Adam(learning_rate=0.0001)

In [39]: def create_model(hl=2, hu=128, dropout=False, rate=0.3,
                          regularize=False, reg=l1(0.0005),
                          optimizer=optimizer, input_dim=len(cols)):
             if not regularize:
                 reg = None
             model = Sequential()
             model.add(Dense(hu, input_dim=input_dim,
                          activity_regularizer=reg,
                          activation='relu'))
             if dropout:
                 model.add(Dropout(rate, seed=100))
             for _ in range(hl):
                 model.add(Dense(hu, activation='relu',
                              activity_regularizer=reg))
                 if dropout:
                     model.add(Dropout(rate, seed=100))
             model.add(Dense(1, activation='sigmoid'))
             model.compile(loss='binary_crossentropy',
                           optimizer=optimizer,
                           metrics=['accuracy'])
             return model
```

基于历史数据的序列训练 – 测试分割，下面的 Python 代码首先会根据标准化特征数据来训练 DNN 模型。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [40]: split = '2018-01-01'  ❶

In [41]: train = data.loc[:split].copy()  ❶

In [42]: np.bincount(train['d'])  ❷
Out[42]: array([ 982, 1006])

In [43]: mu, std = train.mean(), train.std()  ❸

In [44]: train_ = (train - mu) / std  ❸

In [45]: set_seeds()
         model = create_model(hl=2, hu=64)  ❹

In [46]: %%time
         model.fit(train_[cols], train['d'],
                 epochs=20, verbose=False,
                 validation_split=0.2, shuffle=False)  ❺
         CPU times: user 2.93 s, sys: 574 ms, total: 3.5 s
         Wall time: 1.93 s

Out[46]: <keras.callbacks.callbacks.History at 0x7fc9392f38d0>

In [47]: model.evaluate(train_[cols], train['d'])  ❻
         1988/1988 [==============================] - 0s 17us/step

Out[47]: [0.6745863538872549, 0.5925553441047668]
```

❶ 将数据拆分为训练数据和测试数据。

❷ 显示标签类的频率。

❸ 标准化训练特征数据。

❹ 创建 DNN 模型。

❺ 在训练数据上训练 DNN 模型。

❻ 在训练数据上评估模型的性能。

到目前为止，这基本上重复了第 7 章的核心方法。向量化回测现在可用于根据模型预测结果来判断**样本内**基于 DNN 的交易策略的经济回报表现（参见图 10-5）。在这种情况下，向上预测自然被解释为多头头寸，向下预测则被解释为空头头寸。

```
In [48]: train['p'] = np.where(model.predict(train_[cols]) > 0.5, 1, 0)  ❶

In [49]: train['p'] = np.where(train['p'] == 1, 1, -1)  ❷

In [50]: train['p'].value_counts()  ❸
Out[50]: -1    1098
          1     890
         Name: p, dtype: int64

In [51]: train['s'] = train['p'] * train['r']  ❹

In [52]: train[['r', 's']].sum().apply(np.exp)  ❺
Out[52]: r   0.8787
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
         s   5.0766
         dtype: float64

In [53]: train[['r', 's']].sum().apply(np.exp)  - 1  ❺
Out[53]: r   -0.1213
         s    4.0766
         dtype: float64

In [54]: train[['r', 's']].cumsum().apply(np.exp).plot(figsize=(10, 6));  ❻
```

❶ 生成二分类（元）预测。

❷ 将预测值转换为头寸值。

❸ 显示多头头寸和空头头寸的数量。

❹ 计算策略收益值。

❺ 计算总收益和净收益（样本内）。

❻ 可视化总收益随时间的变化（样本内）。

图 10-5：被动基准投资和每日 DNN 策略的总体表现（样本内）

接下来对测试数据集进行相同的序列计算。尽管样本内的输出性能是显著的，样本外的表现虽然并不是那么令人印象深刻，但仍然令人信服（参见图 10-6）。

```
In [55]: test = data.loc[split:].copy()  ❶

In [56]: test_ = (test - mu) / std  ❷

In [57]: model.evaluate(test_[cols], test['d'])  ❸
         503/503 [==============================] - 0s 17us/step

Out[57]: [0.6933823573897421, 0.5407554507255554]
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [58]: test['p'] = np.where(model.predict(test_[cols]) > 0.5, 1, -1)

In [59]: test['p'].value_counts()
Out[59]: -1    406
          1     97
         Name: p, dtype: int64

In [60]: test['s'] = test['p'] * test['r']

In [61]: test[['r', 's']].sum().apply(np.exp)
Out[61]: r   0.9345
         s   1.2431
         dtype: float64

In [62]: test[['r', 's']].sum().apply(np.exp) - 1
Out[62]: r   -0.0655
         s    0.2431
         dtype: float64

In [63]: test[['r', 's']].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ 生成测试数据子集。

❷ 标准化测试数据。

❸ 根据测试数据评估模型性能。

图 10-6：被动基准投资和每日 DNN 策略的总体表现（样本外）

相对于 SMA 的策略，基于 DNN 的交易策略会导致更多的交易。这使得在判断策略的经济绩效时，交易成本的纳入变得更加重要。

以下代码假设欧元 / 美元的实际买卖价差为 1.2 点（0.000 12 的货币单位）。[1] 为了简化计算，请根据欧元 / 美元的平均收盘价计算成比例交易成本 pc 的平均值（参见图 10-7）。

注 1：这是 Oanda 为零售交易员提供的典型价差。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [64]: sum(test['p'].diff() != 0)
Out[64]: 147

In [65]: spread = 0.00012  ❶
         pc = spread / data[symbol].mean()  ❷
         print(f'{pc:.6f}')
         0.000098

In [66]: test['s_'] = np.where(test['p'].diff() != 0,
                               test['s'] - pc, test['s'])

In [67]: test['s_'].iloc[0] -= pc

In [68]: test['s_'].iloc[-1] -= pc

In [69]: test[['r', 's', 's_']].sum().apply(np.exp)
Out[69]: r     0.9345
         s     1.2431
         s_    1.2252
         dtype: float64

In [70]: test[['r', 's', 's_']].sum().apply(np.exp) - 1
Out[70]: r    -0.0655
         s     0.2431
         s_    0.2252
         dtype: float64

In [71]: test[['r', 's', 's_']].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ 固定平均买卖价差。

❷ 计算平均比例交易成本。

图 10-7：设置交易成本前后每日 DNN 策略的总体表现（样本外）

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

基于 DNN 的交易策略在设置典型交易成本前后似乎都很有前景。然而，当观察到更多的交易时，类似的策略在日内交易是否经济上也可行？下一节将基于 DNN 的日内策略进行分析。

10.3 基于DNN的日内策略的回测

要在日内数据上训练和回测 DNN 模型，需要另一个数据集。

```
In [72]: url = 'http://hilpisch.com/aiif_eikon_id_eur_usd.csv'  ❶

In [73]: symbol = 'EUR='  ❶

In [74]: data = pd.DataFrame(pd.read_csv(url, index_col=0,
                             parse_dates=True).dropna()['CLOSE'])  ❶
         data.columns = [symbol]

In [75]: data = data.resample('5min', label='right').last().ffill()  ❷

In [76]: data.info()  ❷
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 26486 entries, 2019-10-01 00:05:00 to 2019-12-31 23:10:00
         Freq: 5T
         Data columns (total 1 columns):
          #   Column  Non-Null Count  Dtype
         ---  ------  --------------  -----
          0   EUR=    26486 non-null  float64
         dtypes: float64(1)
         memory usage: 413.8 KB

In [77]: lags = 5

In [78]: data, cols = add_lags(data, symbol, lags, window=20)
```

❶ 检索欧元 / 美元的日内数据并选择收盘价。

❷ 将数据重新采样为 5 分钟间隔数据。

现在可以使用新的数据集重复上一节的步骤。先从训练 DNN 模型开始。

```
In [79]: split = int(len(data) * 0.85)

In [80]: train = data.iloc[:split].copy()

In [81]: np.bincount(train['d'])
Out[81]: array([16284, 6207])

In [82]: def cw(df):
             c0, c1 = np.bincount(df['d'])
             w0 = (1 / c0) * (len(df)) / 2
             w1 = (1 / c1) * (len(df)) / 2
             return {0: w0, 1: w1}
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [83]: mu, std = train.mean(), train.std()

In [84]: train_ = (train - mu) / std

In [85]: set_seeds()
         model = create_model(hl=1, hu=128,
                              reg=True, dropout=False)

In [86]: %%time
         model.fit(train_[cols], train['d'],
                   epochs=40, verbose=False,
                   validation_split=0.2, shuffle=False,
                   class_weight=cw(train))
         CPU times: user 40.6 s, sys: 5.49 s, total: 46 s
         Wall time: 25.2 s

Out[86]: <keras.callbacks.callbacks.History at 0x7fc91a6b2a90>

In [87]: model.evaluate(train_[cols], train['d'])
         22491/22491 [==============================] - 0s 13us/step

Out[87]: [0.5218664327576152, 0.6729803085327148]
```

在样本内，策略的表现看起来很不错，如图 10-8 所示。

```
In [88]: train['p'] = np.where(model.predict(train_[cols]) > 0.5, 1, -1)

In [89]: train['p'].value_counts()
Out[89]: -1    11519
          1    10972
         Name: p, dtype: int64

In [90]: train['s'] = train['p'] * train['r']

In [91]: train[['r', 's']].sum().apply(np.exp)
Out[91]: r   1.0223
         s   1.6665
         dtype: float64

In [92]: train[['r', 's']].sum().apply(np.exp) - 1
Out[92]: r   0.0223
         s   0.6665
         dtype: float64

In [93]: train[['r', 's']].cumsum().apply(np.exp).plot(figsize=(10, 6));
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

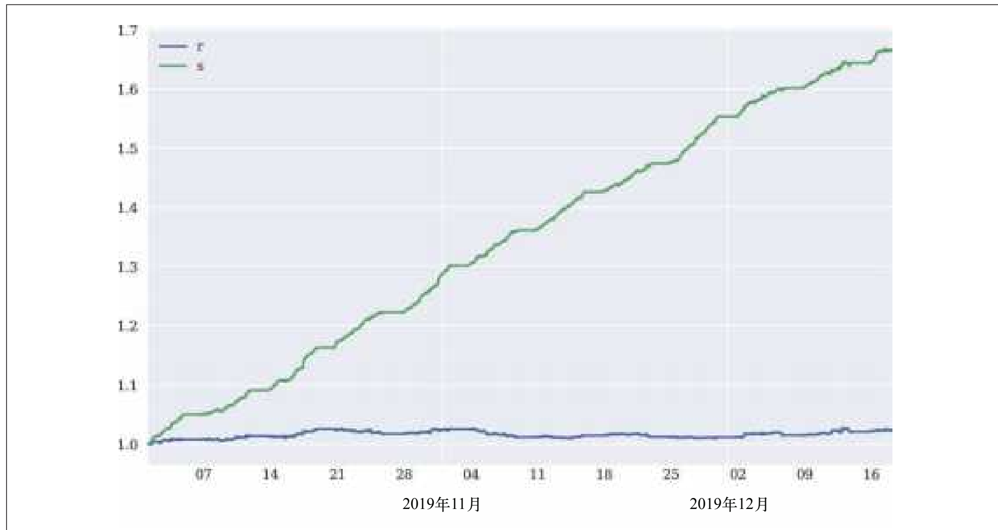


图 10-8：被动基准投资和 DNN 日内策略的总体表现（样本内）

在没有交易成本的情况下，样本外的表现看起来也很不错。该策略似乎系统性地优于被动基准投资（参见图 10-9）。

```
In [94]: test = data.iloc[split:].copy()

In [95]: test_ = (test - mu) / std

In [96]: model.evaluate(test_[cols], test['d'])
         3970/3970 [==============================] - 0s 19us/step

Out[96]: [0.5226116042706168, 0.668513834476471]

In [97]: test['p'] = np.where(model.predict(test_[cols]) > 0.5, 1, -1)

In [98]: test['p'].value_counts()
Out[98]: -1    2273
          1    1697
         Name: p, dtype: int64

In [99]: test['s'] = test['p'] * test['r']

In [100]: test[['r', 's']].sum().apply(np.exp)
Out[100]: r    1.0071
          s    1.0658
          dtype: float64

In [101]: test[['r', 's']].sum().apply(np.exp) - 1
Out[101]: r    0.0071
          s    0.0658
          dtype: float64

In [102]: test[['r', 's']].cumsum().apply(np.exp).plot(figsize=(10, 6));
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

图 10-9：被动基准投资和 DNN 日内策略的总体表现（样本外）

策略最后的试金石是看在增加交易成本时，其在纯经济绩效上的表现。这一策略能在相对较短的时间内促成数百笔交易。以下分析表明，基于标准零售买卖价差，DNN 的策略是不可行的。

将利差降低到专业的高交易量交易者可能达到的水平，该策略仍然不能实现盈亏平衡，反而因交易成本而损失了很大一部分利润（参见图 10-10）。

```
In [103]: sum(test['p'].diff() != 0)
Out[103]: 1303

In [104]: spread = 0.00012  ❶
          pc_1 = spread / test[symbol]  ❶

In [105]: spread = 0.00006  ❷
          pc_2 = spread / test[symbol]  ❷

In [106]: test['s_1'] = np.where(test['p'].diff() != 0,
                                 test['s'] - pc_1, test['s'])  ❶

In [107]: test['s_1'].iloc[0] -= pc_1.iloc[0]  ❶
          test['s_1'].iloc[-1] -= pc_1.iloc[0]  ❶

In [108]: test['s_2'] = np.where(test['p'].diff() != 0,
                                 test['s'] - pc_2, test['s'])  ❷

In [109]: test['s_2'].iloc[0] -= pc_2.iloc[0]  ❷
          test['s_2'].iloc[-1] -= pc_2.iloc[0]  ❷

In [110]: test[['r', 's', 's_1', 's_2']].sum().apply(np.exp)
Out[110]: r     1.0071
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
         s      1.0658
         s_1    0.9259
         s_2    0.9934
         dtype: float64

In [111]: test[['r', 's', 's_1', 's_2']].sum().apply(np.exp) - 1
Out[111]: r       0.0071
          s       0.0658
          s_1    -0.0741
          s_2    -0.0066
          dtype: float64

In [112]: test[['r', 's', 's_1', 's_2']].cumsum().apply(
              np.exp).plot(figsize=(10, 6), style=['-', '-', '--', '--']);
```

❶ 假设零售水平上的买卖价差。

❷ 假设专业水平上的买卖价差。

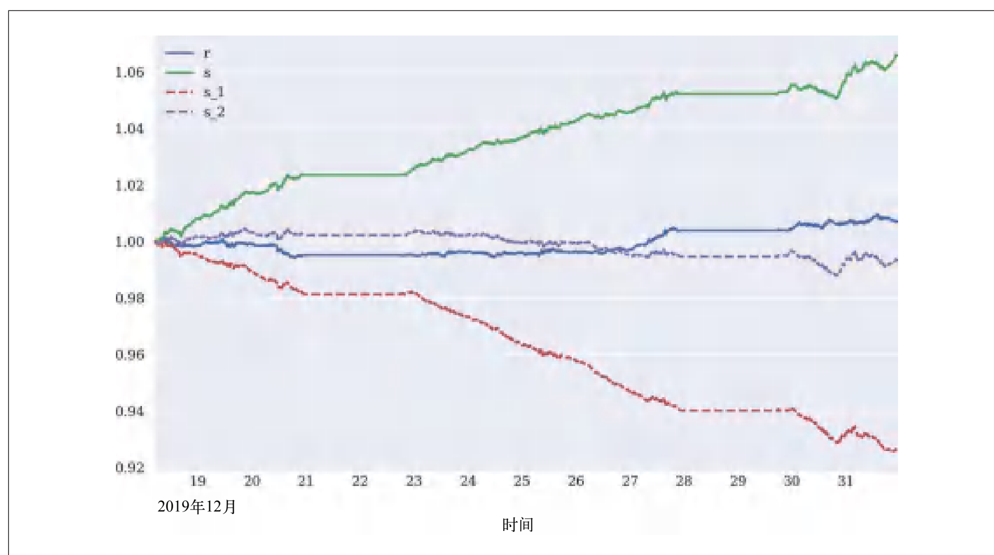
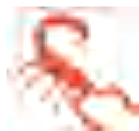


图 10-10：交易成本上升 / 下降前后 DNN 日内策略的总体表现（样本外）



日内交易

从统计的角度来看，本章讨论的日内算法交易似乎很有吸引力。无论是样本内还是样本外，DNN 模型在预测市场方向时都达到了很高的准确率。不考虑交易成本，不论是样本内还是样本外，基于 DNN 的策略的性能都显著优于被动基准投资。然而，在组合中增加交易成本会大大降低 DNN 策略的性能，对标准的零售买卖价差而言该策略是不可行的，而对较低且高交易量的买卖价差来说，该策略也并不具有真正的吸引力。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

10.4 结论

向量化回测被证明是对人工智能算法交易策略进行性能回测的一种有效且有价值的方法。本章首先以两个 SMA 为例，阐述了该方法的基本理念。这允许对策略和最终头寸进行简单的可视化。然后，结合日终数据，对基于 DNN 的交易策略进行了回测，详细信息可以参见第 7 章。在考虑交易成本前后，如第 7 章所述，将统计失效转化为经济失效，这意味着是有利可图的交易策略。当对日内数据使用相同的向量化回测方法时，DNN 策略在样本内和样本外的性能都明显优于被动基准投资，至少在考虑交易成本之前是这样。当交易成本被加到回测中时，要使交易策略在经济上可行，交易成本必须非常低，通常大型专业交易员都很难达到这个水平。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 11 章

风险管理

> 大规模部署自动驾驶汽车的一个重要障碍是安全保障。
>
> ——Majid Khonji 等，2019 年

> 拥有更好的预测能够提高判断的价值。毕竟，如果你不知道自己有多不喜欢被雨淋湿或有多讨厌带伞，那么知道下雨的可能性是没有用的。
>
> ——Ajay Agrawal 等，2018 年

一般来说，向量化回测使人们能够根据原来的策略（纯形式上的同样方式）判断基于预测的算法交易策略的经济潜力。在实际应用中，大多数人工智能体除了预测模型之外，还有更多的组成部分。例如，自动驾驶汽车的人工智能不是独立的，而是拥有大量规则和启发式算法，这些规则和算法用于限制人工智能采取或能够采取的行动。在自动驾驶汽车的环境中，这主要与管理风险有关，比如由碰撞导致的风险。

在金融环境中，人工智能体或交易机器人也不会按原样部署。相反，通常会有许多标准的风控措施，比如（**跟踪**）**止损单或获利单**。道理是很清楚的。在金融市场上进行定向押注时，要避免过大的损失。同样，当达到一定的利润水平时，投资的成功则需通过提前平仓来保护。如何处理这些风控措施往往取决于人类的判断，可能需要对相关数据和统计数字进行正式分析。从概念上讲，这是 Agrawal 等（2018）在书中讨论的一个主要观点：人工智能提供了改进的预测，但人类的判断在制定决策规则和行动边界方面仍然发挥着作用。

本章有 3 个目的。第一，对经过训练的深度 Q 学习智能体所产生的**向量化**和**基于事件**的流行算法交易策略进行回测。此后，这种智能体被称为**交易机器人**。第二，评估与应用交易策略的金融工具相关的风险。第三，使用本章介绍的事件法，对止损单等典型风控措施进行回测。与向量化回测相比，基于事件的回测的主要优点是在建模和分析决策规则与风控管理措施方面具有更高的灵活性。换句话说，它允许人们在使用向量化编程方法时放大事

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

件发展的背景细节。

11.1 节介绍并训练了基于第 9 章 Q 学习金融智能体的交易机器人。11.2 节使用第 10 章的向量化回测来判断交易机器人的（纯）经济性能。11.3 节介绍了基于事件的回测。首先讨论了基类。然后基于基类对交易机器人进行了回测。在这方面，也可参考 Hilpisch（2020）的第 6 章。11.4 节分析了对制定风险管理规则非常重要的选定的统计指标，比如**最大回撤**和**真实波动幅度均值**（ATR）。11.5 节回测了主要风控措施对交易机器人性能的影响。

11.1　交易机器人

本节介绍了一款基于第 9 章的 Q 学习金融智能体 FQLAgent 的交易机器人，这是后面几节将要分析的交易机器人。和往常一样，首先导入必要的库并进行环境设置。

```
In [1]: import os
        import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        pd.set_option('mode.chained_assignment', None)
        pd.set_option('display.float_format', '{:.4f}'.format)
        np.set_printoptions(suppress=True, precision=4)
        os.environ['PYTHONHASHSEED'] = '0'
```

11.7 节展示了一个 Python 模块，其中包括下面将使用的 Finance 类。本节提供了带有 TradingBot 类和一些辅助函数的 Python 模块，以用于绘制训练结果和验证结果。这两个类都非常接近第 9 章中介绍的类，这就是这里使用它们而没有做进一步解释的原因。

下面的代码会训练交易机器人使用历史的日终数据，包括用于验证的子数据集。图 11-1 显示了不同训练回合获得的平均总奖励。

```
In [2]: import finance
        import tradingbot
        Using TensorFlow backend.

In [3]: symbol = 'EUR='
        features = [symbol, 'r', 's', 'm', 'v']

In [4]: a = 0
        b = 1750
        c = 250

In [5]: learn_env = finance.Finance(symbol, features, window=20, lags=3,
                          leverage=1, min_performance=0.9, min_accuracy=0.475,
                          start=a, end=a + b, mu=None, std=None)

In [6]: learn_env.data.info()
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 1750 entries, 2010-02-02 to 2017-01-12
        Data columns (total 6 columns):
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   EUR=    1750 non-null   float64
 1   r       1750 non-null   float64
 2   s       1750 non-null   float64
 3   m       1750 non-null   float64
 4   v       1750 non-null   float64
 5   d       1750 non-null   int64
 dtypes: float64(5), int64(1)
 memory usage: 95.7 KB

In [7]: valid_env = finance.Finance(symbol, features=learn_env.features,
                                    window=learn_env.window,
                                    lags=learn_env.lags,
                                    leverage=learn_env.leverage,
                                    min_performance=0.0, min_accuracy=0.0,
                                    start=a + b, end=a + b + c,
                                    mu=learn_env.mu, std=learn_env.std)

In [8]: valid_env.data.info()
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 250 entries, 2017-01-13 to 2018-01-10
        Data columns (total 6 columns):
         #   Column  Non-Null Count  Dtype
        ---  ------  --------------  -----
         0   EUR=    250 non-null    float64
         1   r       250 non-null    float64
         2   s       250 non-null    float64
         3   m       250 non-null    float64
         4   v       250 non-null    float64
         5   d       250 non-null    int64
        dtypes: float64(5), int64(1)
        memory usage: 13.7 KB

In [9]: tradingbot.set_seeds(100)
        agent = tradingbot.TradingBot(24, 0.001, learn_env, valid_env)

In [10]: episodes = 61

In [11]: %time agent.learn(episodes)
        ========================================================================
        episode: 10/61 | VALIDATION | treward: 247 | perf: 0.936 | eps: 0.95
        ========================================================================
        ========================================================================
        episode: 20/61 | VALIDATION | treward: 247 | perf: 0.897 | eps: 0.86
        ========================================================================
        ========================================================================
        episode: 30/61 | VALIDATION | treward: 247 | perf: 1.035 | eps: 0.78
        ========================================================================
        ========================================================================
        episode: 40/61 | VALIDATION | treward: 247 | perf: 0.935 | eps: 0.70
        ========================================================================
        ========================================================================
        episode: 50/61 | VALIDATION | treward: 247 | perf: 0.890 | eps: 0.64
        ========================================================================
        ========================================================================
        episode: 60/61 | VALIDATION | treward: 247 | perf: 0.998 | eps: 0.58
        ========================================================================
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
episode: 61/61 | treward: 17 | perf: 0.979 | av: 475.1 | max: 1747
CPU times: user 51.4 s, sys: 2.53 s, total: 53.9 s
Wall time: 47 s
```

```
In [12]: tradingbot.plot_treward(agent)
```

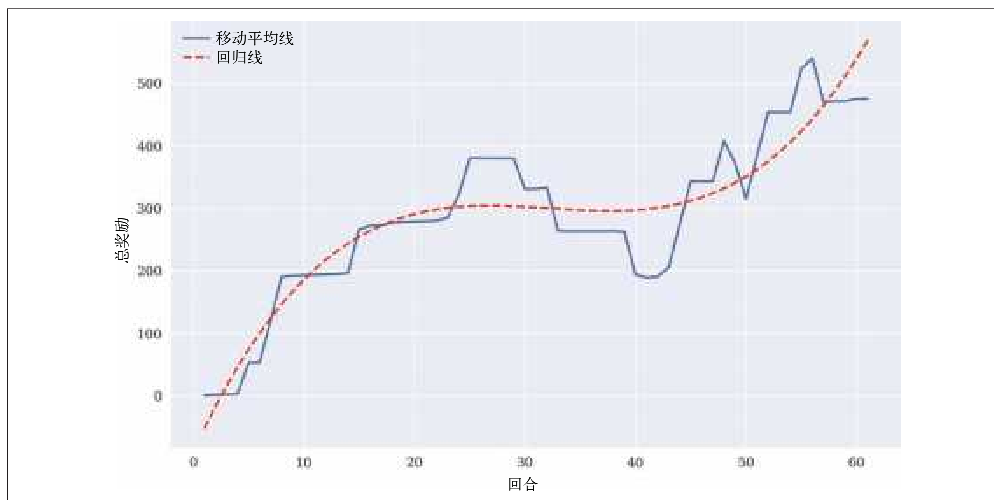


图 11-1：每个训练回合的平均总奖励

图 11-2 比较了交易机器人在训练数据集与验证数据集上的总体表现，由于在探索和利用间交替使用，训练数据集表现出了相当大的差异，验证数据集则仅用于利用。

```
In [13]: tradingbot.plot_performance(agent)
```

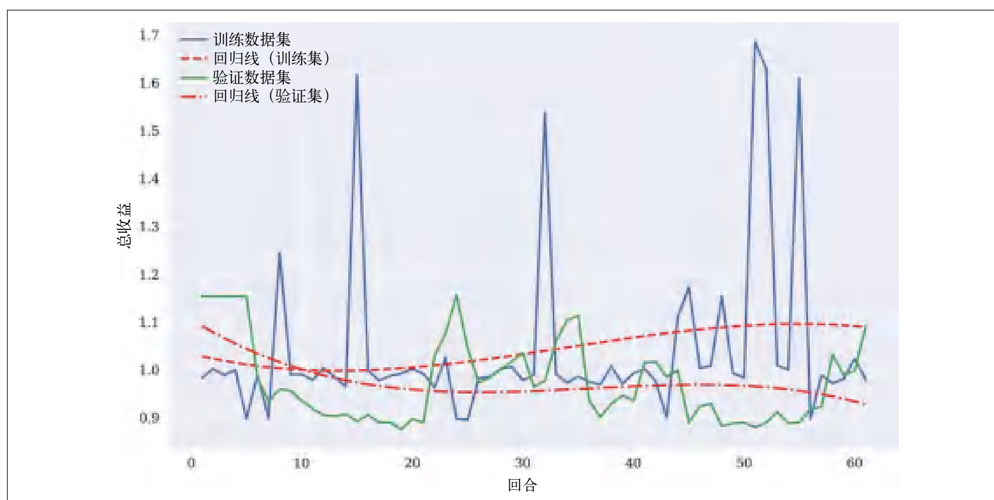


图 11-2：交易机器人在训练数据集和验证数据集上的总体表现

这个训练好的交易机器人在下一节中用于回测。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

11.2 向量化回测

向量化回测不能直接应用于交易机器人。第 10 章使用 DNN 说明过该方法。在这种情况下，首先准备好具有特征和标签子集的数据，然后将其输入 DNN 模型以立即生成所有预测。在强化学习环境中，数据是通过一步一步地与环境交互来生成和收集的。

为此，下面的 Python 代码定义了 `backtest` 函数，该函数将 `TradingBot` 实例和 `Finance` 实例作为输入。它在 `Finance` 环境列的原始 `Data Frame` 对象中生成了交易机器人所持有的头寸和最终策略收益。

```
In [14]: def reshape(s):
             return np.reshape(s, [1, learn_env.lags,
                                   learn_env.n_features])  ❶

In [15]: def backtest(agent, env):
             env.min_accuracy = 0.0
             env.min_performance = 0.0
             done = False
             env.data['p'] = 0  ❷
             state = env.reset()
             while not done:
                 action = np.argmax(
                     agent.model.predict(reshape(state))[0, 0])  ❸
                 position = 1 if action == 1 else -1  ❹
                 env.data.loc[:, 'p'].iloc[env.bar] = position  ❺
                 state, reward, done, info = env.step(action)
             env.data['s'] = env.data['p'] * env.data['r'] * learn_env.leverage  ❻
```

❶ 改变单个特性 – 标签组合的形式。

❷ 生成头寸值列。

❸ 给定训练好的 DNN 模型，得到最优动作（预测）。

❹ 导出最终头寸（+1 表示多头 / 向上，-1 表示空头 / 向下）……

❺ ……并存储在相应列的合适的索引位置。

❻ 计算给定头寸值策略的对数收益率。

配置了 `backtest` 函数后，向量化回测可以像第 10 章那样简化为几行 Python 代码。

图 11-3 比较了被动基准投资与交易机器人（策略投资）的总体表现。

```
In [16]: env = agent.learn_env  ❶

In [17]: backtest(agent, env)  ❷

In [18]: env.data['p'].iloc[env.lags:].value_counts()  ❸
Out[18]:  1    961
         -1    786
         Name: p, dtype: int64

In [19]: env.data[['r', 's']].iloc[env.lags:].sum().apply(np.exp)  ❹
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
Out[19]: r    0.7725
         s    1.5155
         dtype: float64

In [20]: env.data[['r', 's']].iloc[env.lags:].sum().apply(np.exp) - 1  ❺
Out[20]: r   -0.2275
         s    0.5155
         dtype: float64

In [21]: env.data[['r', 's']].iloc[env.lags:].cumsum(
                 ).apply(np.exp).plot(figsize=(10, 6));
```

❶ 指定相关环境。

❷ 生成所需的额外数据。

❸ 计算多头头寸和空头头寸的数量。

❹ 计算被动基准投资（r）和策略（s）投资的总收益……

❺ ……以及相应的净收益。

图 11-3：被动基准投资和交易机器人的总体表现（样本内）

为了更真实地了解交易机器人的性能，下面的 Python 代码使用交易机器人尚未看到的数据创建了一个测试环境。图 11-4 显示了与被动基准投资相比，交易机器人的表现情况。

```
In [22]: test_env = finance.Finance(symbol, features=learn_env.features,
                                    window=learn_env.window,
                                    lags=learn_env.lags,
                                    leverage=learn_env.leverage,
                                    min_performance=0.0, min_accuracy=0.0,
                                    start=a + b + c, end=None,
                                    mu=learn_env.mu, std=learn_env.std)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [23]: env = test_env

In [24]: backtest(agent, env)

In [25]: env.data['p'].iloc[env.lags:].value_counts()
Out[25]: -1    437
          1     56
         Name: p, dtype: int64

In [26]: env.data[['r', 's']].iloc[env.lags:].sum().apply(np.exp)
Out[26]: r   0.9144
         s   1.0992
         dtype: float64

In [27]: env.data[['r', 's']].iloc[env.lags:].sum().apply(np.exp) - 1
Out[27]: r   -0.0856
         s    0.0992
         dtype: float64

In [28]: env.data[['r', 's']].iloc[env.lags:].cumsum(
                     ).apply(np.exp).plot(figsize=(10, 6));
```

图 11-4：被动基准投资和交易机器人的总体表现（样本外）

在没有设置任何风控措施的情况下，样本外的表现似乎已经很不错了。然而，为了能够正确判断一个交易策略的真实表现，应该包括风控措施。这就是基于事件的回测发挥作用的地方。

11.3　基于事件的回测

鉴于上一节的结果，没有任何风控措施的样本外表现似乎已经很不错了。然而，为了能够正确分析风控措施（比如跟踪止损单），需要使用基于事件的回测，这也是本节将介绍的判断算法交易策略收益的另一种方法。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

11.7.3 节会介绍 `BacktestingBase` 类，它可以灵活地用于测试不同类型的方向性交易策略。代码的重要行上有详细的注释。此基类提供了以下方法。

`get_date_price()`

对于给定的 `bar`（包含金融数据的 `DataFrame` 对象的索引值），该方法会返回相关的 `date` 和 `price`。

`print_balance()`

对于给定的 `bar`，该方法会打印交易机器人的当前（现金）余额。

`calculate_net_wealth()`

对于给定的 `price`，该方法会返回由当前（现金）余额和工具头寸组成的净资产。

`print_net_wealth()`

对于给定的 `bar`，该方法会打印交易机器人的净资产。

`place_buy_order()`，`place_sell_order()`

对于给定的 `bar` 和给定的 `units` 数量或给定的 `amount`，这两种方法会发出买入单或卖出单，并相应地调整相关的数量（比如，计算交易成本）。

`close_out()`

在给定的 `bar` 上，该方法会关闭未平仓头寸，并计算和报告业绩统计数据。

下面的 Python 代码演示了 `BacktestingBase` 类函数的实例如何基于一些简单的步骤运行。

```
In [29]: import backtesting as bt

In [30]: bb = bt.BacktestingBase(env=agent.learn_env, model=agent.model,
                                 amount=10000, ptc=0.0001, ftc=1.0,
                                 verbose=True)  ❶

In [31]: bb.initial_amount  ❷
Out[31]: 10000

In [32]: bar = 100  ❸

In [33]: bb.get_date_price(bar)  ❹
Out[33]: ('2010-06-25', 1.2374)

In [34]: bb.env.get_state(bar)  ❺
Out[34]:                EUR=       r       s       m      v
         Date
         2010-06-22 -0.0242 -0.5622 -0.0916 -0.2022 1.5316
         2010-06-23  0.0176  0.6940 -0.0939 -0.0915 1.5563
         2010-06-24  0.0354  0.3034 -0.0865  0.6391 1.0890

In [35]: bb.place_buy_order(bar, amount=5000)  ❻
         2010-06-25 | buy 4040 units for 1.2374
         2010-06-25 | current balance = 4999.40

In [36]: bb.print_net_wealth(2 * bar)  ❼
         2010-11-16 | net wealth = 10450.17
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [37]: bb.place_sell_order(2 * bar, units=1000)  ❽
         2010-11-16 | sell 1000 units for 1.3492
         2010-11-16 | current balance = 6347.47

In [38]: bb.close_out(3 * bar)  ❾
         ==================================================
         2011-04-11 | *** CLOSING OUT ***
         2011-04-11 | sell 3040 units for 1.4434
         2011-04-11 | current balance = 10733.97
         2011-04-11 | net performance [%] = 7.3397
         2011-04-11 | number of trades [#] = 3
         ==================================================
```

❶ 实例化 BacktestingBase 对象。

❷ 查找 initial_amount 属性值。

❸ 固定 bar 值。

❹ 检索 bar 值的 date 值和 price 值。

❺ 检索 bar 值的 Finance 环境的状态。

❻ 使用 amount 参数下买入单。

❼ 打印稍后的时间点的净资产（2 * bar）。

❽ 使用 units 参数在稍后的时间点下卖出单。

❾ 了结剩余的多头头寸（3 * bar）。

继承自 BacktestingBase 类的 TBBacktester 类为交易机器人实现了基于事件的回测。

```
In [39]: class TBBacktester(bt.BacktestingBase):
             def _reshape(self, state):
                 ''' 用来对状态对象进行重塑的辅助函数
                 '''
                 return np.reshape(state, [1, self.env.lags, self.env.n_features])
             def backtest_strategy(self):
                 ''' 基于事件对交易机器人进行性能回测
                 '''
                 self.units = 0
                 self.position = 0
                 self.trades = 0
                 self.current_balance = self.initial_amount
                 self.net_wealths = list()
                 for bar in range(self.env.lags, len(self.env.data)):
                     date, price = self.get_date_price(bar)
                 if self.trades == 0:
                     print(50 * '=')
                     print(f'{date} | *** START BACKTEST ***')
                     self.print_balance(bar)
                     print(50 * '=')
                 state = self.env.get_state(bar)  ❶
                 action = np.argmax(self.model.predict(
                             self._reshape(state.values))[0, 0])  ❷
                 position = 1 if action == 1 else -1  ❸
                 if self.position in [0, -1] and position == 1:  ❹
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
                if self.verbose:
                    print(50 * '-')
                    print(f'{date} | *** GOING LONG ***')
                if self.position == -1:
                    self.place_buy_order(bar - 1, units=-self.units)
                self.place_buy_order(bar - 1,
                                     amount=self.current_balance)
                if self.verbose:
                    self.print_net_wealth(bar)
                self.position = 1
            elif self.position in [0, 1] and position == -1:  ❺
                if self.verbose:
                    print(50 * '-')
                    print(f'{date} | *** GOING SHORT ***')
                if self.position == 1:
                    self.place_sell_order(bar - 1, units=self.units)
                self.place_sell_order(bar - 1,
                                      amount=self.current_balance)
                if self.verbose:
                    self.print_net_wealth(bar)
                self.position = -1
            self.net_wealths.append((date,
                                     self.calculate_net_wealth(price)))  ❻
        self.net_wealths = pd.DataFrame(self.net_wealths,
                                        columns=['date', 'net_wealth'])  ❻
        self.net_wealths.set_index('date', inplace=True)  ❻
        self.net_wealths.index = pd.DatetimeIndex(
                                        self.net_wealths.index)  ❻
        self.close_out(bar)
```

❶ 检索 Finance 环境的状态。

❷ 根据状态和 model 对象生成最优操作（预测）。

❸ 根据最优操作（预测）得到最优头寸（多头 / 空头）。

❹ 如果满足条件，就进入**多头**头寸。

❺ 如果满足条件，就进入**空头**头寸。

❻ 收集随时间变化的净资产值，并将其转换为 DataFrame 对象。

TBBacktester 类的应用很简单，因为 Finance 实例和 TradingBot 实例已经可用。下面的代码首先在**学习环境**数据上对交易机器人进行回测，其中既包括无交易成本的数据也包括有交易成本的数据。图 11-5 直观地比较了这两种情况。

```
In [40]: env = learn_env

In [41]: tb = TBBacktester(env, agent.model, 10000,
                           0.0, 0, verbose=False)  ❶

In [42]: tb.backtest_strategy()  ❶
         ==================================================
         2010-02-05 | *** START BACKTEST ***
         2010-02-05 | current balance = 10000.00
         ==================================================
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        ==================================================
        2017-01-12 | *** CLOSING OUT ***
        2017-01-12 | current balance = 14601.85
        2017-01-12 | net performance [%] = 46.0185
        2017-01-12 | number of trades [#] = 828
        ==================================================

In [43]: tb_ = TBBacktester(env, agent.model, 10000,
                            0.00012, 0.0, verbose=False)

In [44]: tb_.backtest_strategy()  ❷
        ==================================================
        2010-02-05 | *** START BACKTEST ***
        2010-02-05 | current balance = 10000.00
        ==================================================
        ==================================================
        2017-01-12 | *** CLOSING OUT ***
        2017-01-12 | current balance = 13222.08
        2017-01-12 | net performance [%] = 32.2208
        2017-01-12 | number of trades [#] = 828
        ==================================================

In [45]: ax = tb.net_wealths.plot(figsize=(10, 6))
         tb_.net_wealths.columns = ['net_wealth (after tc)']
         tb_.net_wealths.plot(ax=ax);
```

❶ 无交易成本基于事件的样本内回测。

❷ 有交易成本基于事件的样本内回测。

图 11-5：交易机器人在有交易成本和无交易成本两种情况下的总体表现（样本内）

图 11-6 再次对包含交易成本前后**测试环境**数据的交易机器人的总体表现进行了比较。

```
In [46]: env = test_env

In [47]: tb = TBBacktester(env, agent.model, 10000,
                           0.0, 0, verbose=False)  ❶
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [48]: tb.backtest_strategy()  ❶
         ==================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ==================================================
         ==================================================
         2019-12-31 | *** CLOSING OUT ***
         2019-12-31 | current balance = 10936.79
         2019-12-31 | net performance [%] = 9.3679
         2019-12-31 | number of trades [#] = 186
         ==================================================

In [49]: tb_ = TBBacktester(env, agent.model, 10000,
                            0.00012, 0.0, verbose=False)

In [50]: tb_.backtest_strategy()  ❷
         ==================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ==================================================
         ==================================================
         2019-12-31 | *** CLOSING OUT ***
         2019-12-31 | current balance = 10695.72
         2019-12-31 | net performance [%] = 6.9572
         2019-12-31 | number of trades [#] = 186
         ==================================================

In [51]: ax = tb.net_wealths.plot(figsize=(10, 6))
         tb_.net_wealths.columns = ['net_wealth (after tc)']
         tb_.net_wealths.plot(ax=ax);
```

❶ 无交易成本基于事件的样本外回测。

❷ 有交易成本基于事件的样本外回测。

图 11-6：交易机器人在有交易成本和无交易成本两种情况下的总体表现（样本外）

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

在考虑交易成本前，如何比较基于事件的回测性能与向量化回测性能？图 11-7 对比了一段时间内标准化的净资产与总收益间的关系。由于所采用的技术方法不同，因此这两个时间序列并不完全相同，但非常相似。性能差异主要可以解释为基于事件的回测对每个头寸假定的数量相同。向量化回测考虑了复合效应，导致报告的性能略有提高。

```
In [52]: ax = (tb.net_wealths / tb.net_wealths.iloc[0]).plot(figsize=(10, 6))
         tp = env.data[['r', 's']].iloc[env.lags:].cumsum().apply(np.exp)
         (tp / tp.iloc[0]).plot(ax=ax);
```

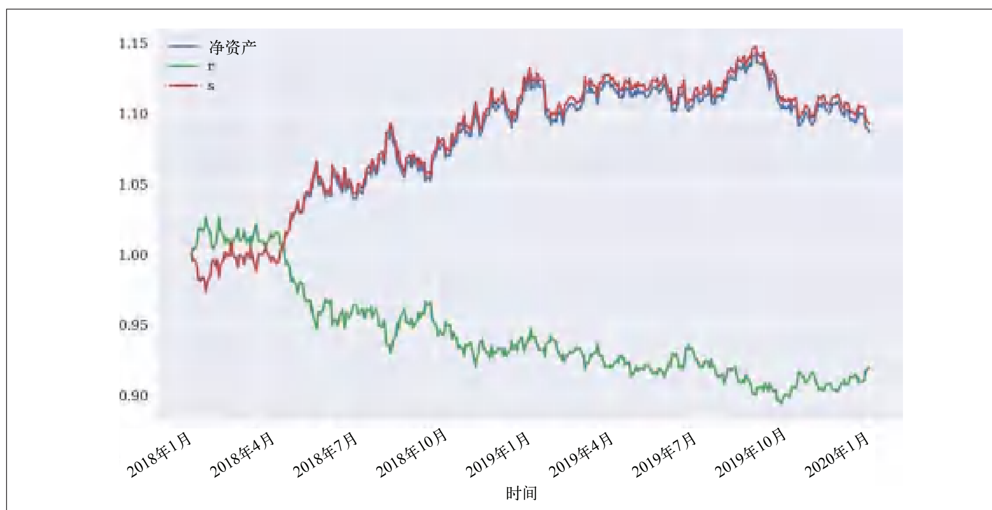


图 11-7：在向量化回测和基于事件的回测两种情况下，被动基准投资和交易机器人的总体表现

性能差异

向量化回测的性能数据和基于事件的回测的性能数据非常接近，但并不完全相同。在前一种情况下，假设金融工具是完全可分割的，复利也是连续进行的。在后一种情况下，只接受金融工具的完整单位进行交易，这更接近现实。净资产的计算是基于价格差异，例如，其所使用的基于事件的代码不会检查当前余额是否足够大，以用现金支付某笔交易。这无疑是一个简化的假设，例如，保证金购买可能并不总是可行。这方面的代码调整可以轻松地添加到 `BacktestingBase` 类中。

11.4　风险评估

风控措施的应用需要了解交易所选择的金融工具所涉及的风险。因此，为了合理设置止损单等风控措施参数，对标的工具的风险进行评估非常重要。有很多方法可以用来衡量金融工具的风险，既有**非方向性的风控措施**，比如波动率或 ATR；也有**方向性的风控措施**，比如最大回撤或风险价值（VaR）。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

当设定止损（SL）单、跟踪止损（TSL）单或止盈（TP）单的目标水平时，通常的做法是将这些水平与 ATR 关联起来。[1] 下面的 Python 代码会以绝对和相对的方式计算金融工具的 ATR，交易机器人会在该金融工具上进行训练和回测（欧元 / 美元汇率）。该计算依赖于来自学习环境的数据，并使用 14 天（条）这种典型窗口长度。图 11-8 显示了计算值，这些值随时间显著变化。

```
In [53]: data = pd.DataFrame(learn_env.data[symbol])  ❶

In [54]: data.head()  ❶
Out[54]:               EUR=
         Date
         2010-02-02 1.3961
         2010-02-03 1.3898
         2010-02-04 1.3734
         2010-02-05 1.3662
         2010-02-08 1.3652

In [55]: window = 14  ❷

In [56]: data['min'] = data[symbol].rolling(window).min()  ❸

In [57]: data['max'] = data[symbol].rolling(window).max()  ❹

In [58]: data['mami'] = data['max'] - data['min']  ❺

In [59]: data['mac'] = abs(data['max'] - data[symbol].shift(1))  ❻

In [60]: data['mic'] = abs(data['min'] - data[symbol].shift(1))  ❼

In [61]: data['atr'] = np.maximum(data['mami'], data['mac'])  ❽

In [62]: data['atr'] = np.maximum(data['atr'], data['mic'])  ❾

In [63]: data['atr%'] = data['atr'] / data[symbol]  ❿

In [64]: data[['atr', 'atr%']].plot(subplots=True, figsize=(10, 6));
```

❶ 原始 `DataFrame` 对象中的价格序列。

❷ 用于计算的窗口长度。

❸ 滚动最小值。

❹ 滚动最大值。

❺ 滚动最大值与最小值之差。

❻ 滚动最大值与前一天价格的绝对差值。

❼ 滚动最小值与前一天价格的绝对差值。

❽ 最大 / 最小差价与最大价差中的最大值。

注 1：有关 ATR 度量的详细信息，请参阅 ATR(1)Investopedia 或 ATR(2)Investopedia。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❾ 上一个最大值和最小价差之间的最大值（=ATR）。

❿ ATR 的绝对值和价格的百分比。

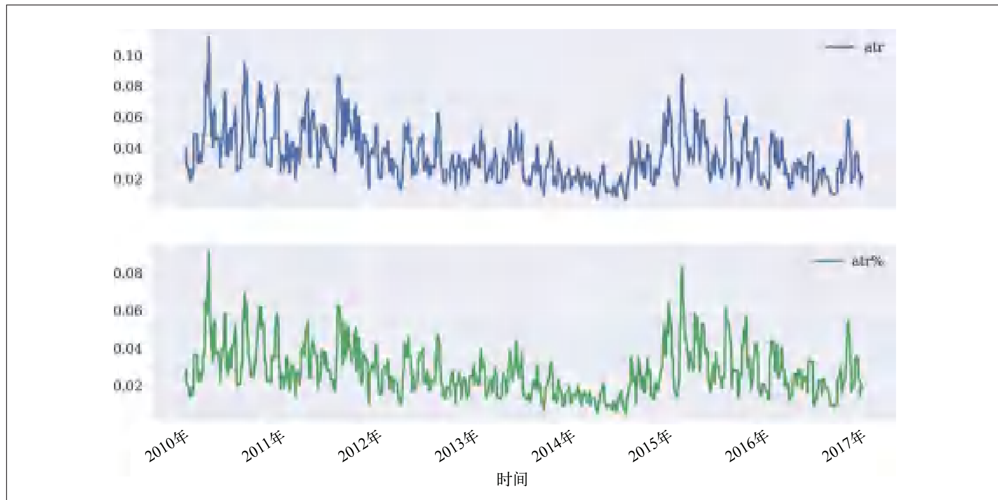


图 11-8：以绝对值（价格）和相对值（%）形式表示的 ATR

下面的代码以绝对值和相对值的形式显示了 ATR 的最终值。例如，一个典型的规则是设置止损水平为进入价格减去 x 倍 ATR。根据交易员或投资者的风险偏好，x 可能小于或大于 1。这就是人类判断或正式风险政策发挥作用的地方。如果 $x = 1$，则将止损水平设置为低于入门水平的 2% 左右。

```
In [65]: data[['atr', 'atr%']].tail()
Out[65]:                atr   atr%
         Date
         2017-01-06 0.0218 0.0207
         2017-01-09 0.0218 0.0206
         2017-01-10 0.0218 0.0207
         2017-01-11 0.0199 0.0188
         2017-01-12 0.0206 0.0194
```

然而，**杠杆**在这种情况下扮演着重要的角色。如果使用的杠杆率为 10（这对外汇交易来说实际上是很低的），那么 ATR 数字需要乘以杠杆率。因此，对于假定的 ATR 系数 1，与之前相同的止损水平将被设定为 20% 左右，而不是仅仅 2%。或者取整个数据集的 ATR 中值，设定为 25% 左右。

```
In [66]: leverage = 10

In [67]: data[['atr', 'atr%']].tail() * leverage
Out[67]:                atr   atr%
         Date
         2017-01-06 0.2180 0.2070
         2017-01-09 0.2180 0.2062
         2017-01-10 0.2180 0.2066
```


专为量化交易设计的实时行情数据API: www.alltick.co

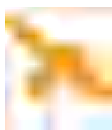
专为量化交易设计的实时行情数据API: www.alltick.co

```
        2017-01-11 0.1990 0.1881
        2017-01-12 0.2060 0.1942

In [68]: data[['atr', 'atr%']].median() * leverage
Out[68]: atr     0.3180
         atr%    0.2481
         dtype: float64
```

将止损水平或止盈水平与 ATR 联系起来的基本理念是，应避免将它们设置得过低或过高。考虑一个杠杆率为 10 倍、ATR 为 20% 的头寸。仅将止损水平设置为 3% 或 5% 可能会降低头寸的金融风险，但它会引入过早发生的止损风险，这是由于金融工具的典型波动造成的。这种在一定范围内的“典型波动”通常被称为**噪声**。一般而言，止损单应保护市场免受比典型价格波动（噪声）更大的不利市场波动的影响。

对于获利回吐水平也是如此。如果将它设得太高（比如是 ATR 水平的 3 倍），则无法获得可观的利润，头寸可能会保持太长时间，直到它们放弃以前的利润。即使在这种情况下可以使用正式的分析和数学公式，但设定这样的目标水平更多是为了艺术而不是科学。在金融环境中，设定目标水平有相当大的自由度，而人类的判断可以起到拯救的作用。在其他情况下，比如自动驾驶汽车，这是不同的，因为不需要人类的判断来指示人工智能避免与人类的任何碰撞。



非正态性与非线性

当保证金或投资权益用完时，**保证金止损**会结束交易头寸。假设一个有保证金止损的杠杆交易头寸，例如，杠杆率为 10 时，保证金是 10% 的权益。交易工具中 10% 或更大的**不利**变动会吞噬所有的权益，并触发头寸的平仓——损失 100% 的权益。标的资产**有利**的变动，比如 25% 的变动，就会带来 150% 的权益收益率。即使交易工具的收益率是正态分布的，杠杆和保证金止损也会导致收益率的非正态分布以及交易工具与交易头寸间的非对称、非线性关系。

11.5 风控措施回测

了解金融工具的 ATR 通常是实施风控措施的良好开端。为了正确回测典型的风控订单的效果，对 `BacktestingBase` 类进行一些调整是有帮助的。下面的 Python 代码提供了一个新的基类 `BacktestBaseRM`，它继承自 `BacktestingBase`，有助于跟踪前一次交易的进入价格以及自那次交易以来的最高价格和最低价格。这些值用于计算止损单、跟踪止损单和止盈单所涉及的基于事件的回测性能。

```
#
# 基于事件的回测
# --基类 (2)
#
# (c) Dr. Yves J. Hilpisch
#
from backtesting import *
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
class BacktestingBaseRM(BacktestingBase):

    def set_prices(self, price):
        ''' 设置用来进行性能回测的价格，比如测试跟踪止损是否命中
        '''
        self.entry_price = price  ❶
        self.min_price = price  ❷
        self.max_price = price  ❸

    def place_buy_order(self, bar, amount=None, units=None, gprice=None):
        ''' 对于给定的bar和给定的amount或给定的units数量下一个买入单
        '''
        date, price = self.get_date_price(bar)
        if gprice is not None:
            price = gprice
        if units is None:
            units = int(amount / price)
        self.current_balance -= (1 + self.ptc) * units * price + self.ftc
        self.units += units
        self.trades += 1
        self.set_prices(price)  ❹
        if self.verbose:
            print(f'{date} | buy {units} units for {price:.4f}')
            self.print_balance(bar)

    def place_sell_order(self, bar, amount=None, units=None, gprice=None):
        ''' 对于给定的bar和给定的amount或给定的units数量下一个卖出单
        '''
        date, price = self.get_date_price(bar)
        if gprice is not None:
            price = gprice
        if units is None:
            units = int(amount / price)
        self.current_balance += (1 - self.ptc) * units * price - self.ftc
        self.units -= units
        self.trades += 1
        self.set_prices(price)  ❹
        if self.verbose:
            print(f'{date} | sell {units} units for {price:.4f}')
            self.print_balance(bar)
```

❶ 设置最近交易的**进入**价格。

❷ 设置自最近交易以来的初始**最低**价格。

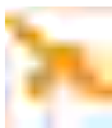
❸ 设置自最近交易以来的初始**最高**价格。

❹ 设置交易执行后的相关价格。

基于这个新的基类，11.7.4 节提供了一个新的回测类 `TBBacktesterRM`，它允许包含止损单、跟踪止损单和止盈单。相关代码部分将在接下来的内容中进行讨论。如上一节所计算的，回测示例的参数将定位在大约 2% 的 ATR 水平上。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co



EUT 和风控措施

EUT、MVP 和 CAPM（参见第 3 章和第 4 章）假设金融主体知道金融工具收益率的未来分布。MPT 和 CAPM 进一步假设收益率是呈正态分布的，并且在市场投资组合的收益率和交易的金融工具的收益率之间存在线性关系。止损单、跟踪止损单和止盈单的使用除了会导致杠杆作用外，还会导致杠杆作用与保证金止损相结合形成“有保证的非正态”分布，以及交易头寸相对于交易工具的高度不对称、非线性回报。

11.5.1 止损

第一个风控措施是止损单。它固定了一个特定的价格水平，或者更常见的是，固定了能够触发平仓的百分比值。如果一个非杠杆头寸的进入价格为 100，止损水平被设置为 5%，则多头头寸会在 95 处平仓，而空头头寸会在 105 处平仓。

下面的 Python 代码是处理止损单的 `TBBacktesterRM` 类的相关部分。对止损单来说，该类允许用户指定订单的价格水平是否有保证。[2] 使用有保证的止损价格水平可能会导致过于乐观的性能结果。

```
# 止损单
if sl is not None and self.position != 0:  ❶
    rc = (price - self.entry_price) / self.entry_price  ❷
    if self.position == 1 and rc < -self.sl:  ❸
        print(50 * '-')
        if guarantee:
            price = self.entry_price * (1 - self.sl)
            print(f'*** STOP LOSS (LONG  | {-self.sl:.4f}) ***')
        else:
            print(f'*** STOP LOSS (LONG  | {rc:.4f}) ***')
        self.place_sell_order(bar, units=self.units, gprice=price)  ❹
        self.wait = wait  ❺
        self.position = 0  ❻
    elif self.position == -1 and rc > self.sl:  ❼
        print(50 * '-')
        if guarantee:
            price = self.entry_price * (1 + self.sl)
            print(f'*** STOP LOSS (SHORT | -{self.sl:.4f}) ***')
        else:
            print(f'*** STOP LOSS (SHORT | -{rc:.4f}) ***')
        self.place_buy_order(bar, units=-self.units, gprice=price)  ❽
        self.wait = wait  ❺
        self.position = 0  ❻
```

❶ 检查是否定义了止损，头寸是否为中性。

❷ 根据最后一笔交易的进入价格计算收益。

❸ 检查是否给定了一个**多头**头寸的止损事件。

注 2：有担保的止损单可能只适用于某些司法管辖区的某些经纪客户群体，比如散户投资者 / 交易员。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❹ 以当前价格或保证价格平仓**多头**头寸。

❺ 设置在下一个交易发生之前等待的条数。

❻ 设置头寸为中性。

❼ 检查是否为**空头**头寸给出了一个止损事件。

❽ 以当前价格或保证价格平仓**空头**头寸。

下面的 Python 代码分别对没有止损单和有止损单的交易机器人的交易策略进行了回测。对于给定的参数，止损单对策略性能有负面影响。

```
In [69]: import tbbacktesterrm as tbbrm

In [70]: env = test_env

In [71]: tb = tbbrm.TBBacktesterRM(env, agent.model, 10000,
                                   0.0, 0, verbose=False)  ❶

In [72]: tb.backtest_strategy(sl=None, tsl=None, tp=None, wait=5)  ❷
         ==================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ==================================================
         ==================================================
         2019-12-31 | *** CLOSING OUT ***
         2019-12-31 | current balance = 10936.79
         2019-12-31 | net performance [%] = 9.3679
         2019-12-31 | number of trades [#] = 186
         ==================================================

In [73]: tb.backtest_strategy(sl=0.0175, tsl=None, tp=None,
                              wait=5, guarantee=False)  ❸
         ==================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ==================================================
         --------------------------------------------------
         *** STOP LOSS (SHORT | -0.0203) ***
         ==================================================
         2019-12-31 | *** CLOSING OUT ***
         2019-12-31 | current balance = 10717.32
         2019-12-31 | net performance [%] = 7.1732
         2019-12-31 | number of trades [#] = 188
         ==================================================

In [74]: tb.backtest_strategy(sl=0.017, tsl=None, tp=None,
                              wait=5, guarantee=True)  ❹
         ==================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ==================================================
         --------------------------------------------------
         *** STOP LOSS (SHORT | -0.0170) ***
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
==================================================
2019-12-31 | *** CLOSING OUT ***
2019-12-31 | current balance = 10753.52
2019-12-31 | net performance [%] = 7.5352
2019-12-31 | number of trades [#] = 188
==================================================
```

❶ 实例化风险管理的回测类。

❷ 在没有任何风控措施的情况下，对交易机器人的性能进行回测。

❸ 使用止损单（无保证金）对交易机器人的性能进行回测。

❹ 使用止损单（有保证金）对交易机器人的性能进行回测。

11.5.2 跟踪止损

与常规的止损单不同，只要在基本指令下达后观察到新的高点，跟踪止损单就会进行调整。假设无杠杆多头头寸的基础订单的进入价格为 95，跟踪止损被设置为 5%。如果工具价格达到 100 并回落到 95，则意味着这是一个跟踪止损事件，头寸会在进入价格水平结束。如果价格达到 110，并回落到 104.5，则意味着这是另一个跟踪止损事件。

下面的 Python 代码是处理跟踪止损单的 `TBBacktesterRM` 类的相关部分。要正确处理这样的风控措施，需要跟踪最高价格和最低价格。最高价格适用于多头头寸，最低价格则适用于空头头寸。

```
# 跟踪止损单
if tsl is not None and self.position != 0:
    self.max_price = max(self.max_price, price)  ❶
    self.min_price = min(self.min_price, price)  ❷
    rc_1 = (price - self.max_price) / self.entry_price  ❸
    rc_2 = (self.min_price - price) / self.entry_price  ❹
    if self.position == 1 and rc_1 < -self.tsl:  ❺
        print(50 * '-')
        print(f'*** TRAILING SL (LONG | {rc_1:.4f}) ***')
        self.place_sell_order(bar, units=self.units)
        self.wait = wait
        self.position = 0
    elif self.position == -1 and rc_2 < -self.tsl:  ❻
        print(50 * '-')
        print(f'*** TRAILING SL (SHORT | {rc_2:.4f}) ***')
        self.place_buy_order(bar, units=-self.units)
        self.wait = wait
        self.position = 0
```

❶ 如有必要，就更新**最高**价格。

❷ 如有必要，就更新**最低**价格。

❸ 计算**多头**头寸的相关收益。

❹ 计算**空头**头寸的相关收益。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❺ 检查是否为**多头**头寸给出了一个跟踪止损事件。

❻ 检查是否为**空头**头寸给出了一个跟踪止损事件。

正如下面的回测结果所示，与没有跟踪止损单的策略相比，在给定参数的情况下使用跟踪止损单会降低总体性能。

```
In [75]: tb.backtest_strategy(sl=None, tsl=0.015,
                              tp=None, wait=5)  ❶
         ==================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ==================================================
         --------------------------------------------------
         *** TRAILING SL (SHORT | -0.0152) ***
         --------------------------------------------------
         *** TRAILING SL (SHORT | -0.0169) ***
         --------------------------------------------------
         *** TRAILING SL (SHORT | -0.0164) ***
         --------------------------------------------------
         *** TRAILING SL (SHORT | -0.0191) ***
         --------------------------------------------------
         *** TRAILING SL (SHORT | -0.0166) ***
         --------------------------------------------------
         *** TRAILING SL (SHORT | -0.0194) ***
         --------------------------------------------------
         *** TRAILING SL (SHORT | -0.0172) ***
         --------------------------------------------------
         *** TRAILING SL (SHORT | -0.0181) ***
         --------------------------------------------------
         *** TRAILING SL (SHORT | -0.0153) ***
         --------------------------------------------------
         *** TRAILING SL (SHORT | -0.0160) ***
         ==================================================
         2019-12-31 | *** CLOSING OUT ***
         2019-12-31 | current balance = 10577.93
         2019-12-31 | net performance [%] = 5.7793
         2019-12-31 | number of trades [#] = 201
         ==================================================
```

❶ 用跟踪止损单对交易机器人的性能进行回测。

11.5.3 止盈

最后是止盈单。一个止盈单将平仓达到一定利润水平的头寸。假设一个非杠杆多头头寸以 100 的价格开仓，止盈单被设置为 5% 的水平。如果价格达到 105，则平仓。

下面 `TBBacktesterRM` 类的代码最终显示了处理止盈单的部分。给定止损单和跟踪止损单代码的引用，止盈实现是很简单的。对于止盈单，可以选择使用与相关高 / 低价格水平相比的保证价格水平进行回测，但这很可能导致性能值过于乐观。[3]

注 3：止盈单有一个固定的目标价格水平。因此，用一个时间间隔的高价来计算多头头寸或用这个时间间隔的低价来计算空头头寸是不现实的。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
# 止盈单
if tp is not None and self.position != 0:
    rc = (price - self.entry_price) / self.entry_price
    if self.position == 1 and rc > self.tp:
        print(50 * '-')
        if guarantee:
            price = self.entry_price * (1 + self.tp)
            print(f'*** TAKE PROFIT (LONG | {self.tp:.4f}) ***')
        else:
            print(f'*** TAKE PROFIT (LONG | {rc:.4f}) ***')
        self.place_sell_order(bar, units=self.units, gprice=price)
        self.wait = wait
        self.position = 0
    elif self.position == -1 and rc < -self.tp:
        print(50 * '-')
        if guarantee:
            price = self.entry_price * (1 - self.tp)
            print(f'*** TAKE PROFIT (SHORT | {self.tp:.4f}) ***')
    else:
        print(f'*** TAKE PROFIT (SHORT | {-rc:.4f}) ***')
    self.place_buy_order(bar, units=-self.units, gprice=price)
    self.wait = wait
    self.position = 0
```

与被动基准投资相比，在给定的参数条件下，增加一个无保证金的止盈单可以显著提高交易机器人的性能。考虑到之前必须考虑的事，这个结果可能过于乐观了。因此，在这种情况下，带保证金的止盈单使其性能值更加现实。

```
In [76]: tb.backtest_strategy(sl=None, tsl=None, tp=0.015,
                              wait=5, guarantee=False)  ❶
         ==================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ==================================================
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0155) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0155) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0204) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0240) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0168) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0156) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0183) ***
         ==================================================
         2019-12-31 | *** CLOSING OUT ***
         2019-12-31 | current balance = 11210.33
         2019-12-31 | net performance [%] = 12.1033
         2019-12-31 | number of trades [#] = 198
         ==================================================
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [77]: tb.backtest_strategy(sl=None, tsl=None, tp=0.015,
                              wait=5, guarantee=True)  ❷
         ==================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ==================================================
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0150) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0150) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0150) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0150) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0150) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0150) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0150) ***
         ==================================================
         2019-12-31 | *** CLOSING OUT ***
         2019-12-31 | current balance = 10980.86
         2019-12-31 | net performance [%] = 9.8086
         2019-12-31 | number of trades [#] = 198
         ==================================================
```

❶ 用一个止盈单对交易机器人的性能进行回测（无保证金）。

❷ 用一个止盈单对交易机器人的性能进行回测（有保证金）。

当然，止损单 / 跟踪止损单也可以和止盈单合并。下面的 Python 代码的回测结果在这两种情况下都比没有风控措施的策略结果更糟糕。在风险管理方面，几乎没有“免费的午餐”。

```
In [78]: tb.backtest_strategy(sl=0.015, tsl=None,
                              tp=0.0185, wait=5)  ❶
         ==================================================
         2018-01-17 | *** START BACKTEST ***
         2018-01-17 | current balance = 10000.00
         ==================================================
         --------------------------------------------------
         *** STOP LOSS (SHORT | -0.0203) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0202) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0213) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0240) ***
         --------------------------------------------------
         *** STOP LOSS (SHORT | -0.0171) ***
         --------------------------------------------------
         *** TAKE PROFIT (SHORT | 0.0188) ***
         --------------------------------------------------
         *** STOP LOSS (SHORT | -0.0153) ***
         --------------------------------------------------
         *** STOP LOSS (SHORT | -0.0154) ***
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
==================================================
2019-12-31 | *** CLOSING OUT ***
2019-12-31 | current balance = 10552.00
2019-12-31 | net performance [%] = 5.5200
2019-12-31 | number of trades [#] = 201
==================================================

In [79]: tb.backtest_strategy(sl=None, tsl=0.02,
                              tp=0.02, wait=5)  ❷
==================================================
2018-01-17 | *** START BACKTEST ***
2018-01-17 | current balance = 10000.00
==================================================
--------------------------------------------------
*** TRAILING SL (SHORT | -0.0235) ***
--------------------------------------------------
*** TRAILING SL (SHORT | -0.0202) ***
--------------------------------------------------
*** TAKE PROFIT (SHORT | 0.0250) ***
--------------------------------------------------
*** TAKE PROFIT (SHORT | 0.0227) ***
--------------------------------------------------
*** TAKE PROFIT (SHORT | 0.0240) ***
--------------------------------------------------
*** TRAILING SL (SHORT | -0.0216) ***
--------------------------------------------------
*** TAKE PROFIT (SHORT | 0.0241) ***
--------------------------------------------------
*** TRAILING SL (SHORT | -0.0206) ***
==================================================
2019-12-31 | *** CLOSING OUT ***
2019-12-31 | current balance = 10346.38
2019-12-31 | net performance [%] = 3.4638
2019-12-31 | number of trades [#] = 198
==================================================
```

❶ 使用止损单和止盈单对交易机器人的性能进行回测。

❷ 使用跟踪止损单和止盈单对交易机器人的性能进行回测。

性能影响

风控措施有其合理性和好处。然而，降低风险的代价可能是整体业绩下滑。另外，使用止盈单的回测例子显示了业绩的改善，这可以解释为，给定金融工具的 ATR，某一利润水平是能够实现的。但任何希望看到更高利润的愿望，通常都会在市场再次扭转时破灭。

11.6 结论

本章有 3 个主题。首先以向量化和基于事件的方式对样本外交易机器人（训练有素的深度 Q 学习智能体）的性能进行了回测。然后以 ATR 指标的形式评估了风险，该指标衡量了利率金融工具价格的**典型**变化。最后，本章讨论并回测了基于事件的典型风控措施的止损

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

单、跟踪止损单和止盈单形式。

与自动驾驶汽车类似，交易机器人很少只基于人工智能的预测来部署。为了避免巨大的下行风险并提高（风险调整后的）业绩，风控措施通常会发挥作用。本章所讨论的标准风控措施既适用于大多数交易平台，也适用于散户交易者。第 12 章将以 Oanda 交易平台为例说明这一点。基于事件的回测方法提供了算法上的灵活性，可以正确地回测此类风控措施的效果。虽然“降低风险”听起来很吸引人，但回测结果表明，降低风险往往是有代价的：与没有任何风控措施的纯策略相比，性能可能更低。然而，当进行精细调试时，结果显示，像止盈单这样的风控措施也可以对性能产生积极影响。

11.7 Python代码

11.7.1 金融环境

下面是带有 Finance 环境类的 Python 模块。

```
#
# 金融环境
#
# (c) Dr. Yves J. Hilpisch
# Artificial Intelligence in Finance
#
import math
import random
import numpy as np
import pandas as pd


class observation_space:
    def __init__(self, n):
        self.shape = (n,)


class action_space:
    def __init__(self, n):
        self.n = n

    def sample(self):
        return random.randint(0, self.n - 1)


class Finance:
    intraday = False
    if intraday:
        url = 'http://hilpisch.com/aiif_eikon_id_eur_usd.csv'
    else:
        url = 'http://hilpisch.com/aiif_eikon_eod_data.csv'

    def __init__(self, symbol, features, window, lags,
                 leverage=1, min_performance=0.85, min_accuracy=0.5,
                 start=0, end=None, mu=None, std=None):
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        self.symbol = symbol
        self.features = features
        self.n_features = len(features)
        self.window = window
        self.lags = lags
        self.leverage = leverage
        self.min_performance = min_performance
        self.min_accuracy = min_accuracy
        self.start = start
        self.end = end
        self.mu = mu
        self.std = std
        self.observation_space = observation_space(self.lags)
        self.action_space = action_space(2)
        self._get_data()
        self._prepare_data()

    def _get_data(self):
        self.raw = pd.read_csv(self.url, index_col=0,
                               parse_dates=True).dropna()
        if self.intraday:
            self.raw = self.raw.resample('30min', label='right').last()
            self.raw = pd.DataFrame(self.raw['CLOSE'])
            self.raw.columns = [self.symbol]

    def _prepare_data(self):
        self.data = pd.DataFrame(self.raw[self.symbol])
        self.data = self.data.iloc[self.start:]
        self.data['r'] = np.log(self.data / self.data.shift(1))
        self.data.dropna(inplace=True)
        self.data['s'] = self.data[self.symbol].rolling(self.window).mean()
        self.data['m'] = self.data['r'].rolling(self.window).mean()
        self.data['v'] = self.data['r'].rolling(self.window).std()
        self.data.dropna(inplace=True)
        if self.mu is None:
            self.mu = self.data.mean()
            self.std = self.data.std()
        self.data_ = (self.data - self.mu) / self.std
        self.data['d'] = np.where(self.data['r'] > 0, 1, 0)
        self.data['d'] = self.data['d'].astype(int)
        if self.end is not None:
            self.data = self.data.iloc[:self.end - self.start]
            self.data_ = self.data_.iloc[:self.end - self.start]

    def _get_state(self):
        return self.data_[self.features].iloc[self.bar -
                                              self.lags:self.bar]

    def get_state(self, bar):
        return self.data_[self.features].iloc[bar - self.lags:bar]

    def seed(self, seed):
        random.seed(seed)
        np.random.seed(seed)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
    def reset(self):
        self.treward = 0
        self.accuracy = 0
        self.performance = 1
        self.bar = self.lags
        state = self.data_[self.features].iloc[self.bar -
                                               self.lags:self.bar]
        return state.values

    def step(self, action):
        correct = action == self.data['d'].iloc[self.bar]
        ret = self.data['r'].iloc[self.bar] * self.leverage
        reward_1 = 1 if correct else 0
        reward_2 = abs(ret) if correct else -abs(ret)
        self.treward += reward_1
        self.bar += 1
        self.accuracy = self.treward / (self.bar - self.lags)
        self.performance *= math.exp(reward_2)
        if self.bar >= len(self.data):
            done = True
        elif reward_1 == 1:
            done = False
        elif (self.performance < self.min_performance and
              self.bar > self.lags + 15):
            done = True
        elif (self.accuracy < self.min_accuracy and
              self.bar > self.lags + 15):
            done = True
        else:
            done = False
        state = self._get_state()
        info = {}
        return state.values, reward_1 + reward_2 * 5, done, info
```

11.7.2 交易机器人

下面是带有 `TradingBot` 类的 Python 模块，它基于一个 Q 学习金融智能体。

```
#
# Q学习金融智能体
#
# (c) Dr. Yves J. Hilpisch
# Artificial Intelligence in Finance
#
import os
import random
import numpy as np
from pylab import plt, mpl
from collections import deque
import tensorflow as tf
from keras.layers import Dense, Dropout
from keras.models import Sequential
from keras.optimizers import Adam, RMSprop
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
os.environ['PYTHONHASHSEED'] = '0'
plt.style.use('seaborn')
mpl.rcParams['savefig.dpi'] = 300
mpl.rcParams['font.family'] = 'serif'
def set_seeds(seed=100):
    ''' 为所有的随机数生成器设置种子
    '''
    random.seed(seed)
    np.random.seed(seed)
    tf.random.set_seed(seed)


class TradingBot:
    def __init__(self, hidden_units, learning_rate, learn_env,
                 valid_env=None, val=True, dropout=False):
        self.learn_env = learn_env
        self.valid_env = valid_env
        self.val = val
        self.epsilon = 1.0
        self.epsilon_min = 0.1
        self.epsilon_decay = 0.99
        self.learning_rate = learning_rate
        self.gamma = 0.5
        self.batch_size = 128
        self.max_treward = 0
        self.averages = list()
        self.trewards = []
        self.performances = list()
        self.aperformances = list()
        self.vperformances = list()
        self.memory = deque(maxlen=2000)
        self.model = self._build_model(hidden_units,
                             learning_rate, dropout)

    def _build_model(self, hu, lr, dropout):
        ''' 创建DNN模型
        '''
        model = Sequential()
        model.add(Dense(hu, input_shape=(
            self.learn_env.lags, self.learn_env.n_features),
            activation='relu'))
        if dropout:
            model.add(Dropout(0.3, seed=100))
        model.add(Dense(hu, activation='relu'))
        if dropout:
            model.add(Dropout(0.3, seed=100))
        model.add(Dense(2, activation='linear'))
        model.compile(
            loss='mse',
            optimizer=RMSprop(lr=lr)
        )
        return model

    def act(self, state):
        ''' 基于探索或者利用而选择不同的动作
        '''
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
    if random.random() <= self.epsilon:
        return self.learn_env.action_space.sample()
    action = self.model.predict(state)[0, 0]
    return np.argmax(action)

def replay(self):
    ''' 基于存储的经验重新训练DNN模型
    '''
    batch = random.sample(self.memory, self.batch_size)
    for state, action, reward, next_state, done in batch:
        if not done:
            reward += self.gamma * np.amax(
                self.model.predict(next_state)[0, 0])
        target = self.model.predict(state)
        target[0, 0, action] = reward
        self.model.fit(state, target, epochs=1,
                       verbose=False)
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay

def learn(self, episodes):
    ''' 训练DQL智能体
    '''
    for e in range(1, episodes + 1):
        state = self.learn_env.reset()
        state = np.reshape(state, [1, self.learn_env.lags,
                                   self.learn_env.n_features])
        for _ in range(10000):
            action = self.act(state)
            next_state, reward, done, info = self.learn_env.step(action)
            next_state = np.reshape(next_state,
                                    [1, self.learn_env.lags,
                                     self.learn_env.n_features])
            self.memory.append([state, action, reward,
                                next_state, done])
            state = next_state
            if done:
                treward = _ + 1
                self.trewards.append(treward)
                av = sum(self.trewards[-25:]) / 25
                perf = self.learn_env.performance
                self.averages.append(av)
                self.performances.append(perf)
                self.aperformances.append(
                    sum(self.performances[-25:]) / 25)
                self.max_treward = max(self.max_treward, treward)
                templ = 'episode: {:2d}/{} | treward: {:4d} | '
                templ += 'perf: {:5.3f} | av: {:5.1f} | max: {:4d}'
                print(templ.format(e, episodes, treward, perf,
                                   av, self.max_treward), end='\r')
                break
        if self.val:
            self.validate(e, episodes)
        if len(self.memory) > self.batch_size:
            self.replay()
    print()
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
    def validate(self, e, episodes):
        ''' 验证DQL智能体
        '''
        state = self.valid_env.reset()
        state = np.reshape(state, [1, self.valid_env.lags,
                                   self.valid_env.n_features])
        for _ in range(10000):
            action = np.argmax(self.model.predict(state)[0, 0])
            next_state, reward, done, info = self.valid_env.step(action)
            state = np.reshape(next_state, [1, self.valid_env.lags,
                                            self.valid_env.n_features])
            if done:
                treward = _ + 1
                perf = self.valid_env.performance
                self.vperformances.append(perf)
                if e % int(episodes / 6) == 0:
                    templ = 71 * '='
                    templ += '\nepisode: {:2d}/{} | VALIDATION | '
                    templ += 'treward: {:4d} | perf: {:5.3f} | eps: {:.2f}\n'
                    templ += 71 * '='
                    print(templ.format(e, episodes, treward,
                                       perf, self.epsilon))
                break


def plot_treward(agent):
    ''' 为每个训练回合的总奖励绘图
    '''
    plt.figure(figsize=(10, 6))
    x = range(1, len(agent.averages) + 1)
    y = np.polyval(np.polyfit(x, agent.averages, deg=3), x)
    plt.plot(x, agent.averages, label='moving average')
    plt.plot(x, y, 'r--', label='regression')
    plt.xlabel('episodes')
    plt.ylabel('total reward')
    plt.legend()


def plot_performance(agent):
    ''' 为每个训练回合的总收益绘图
    '''
    plt.figure(figsize=(10, 6))
    x = range(1, len(agent.performances) + 1)
    y = np.polyval(np.polyfit(x, agent.performances, deg=3), x)
    plt.plot(x, agent.performances[:], label='training')
    plt.plot(x, y, 'r--', label='regression (train)')
    if agent.val:
        y_ = np.polyval(np.polyfit(x, agent.vperformances, deg=3), x)
        plt.plot(x, agent.vperformances[:], label='validation')
        plt.plot(x, y_, 'r-.', label='regression (valid)')
    plt.xlabel('episodes')
    plt.ylabel('gross performance')
    plt.legend()
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

11.7.3 回测基类

下面是带有 `BacktestingBase` 类的 Python 模块，用于基于事件的回测。

```
#
# 基于事件的回测
# --基类 (1)
#
# (c) Dr. Yves J. Hilpisch
# Artificial Intelligence in Finance
#


class BacktestingBase:
    def __init__(self, env, model, amount, ptc, ftc, verbose=False):
        self.env = env  ❶
        self.model = model  ❷
        self.initial_amount = amount  ❸
        self.current_balance = amount  ❸
        self.ptc = ptc   ❹
        self.ftc = ftc   ❺
        self.verbose = verbose  ❻
        self.units = 0  ❼
        self.trades = 0  ❽

    def get_date_price(self, bar):
        ''' 返回给定的bar所对应的日期和价格
        '''
        date = str(self.env.data.index[bar])[:10]  ❾
        price = self.env.data[self.env.symbol].iloc[bar]  ❿
        return date, price

    def print_balance(self, bar):
        ''' 打印给定的bar所对应的现金余额
        '''
        date, price = self.get_date_price(bar)
        print(f'{date} | current balance = {self.current_balance:.2f}')  ⓫

    def calculate_net_wealth(self, price):
        return self.current_balance + self.units * price  ⓬

    def print_net_wealth(self, bar):
        ''' 打印给定的bar所对应的净资产（现金+头寸）
        '''
        date, price = self.get_date_price(bar)
        net_wealth = self.calculate_net_wealth(price)
        print(f'{date} | net wealth = {net_wealth:.2f}')  ⓭

    def place_buy_order(self, bar, amount=None, units=None):
        ''' 对于给定的bar和给定的amount或给定的units数量下一个买入单
        '''
        date, price = self.get_date_price(bar)
        if units is None:
            units = int(amount / price)  ⓮
            # units = amount / price  ⓮
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            self.current_balance -= (1 + self.ptc) * \
                units * price + self.ftc  ⓯
            self.units += units  ⓰
            self.trades += 1  ⓱
            if self.verbose:
                print(f'{date} | buy {units} units for {price:.4f}')
                self.print_balance(bar)

        def place_sell_order(self, bar, amount=None, units=None):
            ''' 对于给定的bar和给定的amount或给定的units数量下一个卖出单
            '''
            date, price = self.get_date_price(bar)
            if units is None:
                units = int(amount / price)  ⓮
                # units = amount / price  ⓮
            self.current_balance += (1 - self.ptc) * \
                units * price - self.ftc  ⓯
            self.units -= units  ⓰
            self.trades += 1  ⓱
            if self.verbose:
                print(f'{date} | sell {units} units for {price:.4f}')
                self.print_balance(bar)

        def close_out(self, bar):
            ''' 在给定的bar上关闭所有未平仓头寸
            '''
            date, price = self.get_date_price(bar)
            print(50 * '=')
            print(f'{date} | *** CLOSING OUT ***')
            if self.units < 0:
                self.place_buy_order(bar, units=-self.units)  ⓲
            else:
                self.place_sell_order(bar, units=self.units)  ⓳
            if not self.verbose:
                print(f'{date} | current balance = {self.current_balance:.2f}')
            perf = (self.current_balance / self.initial_amount - 1) * 100  ⓴
            print(f'{date} | net performance [%] = {perf:.4f}')
            print(f'{date} | number of trades [#] = {self.trades}')
            print(50 * '=')
```

❶ 相关 Finance 环境。

❷ 相关的 DNN 模型（来自交易机器人）。

❸ 初始 / 当前余额。

❹ 成比例交易成本。

❺ 固定交易成本。

❻ 打印是否冗长。

❼ 交易金融工具的初始单位数。

❽ 执行的初始交易数量。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❾ 给定 bar 下的相关**日期**。

❿ 给定 bar 下的**金融工具相关价格**。

⓫ 给定 bar 下的**日期**和**当前余额**的输出。

⓬ 根据当前余额和工具头寸计算**净资产**。

⓭ 给定 bar 下的**日期**和**净资产**输出。

⓮ 在给定交易金额的情况下进行交易的单位数。

⓯ 交易和相关成本对当前余额的影响。

⓰ 调整持有单位数量。

⓱ 调整交易数量。

⓲ 了解**空头**头寸……

⓳ ……或是**多头**头寸。

⓴ 给定初始金额和最终流动余额的净收益。

11.7.4 回测类

以下是 Python 模块，其中 TBBacktesterRM 类用于基于事件的回测，该回测包括风控措施（止损单、跟踪止损单、止盈单）。

```
#
# 基于事件的回测
# --交易机器人回测（包含风险管理）
#
# (c) Dr. Yves J. Hilpisch
#
import numpy as np
import pandas as pd
import backtestingrm as btr


class TBBacktesterRM(btr.BacktestingBaseRM):
    def _reshape(self, state):
        ''' 用来对状态对象进行重塑的辅助函数
        '''
        return np.reshape(state, [1, self.env.lags, self.env.n_features])

    def backtest_strategy(self, sl=None, tsl=None, tp=None,
                          wait=5, guarantee=False):
        ''' 基于事件的交易机器人性能回测，包括止损、跟踪止损和止盈
        '''
        self.units = 0
        self.position = 0
        self.trades = 0
        self.sl = sl
        self.tsl = tsl
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        self.tp = tp
        self.wait = 0
        self.current_balance = self.initial_amount
        self.net_wealths = list()
        for bar in range(self.env.lags, len(self.env.data)):
            self.wait = max(0, self.wait - 1)
            date, price = self.get_date_price(bar)
            if self.trades == 0:
                print(50 * '=')
                print(f'{date} | *** START BACKTEST ***')
                self.print_balance(bar)
                print(50 * '=')
        # 止损单
        if sl is not None and self.position != 0:
            rc = (price - self.entry_price) / self.entry_price
            if self.position == 1 and rc < -self.sl:
                print(50 * '-')
                if guarantee:
                    price = self.entry_price * (1 - self.sl)
                    print(f'*** STOP LOSS (LONG  | {-self.sl:.4f}) ***')
                else:
                    print(f'*** STOP LOSS (LONG  | {rc:.4f}) ***')
                self.place_sell_order(bar, units=self.units, gprice=price)
                self.wait = wait
                self.position = 0
            elif self.position == -1 and rc > self.sl:
                print(50 * '-')
                if guarantee:
                    price = self.entry_price * (1 + self.sl)
                    print(f'*** STOP LOSS (SHORT | -{self.sl:.4f}) ***')
                else:
                    print(f'*** STOP LOSS (SHORT | -{rc:.4f}) ***')
                self.place_buy_order(bar, units=-self.units, gprice=price)
                self.wait = wait
                self.position = 0

        # 跟踪止损单
        if tsl is not None and self.position != 0:
            self.max_price = max(self.max_price, price)
            self.min_price = min(self.min_price, price)
            rc_1 = (price - self.max_price) / self.entry_price
            rc_2 = (self.min_price - price) / self.entry_price
            if self.position == 1 and rc_1 < -self.tsl:
                print(50 * '-')
                print(f'*** TRAILING SL (LONG | {rc_1:.4f}) ***')
                self.place_sell_order(bar, units=self.units)
                self.wait = wait
                self.position = 0
            elif self.position == -1 and rc_2 < -self.tsl:
                print(50 * '-')
                print(f'*** TRAILING SL (SHORT | {rc_2:.4f}) ***')
                self.place_buy_order(bar, units=-self.units)
                self.wait = wait
                self.position = 0
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        # 止盈单
        if tp is not None and self.position != 0:
            rc = (price - self.entry_price) / self.entry_price
            if self.position == 1 and rc > self.tp:
                print(50 * '-')
                if guarantee:
                    price = self.entry_price * (1 + self.tp)
                    print(f'*** TAKE PROFIT (LONG | {self.tp:.4f}) ***')
                else:
                    print(f'*** TAKE PROFIT (LONG | {rc:.4f}) ***')
                self.place_sell_order(bar, units=self.units, gprice=price)
                self.wait = wait
                self.position = 0
            elif self.position == -1 and rc < -self.tp:
                print(50 * '-')
                if guarantee:
                    price = self.entry_price * (1 - self.tp)
                    print(f'*** TAKE PROFIT (SHORT | {self.tp:.4f}) ***')
                else:
                    print(f'*** TAKE PROFIT (SHORT | {-rc:.4f}) ***')
                self.place_buy_order(bar, units=-self.units, gprice=price)
                self.wait = wait
                self.position = 0

        state = self.env.get_state(bar)
        action = np.argmax(self.model.predict(
            self._reshape(state.values))[0, 0])
        position = 1 if action == 1 else -1
        if self.position in [0, -1] and position == 1 and self.wait == 0:
            if self.verbose:
                print(50 * '-')
                print(f'{date} | *** GOING LONG ***')
            if self.position == -1:
                self.place_buy_order(bar - 1, units=-self.units)
            self.place_buy_order(bar - 1, amount=self.current_balance)
            if self.verbose:
                self.print_net_wealth(bar)
            self.position = 1
        elif self.position in [0, 1] and position == -1 and self.wait == 0:
            if self.verbose:
                print(50 * '-')
                print(f'{date} | *** GOING SHORT ***')
            if self.position == 1:
                self.place_sell_order(bar - 1, units=self.units)
            self.place_sell_order(bar - 1, amount=self.current_balance)
            if self.verbose:
                self.print_net_wealth(bar)
            self.position = -1
        self.net_wealths.append((date, self.calculate_net_wealth(price)))
    self.net_wealths = pd.DataFrame(self.net_wealths,
                                    columns=['date', 'net_wealth'])
    self.net_wealths.set_index('date', inplace=True)
    self.net_wealths.index = pd.DatetimeIndex(self.net_wealths.index)
    self.close_out(bar)
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 12 章

执行与部署

> 如果自动驾驶汽车想要在城市混合交通、暴雨和大雪、未铺的路面和地图上未标明的道路，以及无线接入不可靠的地方可靠地运行，那么还需要取得相当大的进展才行。
>
> ——Todd Litman，2020 年

> 从事算法交易的投资公司应拥有适合其经营业务的系统和风控，以确保其交易系统在适当的交易门槛和限制下仍具有弹性和足够的能力，并防止发出错误的指令，或者系统以错误的方式运行从而造成或助长无序市场。
>
> ——MiFID II，第 17 条

第 11 章基于历史数据以 Q 学习金融智能体的形式训练了一款交易机器人。它引入了基于事件的回测，以作为一种足够灵活的方法来考虑典型的风控措施，比如跟踪止损单或获利目标。然而，所有这些都是在仅基于历史数据的沙箱环境中异步发生的。与自动驾驶汽车一样，现实世界中也存在部署人工智能的问题。对自动驾驶汽车来说，这意味着将人工智能与汽车硬件相结合，并在测试街道和公共街道上部署它。对交易机器人而言，这意味着将交易机器人与交易平台连接并对其进行部署，以便自动执行订单。换句话说，算法方面是很清楚的——现在需要添加执行和部署来实现算法交易。

本章介绍了算法交易的 Oanda 交易平台。因此，重点是平台的 v20 API，而不是为用户提供手动交易接口的应用。为了简化代码，我们引入并使用了 `tpqoa` 包。它依赖于 Oanda 的 `v20` Python 包，并提供了更加 Python 化的用户界面。

12.1 节详细说明了使用 Oanda 演示账户的先决条件。12.2 节展示了如何从 API 检索历史数据和实时（流）数据。12.3 节涉及买卖订单的执行，可能包括其他订单，比如跟踪止损单。12.4 节根据 Oanda 的历史日内数据训练交易机器人，并以向量化方式对其性能进行回测。12.5 节展示了如何以实时和自动化的方式部署交易机器人。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

12.1 Oanda账户

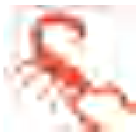
本章中的代码依赖于 Python 包 `tpqoa`。此软件包可通过 `pip` 安装。

要使用此软件包，拥有 Oanda 的演示账户就足够了。打开账户后，会在账户页面生成一个**访问令牌**（登录后）。然后，访问令牌和**账户 id**（也可在账户页面找到）会被存储在配置文本文件中，如下所示。

```
[oanda]
account_id = XYZ-ABC-...
access_token = ZYXCAB...
account_type = practice
```

如果配置文件的名称是 aiif.cfg，并且被存储在当前工作目录中，那么 `tpqoa` 包可以以如下方式使用。

```
import tpqoa
api = tpqoa.tpqoa('aiif.cfg')
```



风险免责声明和披露

Oanda 是一个**外汇**（FX）和**差价合约**（CFD）交易平台。这些工具涉及相当大的风险，特别是在利用杠杆交易时。强烈建议你在继续之前仔细阅读 Oanda 网站上的所有相关风险声明和披露（查看适当的管辖范围）。

本章提供的所有代码及示例只作技术说明，并不构成任何投资建议或类似建议。

12.2 数据检索

像往常一样，先导入并配置一些 Python 包。

```
In [1]: import os
        import time
        import numpy as np
        import pandas as pd
        from pprint import pprint
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        pd.set_option('mode.chained_assignment', None)
        pd.set_option('display.float_format', '{:.5f}'.format)
        np.set_printoptions(suppress=True, precision=4)
        os.environ['PYTHONHASHSEED'] = '0'
```

根据账户的相关管辖区，Oanda 提供了许多可交易的外汇和差价合约工具。以下 Python 代码会检索给定账户的可用工具。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [2]: import tpqoa  ❶

In [3]: api = tpqoa.tpqoa('../aiif.cfg')  ❷

In [4]: ins = api.get_instruments()  ❸

In [5]: ins[:5]  ❹
Out[5]: [('AUD/CAD', 'AUD_CAD'),
         ('AUD/CHF', 'AUD_CHF'),
         ('AUD/CNY', 'AUD_CNY'),
         ('AUD/JPY', 'AUD_JPY'),
         ('AUD/NZD', 'AUD_NZD')]
```

❶ 导入 tpqoa 包。

❷ 实例化给定账户凭证的 API 对象。

❸ 以 (display_name, technical_name) 格式检索可用工具的列表。

❹ 显示选择的少量工具。

Oanda 通过其 v20 API 提供了丰富的历史数据。以下示例会检索数据粒度被设置为 D（每日）的欧元 / 美元货币对的历史数据。

图 12-1 展示了收盘价（卖出价）。

```
In [6]: raw = api.get_history(instrument='EUR_USD',  ❶
                              start='2018-01-01',  ❷
                              end='2020-07-31',  ❸
                              granularity='D',  ❹
                              price='A')  ❺

In [7]: raw.info()
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 671 entries, 2018-01-01 22:00:00 to 2020-07-30 21:00:00
        Data columns (total 6 columns):
         #   Column    Non-Null Count  Dtype
        ---  ------    --------------  -----
         0   o         671 non-null    float64
         1   h         671 non-null    float64
         2   l         671 non-null    float64
         3   c         671 non-null    float64
         4   volume    671 non-null    int64
         5   complete  671 non-null    bool
        dtypes: bool(1), float64(4), int64(1)
        memory usage: 32.1 KB

In [8]: raw.head()
Out[8]:                            o        h        l        c     volume complete
        time
        2018-01-01 22:00:00 1.20101 1.20819 1.20051 1.20610      35630     True
        2018-01-02 22:00:00 1.20620 1.20673 1.20018 1.20170      31354     True
        2018-01-03 22:00:00 1.20170 1.20897 1.20049 1.20710      35187     True
        2018-01-04 22:00:00 1.20692 1.20847 1.20215 1.20327      36478     True
        2018-01-07 22:00:00 1.20301 1.20530 1.19564 1.19717      27618     True

In [9]: raw['c'].plot(figsize=(10, 6));
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❶ 指定工具……

❷ ……开始日期……

❸ ……结束日期……

❹ ……粒度（D = 每日）……

❺ ……价格系列的类型（A = 要价）。

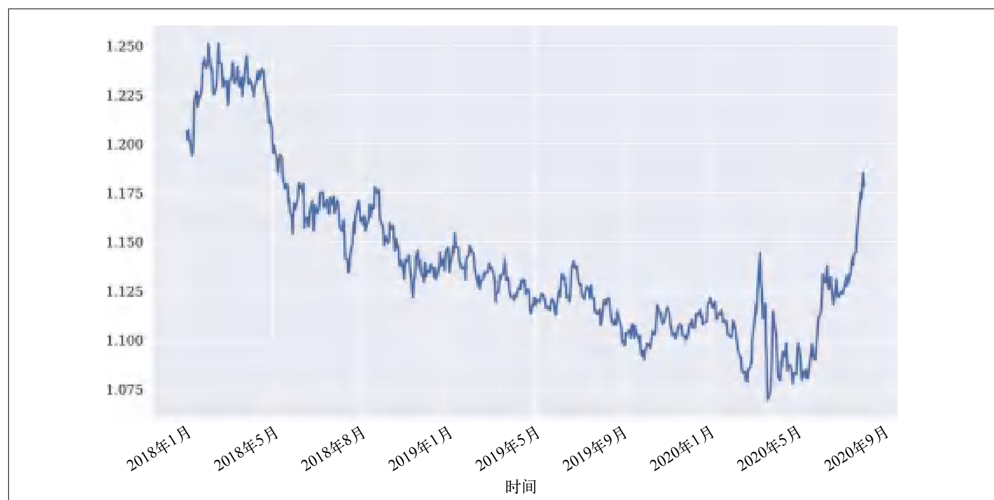


图 12-1：Oanda 欧元 / 美元的每日收盘价历史数据

正如下面的代码所示，与日数据一样，日内数据很容易被检索和使用。图 12-2 显示了分钟（中间）价格数据。

```
In [10]: raw = api.get_history(instrument='EUR_USD',
                              start='2020-07-01',
                              end='2020-07-31',
                              granularity='M1',  ❶
                              price='M')  ❷

In [11]: raw.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 30728 entries, 2020-07-01 00:00:00 to 2020-07-30 23:59:00
         Data columns (total 6 columns):
          #   Column    Non-Null Count  Dtype
         ---  ------    --------------  -----
          0   o         30728 non-null  float64
          1   h         30728 non-null  float64
          2   l         30728 non-null  float64
          3   c         30728 non-null  float64
          4   volume    30728 non-null  int64
          5   complete  30728 non-null  bool
         dtypes: bool(1), float64(4), int64(1)
         memory usage: 1.4 MB
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [12]: raw.tail()
Out[12]:                              o       h       l       c volume complete
         time
         2020-07-30 23:55:00 1.18724 1.18739 1.18718 1.18738     57     True
         2020-07-30 23:56:00 1.18736 1.18758 1.18722 1.18757     57     True
         2020-07-30 23:57:00 1.18756 1.18756 1.18734 1.18734     49     True
         2020-07-30 23:58:00 1.18736 1.18737 1.18713 1.18717     36     True
         2020-07-30 23:59:00 1.18718 1.18724 1.18714 1.18722     31     True

In [13]: raw['c'].plot(figsize=(10, 6));
```

❶ 指定粒度（M1 = 1 分钟）……

❷ ……和价格系列的类型（M = 中间价）。

图 12-2：Oanda 欧元 / 美元的 1 分钟收盘价历史数据

对训练和测试一款交易机器人来说，历史数据很重要，但对进行算法交易的交易机器人来说，实时（流）数据的部署是必要的。tpqoa 允许通过单一方法调用为所有可用的工具同步实时数据流。该方法会默认打印时间戳和出价 / 要价。对算法交易来说，这种默认方式是可以调整的，如 12.5 节所示。

```
In [14]: api.stream_data('EUR_USD', stop=10)
         2020-08-13T12:07:09.735715316Z 1.18328 1.18342
         2020-08-13T12:07:16.245253689Z 1.18329 1.18343
         2020-08-13T12:07:16.397803785Z 1.18328 1.18342
         2020-08-13T12:07:17.240232521Z 1.18331 1.18346
         2020-08-13T12:07:17.358476854Z 1.18334 1.18348
         2020-08-13T12:07:17.778061207Z 1.18331 1.18345
         2020-08-13T12:07:18.016544856Z 1.18333 1.18346
         2020-08-13T12:07:18.144762415Z 1.18334 1.18348
         2020-08-13T12:07:18.689365678Z 1.18331 1.18345
         2020-08-13T12:07:19.148039139Z 1.18331 1.18345
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

12.3 订单执行

自动驾驶汽车的人工智能需要控制实体车辆。为此，它向车辆发送不同类型的信号，比如加速、刹车、左转或右转。交易机器人需要在交易平台上下订单。本节介绍了不同类型的订单，比如市场订单和止损单。

最基本的订单类型是**市场订单**。该订单允许以当前市场价格（买入时的**卖出价**和卖出时的**买入价**）购买或出售金融工具。下面的例子基于 20 倍杠杆的账户和相对较小的订单规模。因此，流动性问题并没有产生影响。当通过 Oanda v20 API 执行订单时，该 API 返回了一个详细的订单对象。首先，发出**买入市场订单**。

```
In [15]: order = api.create_order('EUR_USD', units=25000,
                                  suppress=True, ret=True)  ❶
         pprint(order)  ❶
         {'accountBalance': '98553.3172',
          'accountID': '101-004-13834683-001',
          'batchID': '1625',
          'commission': '0.0',
          'financing': '0.0',
          'fullPrice': {'asks': [{'liquidity': '10000000', 'price': 1.18345}],
                        'bids': [{'liquidity': '10000000', 'price': 1.18331}],
                        'closeoutAsk': 1.18345,
                        'closeoutBid': 1.18331,
                        'type': 'PRICE'},
          'fullVWAP': 1.18345,
          'gainQuoteHomeConversionFactor': '0.840811914585',
          'guaranteedExecutionFee': '0.0',
          'halfSpreadCost': '1.4788',
          'id': '1626',
          'instrument': 'EUR_USD',
          'lossQuoteHomeConversionFactor': '0.849262285586',
          'orderID': '1625',
          'pl': '0.0',
          'price': 1.18345,
          'reason': 'MARKET_ORDER',
          'requestID': '78757241547812154',
          'time': '2020-08-13T12:07:19.434407966Z',
          'tradeOpened': {'guaranteedExecutionFee': '0.0',
                          'halfSpreadCost': '1.4788',
                          'initialMarginRequired': '832.5',
                          'price': 1.18345,
                          'tradeID': '1626',
                          'units': '25000.0'},
          'type': 'ORDER_FILL',
          'units': '25000.0',
          'userID': 13834683}

In [16]: def print_details(order):  ❷
             details = (order['time'][:-7], order['instrument'], order['units'],
                        order['price'], order['pl'])
             return details
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [17]: print_details(order)  ❷
Out[17]: ('2020-08-13T12:07:19.434', 'EUR_USD', '25000.0', 1.18345, '0.0')

In [18]: time.sleep(1)
```

❶ 发出**买入市场订单**并打印订单对象详细信息。

❷ 选择并显示订单的 `time`、`instrument`、`units`、`price` 和 `pl` 明细。

然后，该头寸通过相同规模的**卖出市场订单**平仓。鉴于第一笔交易的损益（P&L）在计入交易成本之前为零，第二笔交易的损益一般为非零。

```
In [19]: order = api.create_order('EUR_USD', units=-25000,
                                  suppress=True, ret=True)  ❶
         pprint(order)  ❶
         {'accountBalance': '98549.283',
          'accountID': '101-004-13834683-001',
          'batchID': '1627',
          'commission': '0.0',
          'financing': '0.0',
          'fullPrice': {'asks': [{'liquidity': '9975000', 'price': 1.18339}],
                        'bids': [{'liquidity': '10000000', 'price': 1.18326}],
                        'closeoutAsk': 1.18339,
                        'closeoutBid': 1.18326,
                        'type': 'PRICE'},
          'fullVWAP': 1.18326,
          'gainQuoteHomeConversionFactor': '0.840850994445',
          'guaranteedExecutionFee': '0.0',
          'halfSpreadCost': '1.3732',
          'id': '1628',
          'instrument': 'EUR_USD',
          'lossQuoteHomeConversionFactor': '0.849301758209',
          'orderID': '1627',
          'pl': '-4.0342',
          'price': 1.18326,
          'reason': 'MARKET_ORDER',
          'requestID': '78757241552009237',
          'time': '2020-08-13T12:07:20.586564454Z',
          'tradesClosed': [{'financing': '0.0',
                            'guaranteedExecutionFee': '0.0',
                            'halfSpreadCost': '1.3732',
                            'price': 1.18326,
                            'realizedPL': '-4.0342',
                            'tradeID': '1626',
                            'units': '-25000.0'}],
          'type': 'ORDER_FILL',
          'units': '-25000.0',
          'userID': 13834683}

In [20]: print_details(order)  ❷
Out[20]: ('2020-08-13T12:07:20.586', 'EUR_USD', '-25000.0', 1.18326, '-4.0342')

In [21]: time.sleep(1)
```

❶ 发出**卖出市场订单**并打印订单对象详细信息。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❷ 选择并显示订单的 time、instrument、units、price 和 pl 明细。

限价订单

本章仅将**市场订单**作为基础订单的一种类型。在市场订单中，购买或出售金融工具的价格是在订单发出时的当前价格。相比之下，**限价订单**作为基础订单的另一种主要类型，允许以最低价格或最高价格下订单。只有达到最低 / 最高价格时，订单才会执行。在此之前，没有任何交易发生。

接下来，考虑一个相同交易组合的例子，但这次使用**止损**单。止损单被作为单独的（限制）订单来处理。下面的 Python 代码设置了订单并显示了止损单对象的详细信息。

```
In [22]: order = api.create_order('EUR_USD', units=25000,
                                  sl_distance=0.005,  ❶
                                  suppress=True, ret=True)

In [23]: print_details(order)
Out[23]: ('2020-08-13T12:07:21.740', 'EUR_USD', '25000.0', 1.18343, '0.0')

In [24]: sl_order = api.get_transaction(tid=int(order['id']) + 1)  ❷

In [25]: sl_order  ❷
Out[25]: {'id': '1631',
          'time': '2020-08-13T12:07:21.740825489Z',
          'userID': 13834683,
          'accountID': '101-004-13834683-001',
          'batchID': '1629',
          'requestID': '78757241556206373',
          'type': 'STOP_LOSS_ORDER',
          'tradeID': '1630',
          'price': 1.17843,
          'distance': '0.005',
          'timeInForce': 'GTC',
          'triggerCondition': 'DEFAULT',
          'reason': 'ON_FILL'}

In [26]: (sl_order['time'], sl_order['type'], order['price'],
          sl_order['price'], sl_order['distance'])  ❸
Out[26]: ('2020-08-13T12:07:21.740825489Z',
          'STOP_LOSS_ORDER',
          1.18343,
          1.17843,
          '0.005')

In [27]: time.sleep(1)

In [28]: order = api.create_order('EUR_USD', units=-25000, suppress=True, ret=True)

In [29]: print_details(order)
Out[29]: ('2020-08-13T12:07:23.059', 'EUR_USD', '-25000.0', 1.18329, '-2.9725')
```

❶ 止损距离以货币单位定义。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❷ 选择并显示止损单对象数据。

❸ 选择并显示两个订单对象的一些相关细节。

跟踪止损单以同样的方式处理。唯一的区别是跟踪止损单没有固定的价格。

```
In [30]: order = api.create_order('EUR_USD', units=25000,
                                  tsl_distance=0.005,  ❶
                                  suppress=True, ret=True)

In [31]: print_details(order)
Out[31]: ('2020-08-13T12:07:23.204', 'EUR_USD', '25000.0', 1.18341, '0.0')

In [32]: tsl_order = api.get_transaction(tid=int(order['id']) + 1)  ❷

In [33]: tsl_order  ❷
Out[33]: {'id': '1637',
          'time': '2020-08-13T12:07:23.204457044Z',
          'userID': 13834683,
          'accountID': '101-004-13834683-001',
          'batchID': '1635',
          'requestID': '78757241564598562',
          'type': 'TRAILING_STOP_LOSS_ORDER',
          'tradeID': '1636',
          'distance': '0.005',
          'timeInForce': 'GTC',
          'triggerCondition': 'DEFAULT',
          'reason': 'ON_FILL'}

In [34]: (tsl_order['time'][:-7], tsl_order['type'],
          order['price'], tsl_order['distance'])  ❸
Out[34]: ('2020-08-13T12:07:23.204', 'TRAILING_STOP_LOSS_ORDER', 1.18341, '0.005')

In [35]: time.sleep(1)

In [36]: order = api.create_order('EUR_USD', units=-25000,
                                  suppress=True, ret=True)

In [37]: print_details(order)
Out[37]: ('2020-08-13T12:07:24.551', 'EUR_USD', '-25000.0', 1.1833, '-2.3355')

In [38]: time.sleep(1)
```

❶ 跟踪止损距离以货币单位定义。

❷ 选择并显示跟踪止损单对象数据。

❸ 选择并显示两个订单对象的一些相关细节。

最后，这是一个**止盈**单。这个订单需要一个固定的止盈目标价格。因此，下面的代码使用前一个订单中的执行价格来相对定义止盈价格。除了这个小小的差别之外，处理方式和以前一样。

```
In [39]: tp_price = round(order['price'] + 0.01, 4)
         tp_price
Out[39]: 1.1933
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [40]: order = api.create_order('EUR_USD', units=25000,
                                  tp_price=tp_price,  ❶
                                  suppress=True, ret=True)
In [41]: print_details(order)
Out[41]: ('2020-08-13T12:07:25.712', 'EUR_USD', '25000.0', 1.18344, '0.0')

In [42]: tp_order = api.get_transaction(tid=int(order['id']) + 1)  ❷

In [43]: tp_order  ❷
Out[43]: {'id': '1643',
          'time': '2020-08-13T12:07:25.712531725Z',
          'userID': 13834683,
          'accountID': '101-004-13834683-001',
          'batchID': '1641',
          'requestID': '78757241572993078',
          'type': 'TAKE_PROFIT_ORDER',
          'tradeID': '1642',
          'price': 1.1933,
          'timeInForce': 'GTC',
          'triggerCondition': 'DEFAULT',
          'reason': 'ON_FILL'}

In [44]: (tp_order['time'][:-7], tp_order['type'],
          order['price'], tp_order['price'])  ❸
Out[44]: ('2020-08-13T12:07:25.712', 'TAKE_PROFIT_ORDER', 1.18344, 1.1933)

In [45]: time.sleep(1)

In [46]: order = api.create_order('EUR_USD', units=-25000,
                                  suppress=True, ret=True)

In [47]: print_details(order)
Out[47]: ('2020-08-13T12:07:27.020', 'EUR_USD', '-25000.0', 1.18332, '-2.5478')
```

❶ 止盈目标价格是相对于之前的执行价格定义的。

❷ 选择并显示止盈单对象数据。

❸ 选择并显示两个订单对象的一些相关细节。

到目前为止，代码只处理了单个订单的交易细节。然而，对多笔**历史交易**进行说明也是很有意义的。为此，下面的方法调用提供了本节中所有主要订单的概览数据，包括损益数据。

```
In [48]: api.print_transactions(tid=int(order['id']) - 22)
          1626 | 2020-08-13T12:07:19.434407966Z |   EUR_USD |  25000.0 |     0.0
          1628 | 2020-08-13T12:07:20.586564454Z |   EUR_USD | -25000.0 | -4.0342
          1630 | 2020-08-13T12:07:21.740825489Z |   EUR_USD |  25000.0 |     0.0
          1633 | 2020-08-13T12:07:23.059178023Z |   EUR_USD | -25000.0 | -2.9725
          1636 | 2020-08-13T12:07:23.204457044Z |   EUR_USD |  25000.0 |     0.0
          1639 | 2020-08-13T12:07:24.551026466Z |   EUR_USD | -25000.0 | -2.3355
          1642 | 2020-08-13T12:07:25.712531725Z |   EUR_USD |  25000.0 |     0.0
          1645 | 2020-08-13T12:07:27.020414342Z |   EUR_USD | -25000.0 | -2.5478
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

这里调用了另外一个函数来打印**账户明细**快照。这个账户为用于技术测试的 Oanda 演示账户。

```
In [49]: api.get_account_summary()
Out[49]: {'id': '101-004-13834683-001',
          'alias': 'Primary',
          'currency': 'EUR',
          'balance': '98541.4272',
          'createdByUserID': 13834683,
          'createdTime': '2020-03-19T06:08:14.363139403Z',
          'guaranteedStopLossOrderMode': 'DISABLED',
          'pl': '-1248.5543',
          'resettablePL': '-1248.5543',
          'resettablePLTime': '0',
          'financing': '-210.0185',
          'commission': '0.0',
          'guaranteedExecutionFees': '0.0',
          'marginRate': '0.0333',
          'openTradeCount': 1,
          'openPositionCount': 1,
          'pendingOrderCount': 0,
          'hedgingEnabled': False,
          'unrealizedPL': '941.9536',
          'NAV': '99483.3808',
          'marginUsed': '380.83',
          'marginAvailable': '99107.2283',
          'positionValue': '3808.3',
          'marginCloseoutUnrealizedPL': '947.9546',
          'marginCloseoutNAV': '99489.3818',
          'marginCloseoutMarginUsed': '380.83',
          'marginCloseoutPercent': '0.00191',
          'marginCloseoutPositionValue': '3808.3',
          'withdrawalLimit': '98541.4272',
          'marginCallMarginUsed': '380.83',
          'marginCallPercent': '0.00383',
          'lastTransactionID': '1646'}
```

这就结束了对使用 Oanda 执行订单的基本知识的讨论。现在将所有元素整合在一起就能支持交易机器人的部署了。本章余下内容将在 Oanda 数据上训练一款交易机器人，并以自动化的方式部署它。

12.4 交易机器人

第 11 章详细展示过如何训练一款深度 Q 学习交易机器人，以及如何以向量化和基于事件的方式对其进行回测。本节会根据 Oanda 的历史数据重复这方面选定的核心步骤。12.7.1 节会展示一个 Python 模块，其中包含用于处理 Oanda 数据的环境类 `OandaEnv`，其用法与第 11 章的 `Finance` 类相同。

下面的 Python 代码实例化了学习环境对象。在此步骤中，驱动学习、验证和测试的主要数据相关的参数是固定的。`OandaEnv` 类允许包含杠杆，这在外汇和差价合约交易中是典型情况。杠杆放大了已实现的收益率，从而增加了潜在的利润，但也增加了损失风险。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [50]: import oandaenv as oe

In [51]: symbol = 'EUR_USD'

In [52]: date = '2020-08-11'

In [53]: features = [symbol, 'r', 's', 'm', 'v']

In [54]: %%time
         learn_env = oe.OandaEnv(symbol=symbol,
                                 start=f'{date} 08:00:00',
                                 end=f'{date} 13:00:00',
                                 granularity='S30',  ❶
                                 price='M',  ❷
                                 features=features,  ❸
                                 window=20,  ❹
                                 lags=3,  ❺
                                 leverage=20,  ❻
                                 min_accuracy=0.4,  ❼
                                 min_performance=0.85  ❽
                                )
         CPU times: user 23.1 ms, sys: 2.86 ms, total: 25.9 ms
         Wall time: 26.8 ms

In [55]: np.bincount(learn_env.data['d'])
Out[55]: array([299, 281])

In [56]: learn_env.data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 580 entries, 2020-08-11 08:10:00 to 2020-08-11 12:59:30
         Data columns (total 6 columns):
          #   Column  Non-Null Count   Dtype
         ---  ------  --------------   -----
          0   EUR_USD 580 non-null     float64
          1   r       580 non-null     float64
          2   s       580 non-null     float64
          3   m       580 non-null     float64
          4   v       580 non-null     float64
          5   d       580 non-null     int64
         dtypes: float64(5), int64(1)
         memory usage: 31.7 KB
```

❶ 将数据的粒度设置为 5 秒。

❷ 将价格类型设置为中间价格。

❸ 定义要使用的特性集。

❹ 定义滚动统计数据的窗口长度。

❺ 指定滞后的数量。

❻ 固定杠杆。

❼ 设置所需的最小准确率。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❽ 设置所需的最低性能。

在下一步中，我们将实例化验证环境，很明显，除了时间间隔之外，我们还依赖于学习环境的参数。

图 12-3 显示了学习环境、验证环境和测试环境中使用的欧元 / 美元收盘价（从左到右）。

```
In [57]: valid_env = oe.OandaEnv(symbol=learn_env.symbol,
                                  start=f'{date} 13:00:00',
                                  end=f'{date} 14:00:00',
                                  granularity=learn_env.granularity,
                                  price=learn_env.price,
                                  features=learn_env.features,
                                  window=learn_env.window,
                                  lags=learn_env.lags,
                                  leverage=learn_env.leverage,
                                  min_accuracy=0,
                                  min_performance=0,
                                  mu=learn_env.mu,
                                  std=learn_env.std
                                 )

In [58]: valid_env.data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 100 entries, 2020-08-11 13:10:00 to 2020-08-11 13:59:30
         Data columns (total 6 columns):
          #   Column   Non-Null Count  Dtype
         ---  ------   --------------  -----
          0   EUR_USD  100 non-null    float64
          1   r        100 non-null    float64
          2   s        100 non-null    float64
          3   m        100 non-null    float64
          4   v        100 non-null    float64
          5   d        100 non-null    int64
         dtypes: float64(5), int64(1)
         memory usage: 5.5 KB

In [59]: test_env = oe.OandaEnv(symbol=learn_env.symbol,
                                  start=f'{date} 14:00:00',
                                  end=f'{date} 17:00:00',
                                  granularity=learn_env.granularity,
                                  price=learn_env.price,
                                  features=learn_env.features,
                                  window=learn_env.window,
                                  lags=learn_env.lags,
                                  leverage=learn_env.leverage,
                                  min_accuracy=0,
                                  min_performance=0,
                                  mu=learn_env.mu,
                                  std=learn_env.std
                                 )

In [60]: test_env.data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 340 entries, 2020-08-11 14:10:00 to 2020-08-11 16:59:30
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
Data columns (total 6 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   EUR_USD  340 non-null    float64
 1   r        340 non-null    float64
 2   s        340 non-null    float64
 3   m        340 non-null    float64
 4   v        340 non-null    float64
 5   d        340 non-null    int64
dtypes: float64(5), int64(1)
memory usage: 18.6 KB
```

```
In [61]: ax = learn_env.data[learn_env.symbol].plot(figsize=(10, 6))
         plt.axvline(learn_env.data.index[-1], ls='--')
         valid_env.data[learn_env.symbol].plot(ax=ax, style='-.')
         plt.axvline(valid_env.data.index[-1], ls='--')
         test_env.data[learn_env.symbol].plot(ax=ax, style='-.');
```

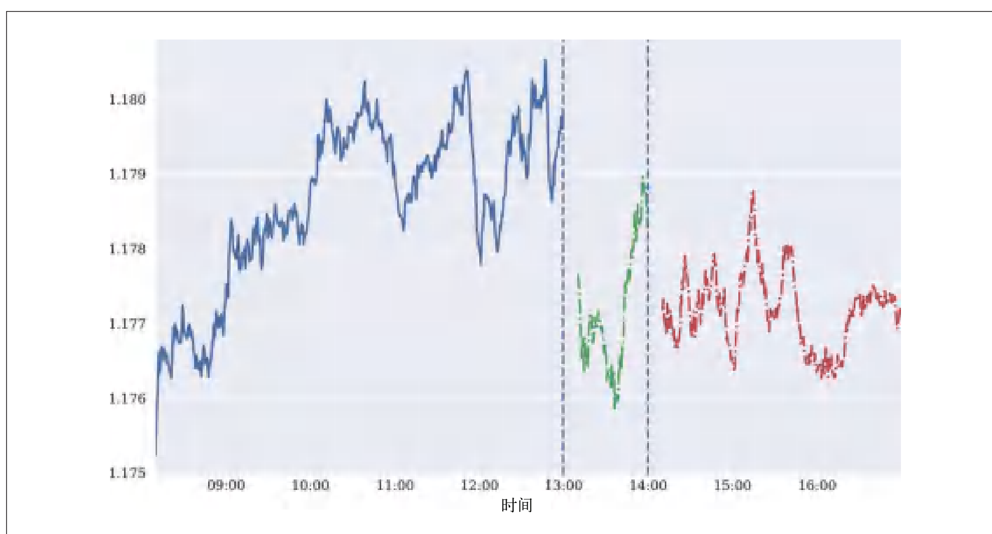


图 12-3：Oanda 欧元 / 美元的 30 秒收盘价历史数据（左：训练集；中：验证集；右：测试集）

基于 Oanda 环境，可以对第 11 章中的交易机器人进行训练和验证。下面的 Python 代码会执行此任务并可视化性能结果（参见图 12-4）。

```
In [62]: import sys
         sys.path.append('../ch11/')  ❶

In [63]: import tradingbot  ❶
         Using TensorFlow backend.

In [64]: tradingbot.set_seeds(100)
         agent = tradingbot.TradingBot(24, 0.001, learn_env=learn_env,
                                       valid_env=valid_env)  ❷
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [65]: episodes = 31

In [66]: %time agent.learn(episodes)  ❷
         ======================================================================
         episode:  5/31 | VALIDATION | treward: 97 | perf: 1.004 | eps: 0.96
         ======================================================================
         ======================================================================
         episode: 10/31 | VALIDATION | treward: 97 | perf: 1.005 | eps: 0.91
         ======================================================================
         ======================================================================
         episode: 15/31 | VALIDATION | treward: 97 | perf: 0.986 | eps: 0.87
         ======================================================================
         ======================================================================
         episode: 20/31 | VALIDATION | treward: 97 | perf: 1.012 | eps: 0.83
         ======================================================================
         ======================================================================
         episode: 25/31 | VALIDATION | treward: 97 | perf: 0.995 | eps: 0.79
         ======================================================================
         ======================================================================
         episode: 30/31 | VALIDATION | treward: 97 | perf: 0.972 | eps: 0.75
         ======================================================================
         episode: 31/31 | treward: 16 | perf: 0.981 | av: 376.0 | max: 577
         CPU times: user 22.1 s, sys: 1.17 s, total: 23.3 s
         Wall time: 20.1 s

In [67]: tradingbot.plot_performance(agent)  ❸
```

❶ 从第 11 章导入 `tradingbot` 模块。

❷ 基于 Oanda 数据训练和验证交易机器人。

❸ 可视化性能结果。

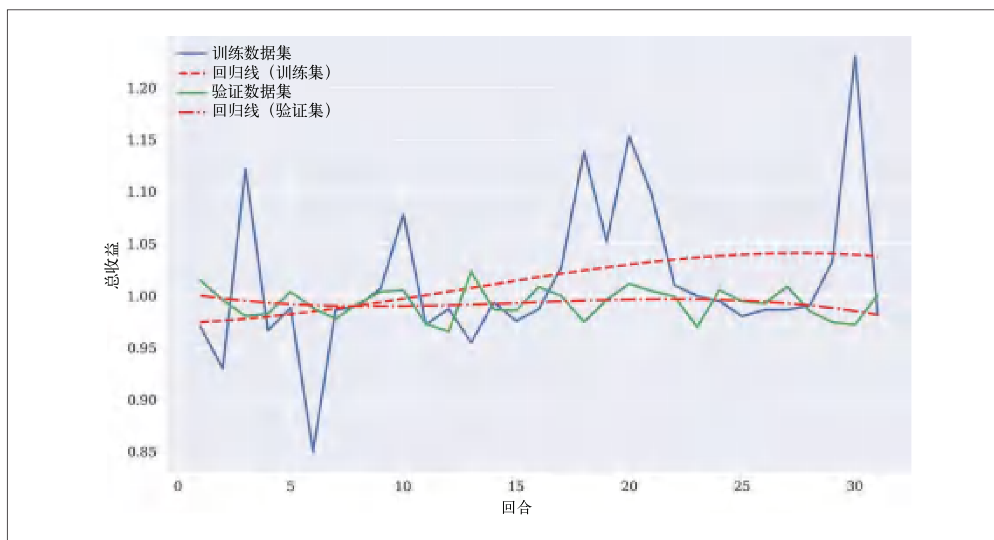


图 12-4：交易机器人基于 Oanda 数据的训练集和验证集性能结果

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

正如前两章所讨论的，训练集性能和验证集性能只是交易机器人性能的一个指标。

下面的代码为测试环境实现了交易机器人性能的向量化回测——除了使用的时间间隔参数不同外，再次使用了与学习环境相同的参数。该代码使用了 12.7.2 节展示的 Python 模块中的函数 backtest。报告的性能数字包括 20 倍的杠杆率。这对随时间变化的被动基准投资和交易机器人的总体性能都是适用的，如图 12-5 所示。

```
In [68]: import backtest as bt

In [69]: env = test_env

In [70]: bt.backtest(agent, env)

In [71]: env.data['p'].iloc[env.lags:].value_counts()  ❶
Out[71]:  1    263
         -1     74
         Name: p, dtype: int64

In [72]: sum(env.data['p'].iloc[env.lags:].diff() != 0)  ❷
Out[72]: 25

In [73]: (env.data[['r', 's']].iloc[env.lags:] * env.leverage).sum(
                 ).apply(np.exp)  ❸
Out[73]: r   0.99966
         s   1.05910
         dtype: float64

In [74]: (env.data[['r', 's']].iloc[env.lags:] * env.leverage).sum(
                 ).apply(np.exp) - 1  ❹
Out[74]: r   -0.00034
         s    0.05910
         dtype: float64

In [75]: (env.data[['r', 's']].iloc[env.lags:] * env.leverage).cumsum(
                 ).apply(np.exp).plot(figsize=(10, 6)); ❺
```

❶ 显示多头头寸和空头头寸的总数。

❷ 显示执行策略所需的交易数量。

❸ 计算包括杠杆在内的总收益。

❹ 计算包括杠杆在内的净收益。

❺ 可视化包括杠杆的随时间变化的总体表现。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

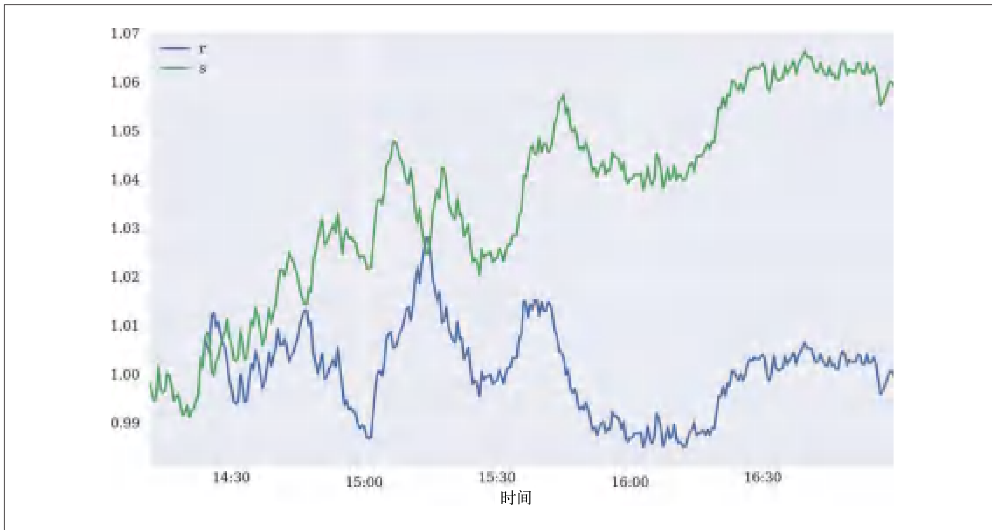
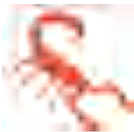


图 12-5：随时间变化的被动基准投资和交易机器人的总体表现（包括杠杆）



简化的回测

本节中对交易机器人的训练和回测是在不现实的假设下进行的。基于 30 秒间隔数据的交易策略可能会在短时间内导致大量交易。假设存在典型的交易成本（买卖价差），那么这样的策略在经济上通常是不可行的。较长的间隔数据或交易较少的策略更为现实。然而，为了在下一节中进行“快速”部署演示，我们有意在相对较短的 30 秒间隔数据中执行训练和回测。

12.5 部署

本节会结合前面几节的主要内容，以自动化方式部署经过训练的交易机器人。这与准备在街道上部署自动驾驶汽车的方式相当。以下代码中提供的 `OandaTradingBot` 类继承自 `tpqoa` 类，并添加了一些辅助函数和交易逻辑。

```
In [76]: import tpqoa

In [77]: class OandaTradingBot(tpqoa.tpqoa):
             def __init__(self, config_file, agent, granularity, units,
                          verbose=True):
                 super(OandaTradingBot, self).__init__(config_file)
                 self.agent = agent
                 self.symbol = self.agent.learn_env.symbol
                 self.env = agent.learn_env
                 self.window = self.env.window
                 if granularity is None:
                     self.granularity = agent.learn_env.granularity
                 else:
                     self.granularity = granularity
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        self.units = units
        self.trades = 0
        self.position = 0
        self.tick_data = pd.DataFrame()
        self.min_length = (self.agent.learn_env.window +
                           self.agent.learn_env.lags)
        self.pl = list()
        self.verbose = verbose
    def _prepare_data(self):
        self.data['r'] = np.log(self.data / self.data.shift(1))
        self.data.dropna(inplace=True)
        self.data['s'] = self.data[self.symbol].rolling(
                                    self.window).mean()
        self.data['m'] = self.data['r'].rolling(self.window).mean()
        self.data['v'] = self.data['r'].rolling(self.window).std()
        self.data.dropna(inplace=True)
        # self.data_ = (self.data - self.env.mu) / self.env.std  ❶
        self.data_ = (self.data - self.data.mean()) / self.data.std()  ❶
    def _resample_data(self):
        self.data = self.tick_data.resample(self.granularity,
                        label='right').last().ffill().iloc[:-1]  ❷
        self.data = pd.DataFrame(self.data['mid'])  ❷
        self.data.columns = [self.symbol,]  ❷
        self.data.index = self.data.index.tz_localize(None)  ❷
    def _get_state(self):
        state = self.data_[self.env.features].iloc[-self.env.lags:]  ❸
        return np.reshape(state.values, [1, self.env.lags,
                                         self.env.n_features])  ❸
    def report_trade(self, time, side, order):
        self.trades += 1
        pl = float(order['pl'])  ❹
        self.pl.append(pl)  ❹
        cpl = sum(self.pl)  ❺
        print('\n' + 75 * '=')
        print(f'{time} | *** GOING {side} ({self.trades}) ***')
        print(f'{time} | PROFIT/LOSS={pl:.2f} | CUMULATIVE={cpl:.2f}')
        print(75 * '=')
        if self.verbose:
            pprint(order)
            print(75 * '=')
    def on_success(self, time, bid, ask):
        df = pd.DataFrame({'ask': ask, 'bid': bid,
                           'mid': (bid + ask) / 2},
                          index=[pd.Timestamp(time)])
        self.tick_data = self.tick_data.append(df)  ❷
        self._resample_data()  ❷
        if len(self.data) > self.min_length:
            self.min_length += 1
            self._prepare_data()
            state = self._get_state()  ❻
            prediction = np.argmax(
                self.agent.model.predict(state)[0, 0])  ❻
            position = 1 if prediction == 1 else -1  ❻
            if self.position in [0, -1] and position == 1:  ❼
                order = self.create_order(self.symbol,
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
                                units=(1 - self.position) * self.units,
                                        suppress=True, ret=True)
                        self.report_trade(time, 'LONG', order)
                        self.position = 1
                    elif self.position in [0, 1] and position == -1:  ❽
                        order = self.create_order(self.symbol,
                                units=-(1 + self.position) * self.units,
                                        suppress=True, ret=True)
                        self.report_trade(time, 'SHORT', order)
                        self.position = -1
```

❶ 为了便于演示，用实时数据统计进行了标准化处理。[1]

❷ 收集逐笔数据并将其重新采样到所需的粒度。

❸ 返回金融市场的当前状态。

❹ 收集每笔交易的损益数据。

❺ 计算所有交易的累计损益。

❻ 预测市场方向并得出信号（头寸）。

❼ 检查是否满足多头头寸（买入单）的条件。

❽ 检查是否满足空头头寸（卖出单）的条件。

这个类的应用非常简单。首先，实例化一个对象，将上一节中训练过的交易机器人 agent 作为主要输入。然后，启动待交易工具的流数据。每当新的逐笔数据到达时，都会调用 .on_success() 方法，该方法包含处理逐笔数据和安排交易的主要逻辑。为了加快速度，部署示例依赖于 30 秒间隔数据，就像之前的回测一样。在生产环境中，当管理实际资金时，如果只是为了减少交易数量和交易成本，则更长的时间间隔可能是更好的选择。

```
In [78]: otb = OandaTradingBot('../aiif.cfg', agent, '30s',
                               25000, verbose=False)  ❶

In [79]: otb.tick_data.info()
         <class 'pandas.core.frame.DataFrame'>
         Index: 0 entries
         Empty DataFrame
In [80]: otb.stream_data(agent.learn_env.symbol, stop=1000)  ❷

         ===========================================================================
         2020-08-13T12:19:32.320291893Z | *** GOING SHORT (1) ***
         2020-08-13T12:19:32.320291893Z | PROFIT/LOSS=0.00 | CUMULATIVE=0.00
         ===========================================================================

         ===========================================================================
         2020-08-13T12:20:00.083985447Z | *** GOING LONG (2) ***
         2020-08-13T12:20:00.083985447Z | PROFIT/LOSS=-6.80 | CUMULATIVE=-6.80
         ===========================================================================
```

注 1：鉴于所使用的数据，这个小技巧可以更快地在这种特定环境下进行交易。对于实际部署，将使用来自学习环境数据的统计信息进行标准化。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
===========================================================================
2020-08-13T12:25:00.099901587Z | *** GOING SHORT (3) ***
2020-08-13T12:25:00.099901587Z | PROFIT/LOSS=-7.86 | CUMULATIVE=-14.66
===========================================================================

In [81]: print('\n' + 75 * '=')
         print('*** CLOSING OUT ***')
         order = otb.create_order(otb.symbol,
                         units=-otb.position * otb.units,
                         suppress=True, ret=True)  ❸
         otb.report_trade(otb.time, 'NEUTRAL', order)  ❸
         if otb.verbose:
             pprint(order)
         print(75 * '=')

         ===========================================================================
         *** CLOSING OUT ***

         ===========================================================================
         2020-08-13T12:25:16.870357562Z | *** GOING NEUTRAL (4) ***
         2020-08-13T12:25:16.870357562Z | PROFIT/LOSS=-3.19 | CUMULATIVE=-17.84
         ===========================================================================
         ===========================================================================
```

❶ 实例化 `OandaTradingBot` 对象。

❷ 启动实时数据流和交易。

❸ 在检索到一定数量交易数据后平仓最终头寸。

在部署过程中，损益数据被收集在 `pl` 属性中，`pl` 属性是一个 `list` 对象。一旦交易停止，就可以分析损益数据了。

```
In [82]: pl = np.array(otb.pl)  ❶

In [83]: pl  ❶
Out[83]: array([ 0.    , -6.7959, -7.8594, -3.1862])

In [84]: pl.cumsum()  ❷
Out[84]: array([  0.    ,  -6.7959, -14.6553, -17.8415])
```

❶ 所有交易的损益数据。

❷ 累计损益数据。

这个简单的部署示例表明，我们可以用不到 100 行的 Python 代码以算法和自动化的方式与深度 Q 学习交易机器人进行交易。主要的先决条件是训练有素的交易机器人（比如，`tradingbot` 类的实例）。这里有意忽略了许多重要的方面。例如，在生产环境中，我们可能既想持久化数据，又想持久化 `order` 对象。确保套接字连接仍然活跃的措施也很重要（比如，通过监视心跳）。总的来说，安全性、可靠性、日志记录和监视并没有得到真正的解决。Hilpisch（2020）提供了这方面的更多细节。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

12.7.3 节的 Python 脚本展示了 `OandaTradingBot` 类的独立可执行版本。与交互式环境（比如 Jupyter Notebook 或 Jupyter Lab）相比，这是朝着更稳健的部署选项迈出的重要一步。该脚本还包括为执行添加止损单、跟踪止损单或止盈单的功能。该脚本期望当前工作目录中的 `agent` 对象是 `pickle` 版本。下面的 Python 代码会对对象进行 `pickle` 处理，以供脚本以后使用。

```
In [85]: import pickle

In [86]: pickle.dump(agent, open('trading.bot', 'wb'))
```

12.6 结论

本章讨论了执行算法交易策略和部署交易机器人的核心方面。Oanda 交易平台直接或间接地为其 v20 API 提供了执行以下操作所需的所有功能。

- 检索历史数据
- 训练和回测交易机器人（深度 Q 学习智能体）
- 实时数据流
- 发出市场订单和限价订单
- 使用止损单、跟踪止损单和止盈单
- 以自动化方式部署交易机器人

实现以上所有步骤的先决条件是拥有 Oanda 演示账户、标准的硬件和软件（仅限开源）以及稳定的互联网连接。换言之，以利用经济失效为目的的算法交易进入门槛相当低。这与训练、设计和制造在公共街道上行驶的自动驾驶汽车形成了鲜明的对比——各公司在自动驾驶汽车领域的预算高达数十亿美元。而与其他行业和领域相比，在现实世界中部署人工智能体（比如本章和第 11 章所关注的交易机器人）方面，金融领域具有独特的优势。

12.7 Python代码

本节包含本章中使用和引用的代码。

12.7.1 Oanda环境

下面是带有 `OandaEnv` 类的 Python 模块，用于基于 Oanda 历史数据训练交易机器人。

```
#
# 金融环境
#
# (c) Dr. Yves J. Hilpisch
# Artificial Intelligence in Finance
#
#
import math
import tpqoa
import random
import numpy as np
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
import pandas as pd


class observation_space:
    def __init__(self, n):
        self.shape = (n,)


class action_space:
    def __init__(self, n):
        self.n = n

    def sample(self):
        return random.randint(0, self.n - 1)


class OandaEnv:
    def __init__(self, symbol, start, end, granularity, price,
                 features, window, lags, leverage=1,
                 min_accuracy=0.5, min_performance=0.85,
             mu=None, std=None):
    self.symbol = symbol
    self.start = start
    self.end = end
    self.granularity = granularity
    self.price = price
    self.api = tpqoa.tpqoa('../aiif.cfg')
    self.features = features
    self.n_features = len(features)
    self.window = window
    self.lags = lags
    self.leverage = leverage
    self.min_accuracy = min_accuracy
    self.min_performance = min_performance
    self.mu = mu
    self.std = std
    self.observation_space = observation_space(self.lags)
    self.action_space = action_space(2)
    self._get_data()
    self._prepare_data()

def _get_data(self):
    ''' 从Oanda检索数据
    '''
    self.fn = f'../../source/oanda/'  ❶
    self.fn += f'oanda_{self.symbol}_{self.start}_{self.end}_'  ❷
    self.fn += f'{self.granularity}_{self.price}.csv'  ❷
    self.fn = self.fn.replace(' ', '_').replace('-', '_').replace(':', '_')
    try:
        self.raw = pd.read_csv(self.fn, index_col=0, parse_dates=True)  ❸
    except:
        self.raw = self.api.get_history(self.symbol, self.start,
                                        self.end, self.granularity,
                                        self.price)  ❹
        self.raw.to_csv(self.fn)  ❺
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        self.data = pd.DataFrame(self.raw['c'])  ❻
        self.data.columns = [self.symbol]  ❼

    def _prepare_data(self):
        ''' 准备额外的时间序列数据（如特征数据）
        '''
        self.data['r'] = np.log(self.data / self.data.shift(1))
        self.data.dropna(inplace=True)
        self.data['s'] = self.data[self.symbol].rolling(self.window).mean()
        self.data['m'] = self.data['r'].rolling(self.window).mean()
        self.data['v'] = self.data['r'].rolling(self.window).std()
        self.data.dropna(inplace=True)
        if self.mu is None:
            self.mu = self.data.mean()
            self.std = self.data.std()
        self.data_ = (self.data - self.mu) / self.std
        self.data['d'] = np.where(self.data['r'] > 0, 1, 0)
        self.data['d'] = self.data['d'].astype(int)

    def _get_state(self):
        ''' 获取环境的当前状态的私有方法
        '''
        return self.data_[self.features].iloc[self.bar -
                                    self.lags:self.bar].values

    def get_state(self, bar):
        ''' 获取环境的当前状态
        '''
        return self.data_[self.features].iloc[bar - self.lags:bar].values

    def reset(self):
        ''' 对环境进行复位重置
        '''
        self.treward = 0
        self.accuracy = 0
        self.performance = 1
        self.bar = self.lags
        state = self._get_state()
        return state

    def step(self, action):
        ''' 将环境状态向前推进一步
        '''
        correct = action == self.data['d'].iloc[self.bar]
        ret = self.data['r'].iloc[self.bar] * self.leverage
        reward_1 = 1 if correct else 0  ❽
        reward_2 = abs(ret) if correct else -abs(ret)  ❾
        reward = reward_1 + reward_2 * self.leverage  ❿
        self.treward += reward_1
        self.bar += 1
        self.accuracy = self.treward / (self.bar - self.lags)
        self.performance *= math.exp(reward_2)
        if self.bar >= len(self.data):
            done = True
        elif reward_1 == 1:
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        done = False
    elif (self.accuracy < self.min_accuracy and
          self.bar > self.lags + 15):
        done = True
    elif (self.performance < self.min_performance and
          self.bar > self.lags + 15):
        done = True
    else:
        done = False
    state = self._get_state()
    info = {}
    return state, reward, done, info
```

❶ 定义数据文件的路径。

❷ 定义数据文件的文件名。

❸ 如果存在相应的数据文件，则读取数据。

❹ 如果不存在这样的文件，则检索 API 的数据。

❺ 将数据作为 CSV 文件写入磁盘。

❻ 选择包含收盘价的列。

❼ 将列重命名为工具名称（标记）。

❽ 对正确预测的奖励。

❾ 对实现的收益的奖励（收益率）。

❿ 对预测和收益的综合奖励。

12.7.2 向量化回测

下面的 Python 模块会使用辅助函数 backtest 生成数据，以便对深度 Q 学习交易机器人进行向量化回测。该代码在第 11 章中也使用过。

```
#
# 交易机器人向量化回测
# （Q学习金融智能体）
#
# (c) Dr. Yves J. Hilpisch
# Artificial Intelligence in Finance
#
import numpy as np
import pandas as pd
pd.set_option('mode.chained_assignment', None)

def reshape(s, env):
    return np.reshape(s, [1, env.lags, env.n_features])

def backtest(agent, env):
    done = False
    env.data['p'] = 0
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
    state = env.reset()
    while not done:
        action = np.argmax(
            agent.model.predict(reshape(state, env))[0, 0])
        position = 1 if action == 1 else -1
    env.data.loc[:, 'p'].iloc[env.bar] = position
    state, reward, done, info = env.step(action)
env.data['s'] = env.data['p'] * env.data['r']
```

12.7.3 Oanda交易机器人

下面是带有 `OandaTradingBot` 类和部署该类的代码的 Python 脚本。

```
#
# Oanda交易机器人
# 和部署代码
#
# (c) Dr. Yves J. Hilpisch
# Artificial Intelligence in Finance
#
import sys
import tpqoa
import pickle
import numpy as np
import pandas as pd

sys.path.append('../ch11/')


class OandaTradingBot(tpqoa.tpqoa):
    def __init__(self, config_file, agent, granularity, units,
                 sl_distance=None, tsl_distance=None, tp_price=None,
                 verbose=True):
        super(OandaTradingBot, self).__init__(config_file)
        self.agent = agent
        self.symbol = self.agent.learn_env.symbol
        self.env = agent.learn_env
        self.window = self.env.window
        if granularity is None:
            self.granularity = agent.learn_env.granularity
        else:
            self.granularity = granularity
        self.units = units
        self.sl_distance = sl_distance
        self.tsl_distance = tsl_distance
        self.tp_price = tp_price
        self.trades = 0
        self.position = 0
        self.tick_data = pd.DataFrame()
        self.min_length = (self.agent.learn_env.window +
                           self.agent.learn_env.lags)
        self.pl = list()
        self.verbose = verbose
    def _prepare_data(self):
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        ''' 准备（滞后）特征数据
        '''
        self.data['r'] = np.log(self.data / self.data.shift(1))
        self.data.dropna(inplace=True)
        self.data['s'] = self.data[self.symbol].rolling(self.window).mean()
        self.data['m'] = self.data['r'].rolling(self.window).mean()
        self.data['v'] = self.data['r'].rolling(self.window).std()
        self.data.dropna(inplace=True)
        self.data_ = (self.data - self.env.mu) / self.env.std
    def _resample_data(self):
        ''' 对数据进行重采样以适配交易区间长度
        '''
        self.data = self.tick_data.resample(self.granularity,
                                label='right').last().ffill().iloc[:-1]
        self.data = pd.DataFrame(self.data['mid'])
        self.data.columns = [self.symbol,]
        self.data.index = self.data.index.tz_localize(None)
    def _get_state(self):
        ''' 返回金融市场的（当前）状态
        '''
        state = self.data_[self.env.features].iloc[-self.env.lags:]
        return np.reshape(state.values, [1, self.env.lags, self.env.n_features])
    def report_trade(self, time, side, order):
        ''' 返回交易和订单明细
        '''
        self.trades += 1
        pl = float(order['pl'])
        self.pl.append(pl)
        cpl = sum(self.pl)
        print('\n' + 71 * '=')
        print(f'{time} | *** GOING {side} ({self.trades}) ***')
        print(f'{time} | PROFIT/LOSS={pl:.2f} | CUMULATIVE={cpl:.2f}')
        print(71 * '=')
        if self.verbose:
            pprint(order)
            print(71 * '=')
    def on_success(self, time, bid, ask):
        ''' 包含主要交易逻辑
        '''
        df = pd.DataFrame({'ask': ask, 'bid': bid, 'mid': (bid + ask) / 2},
                          index=[pd.Timestamp(time)])
        self.tick_data = self.tick_data.append(df)
        self._resample_data()
        if len(self.data) > self.min_length:
            self.min_length += 1
            self._prepare_data()
            state = self._get_state()
            prediction = np.argmax(self.agent.model.predict(state)[0, 0])
            position = 1 if prediction == 1 else -1
            if self.position in [0, -1] and position == 1:
                order = self.create_order(self.symbol,
                        units=(1 - self.position) * self.units,
                        sl_distance=self.sl_distance,
                        tsl_distance=self.tsl_distance,
                        tp_price=self.tp_price,
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
                        suppress=True, ret=True)
                self.report_trade(time, 'LONG', order)
                self.position = 1
            elif self.position in [0, 1] and position == -1:
                order = self.create_order(self.symbol,
                        units=-(1 + self.position) * self.units,
                        sl_distance=self.sl_distance,
                        tsl_distance=self.tsl_distance,
                        tp_price=self.tp_price,
                        suppress=True, ret=True)
                self.report_trade(time, 'SHORT', order)
                self.position = -1


if __name__ == '__main__':
    agent = pickle.load(open('trading.bot', 'rb'))
    otb = OandaTradingBot('../aiif.cfg', agent, '5s',
                          25000, verbose=False)
    otb.stream_data(agent.learn_env.symbol, stop=1000)
    print('\n' + 71 * '=')
    print('*** CLOSING OUT ***')
    order = otb.create_order(otb.symbol,
                    units=-otb.position * otb.units,
                    suppress=True, ret=True)
    otb.report_trade(otb.time, 'NEUTRAL', order)
    if otb.verbose:
        pprint(order)
    print(71 * '=')
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第五部分

展望

第五部分对人工智能被金融领域广泛采用后可能产生的后果进行了展望。与本书其他部分一样，这里主要关注交易领域，以保持讨论的重点。这一部分由两章组成。

- 第 13 章讨论了金融行业中人工智能驱动竞争的各个方面，比如对金融教育的新要求或可能出现的新竞争场景。
- 第 14 章讨论了金融奇点的前景和人工金融智能的出现。人工金融智能是一种交易机器人通过算法交易持续产生利润，远远超出已知的人类或机构的能力。

这一部分主要是推测性的形而上的讨论，忽略了许多相关且有趣的细节，然而它可以作为对其中所涉及的重要主题进行更深入讨论和分析的起点。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 13 章 基于人工智能的竞争

当今，全球金融市场是一个高风险且竞争异常激烈的人工智能系统运行环境。

——Nick Bostrom，2014 年

金融服务公司越来越依赖人工智能，用它来自动化琐碎的任务、分析数据、改善客户服务并帮助公司遵守法规。

——Nick Huber，2020 年

基于人工智能的系统性和战略性应用，本章讨论了与金融业中的竞争相关的主题。13.1 节对人工智能对未来金融的重要性进行了回顾和总结。13.2 节提出金融中的人工智能仍处于初级阶段，因此在许多情况下实施起来并不容易。另外，这为金融参与者通过人工智能获得竞争优势留下了广阔的竞争空间。13.3 节指出人工智能在金融领域的兴起使得我们必须重新思考并重新设计金融教育和培训，因为传统金融课程已无法满足当今的要求。13.4 节讨论了金融机构将如何争夺必要的资源以在金融领域大规模应用人工智能。与其他许多领域一样，人工智能专家往往是金融公司与科技公司、初创公司和其他行业的公司竞争的焦点。

13.5 节解释了人工智能是形成**微观 alpha** 收益的时代的主要原因，也是唯一解决方案。微观 alpha 收益与当今的黄金一样，仍然有待发现，但只能在小规模上进行，并且在许多情况下只能用工业方法开采。13.6 节讨论了支持和反对金融业以垄断、寡头垄断或完全竞争为特征的未来情景的理由。13.7 节简要介绍了人工智能在金融领域的总体风险以及监管机构和行业监督部门面临的主要问题。

13.1 人工智能和金融

本书主要关注了人工智能在预测金融时间序列中的应用。目的是发现**统计失效**情况，也就是人工智能算法在预测未来市场走势方面优于基线算法的情况。这种统计失效是经济失效

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

的基础。在**经济失效**情况下，需要有一种能够利用统计失效来实现高于市场收益率的交易策略。换句话说，有一种策略（由预测算法和执行算法组成）可以生成 alpha 收益。

当然，还有其他许多领域可以将人工智能算法应用于金融。示例如下。

信用评分

人工智能算法可用于为潜在债务人推导出信用评分，从而支持贷款决策，甚至将贷款决策完全自动化。例如，Golbayani 等（2020）将基于神经网络的方法应用于企业信用评级，而 Babaev 等（2019）将循环神经网络应用于零售贷款的申请。

欺诈识别

人工智能算法可以识别异常模式（比如，在与信用卡相关的交易中），从而防止欺诈行为未被发现，甚至可以防止欺诈发生。Yousefi 等（2019）提供了有关这个研究主题的文献总结。

交易执行

人工智能算法可以学习如何最好地执行与大宗股票相关的交易，从而最大限度地减小市场影响并降低交易成本。Ning 等（2020）的论文应用双深度 Q 学习算法研究了最佳交易执行策略。

衍生品对冲

人工智能算法可以通过训练来最优地执行单一衍生工具或由此类工具组成的投资组合的对冲交易，这种方法通常被称为**深度对冲**。Buehler 等（2019）使用一种强化学习方法实施了深度对冲。

投资组合管理

人工智能算法可用于构建和重新平衡金融工具的投资组合，比如在长期退休储蓄计划的背景下。López de Prado（2020）最近的一本书详细介绍了这个主题。

客户服务

人工智能算法可用于处理自然语言，比如在客户查询的情况下。因此，就像在其他许多行业一样，聊天机器人在金融领域变得非常流行。Yu 等（2020）的论文讨论了一种基于当下流行的由 Transformer 生成的双向编码表征模型的金融聊天机器人，该模型起源于谷歌。

人工智能在金融领域的以上所有应用，以及此处未列出的其他应用都受益于大量相关数据的程序化可用性。为什么我们可以期望机器学习、深度学习和强化学习算法比金融计量经济学的传统方法（如 OLS 回归）表现更好呢？原因有很多。

大数据

虽然传统的统计学方法通常可以处理更大的数据集，但它们并没有从增加的数据量中获得太多的性能优势。另外，在相关性能指标的更大数据集上进行训练时，基于神经网络的方法通常会受益匪浅。

不稳定性

与现实世界不同，金融市场并不遵循恒定的规律。它们随着时间的推移而发生变化，有时变化很快。人工智能算法更容易处理这个问题，比如通过在线训练逐步更新神经网络。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

非线性

以 OLS 回归为例，它会假设自变量和因变量之间存在固有的线性关系。人工智能算法（如神经网络）通常可以更轻松地处理非线性关系。

非正态性

在金融计量经济学中，变量是正态分布的这种假设无处不在。人工智能算法一般不太依赖这种限制性假设。

高维数据

金融计量经济学的传统方法已被证明对低维问题很有用。金融中的许多问题被设置在了具有非常少的特征变量（自变量）的环境中，比如只有一个特征变量（如 CAPM 模型）或者可能再多几个。更先进的人工智能算法可以轻松处理高维问题，如果需要，甚至可以考虑数百种不同的特征。

分类问题

传统计量经济学的分析工具主要基于估计（回归）问题的方法。这些回归问题无疑构成了金融的一个重要范畴。然而，分类问题可能同样重要。机器学习和深度学习的分析工具为解决分类问题提供了大量的选择。

非结构化数据

金融计量经济学的传统方法基本上只能处理结构化的数值数据。机器学习和深度学习算法还能够有效地处理非结构化且基于文本的数据。同时它们还可以有效地处理结构化和非结构化数据。

尽管人工智能在与金融相关的许多领域的应用仍处于起步阶段，但事实证明，某些应用领域在向人工智能优先范式的转变中受益匪浅。因此，可以预测，机器学习、深度学习和强化学习算法将重新塑造金融在实践中的处理和实施方式。此外，人工智能已经成为追求竞争优势的首要工具。

13.2 标准的缺失

传统的规范性金融（参见第 3 章）已达到高度标准化。有许多教科书在不同层面教授和解释了相同的理论和模型，比如 Copeland 等（2005）编著的教科书和 Jones（2012）编著的教科书。反过来，这些理论和模型通常又依赖于过去几十年发表的研究性论文。

当 Black 和 Scholes（1973）以及 Merton（1973）发表了他们用封闭式分析公式为欧式期权合约定价的理论和模型时，金融业立即接受了该公式及其背后的理念，并以此作为相关分析的基准。近 50 年过去了，随着许多改进的理论和模型的提出，Black-Scholes-Merton 模型和公式仍然被认为是期权定价的一个基准，即便不是唯一基准。

另外，人工智能优先的金融的标准化程度明显不足。在这个领域每天都会有大量的论文发表。传统的需要经过同行评审的出版速度通常太慢，无法跟上人工智能领域的快速发展。研究人员通常热衷于尽快与公众分享他们的工作，以免被竞争团队超越。虽然同行评审往往在质量保证方面更有优势，但这一过程可能需要数月时间，而在此期间研究不能被发表。因此从这个意义上说，研究人员越来越依赖大众来进行评审，以确保他们的发现尽早获得认可。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

几十年前，一篇新的金融领域的研究论文在通过同行评审最终被发表之前，可能已经在相关领域专家手中传看多年，这种情况并不罕见，但今天的研究环境已经大为不同，论文的发表速度要快得多，而且研究人员愿意尽早开展可能还没有被完全关注和检验的工作。因此，对于众多被应用于金融问题的人工智能算法，几乎没有任何标准或基准可以参考。

研究发表得如此快速，很大程度上是因为人工智能算法非常容易应用到金融数据的分析中。学生、研究人员和从业人员可以轻而易举地将人工智能的最新突破用于金融领域的研究。与几十年前计量经济学研究受到的限制相比（比如，有限的数据可得性和有限的计算能力），这是一个优势。但这也经常会导致“广撒网，多捕鱼”的想法。

在某种程度上，这种对发表的渴望和紧迫感也是由投资者引起的。投资者会督促投资经理更快地提出新的投资方式，这通常需要摒弃传统的金融研究方法，转而采用更实用的方法。正如 Lopéz de Prado（2018）总结的那样：

> 问题：数学证明可能需要数年、数十年甚至数百年的时间才能完成，但没有一个投资者会等那么久。
>
> 解决方案：使用实验数学。解决棘手的问题时，不是通过证明而是通过实验来实现。

总体而言，标准的缺失为个体金融参与者在竞争环境中利用人工智能优先的金融的优势提供了充足的机会。当我在 2020 年年中撰写本书时，已经感觉到利用人工智能来彻底改变金融的处理方式的竞赛正在全速展开。本章余下内容将讨论基于人工智能的竞争的其他重要方面。

13.3 教育和培训

进入金融领域和金融行业的途径通常是接受与该领域相关的正规教育。相关的常见学位名称如下。

- 金融学硕士
- 计量金融硕士
- 计算金融硕士
- 金融工程硕士
- 量化企业风险管理硕士

从本质上讲，当下所有这些学位都要求学生至少掌握一门编程语言（通常是 Python），以满足数据驱动的金融的数据处理要求。在这方面，大学教育解决了行业对这些技能的需求。Murray（2019）指出：

> 随着公司将人工智能应用于更多的任务，劳动力将不得不适应这种变化。
>
> 金融硕士毕业生有很多的就业机会。技术和金融知识的融合对他们来说是一个很大的优势。
>
> 也许最大的需求来自量化投资者，他们使用人工智能来捕获市场和庞大数据集中的信息，以识别潜在的交易机会。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

大学里与金融相关的学位课程都进行了调整，加入了编程、数据科学和人工智能。公司也在新员工和现有员工的培训上投入了大量的资金，从而为数据驱动的金融和人工智能优先的金融做好准备。Noonan（2018）描述了世界上最大的银行之一——摩根大通，在大规模培训工作中所做的努力：

> 摩根大通正在为数百名新的投资银行家和资产经理提供强制性编码课程，这表明华尔街对技术技能的需求日益增加。
>
> 从人工智能交易到在线借贷平台，技术都在塑造着银行业的未来。金融服务机构正在开发软件以提高效率、创造新产品并抵御来自初创企业和科技巨头的威胁。
>
> 今年，大三学生的编码培训基于 Python 编程，这将帮助他们分析超大数据集并解释自由语言文本等非结构化数据。明年，资产管理部门将增加强制性技术培训内容，将数据科学概念、机器学习和云计算包括进来。

总而言之，金融行业中越来越多的工作将需要能够熟练编程、掌握基础和高级数据科学概念、掌握机器学习以及拥有其他方面技术（如云计算）的员工。在供给和需求这两个方面，大学和金融机构分别通过调整课程和加大员工培训力度来应对这一趋势。在这两种情况下，关键在于有效竞争，至少要保持和该领域的相关性，并能够在因人工智能而永远改变的金融环境中生存。

13.4 资源争夺

为了在金融领域以可扩展且有意义的方式利用人工智能，金融市场的参与者会竞相争夺最好的资源，其中，有 4 大资源至关重要：人力、算法、数据和硬件。

最重要但同时也是最稀缺的资源通常是人工智能领域的专家，尤其是金融领域的人工智能专家。在这方面，金融机构会与科技公司、金融科技初创公司和其他机构竞争最优秀的人才。尽管银行愿意支付较高的薪水，也很难从科技公司吸引到顶尖人才，因为科技公司的文化氛围更适合这些人才，并且薪酬待遇中承诺的股票期权也更有吸引力。因此，金融机构通常采取内部培养人才的方式。

机器学习和深度学习中的许多算法和模型可以被视为经过充分研究、测试和记录的标准算法。然而，在许多情况下，从一开始就不清楚如何在金融环境中最好地应用它们，这也成了金融机构大力投资进行研究的方面。对于许多大型的买方机构（如系统对冲基金），投资和交易策略研究是其商业模式的核心。然而，正如第 12 章所述，部署和生产同等重要。当然，在这种情况下，战略研究和部署都是技术性非常强的议题。

没有数据的算法通常毫无价值。同样，使用来自交易所或数据服务提供商（如 Refinitiv 或 Bloomberg）的“标准”数据的算法也可能只有有限的价值。因为市场上许多相关的参与者对此类数据进行了深入分析，所以很难甚至不可能发现任何能够产生 alpha 收益的机会或类似的竞争优势。因此，大型买方机构在获取**替代数据**方面投入了大量资金（参见 4.3 节）。

如今，替代数据的重要性体现在买方参与者和其他投资者对活跃于该领域的公司的投资中。例如，2018 年一批投资公司向数据集团 Enigma 投资了 9500 万美元。Fortado（2018）将这笔投资及其理由描述如下：

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

> 对冲基金、银行和风险投资公司正纷纷投资于数据公司，并希望从自己使用的更多的数据业务中获利。
>
> 近年来，越来越多的初创企业在大量数据中搜寻有价值的数据，并将其出售给寻求边际优势的投资集团。
>
> 最近，吸引投资者兴趣的公司是总部位于纽约的初创公司 Enigma，该公司周二宣布从量化巨头 Two Sigma、激进对冲基金 Third Point 以及风险投资公司 NEA 和 Glynn Capital 获得了 9500 万美元的融资。

金融机构争夺的第 4 种资源是能够处理金融大数据、实现基于传统数据集和替代数据集的算法，从而有效地将人工智能应用于金融的最佳硬件资源。近年来，在致力于使机器学习和深度学习工作更快、更节能和更具成本效益的硬件方面取得了巨大的创新。虽然传统处理器（如 CPU）在该领域发挥的作用很小，但专用硬件（比如 Nvidia 生产的 GPU、谷歌开发的 TPU 和初创公司 Graphcore 开发的 IPU）已经接替了人工智能。金融机构对新型专业硬件的兴趣体现在最大的对冲基金和做市商之一 Citadel 对 IPU 的研究工作中。该研究工作被记录在 Jia 等（2019）撰写的综合性研究报告中，该报告说明了专用硬件与替代选项相比的潜在优势。

为了争夺人工智能优先的金融中的主导地位，金融机构每年在人才、研究、数据和硬件方面投资数十亿美元。大型机构似乎有能力跟上该领域的步伐，中小型企业则很难将其业务全面转型为人工智能优先的方式。

13.5 市场影响

毫无疑问，数据科学、机器学习和深度学习算法在金融行业的日益广泛使用，对金融市场、投资和交易机会产生了影响。正如本书中许多示例所示，机器学习和深度学习方法能够发现传统计量经济学方法（如多元 OLS 回归）无法发现的统计失效和经济失效问题。因此，可以假设新的和更好的分析方法会使发现能够产生 alpha 收益的机会和策略变得更加困难。

Lopéz de Prado（2018）将金融市场的现状与金矿行业的现状进行了对比，并进行了如下描述：

> 如果十年前个体发现宏观 alpha 收益（使用简单的数学工具，如计量经济学）是相对普遍的，那么目前这种情况发生的可能性正在迅速趋于零。如今，无论经验或知识储备如何，寻找宏观 alpha 收益的个体都面临着巨大的挑战。剩下的唯一真正的 alpha 收益是微观的，找到它需要资本密集型的工业方法。就像黄金一样，微观 alpha 收益并不意味着整体利润的减少。如今，微观 alpha 收益比历史上曾经出现的宏观 alpha 收益要丰富得多。现在，赚钱的机会仍然很多，但需要使用更深度的机器学习工具。

在这种背景下，金融机构看起来更像是在被迫接受人工智能优先的金融，以免掉队甚至倒闭。这不仅适用于投资和交易，也适用于其他领域。虽然银行与商业和零售债务人已经建立起了长期的关系，并自然地具备做出合理信贷决策的能力，但当今的人工智能使竞争环

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

境变得更加公平，并使这些长期关系几乎毫无价值。因此，依靠人工智能，该领域的新进入者（如金融科技初创公司）通常可以用可控且可行的方式快速从现有企业手中抢夺市场份额。另外，这些发展也激励现有企业收购或合并年轻的、有创造力的金融科技初创公司，以保持自己的竞争力。

13.6 竞争场景

未来 3~5 年，人工智能优先的金融驱动的竞争格局会是什么样子？可以想象以下 3 种场景。

垄断

通过将人工智能应用于算法交易等方面的重大且无可比拟的突破，一家金融机构占据了主导地位。例如，在互联网搜索方面，谷歌拥有大约 90% 的全球市场份额。

寡头垄断

少数金融机构能够利用人工智能优先的金融来确立其领先地位。例如，在对冲基金行业存在的寡头垄断，其中少数大公司在管理资产方面占据主导地位。

完全竞争

金融市场的所有参与者都以类似的方式受益于人工智能优先的金融带来的进步，没有一个参与者或一组参与者享有任何竞争优势。从技术上来说，这和如今的计算机国际象棋的情况差不多。许多运行在智能手机等标准硬件上的国际象棋程序在下棋方面明显优于当前的世界冠军（在撰写本书时，国际象棋世界冠军是 Magnus Carlsen）。

很难预测以上哪种情况更有可能发生。我们可以找到论据并描述出这 3 种情况是如何发生的。例如，发生垄断的一个论据可能是算法交易的重大突破，这种突破将带来快速且显著的市场表现，从而有助于通过再投资以及新的资本流入积累更多资本。反过来，这又将增加技术和研究的预算，以保护竞争优势并吸引原本难以争取的人才。整个周期是自我强化型的，谷歌在搜索领域的与在线广告相关的核心业务，就是这方面的一个很好的例子。

同样，我们也有充分的理由预测寡头垄断的出现。现在，几乎可以肯定，交易业务的所有大型参与者都在研究和技术上进行了大量的投资，其中与人工智能相关的投资占预算的很大一部分。至于在其他领域［比如推荐引擎（想想 Netflix 的电影推荐和 Spotify 的音乐推荐)］，多家公司可能也能同时实现类似的突破。可以想象，目前领先的系统性交易者将能够使用人工智能优先的金融来巩固他们的领先地位。

最后要说的是完全竞争的情况。近年来，许多技术变得无处不在。强大的国际象棋程序只是其中的一个例子。其他例子还有地图和导航系统或个人语音助理。在一种完全竞争的情境下，会有相当多的金融参与者争夺微小的创造 alpha 收益的机会，但他们甚至可能无法实现任何与普通市场收益率不同的收益率。

与此同时，也有反对这 3 种情况的观点。当前形势下，为了在金融领域中利用人工智能，许多金融市场参与者具有同样的手段和动机。这使得投资管理领域中不可能只有一个参与者脱颖而出，并在投资管理的市场中抢占类似谷歌在搜索领域的市场份额。同时，在该领域进行研究的中小型和大型参与者的数量以及算法交易的低准入门槛，使得个别参与者不可能获得稳固的竞争优势。关于完全竞争，一个反对的观点是，在可预见的未来，大规模

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

的算法交易需要大量的资金和其他资源。例如，在国际象棋方面，DeepMind 与 AlphaZero 都表明了同一个观点：即使一个领域似乎看起来已经完全定型了，但始终存在创新和重大改进的空间。

13.7 风险、监管和监督

一个简单的谷歌搜索显示，关于人工智能及其监管的总体风险以及金融服务行业的讨论非常活跃。[1] 本节无法涵盖与此相关的所有方面，但至少可以解决一些重要的问题。

以下是人工智能在金融中的应用带来的一些风险。

隐私

金融是一个拥有严格的隐私法的敏感领域。大规模使用人工智能需要（至少部分需要）使用来自客户的私人数据。这便增加了私人数据被泄露或以不当方式使用的风险。这样的风险显然在使用那些公开可用的数据库（如金融时间序列数据）时不存在。

偏置

人工智能算法可以轻松学习到数据（如零售数据或企业客户数据）本身固有的偏置。算法只能在数据允许的情况下尽可能准确且客观地判断潜在债务人的信用度。[2] 同样，在处理市场数据时，学习偏置问题并不是真正的问题。

不可解释性

在许多领域，决策能够被解释（有时是详细的事后解释）是非常重要的。这既可能是法律要求的，也可能是投资者要求的，因为他们想要了解为何要做出特定的投资决定。以投资和交易决策为例，如果一个基于大型神经网络的人工智能通过**算法**决定了何时交易以及交易什么，那么通常很难或不可能详细解释为何人工智能以这种方式进行了交易。研究人员在“可解释的人工智能”方面积极深入地进行了探索，但很显然这方面的研究还存在明显的局限。

羊群效应

自 1987 年股市崩盘以来，金融交易中的羊群效应就很明显了。1987 年，在看跌期权的大规模合成复制程序背景下，正反馈交易与止损单相结合触发了螺旋式下降。在 2008 年对冲基金的崩盘中可以观察到类似的羊群效应，这首次揭示了不同对冲基金实施类似策略的程度。至于 2010 年的闪电崩盘，有些人将其归咎于算法交易，但证据似乎并不明确。然而，当越来越多的机构采用已被证明卓有成效的类似方法时，在交易中更广泛地使用人工智能可能会带来类似的风险。其他领域也容易受到这种影响。信用决策主体可能会根据不同的数据集学习到相同的偏置，并可能使某些群体或个人根本无法获得信用。

消失的 alpha 收益

正如之前所讨论的那样，人工智能在金融领域的更广泛应用可能会导致市场中的 alpha 收益消失。技术必须变得更好，数据可能会变得“更具替代性”，以确保竞争优势。

注 1：有关这些主题的简要概述，请参阅 McKinsey 的以下两篇文章：“Confronting the risks of artificial intelligence”和“Derisking machine learning and artificial intelligence”。

注 2：有关人工智能偏见问题及其解决方案的更多信息，请参阅 Klein（2020）。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 14 章会在潜在的“金融奇点”背景下仔细研究这一点。

除了自身的典型风险，人工智能还给金融领域带来了特有的新风险。与此同时，立法者和监管机构很难跟上该领域的发展，也很难全面评估人工智能优先的金融带来的个人和系统性风险。这有以下几个原因。

专业知识

立法者和监管机构需要像金融参与者一样获得与金融人工智能相关的新的专业知识。在这方面，他们要与大型金融机构和科技公司竞争，众所周知，这些金融机构和科技公司所支付的薪水远高于立法者和监管机构。

数据不足

在许多应用领域，监管机构可以用来判断人工智能实际影响的可用数据很少，甚至根本没有。在某些情况下，甚至可能不知道人工智能是否起作用。即使这种作用是已知的且相关数据是可得的，也可能很难将人工智能的影响与其他相关因素的影响区分开来。

不透明

尽管大多数金融机构试图利用人工智能来确保或获得竞争优势，但单个机构在这方面所做的工作及其实施和使用方式是不透明的。许多人将他们在这种情况下的努力视为知识产权和自己的“秘密武器”。

模型验证

模型验证在许多金融领域中是核心风险管理和监管工具。以基于 Black-Scholes-Merton（1973）期权定价模型的欧式期权定价为例，该模型的一个特定应用所生成的价格可以用 Cox 等（1979）的二项式期权定价模型相互印证。这通常与人工智能算法完全不同，几乎没有一个模型可以基于一组简约的参数来验证复杂人工智能算法的输出。然而，再现性可能是一个可实现的目标（可以选择让第三方通过精确复制所涉及的所有步骤来验证输出）。但这反过来又要求第三方（比如监管机构或审计师）能够访问相同的数据、使用与金融机构的计算设备一样强大的计算设备等。这对需要庞大计算量的人工智能任务来说几乎是不可能实现的。

难以监管

回到期权定价的例子，监管机构可以指定 Black-Scholes-Merton（1973）和 Cox 等（1979）的期权定价模型对于欧式期权的定价都是可以接受的。即使立法者和监管机构指定支持向量机（SVM）算法和神经网络都是“可接受的算法”，也仍然为如何训练、使用这些算法等问题保留了空间。事实上，在这些方面也很难再具体了。例如，监管机构是否应该限制神经网络中隐藏层和 / 或隐藏单元的数量？关于软件包应该做何规定？类似难以回答的问题似乎无穷无尽。因此，只能制定一些一般规则。

科技公司和金融机构通常更喜欢宽松的人工智能监管方式，其原因显而易见。在 Bradshaw（2019）中，谷歌首席执行官 Sundar Pichai 呼吁“智能”监管，并提出应该使用一种区分不同行业的监管方法：

> 谷歌首席执行官警告政治家不要对人工智能进行下意识的监管，他们认为现有规则足以管理新技术。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

> Sundar Pichai 表示，人工智能需要“智能监管”，要在创新与保护公民之间取得平衡……“这是一项如此广泛的跨领域技术，因此在某些垂直领域中更多地进行审视（监管）是非常重要的”。

另外，也有更加严格的人工智能监管的支持者，比如 Elon Musk 在 Matyus（2020）中提到：

> “记住我说的话，”Musk 警告说，“人工智能远比核武器危险。那么为什么我们没有相应的监管呢？”

人工智能在金融领域的风险是多方面的，立法者和监管机构面临的问题也是如此。尽管如此，可以肯定的是，许多司法管辖区会针对金融领域的人工智能进行更严格的监管和监督。

13.8 结论

本章讨论了使用人工智能在金融行业进行竞争的一些问题。人工智能在许多应用领域中带来的好处是显而易见的。然而，到目前为止，几乎还没有建立起任何相关的标准，这个领域似乎仍然为市场参与者争取竞争优势敞开着大门。在提供金融教育和培训时，必须意识到来自数据科学、机器学习、深度学习和更普遍的人工智能的新技术和方法已经渗透到了金融学科的方方面面。许多硕士课程已经调整了课程体系，而大型金融机构则在新员工和现有员工所需技能培训方面进行了大力投资。除了人力资源，金融机构还会争夺该领域的其他资源，比如替代数据。在金融市场中，人工智能驱动的投资和交易使得识别可持续的 alpha 收益的机会变得更加困难。另外，如今，传统的计量经济学方法可能已无法识别和挖掘微观 alpha 收益。

在人工智能接管金融业的时候，很难预测金融业竞争的最终情形。从垄断到寡头垄断再到完全竞争的情形似乎都有其合理性。第 14 章会重新讨论这个话题。人工智能优先的金融使研究人员、从业人员和监管机构面临着新的风险和妥善应对这些风险的挑战，其中一个很重要的风险是许多人工智能算法的黑箱特征。这种风险通常只能一定程度上利用当今最先进的可解释人工智能来缓解。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第 14 章
金融奇点

我们发现自己处于战略复杂性的丛林中，同时周围密布着不确定性的浓雾。

——Nick Bostrom，2014 年

“随着时间的推移，大多数交易和投资角色将消失，大多数需要人工服务的角色可能会实现自动化，”Skinner 先生说，“最终我们看到的会是主要由经理和机器运营的银行。经理决定机器需要做什么，机器负责完成工作。”

——Nick Huber，2020 年

在金融行业中基于人工智能的竞争能否导致金融奇点？这是本章将讨论的主要问题。14.1 节定义了**金融奇点**。14.2 节说明了在潜在财富积累方面 AFI 竞争中的风险。14.3 节在第 2 章的背景下指出了通往 AFI 的可能途径。14.4 节和 14.5 节提出许多资源对创建 AFI 的目标是有用且正交的，任何参与 AFI 竞赛的人都将争夺这些资源。14.6 节分析了本章中讨论的 AFI 是只会使少数人受益，还是会使整个人类受益。

14.1 概念和定义

金融奇点这种说法至少可以追溯到 Shiller 于 2015 年所写的博客文章，在这篇文章中，Shiller 写道：

对于每一种可以想象的投资策略，alpha 收益最终都会变为零吗？更为根本的是，有了这么多聪明的人和更聪明的计算机，金融市场真的会变得完美，我们可以坐下来放松并假设所有资产的定价都正确，这一天会到来吗？

当计算机取代人类智能时，这种想象中的状态可能会被称为金融奇点，类似于假设的未来技术奇点。金融奇点意味着所有投资决策最好由计算机程序来做，因为拥有算法的专家已经找出驱动市场结果的因素并将其简化为一个无缝系统。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

更笼统地说，人们可以将金融奇点定义为计算机和算法开始接管金融和整个金融行业（包括银行、资产管理公司、交易所等）的时间点，人类则作为经理、主管和控制者退居二线。

另外，本着本书强调的精神，我们可以将金融奇点定义为交易机器人存在的时间点，**它显示出了以超人和超级机构的水平，以前所未有的准确率持续预测金融市场走势的能力**。从这个意义上讲，这样的交易机器人将被称为狭义人工智能（ANI），而不是通用人工智能（AGI）或超级智能（参见第 2 章）。

可以假设以交易机器人的形式构建这样的 AFI 比 AGI 甚至超级智能要容易得多。这同样适用于 AlphaZero，因为构建一个在围棋游戏中优于任何人类或其他智能体的人工智能体更容易。因此，即使尚不清楚是否会有符合 AGI 或超级智能标准的人工智能体，但无论如何，出现符合 ANI 或 AFI 标准的交易机器人的可能性要大得多。

为了让讨论尽可能具体并符合本书的内容，本章的重点在于讨论符合 AFI 标准的交易机器人。

14.2 风险是什么

追求 AFI 本身可能具有挑战性且令人兴奋。然而，正如金融领域通常的情况一样，没有多少行为是由利他动机驱动的。相反，大多数行为由财务激励（现金）驱动。但是，在建立 AFI 的竞赛中究竟有什么风险呢？我们不能肯定地或笼统地回答这个问题，但可以通过一些简单的计算来阐明。

要了解与劣质交易策略相比，拥有 AFI 的价值有多大，请考虑以下基准。

牛市策略
: 一种仅在预期价格上涨的金融工具上做多的交易策略。

随机策略
: 为给定金融工具随机选择多头头寸或空头头寸的交易策略。

熊市策略
: 一种仅在预期价格下跌的金融工具上做空的交易策略。

这些基准策略应与具有以下成功特征的 AFI 进行比较。

顶部 *X*%
: AFI 能准确预测顶部 *X*% 的上下波动，对其余市场波动的预测则是随机的。

X% AFI
: AFI 能准确预测所有随机选择的市场波动中的 *X*%，对其余市场波动的预测则是随机的。

以下 Python 代码导入了已知时间序列数据集，其中包含许多金融工具的 EOD 数据。下面的示例依赖于单一金融工具 5 年的 EOD 数据。

```
In [1]: import random
        import numpy as np
        import pandas as pd
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'

In [2]: url = 'https://hilpisch.com/aiif_eikon_eod_data.csv'

In [3]: raw = pd.read_csv(url, index_col=0, parse_dates=True)

In [4]: symbol = 'EUR='

In [5]: raw['bull'] = np.log(raw[symbol] / raw[symbol].shift(1))  ❶

In [6]: data = pd.DataFrame(raw['bull']).loc['2015-01-01':]  ❶

In [7]: data.dropna(inplace=True)

In [8]: data.info()
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 1305 entries, 2015-01-01 to 2020-01-01
        Data columns (total 1 columns):
         #   Column  Non-Null Count  Dtype
        ---  ------  --------------  -----
         0   bull    1305 non-null   float64
        dtypes: float64(1)
        memory usage: 20.4 KB
```

❶ **牛市**基准收益率（仅做多）。

由于牛市策略已经由基础金融工具的对数收益率定义，因此以下 Python 代码指定了另外两个基准策略并推导出了 AFI 策略的表现。在这种情况下，考虑用一些 AFI 策略来说明提升 AFI 预测准确率的影响。

```
In [9]: np.random.seed(100)

In [10]: data['random'] = np.random.choice([-1, 1], len(data)) * data['bull']  ❶

In [11]: data['bear'] = -data['bull']  ❷

In [12]: def top(t):
             top = pd.DataFrame(data['bull'])
             top.columns = ['top']
             top = top.sort_values('top')
             n = int(len(data) * t)
             top['top'].iloc[:n] = abs(top['top'].iloc[:n])
             top['top'].iloc[n:] = abs(top['top'].iloc[n:])
             top['top'].iloc[n:-n] = np.random.choice([-1, 1],
                             len(top['top'].iloc[n:-n])) * top['top'].iloc[n:-n]
             data[f'{int(t * 100)}_top'] = top.sort_index()

In [13]: for t in [0.1, 0.15]:
             top(t)  ❸
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [14]: def afi(ratio):
             correct = np.random.binomial(1, ratio, len(data))
             random = np.random.choice([-1, 1], len(data))
             strat = np.where(correct, abs(data['bull']), random * data['bull'])
             data[f'{int(ratio * 100)}_afi'] = strat

In [15]: for ratio in [0.51, 0.6, 0.75, 0.9]:
             afi(ratio)  ❹
```

❶ **随机**基准收益率。

❷ **熊市**基准收益率（仅做空）。

❸ **顶部** *X*% 策略收益率。

❹ *X*% AFI 策略收益率。

使用第 10 章中介绍的标准的向量化回测方法（忽略交易成本），可以清楚地看出预测准确率的显著提高在金融方面的意义。考虑一下“90% AFI”，它的预测并不完美，而是在 10% 的情况下没有任何优势。假设 90% 的准确率导致 5 年内的总收益几乎是投资资本的 100 倍（不考虑交易成本）。以 75% 的准确率，AFI 仍将有几乎是投资资本 50 倍的收益（参见图 14-1）。这不包括杠杆，在此类预测准确率之下，杠杆可以以几乎无风险的方式被轻松添加。

```
In [16]: data.head()
Out[16]:                 bull    random      bear    10_top    15_top    51_afi  \
         Date
         2015-01-01  0.000413 -0.000413 -0.000413  0.000413 -0.000413  0.000413
         2015-01-02 -0.008464  0.008464  0.008464  0.008464  0.008464  0.008464
         2015-01-05 -0.005767 -0.005767  0.005767 -0.005767  0.005767 -0.005767
         2015-01-06 -0.003611 -0.003611  0.003611 -0.003611  0.003611  0.003611
         2015-01-07 -0.004299 -0.004299  0.004299  0.004299  0.004299  0.004299

                       60_afi    75_afi    90_afi
         Date
         2015-01-01  0.000413  0.000413  0.000413
         2015-01-02  0.008464  0.008464  0.008464
         2015-01-05  0.005767 -0.005767  0.005767
         2015-01-06  0.003611  0.003611  0.003611
         2015-01-07  0.004299  0.004299  0.004299

In [17]: data.sum().apply(np.exp)
Out[17]: bull       0.926676
         random     1.097137
         bear       1.079126
         10_top     9.815383
         15_top    21.275448
         51_afi    12.272497
         60_afi    22.103642
         75_afi    49.227314
         90_afi    98.176658
         dtype: float64

In [18]: data.cumsum().apply(np.exp).plot(figsize=(10, 6));
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

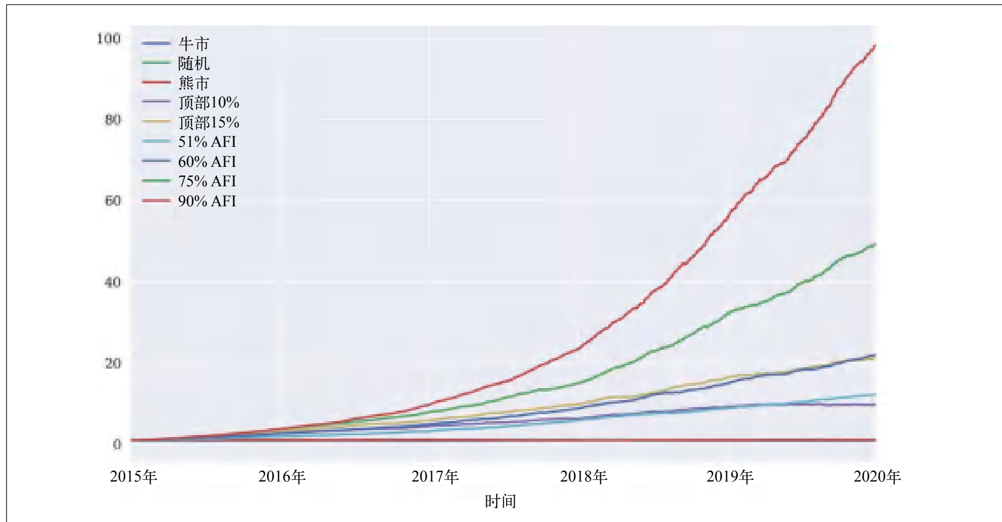


图 14-1：随时间变化的基准策略和理论 AFI 策略的总体表现

分析表明，尽管需要做几个简化的假设，但仍有很多问题有很大的不确定性。在这种情况下，时间起着重要的作用。在 10 年内重新实施相同的分析会使这些数字更加令人印象深刻——在交易环境中几乎是不可想象的。正如以下输出所示，在“90% AFI”的情况下，总收益将超过投资资本的 16 000 倍（不考虑交易成本）。复利和再投资的效果是巨大的。

```
bull          0.782657
random        0.800253
bear          1.277698
10_top      165.066583
15_top     1026.275100
51_afi      206.639897
60_afi      691.751006
75_afi     2947.811043
90_afi    16581.526533
dtype: float64
```

14.3　通往金融奇点的途径

AFI 的出现将是一个特定环境中的特定事件。例如，没有必要模拟人类的大脑，因为 AGI 或超级智能不是主要目标。鉴于似乎没有人在金融市场的交易中始终表现出色，因此试图模仿人脑来达到 AFI 可能是一条死路。我们也无须担心 AFI 会变成人的化身，因为 AFI 只能作为软件在连接到所需数据和交易 API 的适当基础设施上存在。

另外，鉴于此问题的本质，人工智能似乎是通往 AFI 的一条很有前途的道路：将大量金融数据和其他数据作为输入，并生成对未来价格走势方向的预测。这正是本书中介绍和应用的算法的全部内容，尤其是那些属于监督学习和强化学习类别的算法。

还有一种选择可能是人类和机器智能的混合体。几十年来，机器一直支持人类交易者，但

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

在许多情况下，角色已经发生了变化。人类通过提供理想环境和最新数据、仅在极端情况下进行干预等方式来支持机器进行交易。在许多情况下，机器的算法交易决策已经完全自主。或者正如文艺复兴科技公司（最成功且最神秘的系统交易对冲基金之一）的创始人 Jim Simons 所说："唯一的规则是我们永远不要更改计算机的决定。"

虽然目前还不清楚哪些路径可能会生成超级智能，但从今天的角度来看，人工智能似乎最有可能为金融奇点和 AFI 铺平道路。

14.4 正交技能和资源

第 13 章讨论了金融行业中基于人工智能的竞争背景下的资源竞争。这方面的 4 大资源分别是人力资源（专家）、算法和软件、金融和替代数据，以及高性能硬件。这里可以加入第 5 种资源，即获取其他资源所需的资本。

根据正交性假设，获得这种正交技能和资源是明智的，甚至是必要的，无论 AFI 将如何实现，这些都是有用的。参与 AFI 建造竞赛的金融机构将尝试获得尽可能多的、他们能够负担和合理化的高质量资源，在某一条通往 AFI 的道路变得清晰的时候将自己置于尽可能有利的地位。

在一个由人工智能优先的金融驱动的世界中，这种行为和定位可能会决定是繁荣、仅仅生存还是离开市场。不排除这一进展可能比预期要快得多。当 Nick Bostrom 在 2014 年预测人工智能可能需要 10 年才能在围棋比赛中击败世界冠军时，基本上没有人预料到它会在两年后发生。主要驱动力是强化学习在此类游戏中的应用取得了突破，并且如今的许多其他应用也仍然从中受益。金融界也不能排除这种不可预见的突破。

14.5 之前和之后的情景

我们有理由认为，世界各地所有主要金融机构和许多非金融实体目前正在研究如何将人工智能应用于金融领域，或者已有一定的实践经验。然而，并非金融行业的所有参与者都处于同样有利的地位，能够首先实现交易 AFI。一些参与者，比如银行，会受到监管要求的限制；另一些参与者，比如交易所，只是遵循不同的商业模式；还有一些参与者，比如某些资产管理公司，会专注于提供与更广泛的市场指数相仿的低成本、商品化的投资产品（如 ETF）。换句话说，生成 alpha 收益并不是每个金融机构的主要目标。

因此，从外部角度来看，大型对冲基金似乎最有能力充分利用人工智能优先的金融和人工智能驱动的算法交易。总的来说，它们已经拥有该领域所需要的大量重要资源，包括才华横溢且受过良好教育的人、交易算法方面的经验、对传统和替代数据源的几乎无限制的访问权限，以及可扩展的专业交易基础设施。如果缺少某些东西，则高额的技术预算可确保快速和有针对性的投资。

目前尚不清楚是否会先出现一个 AFI，随后出现其他 AFI，或者是否可能同时出现多个 AFI。如果存在多个 AFI，则可以说是多极或寡头垄断情景。AFI 可能主要是相互竞争，而"非 AFI"玩家会被边缘化。这些项目的发起人将努力获得优势，无论优势多么小，因为这可能会让一个 AFI 完全接管并最终成为单一的或垄断的 AFI。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

我们也可以想象一个“赢家通吃”的局面从一开始就占据上风，在这种情况下，某个 AFI 会出现并能够迅速在金融交易中达到任何其他竞争对手无法匹敌的主导地位。这可能有几个原因。一个原因可能是第一个 AFI 产生的收益率如此可观，以至于管理的资产以惊人的速度膨胀，使其能够获得更多相关资源的预算越来越多。另一个原因可能是第一个 AFI 迅速达到其行动可以产生市场影响的规模（比如具有操纵市场价格的能力），使其成为金融市场的主要甚至唯一驱动力。

理论上，监管可以防止 AFI 变得过大或获得过多的市场力量。主要问题是这些法律在实践中是否具有可执行性，以及它们究竟需要如何设计才能达到预期效果。

14.6　星际迷航还是星球大战

对许多人来说，金融业代表了资本主义最纯粹的形式，这是贪婪驱动一切的行业。毫无疑问，它也是一个竞争激烈的行业。特别是，交易和投资管理通常是亿万富翁经理和所有者的象征，而这些人愿意下大赌注并与竞争对手正面交锋，以达成下一笔大型交易。如第 13 章所述，人工智能的出现为雄心勃勃的管理者提供了丰富的工具集，从而将竞争推向了新的高度。

然而，问题是，那些最终可能会产生 AFI 的人工智能优先的金融是否会带来金融乌托邦或反乌托邦？理论上，系统的、无误的财富积累可以只为少数人服务，也可以潜在地为全人类服务。不幸的是，可以假设只有生成 AFI 的项目发起人才能直接从本章中想象的 AFI 中受益。这是因为这样的 AFI 只能通过在金融市场上交易而不是通过发明新产品、解决重要问题或发展企业和行业来产生利润。换句话说，一个仅仅通过在金融市场上交易来产生利润的 AFI 是在参与零和游戏，并没有直接增加可分配的财富。

有人可能会争辩，投资于 AFI 管理的基金的养老基金也将从其丰厚的收益中受益。但这同样只会使特定群体而不是整个人类受益。一个成功的 AFI 项目的发起人是否愿意向外部投资者开放也是问题。这方面的一个很好的例子是 Medallion 基金，它由 Renaissance Technologies 管理，是历史上表现最好的投资工具之一。1993 年，Renaissance 对外部投资者关闭了基本上完全由机器运营的 Medallion。它的出色表现肯定已经吸引了大量额外资产。然而，一些特定的因素（比如某些策略的能力）在这种情况下发挥了作用，类似的考虑因素也可能适用于 AFI。

因此，虽然人们可以期望超级智能会帮助克服整个人类面临的一些基本问题（比如严重的疾病、环境问题、来自外太空的未知威胁等），但 AFI 更有可能导致市场上更多的不平等和更激烈的竞争。与以平等和取之不尽的资源为特征的类似《星际迷航》的世界不同，AFI 反而可能会带来类似《星球大战》的世界，在这个世界中，充满了激烈的贸易战和可用资源的竞争。

14.7　结论

本章从较高的层面讨论了金融奇点和 AFI 的概念。AFI 是一种 ANI，它缺乏超级智能的许多能力和特征。AFI 可以与 AlphaZero 相提并论，后者是用于玩棋盘游戏（比如国际象棋

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

或围棋）的 ANI。AFI 则擅长交易金融工具的游戏。当然，与玩棋盘游戏相比，在金融交易中风险更大。

同 AlphaZero 类似，与其他替代路径（如模拟人脑）相比，人工智能更有可能为 AFI 的产生铺平道路。虽然路径还不是完全可见的，人们也无法确定单个项目已经取得了多大进展，但不管对于哪条路径，许多工具资源是非常重要的，包括专家、算法、数据、硬件和资本。大型成功的对冲基金似乎最有能力赢得 AFI 的竞争。

虽然事实证明不可能像本章所描绘的那样创建 AFI，但将人工智能系统地引入金融领域肯定会激励创新，并且在许多情况下会加剧行业竞争。人工智能不是一种时尚，而是最终将导致行业范式转变的趋势。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

第六部分

附录

作为附录，第六部分提供了额外的材料来支持本书其他部分中的内容、代码和示例。这一部分由 3 个附录组成。

- 附录 A 涵盖了与神经网络相关的基本概念，比如张量运算。
- 附录 B 展示了从头开始实现简单神经网络的类和浅层神经网络的类。
- 附录 C 演示了使用 `Keras` 框架的卷积神经网络的应用。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

附录A

交互式神经网络

本附录基于简单的浅层神经网络，使用基本的 Python 代码来探索神经网络的基本概念，目标是在使用标准机器学习包和深度学习包时，能够很好地理解和感知经常被忽略的高级抽象 API 背后的重要概念。

本附录由以下几节组成。

- A.1 节涵盖了**张量**的基础知识以及对它们实施的运算。
- A.2 节讨论了**简单神经网络**，即只有输入层和输出层的神经网络。
- A.3 节侧重于**浅层神经网络**，即具有一个隐藏层的神经网络。

A.1　张量和张量运算

除了实现几个导入和配置之外，以下 Python 代码显示了与本附录的目的相关的 4 种类型的张量：标量张量、向量张量、矩阵张量和三阶张量。在 Python 中，张量通常表示为潜在的多维 ndarray 对象。有关详细信息和示例，请参阅 Chollet（2017）的第 12 章。

```
In [1]: import math
        import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        np.random.seed(1)
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        np.set_printoptions(suppress=True)

In [2]: t0 = np.array(10)  ❶
        t0  ❶
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
Out[2]: array(10)

In [3]: t1 = np.array((2, 1))  ❷
        t1  ❷
Out[3]: array([2, 1])

In [4]: t2 = np.arange(10).reshape(5, 2)  ❸
        t2  ❸
Out[4]: array([[0, 1],
               [2, 3],
               [4, 5],
               [6, 7],
               [8, 9]])

In [5]: t3 = np.arange(16).reshape(2, 4, 2)  ❹
        t3  ❹
Out[5]: array([[[ 0,  1],
                [ 2,  3],
                [ 4,  5],
                [ 6,  7]],

               [[ 8,  9],
                [10, 11],
                [12, 13],
                [14, 15]]])
```

❶ 标量张量。

❷ 向量张量。

❸ 矩阵张量。

❹ 三阶张量。

在神经网络环境中，张量上的几个数学运算很重要，比如逐元素运算或点积。

```
In [6]: t2 + 1  ❶
Out[6]: array([[ 1,  2],
               [ 3,  4],
               [ 5,  6],
               [ 7,  8],
               [ 9, 10]])

In [7]: t2 + t2  ❷
Out[7]: array([[ 0,  2],
               [ 4,  6],
               [ 8, 10],
               [12, 14],
               [16, 18]])

In [8]: t1
Out[8]: array([2, 1])

In [9]: t2
Out[9]: array([[0, 1],
               [2, 3],
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
               [4, 5],
               [6, 7],
               [8, 9]])

In [10]: np.dot(t2, t1)  ❸
Out[10]: array([ 1, 7, 13, 19, 25])

In [11]: t2[:, 0] * 2 + t2[:, 1] * 1  ❹
Out[11]: array([ 1,  7, 13, 19, 25])

In [12]: np.dot(t1, t2.T)  ❸
Out[12]: array([ 1,  7, 13, 19, 25])
```

❶ 广播运算。

❷ 逐元素运算。

❸ 使用 `NumPy` 函数的点积。

❹ 显式的点积运算。

A.2　简单神经网络

基于张量的基础知识，来考虑一个简单神经网络，它只有一个输入层和一个输出层。

A.2.1　估计

第一个问题是**估计问题**，其标签为实数值。

```
In [13]: features = 3  ❶

In [14]: samples = 5  ❷

In [15]: l0 = np.random.random((samples, features))  ❸
         l0  ❸
Out[15]: array([[0.417022  ,  0.72032449, 0.00011437],
                [0.30233257, 0.14675589, 0.09233859],
                [0.18626021, 0.34556073, 0.39676747],
                [0.53881673, 0.41919451, 0.6852195 ],
                [0.20445225, 0.87811744, 0.02738759]])

In [16]: w = np.random.random((features, 1))  ❹
         w  ❹
Out[16]: array([[0.67046751],
                [0.4173048 ],
                [0.55868983]])

In [17]: l2 = np.dot(l0, w)  ❺
         l2  ❺
Out[17]: array([[0.58025848],
                [0.31553474],
                [0.49075552],
                [0.91901616],
                [0.51882238]])
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [18]: y = l0[:, 0] * 0.5 + l0[:, 1] ❻
         y = y.reshape(-1, 1) ❻
         y ❻
Out[18]: array([[0.9288355 ],
                [0.29792218],
                [0.43869083],
                [0.68860288],
                [0.98034356]])
```

❶ 特征数量。

❷ 样本数量。

❸ 随机输入层。

❹ 随机权重。

❺ 点积运算输出层。

❻ 要学习的标签。

以下 Python 代码逐步完成了一个学习回合，从误差的计算到权重更新后的均方误差（MSE）的计算。

```
In [19]: e = l2 - y  ❶
         e  ❶
Out[19]: array([[-0.34857702],
                [ 0.01761256],
                [ 0.05206469],
                [ 0.23041328],
                [-0.46152118]])

In [20]: mse = (e ** 2).mean()  ❷
         mse  ❷
Out[20]: 0.07812379019517127

In [21]: d = e * 1  ❸
         d  ❸
Out[21]: array([[-0.34857702],
                [ 0.01761256],
                [ 0.05206469],
                [ 0.23041328],
                [-0.46152118]])

In [22]: a = 0.01  ❹

In [23]: u = a * np.dot(l0.T, d)  ❺
         u  ❺
Out[23]: array([[-0.0010055 ],
                [-0.00539194],
                [ 0.00167488]])

In [24]: w ❻
Out[24]: array([[0.67046751],
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
                [0.4173048 ],
                [0.55868983]])

In [25]: w -= u  ❻

In [26]: w  ❻
Out[26]: array([[0.67147301],
                [0.42269674],
                [0.55701495]])

In [27]: l2 = np.dot(l0, w)  ❼

In [28]: e = l2 - y  ❽

In [29]: mse = (e ** 2).mean()  ❾
         mse  ❾
Out[29]: 0.07681782193617318
```

❶ 估计误差。

❷ 给定估计的 MSE 值。

❸ 反向传播（这里 d = e[1]）。

❹ 学习率。

❺ 更新值。

❻ 更新前后的权重。

❼ 更新后的新输出层（估计值）。

❽ 更新后的新误差值。

❾ 更新后的新 MSE 值。

为了改进估计，相同的过程通常需要重复多次。在下面的代码中，学习率提高了，并且程序执行了几百次。最终的 MSE 值很低，估计也很好。

```
In [30]: a = 0.025  ❶

In [31]: w = np.random.random((features, 1))  ❷
         w  ❷
Out[31]: array([[0.14038694],
                [0.19810149],
                [0.80074457]])

In [32]: steps = 800  ❸

In [33]: for s in range(1, steps + 1):
             l2 = np.dot(l0, w)
             e = l2 - y
             u = a * np.dot(l0.T, e)
```

注 1：由于没有隐藏层，因此反向传播以 1 作为导数值。输出层和输入层直接相连。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            w -= u
            mse = (e ** 2).mean()
            if s % 50 == 0:
                print(f'step={s:3d} | mse={mse:.5f}')
        step= 50 | mse=0.03064
        step=100 | mse=0.01002
        step=150 | mse=0.00390
        step=200 | mse=0.00195
        step=250 | mse=0.00124
        step=300 | mse=0.00092
        step=350 | mse=0.00074
        step=400 | mse=0.00060
        step=450 | mse=0.00050
        step=500 | mse=0.00041
        step=550 | mse=0.00035
        step=600 | mse=0.00029
        step=650 | mse=0.00024
        step=700 | mse=0.00020
        step=750 | mse=0.00017
        step=800 | mse=0.00014

In [34]: l2 - y  ❹
Out[34]: array([[-0.01240168],
                [-0.01606065],
                [ 0.01274072],
                [-0.00087794],
                [ 0.01072845]])

In [35]: w  ❺
Out[35]: array([[0.41907514],
                [1.02965827],
                [0.04421136]])
```

❶ 调整学习率。

❷ 初始随机权重。

❸ 学习步骤数。

❹ 估计的残差。

❺ 网络的最终权重。

A.2.2 分类

第二个问题是**分类问题**，其标签是二元整数值。为了提高学习算法的性能，将 sigmoid **函数**用于输出层的激活函数。图 A-1 显示了带有一阶导数的 sigmoid 函数，并将其与简单的阶跃函数进行了比较。

```
In [36]: def sigmoid(x, deriv=False):
             if deriv:
                 return sigmoid(x) * (1 - sigmoid(x))
             return 1 / (1 + np.exp(-x))
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [37]: x = np.linspace(-10, 10, 100)

In [38]: plt.figure(figsize=(10, 6))
         plt.plot(x, np.where(x > 0, 1, 0), 'y--', label='step function')
         plt.plot(x, sigmoid(x), 'r', label='sigmoid')
         plt.plot(x, sigmoid(x, True), '--', label='derivative')
         plt.legend();
```

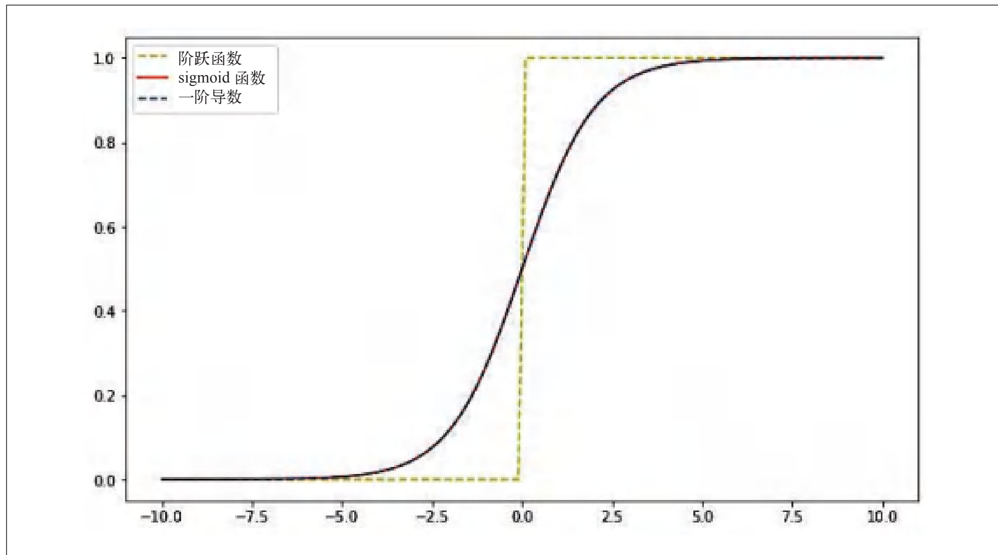


图 A-1：阶跃函数、sigmoid 函数及其一阶导数

为简单起见，分类问题基于随机二元特征和二元标签数据。除了不同的特征和标签数据外，只有输出层的激活函数与估计问题不同，更新神经网络权重的学习算法基本相同。

```
In [39]: features = 4
         samples = 5

In [40]: l0 = np.random.randint(0, 2, (samples, features))  ❶
         l0  ❶
Out[40]: array([[1, 1, 1, 1],
                [0, 1, 1, 0],
                [0, 1, 0, 0],
                [1, 1, 1, 0],
                [1, 0, 0, 1]])

In [41]: w = np.random.random((features, 1))
         w
Out[41]: array([[0.42110763],
                [0.95788953],
                [0.53316528],
                [0.69187711]])

In [42]: l2 = sigmoid(np.dot(l0, w))  ❷
         l2
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
Out[42]: array([[0.93112111],
                [0.81623654],
                [0.72269905],
                [0.87126189],
                [0.75268514]])

In [43]: l2.round()
Out[43]: array([[1.],
                [1.],
                [1.],
                [1.],
                [1.]])

In [44]: y = np.random.randint(0, 2, samples)  ❸
         y = y.reshape(-1, 1)  ❸
         y  ❸
Out[44]: array([[1],
                [1],
                [0],
                [0],
                [0]])

In [45]: e = l2 - y
         e
Out[45]: array([[-0.06887889],
                [-0.18376346],
                [ 0.72269905],
                [ 0.87126189],
                [ 0.75268514]])

In [46]: mse = (e ** 2).mean()
         mse
Out[46]: 0.37728788783411127

In [47]: a = 0.02

In [48]: d = e * sigmoid(l2, True)  ❹
         d
Out[48]: array([[-0.01396723],
                [-0.03906484],
                [ 0.15899479],
                [ 0.18119776],
                [ 0.16384833]])

In [49]: u = a * np.dot(l0.T, d)
         u
Out[49]: array([[0.00662158],
                [0.00574321],
                [0.00256331],
                [0.00299762]])

In [50]: w
Out[50]: array([[0.42110763],
                [0.95788953],
                [0.53316528],
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
                [0.69187711]])
In [51]: w -= u

In [52]: w
Out[52]: array([[0.41448605],
                [0.95214632],
                [0.53060197],
                [0.68887949]])
```

❶ 具有二元特征的输入层。

❷ sigmoid 被激活的输出层。

❸ 二元标签数据。

❹ 通过一阶导数反向传播。

和以前一样，需要对学习步骤进行大量迭代才能获得准确的分类结果。根据抽取的随机数，是可能达到 100% 准确率的，如下所示。

```
In [53]: steps = 3001

In [54]: a = 0.025

In [55]: w = np.random.random((features, 1))
         w
Out[55]: array([[0.41253884],
                [0.03417131],
                [0.62402999],
                [0.66063573]])

In [56]: for s in range(1, steps + 1):
             l2 = sigmoid(np.dot(l0, w))
             e = l2 - y
             d = e * sigmoid(l2, True)
             u = a * np.dot(l0.T, d)
             w -= u
             mse = (e ** 2).mean()
             if s % 200 == 0:
                 print(f'step={s:4d} | mse={mse:.4f}')
         step= 200 | mse=0.1899
         step= 400 | mse=0.1572
         step= 600 | mse=0.1349
         step= 800 | mse=0.1173
         step=1000 | mse=0.1029
         step=1200 | mse=0.0908
         step=1400 | mse=0.0806
         step=1600 | mse=0.0720
         step=1800 | mse=0.0646
         step=2000 | mse=0.0583
         step=2200 | mse=0.0529
         step=2400 | mse=0.0482
         step=2600 | mse=0.0441
         step=2800 | mse=0.0405
         step=3000 | mse=0.0373
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [57]: l2
Out[57]: array([[0.71220474],
               [0.92308745],
               [0.16614971],
               [0.20193503],
               [0.17094583]])

In [58]: l2.round() == y
Out[58]: array([[ True],
               [ True],
               [ True],
               [ True],
               [ True]])

In [59]: w
Out[59]: array([[-3.86002022],
               [-1.61346536],
               [ 4.09895004],
               [ 2.28088807]])
```

A.3 浅层神经网络

上一节的神经网络仅由一个输入层和一个输出层组成，换句话说，输入层和输出层是直接连接的。而**浅层神经网络**在输入层和输出层之间有一个隐藏层，鉴于这种结构，需要两组权重来连接神经网络中的这 3 层。本节会分析用于估计和分类的浅层神经网络。

A.3.1 估计

和上一节一样，首先考虑估计问题。以下 Python 代码构建了具有三层神经元和两组权重的神经网络。第一个步骤序列通常称为**前向传播**。此上下文中的输入层矩阵通常是满秩的，表明可能得到完美的估计结果。

```
In [60]: features = 5
         samples = 5

In [61]: l0 = np.random.random((samples, features))  ❶
         l0  ❶
Out[61]: array([[0.29849529, 0.44613451, 0.22212455, 0.07336417, 0.46923853],
               [0.09617226, 0.90337017, 0.11949047, 0.52479938, 0.083623  ],
               [0.91686133, 0.91044838, 0.29893011, 0.58438912, 0.56591203],
               [0.61393832, 0.95653566, 0.26097898, 0.23101542, 0.53344849],
               [0.94993814, 0.49305959, 0.54060051, 0.7654851 , 0.04534573]])

In [62]: np.linalg.matrix_rank(l0)  ❷
Out[62]: 5

In [63]: units = 3  ❸

In [64]: w0 = np.random.random((features, units))  ❹
         w0  ❹
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
Out[64]: array([[0.13996612, 0.79240359, 0.02980136],
               [0.88312548, 0.54078819, 0.44798018],
               [0.89213587, 0.37758434, 0.53842469],
               [0.65229888, 0.36126102, 0.57100856],
               [0.63783648, 0.12631489, 0.69020459]])

In [65]: l1 = np.dot(l0, w0)  ❺
         l1  ❺
Out[65]: array([[0.98109007, 0.64743919, 0.69411448],
               [1.31351565, 0.81000928, 0.82927653],
               [1.94121167, 1.61435539, 1.32042417],
               [1.65444429, 1.25315104, 1.08742312],
               [1.57892999, 1.50576525, 1.00865941]])

In [66]: w1 = np.random.random((units, 1))  ❻
         w1  ❻
Out[66]: array([[0.6477494 ],
               [0.35393909],
               [0.76323305]])

In [67]: l2 = np.dot(l1, w1)  ❼
         l2  ❼
Out[67]: array([[1.39442565],
               [1.77045418],
               [2.83659354],
               [2.3451617 ],
               [2.32554234]])

In [68]: y = np.random.random((samples, 1))  ❽
         y  ❽
Out[68]: array([[0.35653172],
               [0.75278835],
               [0.88134183],
               [0.01166919],
               [0.49810907]])
```

❶ 随机输入层。

❷ 输入层矩阵的秩。

❸ 隐藏单元数。

❹ 给定 `features` 参数和 `units` 参数的第一组随机权重。

❺ 给定输入层和权重的隐藏层。

❻ 第二组随机权重。

❼ 给定隐藏层和权重的输出层。

❽ 随机标签数据。

相对于估计误差，第二个步骤序列通常称为**反向传播**。更新两组权重，从输出层开始，更新隐藏层和输出层之间的权重集合 `w1`。之后，基于更新后的权重集合 `w1`，更新输入层和隐藏层之间的权重集合 `w0`。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [69]: e2 = l2 - y  ❶
         e2  ❶
Out[69]: array([[1.03789393],
                [1.01766583],
                [1.95525171],
                [2.33349251],
                [1.82743327]])

In [70]: mse = (e2 ** 2).mean()
         mse
Out[70]: 2.9441152813655007

In [71]: d2 = e2 * 1  ❶
         d2  ❶
Out[71]: array([[1.03789393],
                [1.01766583],
                [1.95525171],
                [2.33349251],
                [1.82743327]])

In [72]: a = 0.05

In [73]: u2 = a * np.dot(l1.T, d2)  ❶
         u2  ❶
Out[73]: array([[0.64482837],
                [0.51643336],
                [0.42634283]])

In [74]: w1  ❶
Out[74]: array([[0.6477494 ],
                [0.35393909],
                [0.76323305]])

In [75]: w1 -= u2  ❶

In [76]: w1  ❶
Out[76]: array([[ 0.00292103],
                [-0.16249427],
                [ 0.33689022]])

In [77]: e1 = np.dot(d2, w1.T)  ❷

In [78]: d1 = e1 * 1  ❷

In [79]: u1 = a * np.dot(l0.T, d1)  ❷

In [80]: w0 -= u1  ❷

In [81]: w0  ❷
Out[81]: array([[ 0.13918198, 0.8360247 , -0.06063583],
                [ 0.88220599, 0.59193836,  0.34193342],
                [ 0.89176585, 0.39816855,  0.49574861],
                [ 0.65175984, 0.39124762,  0.50883904],
                [ 0.63739741, 0.15074009,  0.63956519]])
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

❶ 权重集合 w1 的更新过程。

❷ 权重集合 w0 的更新过程。

以下 Python 代码会将学习（网络权重的更新）实现为具有较大迭代次数的 for 循环。通过增加迭代次数，估计结果可以是任意精确的。

```
In [82]: a = 0.015
         steps = 5000

In [83]: for s in range(1, steps + 1):
             l1 = np.dot(l0, w0)
             l2 = np.dot(l1, w1)
             e2 = l2 - y
             u2 = a * np.dot(l1.T, e2)
             w1 -= u2
             e1 = np.dot(e2, w1.T)
             u1 = a * np.dot(l0.T, e1)
             w0 -= u1
             mse = (e2 ** 2).mean()
             if s % 750 == 0:
                 print(f'step={s:5d} | mse={mse:.6f}')
         step= 750 | mse=0.039263
         step= 1500 | mse=0.009867
         step= 2250 | mse=0.000666
         step= 3000 | mse=0.000027
         step= 3750 | mse=0.000001
         step= 4500 | mse=0.000000

In [84]: l2
Out[84]: array([[0.35634333],
                [0.75275415],
                [0.88135507],
                [0.01179945],
                [0.49809208]])

In [85]: y
Out[85]: array([[0.35653172],
                [0.75278835],
                [0.88134183],
                [0.01166919],
                [0.49810907]])

In [86]: (l2 - y)
Out[86]: array([[-0.00018839],
                [-0.00003421],
                [ 0.00001324],
                [ 0.00013025],
                [-0.00001699]])
```

A.3.2 分类

接下来是分类问题。在这种情况下的实现非常接近于估计问题，再次将 sigmoid 函数用于激活函数。以下 Python 代码首先会生成随机样本数据。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [87]: features = 5
         samples = 10
         units = 10

In [88]: np.random.seed(200)
         l0 = np.random.randint(0, 2, (samples, features))  ❶
         w0 = np.random.random((features, units))
         w1 = np.random.random((units, 1))
         y = np.random.randint(0, 2, (samples, 1))  ❷

In [89]: l0  ❶
Out[89]: array([[0, 1, 0, 0, 0],
                [1, 0, 1, 1, 0],
                [1, 1, 1, 1, 0],
                [0, 0, 1, 1, 1],
                [1, 1, 1, 1, 0],
                [1, 1, 0, 1, 0],
                [0, 1, 0, 1, 0],
                [0, 1, 0, 0, 1],
                [0, 1, 1, 1, 1],
                [0, 0, 1, 0, 0]])

In [90]: y  ❷
Out[90]: array([[1],
                [0],
                [1],
                [0],
                [1],
                [0],
                [0],
                [0],
                [1],
                [1]])
```

❶ 二元特征数据（输入层）。

❷ 二元标签数据。

学习算法的实现会根据需要再次使用 for 循环来尽可能频繁地重复权重更新步骤。根据为特征数据和标签数据生成的随机数，经过足够的学习步骤后，准确率可以达到 100%。

```
In [91]: a = 0.1
         steps = 20000

In [92]: for s in range(1, steps + 1):
             l1 = sigmoid(np.dot(l0, w0))  ❶
             l2 = sigmoid(np.dot(l1, w1))  ❶
             e2 = l2 - y  ❷
             d2 = e2 * sigmoid(l2, True)  ❷
             u2 = a * np.dot(l1.T, d2)  ❷
             w1 -= u2  ❷
             e1 = np.dot(d2, w1.T)  ❷
             d1 = e1 * sigmoid(l1, True)  ❷
             u1 = a * np.dot(l0.T, d1)  ❷
             w0 -= u1  ❷
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            mse = (e2 ** 2).mean()
            if s % 2000 == 0:
                print(f'step={s:5d} | mse={mse:.5f}')
        step= 2000 | mse=0.00933
        step= 4000 | mse=0.02399
        step= 6000 | mse=0.05134
        step= 8000 | mse=0.00064
        step=10000 | mse=0.00013
        step=12000 | mse=0.00009
        step=14000 | mse=0.00007
        step=16000 | mse=0.00007
        step=18000 | mse=0.00012
        step=20000 | mse=0.00015

In [93]: acc = l2.round() == y  ❸
         acc  ❸
Out[93]: array([[ True],
                [ True],
                [ True],
                [ True],
                [ True],
                [ True],
                [ True],
                [ True],
                [ True],
                [ True]])

In [94]: sum(acc) / len(acc)  ❸
Out[94]: array([1.])
```

❶ 前向传播。

❷ 后向传播。

❸ 分类准确率。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

附录 B

神经网络类

在附录 A 的基础上，本附录提供了简单且基于类的模仿了 `scikit-learn` 等软件包的 API 的神经网络实现。该实现基于纯粹且简单的 Python 代码，用于演示和说明。本附录中提供的类无法替代标准 Python 包（比如 `scikit-learn` 或 `TensorFlow` 与 `Keras` 的组合）中健壮、高效且可扩展的实现。

本附录由以下几节组成。

- B.1 节介绍了具有不同激活函数的 Python 函数。
- B.2 节介绍了用于**简单神经网络**的 Python 类。
- B.3 节介绍了用于**浅层神经网络**的 Python 类。
- B.4 节介绍了浅层神经网络类在金融数据中的应用。

本附录中的实现和示例简单明了。Python 类不太适合解决更大的估计或分类问题，这里仅仅是从头开始展示易于理解的 Python 实现。

B.1 激活函数

附录 A 隐式或显式地使用了两个激活函数：线性函数和 sigmoid 函数。Python 函数 `activation` 将 `relu`（整流线性单元）函数和 `softplus` 函数添加到了选项集中。对于所有这些激活函数，还定义了它们的一阶导数。

```
In [1]: import math
        import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        mpl.rcParams['font.family'] = 'serif'
        np.set_printoptions(suppress=True)

In [2]: def activation(x, act='linear', deriv=False):
            if act == 'sigmoid':
                if deriv:
                    out = activation(x, 'sigmoid', False)
                    return out * (1 - out)
                return 1 / (1 + np.exp(-x))
            elif act == 'relu':
                if deriv:
                    return np.where(x > 0, 1, 0)
                return np.maximum(x, 0)
            elif act == 'softplus':
                if deriv:
                    return activation(x, act='sigmoid')
                return np.log(1 + np.exp(x))
            elif act == 'linear':
                if deriv:
                    return 1
                return x
            else:
                raise ValueError('Activation function not known.')

In [3]: x = np.linspace(-1, 1, 20)

In [4]: activation(x, 'sigmoid')
Out[4]: array([0.26894142, 0.29013328, 0.31228169, 0.33532221, 0.35917484,
               0.38374461, 0.40892261, 0.43458759, 0.46060812, 0.48684514,
               0.51315486, 0.53939188, 0.56541241, 0.59107739, 0.61625539,
               0.64082516, 0.66467779, 0.68771831, 0.70986672, 0.73105858])

In [5]: activation(x, 'sigmoid', True)
Out[5]: array([0.19661193, 0.20595596, 0.21476184, 0.22288122, 0.23016827,
               0.23648468, 0.24170491, 0.24572122, 0.24844828, 0.24982695,
               0.24982695, 0.24844828, 0.24572122, 0.24170491, 0.23648468,
               0.23016827, 0.22288122, 0.21476184, 0.20595596, 0.19661193])
```

B.2 简单神经网络

本节介绍了一个用于**简单神经网络**的类，其 API 类似于标准 Python 包中用于机器学习或深度学习的模型（特别是 `scikit-learn` 和 `Keras`）。考虑以下 Python 代码中显示的类 `sinn`。它实现了一个简单的神经网络并定义了两个主要方法 `.fit()` 和 `.predict()`。`.metrics()` 方法会计算典型的性能指标：估计的 MSE 和分类的准确率。该类还为前向传播和后向传播步骤实现了两个方法。

```
In [6]: class sinn:
            def __init__(self, act='linear', lr=0.01, steps=100,
                             verbose=False, psteps=200):
                self.act = act
                self.lr = lr
                self.steps = steps
                self.verbose = verbose
                self.psteps = psteps
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        def forward(self):
            ''' 前向传播
            '''
            self.l2 = activation(np.dot(self.l0, self.w), self.act)
        def backward(self):
            ''' 后向传播
            '''
            self.e = self.l2 - self.y
            d = self.e * activation(self.l2, self.act, True)
            u = self.lr * np.dot(self.l0.T, d)
            self.w -= u
        def metrics(self, s):
            ''' 性能指标
            '''
            mse = (self.e ** 2).mean()
            acc = float(sum(self.l2.round() == self.y) / len(self.y))
            self.res = self.res.append(
                pd.DataFrame({'mse': mse, 'acc': acc}, index=[s,])
            )
            if s % self.psteps == 0 and self.verbose:
                    print(f'step={s:5d} | mse={mse:.6f}')
                    print(f'           | acc={acc:.6f}')
        def fit(self, l0, y, steps=None, seed=None):
            ''' 拟合步数
            '''
            self.l0 = l0
            self.y = y
            if steps is None:
                steps = self.steps
            self.res = pd.DataFrame()
            samples, features = l0.shape
            if seed is not None:
                np.random.seed(seed)
            self.w = np.random.random((features, 1))
            for s in range(1, steps + 1):
                self.forward()
                self.backward()
                self.metrics(s)
        def predict(self, X):
            ''' 预测步数
            '''
            return activation(np.dot(X, self.w), self.act)
```

B.2.1 估计

第一个问题是估计问题，可以使用回归技术进行解决。

```
In [7]: features = 5
        samples = 5

In [8]: np.random.seed(10)
        l0 = np.random.standard_normal((samples, features))
        l0
Out[8]: array([[ 1.3315865 ,  0.71527897, -1.54540029, -0.00838385,  0.62133597],
               [-0.72008556,  0.26551159,  0.10854853,  0.00429143, -0.17460021],
               [ 0.43302619,  1.20303737, -0.96506567,  1.02827408,  0.22863013],
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
              [ 0.44513761, -1.13660221,  0.13513688,  1.484537 ,  -1.07980489],
              [-1.97772828, -1.7433723 ,  0.26607016,  2.38496733,  1.12369125]])

In [9]: np.linalg.matrix_rank(l0)
Out[9]: 5

In [10]: y = np.random.random((samples, 1))
         y
Out[10]: array([[0.8052232 ],
                [0.52164715],
                [0.90864888],
                [0.31923609],
                [0.09045935]])

In [11]: reg = np.linalg.lstsq(l0, y, rcond=-1)[0]  ❶

In [12]: reg  ❶
Out[12]: array([[-0.74919308],
                [ 0.00146473],
                [-1.49864704],
                [-0.02498757],
                [-0.82793882]])

In [13]: np.allclose(np.dot(l0, reg), y)  ❶
Out[13]: True
```

❶ 回归精确解。

将 sinn 类应用于估计问题需要以重复训练步数的形式付出相当多的努力。然而，通过增加步数，可以使估计任意精确。

```
In [14]: model = sinn(lr=0.015, act='linear', steps=6000,
                      verbose=True, psteps=1000)

In [15]: %time model.fit(l0, y, seed=100)
         step= 1000 | mse=0.008086
                    | acc=0.000000
         step= 2000 | mse=0.000545
                    | acc=0.000000
         step= 3000 | mse=0.000037
                    | acc=0.000000
         step= 4000 | mse=0.000002
                    | acc=0.000000
         step= 5000 | mse=0.000000
                    | acc=0.000000
         step= 6000 | mse=0.000000
                    | acc=0.000000
         CPU times: user 5.23 s, sys: 29.7 ms, total: 5.26 s
         Wall time: 5.26 s

In [16]: model.predict(l0)
Out[16]: array([[0.80512489],
                [0.52144986],
                [0.90872498],
                [0.31919803],
                [0.09045743]])
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [17]: model.predict(l0) - y  ❶
Out[17]: array([[-0.0000983 ],
                [-0.00019729],
                [ 0.0000761 ],
                [-0.00003806],
                [-0.00000191]])
```

❶ 神经网络估计的残差。

B.2.2 分类

第二个问题是分类问题，也可以使用 `sinn` 类进行解决。标准回归技术在这种情况下通常没有用。对于特定的一组随机特征和标签，`sinn` 模型的准确率达到了 100%。同样，这需要以重复训练步数的形式付出相当多的努力。图 B-1 显示了预测准确率如何随着训练步数的数量而变化。

```
In [18]: features = 5
         samples = 10

In [19]: np.random.seed(3)
         l0 = np.random.randint(0, 2, (samples, features))
         l0
Out[19]: array([[0, 0, 1, 1, 0],
                [0, 0, 1, 1, 1],
                [0, 1, 1, 1, 0],
                [1, 1, 0, 0, 0],
                [0, 1, 1, 0, 0],
                [0, 1, 0, 0, 0],
                [0, 1, 0, 1, 1],
                [0, 1, 0, 0, 1],
                [1, 0, 0, 1, 0],
                [1, 0, 1, 1, 1]])

In [20]: np.linalg.matrix_rank(l0)
Out[20]: 5

In [21]: y = np.random.randint(0, 2, (samples, 1))
         y
Out[21]: array([[1],
                [0],
                [1],
                [0],
                [0],
                [1],
                [1],
                [1],
                [0],
                [0]])

In [22]: model = sinn(lr=0.01, act='sigmoid')  ❶

In [23]: %time model.fit(l0, y, 4000)
         CPU times: user 3.57 s, sys: 9.6 ms, total: 3.58 s
         Wall time: 3.59 s
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [24]: model.l2
Out[24]: array([[0.51118415],
               [0.34390898],
               [0.84733758],
               [0.07601979],
               [0.40505454],
               [0.84145926],
               [0.95592461],
               [0.72680243],
               [0.11219587],
               [0.00806003]])

In [25]: model.predict(l0).round() == y  ❷
Out[25]: array([[ True],
               [ True],
               [ True],
               [ True],
               [ True],
               [ True],
               [ True],
               [ True],
               [ True],
               [ True]])

In [26]: ax = model.res['acc'].plot(figsize=(10, 6),
                     title='Prediction Accuracy | Classification')
         ax.set(xlabel='steps', ylabel='accuracy');
```

❶ sigmoid 函数用于激活函数。

❷ 这个特定数据集的完美准确率。

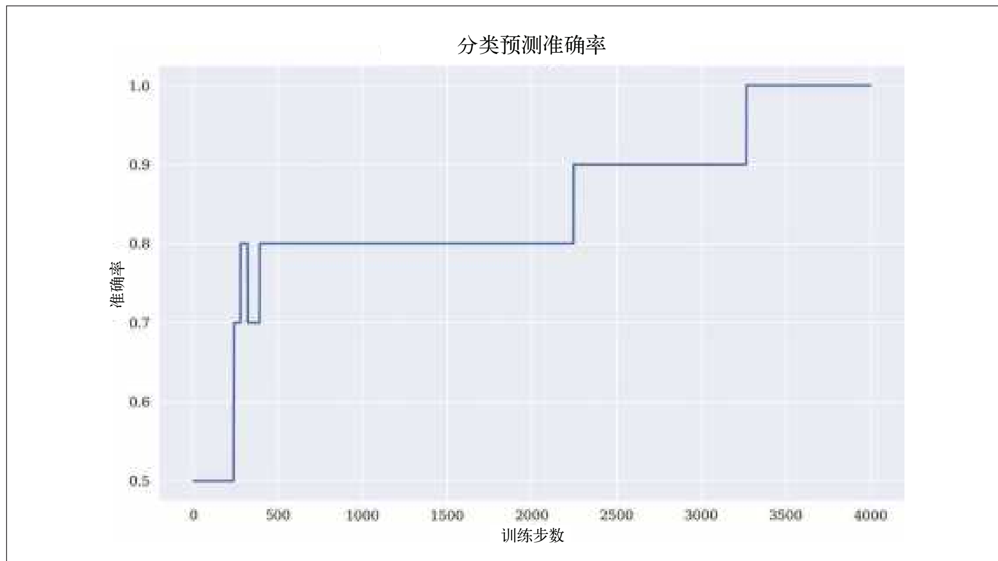


图 B-1：预测准确率与训练步数

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

B.3 浅层神经网络

本节将 shnn 类应用于估计和分类问题，该类实现了具有一个隐藏层的浅层神经网络，类的结构与上一节中的 sinn 类一致。

```
In [27]: class shnn:
             def __init__(self, units=12, act='linear', lr=0.01, steps=100,
                          verbose=False, psteps=200, seed=None):
                 self.units = units
                 self.act = act
                 self.lr = lr
                 self.steps = steps
                 self.verbose = verbose
                 self.psteps = psteps
                 self.seed = seed
             def initialize(self):
                 ''' 初始化随机权重
                 '''
                 if self.seed is not None:
                     np.random.seed(self.seed)
                 samples, features = self.l0.shape
                 self.w0 = np.random.random((features, self.units))
                 self.w1 = np.random.random((self.units, 1))
             def forward(self):
                 ''' 前向传播
                 '''
                 self.l1 = activation(np.dot(self.l0, self.w0), self.act)
                 self.l2 = activation(np.dot(self.l1, self.w1), self.act)
             def backward(self):
                 ''' 后向传播
                 '''
                 self.e = self.l2 - self.y
                 d2 = self.e * activation(self.l2, self.act, True)
                 u2 = self.lr * np.dot(self.l1.T, d2)
                 self.w1 -= u2
                 e1 = np.dot(d2, self.w1.T)
                 d1 = e1 * activation(self.l1, self.act, True)
                 u1 = self.lr * np.dot(self.l0.T, d1)
                 self.w0 -= u1
             def metrics(self, s):
                 ''' 性能指标
                 '''
                 mse = (self.e ** 2).mean()
                 acc = float(sum(self.l2.round() == self.y) / len(self.y))
                 self.res = self.res.append(
                     pd.DataFrame({'mse': mse, 'acc': acc}, index=[s,])
                 )
                 if s % self.psteps == 0 and self.verbose:
                         print(f'step={s:5d} | mse={mse:.5f}')
                         print(f'           | acc={acc:.5f}')
             def fit(self, l0, y, steps=None):
                 ''' 拟合步数
                 '''
                 self.l0 = l0
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
            self.y = y
            if steps is None:
                steps = self.steps
            self.res = pd.DataFrame()
            self.initialize()
            self.forward()
            for s in range(1, steps + 1):
                self.backward()
                self.forward()
                self.metrics(s)
        def predict(self, X):
            ''' 预测步数
            '''
            l1 = activation(np.dot(X, self.w0), self.act)
            l2 = activation(np.dot(l1, self.w1), self.act)
            return l2
```

B.3.1 估计

同样，首先考虑估计问题。对 5 个特征和 10 个样本来说，不可能存在完美回归解决方案，因此，回归的 MSE 值比较高。

```
In [28]: features = 5
         samples = 10

In [29]: l0 = np.random.standard_normal((samples, features))

In [30]: np.linalg.matrix_rank(l0)
Out[30]: 5

In [31]: y = np.random.random((samples, 1))

In [32]: reg = np.linalg.lstsq(l0, y, rcond=-1)[0]

In [33]: (np.dot(l0, reg)  - y)
Out[33]: array([[-0.10226341],
                [-0.42357164],
                [-0.25150491],
                [-0.30984143],
                [-0.85213261],
                [-0.13791373],
                [-0.52336502],
                [-0.50304204],
                [-0.7728686 ],
                [-0.3716898 ]])

In [34]: ((np.dot(l0, reg)  - y) ** 2).mean()
Out[34]: 0.23567187607888118
```

然而，基于 shnn 类的浅层神经网络估计相当不错，并且与回归值相比显示出了相对较低的 MSE 值。

```
In [35]: model = shnn(lr=0.01, units=16, act='softplus',
                      verbose=True, psteps=2000, seed=100)
```


专为量化交易设计的实时行情数据API: www.alltick.co

AllTick

实时行情数据接口

专为量化交易打造

全方位的市场行情数据接口

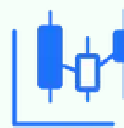
包含实时和历史行情

外汇API

来自世界领先银行机构的 100 多种货币对的逐笔更新。

商品API

所有贵金属（如黄金、白银）和所有能源类别的实时和历史商品数据 API。



股票API

适用于 170,000+ 美国和香港股票的实时和历史股票数据 API。

加密货币API

来自所有主要加密货币交易所的实时和历史加密货币数据，统一在一个易于使用的 API 中。

量化交易神器
回测必备！！

可靠的行情源

我们的系统具有 99.95% 的 SLA。AllTick 倾向于将质量和正常运行时间的标准提升到下一个级别。

低延时接口实时推送

通过 WebSocket 进行的实时数据流具有超低延迟，平均仅约 170 毫秒。

逐笔更新的高频数据

每个交易的实时推送，每个都可追踪，并且与交易所的实时交易行情完全同步。

马上登录ALLTICK.CO

免费试用!

全美股财报免费送!

官方网站：https://www.alltick.co | Telegram：https://t.me/alltick001

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [36]: %time model.fit(l0, y, 8000)
         step= 2000 | mse=0.00205
                    | acc=0.00000
         step= 4000 | mse=0.00098
                    | acc=0.00000
         step= 6000 | mse=0.00043
                    | acc=0.00000
         step= 8000 | mse=0.00022
                    | acc=0.00000
         CPU times: user 8.15 s, sys: 69.2 ms, total: 8.22 s
         Wall time: 8.3 s

In [37]: model.l2 - y
Out[37]: array([[-0.00390976],
                [-0.00522077],
                [ 0.02053932],
                [-0.0042113 ],
                [-0.0006624 ],
                [-0.01001395],
                [ 0.01783203],
                [-0.01498316],
                [-0.0177866 ],
                [ 0.02782519]])
```

B.3.2 分类

分类示例采用估计数并对它们进行四舍五入。浅层神经网络会快速收敛，以 100% 的准确率预测标签（参见图 B-2）。

```
In [38]: model = shnn(lr=0.025, act='sigmoid', steps=200,
                      verbose=True, psteps=50, seed=100)

In [39]: l0.round()
Out[39]: array([[ 0., -1., -2.,  1., -0.],
                [-1., -2., -0., -0., -2.],
                [ 0.,  1., -1., -1., -1.],
                [-0.,  0., -1., -0., -1.],
                [ 1., -1.,  1.,  1., -1.],
                [ 1., -1.,  1., -2.,  1.],
                [-1., -0.,  1., -1.,  1.],
                [ 1.,  2., -1., -0., -0.],
                [-1.,  0.,  0.,  0.,  2.],
                [ 0.,  0., -0.,  1.,  1.]])

In [40]: np.linalg.matrix_rank(l0)
Out[40]: 5

In [41]: y.round()
Out[41]: array([[0.],
                [1.],
                [1.],
                [1.],
                [1.],
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
              [1.],
              [0.],
              [1.],
              [0.],
              [0.]])

In [42]: model.fit(l0.round(), y.round())
         step=   50 | mse=0.26774
                    | acc=0.60000
         step=  100 | mse=0.22556
                    | acc=0.60000
         step=  150 | mse=0.19939
                    | acc=0.70000
         step=  200 | mse=0.16924
                    | acc=1.00000

In [43]: ax = model.res.plot(figsize=(10, 6), secondary_y='mse')
         ax.get_legend().set_bbox_to_anchor((0.2, 0.5));
```

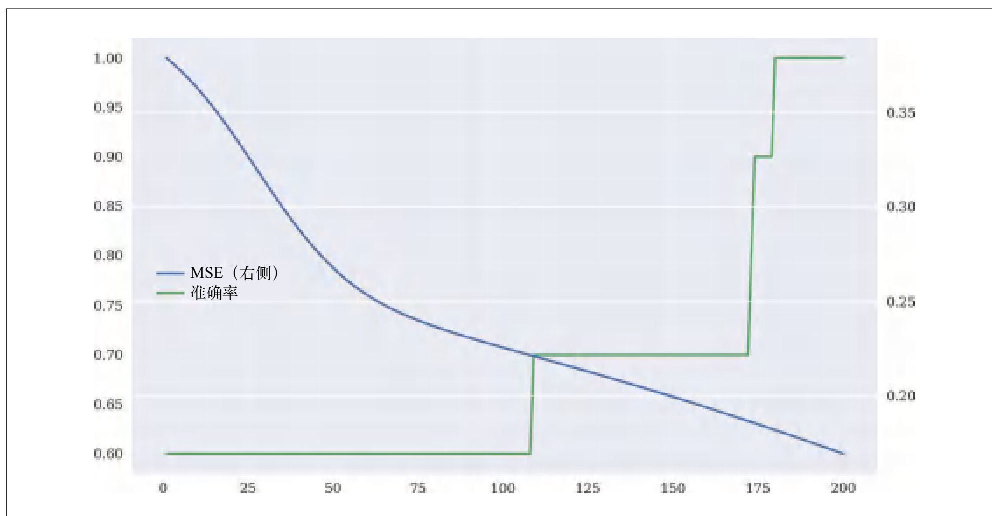


图 B-2：浅层神经网络的性能指标（分类示例）

B.4 预测市场方向

本节应用 shnn 类来预测欧元 / 美元汇率的未来方向。该分析是样本内的，只是为了演示将 shnn 应用于现实世界数据。有关此类基于预测策略的向量化回测的更现实设置的实现，请参见第 10 章。

以下 Python 代码会导入财务数据（相当于 10 年的日终数据），并创建用作特征的滞后的、标准化的对数收益率，标签数据是作为二元化的价格走向序列。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [44]: url = 'http://hilpisch.com/aiif_eikon_eod_data.csv'

In [45]: raw = pd.read_csv(url, index_col=0, parse_dates=True).dropna()

In [46]: sym = 'EUR='

In [47]: data = pd.DataFrame(raw[sym])

In [48]: lags = 5
         cols = []
         data['r'] = np.log(data / data.shift(1))
         data['d'] = np.where(data['r'] > 0, 1, 0)  ❶
         for lag in range(1, lags + 1):
             col = f'lag_{lag}'
             data[col] = data['r'].shift(lag)  ❷
             cols.append(col)
         data.dropna(inplace=True)
         data[cols] = (data[cols] - data[cols].mean()) / data[cols].std()  ❸

In [49]: data.head()
Out[49]:               EUR=         r  d     lag_1     lag_2     lag_3     lag_4  \
         Date
         2010-01-12  1.4494 -0.001310  0  1.256582  1.177935 -1.142025  0.560551
         2010-01-13  1.4510  0.001103  1 -0.214533  1.255944  1.178974 -1.142118
         2010-01-14  1.4502 -0.000551  0  0.213539 -0.214803  1.256989  1.178748
         2010-01-15  1.4382 -0.008309  0 -0.079986  0.213163 -0.213853  1.256758
         2010-01-19  1.4298 -0.005858  0 -1.456028 -0.080289  0.214140 -0.214000

                        lag_5
         Date
         2010-01-12 -0.511372
         2010-01-13  0.560740
         2010-01-14 -1.141841
         2010-01-15  1.178904
         2010-01-19  1.256910
```

❶ 市场方向作为标签数据。

❷ 返回滞后的对数收益率作为特征数据。

❸ 特征数据的高斯归一化。

完成数据预处理后，浅层神经网络类 shnn 在监督分类中的应用很简单。图 B-3 表明，在样本内，基于预测的策略明显优于被动基准投资。

```
In [50]: model = shnn(lr=0.0001, act='sigmoid', steps=10000,
                      verbose=True, psteps=2000, seed=100)

In [51]: y = data['d'].values.reshape(-1, 1)

In [52]: %time model.fit(data[cols].values, y)
         step= 2000 | mse=0.24964
                    | acc=0.51594
         step= 4000 | mse=0.24951
                    | acc=0.52390
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
        step= 6000 | mse=0.24945
                   | acc=0.52231
        step= 8000 | mse=0.24940
                   | acc=0.52510
        step=10000 | mse=0.24936
                   | acc=0.52430
        CPU times: user 9min 1s, sys: 40.9 s, total: 9min 42s
        Wall time: 1min 21s

In [53]: data['p'] = np.where(model.predict(data[cols]) > 0.5, 1, -1)  ❶

In [54]: data['p'].value_counts()  ❶
Out[54]:  1    1257
         -1    1253
         Name: p, dtype: int64

In [55]: data['s'] = data['p'] * data['r']  ❷

In [56]: data[['r', 's']].sum().apply(np.exp)  ❸
Out[56]: r    0.772411
         s    1.885677
         dtype: float64

In [57]: data[['r', 's']].cumsum().apply(np.exp).plot(figsize=(10, 6));  ❹
```

❶ 基于预测值的头寸方向和数量。

❷ 根据头寸价值和对数收益率计算策略收益率。

❸ 计算策略投资和基准投资的总收益。

❹ 显示随时间变化的策略投资和基准投资的总体表现。

图 B-3：与被动基准投资相比，基于预测的策略投资的总体表现（样本内）

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

附录C

卷积神经网络

本书第三部分重点介绍了两种标准的神经网络类型：密集神经网络（DNN）和循环神经网络 (RNN)。DNN 的魅力在于它是很好的通用逼近器，例如，书中关于强化学习的示例会使用 DNN 来逼近最优动作策略。RNN 则专门用于处理序列数据（如时间序列数据），这在尝试预测金融时间序列的未来值时很有帮助。

然而，**卷积神经网络**（CNN）是另一种在实践中被广泛使用的标准神经网络类型，尤其用于计算机视觉领域。CNN 能够在许多标准测试和挑战（如 ImageNet 挑战）中设立新的基准，与此相关的更多信息，请参阅《经济学人》（2016）或 Gerrish（2018）。计算机视觉在自动驾驶汽车、安全和监视等领域也很重要。

本附录简要说明了 CNN 在金融时间序列数据预测中的应用。有关 CNN 的详细信息，请参阅 Chollet（2017）的第 5 章和 Goodfellow 等（2016）的第 9 章。

C.1 特征和标签数据

以下 Python 代码首先处理了所需的引用和自定义设置，然后导入了包含多种金融工具日终数据的数据集，该数据集在本书中被用在了不同的示例中。

```
In [1]: import os
        import math
        import numpy as np
        import pandas as pd
        from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        os.environ['PYTHONHASHSEED'] = '0'
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [2]: url = 'http://hilpisch.com/aiif_eikon_eod_data.csv'  ❶

In [3]: symbol = 'EUR='  ❶

In [4]: data = pd.DataFrame(pd.read_csv(url, index_col=0,
                                        parse_dates=True).dropna()[symbol])  ❶

In [5]: data.info()  ❶
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 2516 entries, 2010-01-04 to 2019-12-31
        Data columns (total 1 columns):
         #   Column  Non-Null Count  Dtype
        ---  ------  --------------  -----
         0   EUR=    2516 non-null   float64
        dtypes: float64(1)
        memory usage: 39.3 KB
```

❶ 提取并选择金融时间序列数据。

下一步是生成特征数据和滞后项，然后将其拆分为训练数据集和测试数据集，最后再根据训练数据集的统计数据对其进行归一化处理。

```
In [6]: lags = 5

In [7]: features = [symbol, 'r', 'd', 'sma', 'min', 'max', 'mom', 'vol']

In [8]: def add_lags(data, symbol, lags, window=20, features=features):
            cols = []
            df = data.copy()
            df.dropna(inplace=True)
            df['r'] = np.log(df / df.shift(1))
            df['sma'] = df[symbol].rolling(window).mean()  ❶
            df['min'] = df[symbol].rolling(window).min()  ❷
            df['max'] = df[symbol].rolling(window).max()  ❸
            df['mom'] = df['r'].rolling(window).mean()  ❹
            df['vol'] = df['r'].rolling(window).std()  ❺
            df.dropna(inplace=True)
            df['d'] = np.where(df['r'] > 0, 1, 0)
            for f in features:
                for lag in range(1, lags + 1):
                    col = f'{f}_lag_{lag}'
                    df[col] = df[f].shift(lag)
                    cols.append(col)
            df.dropna(inplace=True)
            return df, cols

In [9]: data, cols = add_lags(data, symbol, lags, window=20, features=features)

In [10]: split = int(len(data) * 0.8)

In [11]: train = data.iloc[:split].copy()  ❻

In [12]: mu, std = train[cols].mean(), train[cols].std()  ❻

In [13]: train[cols] = (train[cols] - mu) / std  ❻
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
In [14]: test = data.iloc[split:].copy()  ❼

In [15]: test[cols] = (test[cols] - mu) / std  ❼
```

❶ 简单移动平均线特征。

❷ 滚动最小值特征。

❸ 滚动最大值特征。

❹ 时间序列动量特征。

❺ 滚动波动率特性。

❻ 训练数据集的高斯归一化。

❼ 测试数据集的高斯归一化。

C.2 训练模型

CNN 的实现类似于 DNN。下面的 Python 代码会从 Keras 导入所需随机数生成器函数，并设置相关种子值。

```
In [16]: import random
         import tensorflow as tf
         from keras.models import Sequential
         from keras.layers import Dense, Conv1D, Flatten
         Using TensorFlow backend.

In [17]: def set_seeds(seed=100):
             random.seed(seed)
             np.random.seed(seed)
             tf.random.set_seed(seed)
```

以下 Python 代码实现并训练了一个简单的 CNN 模型，该模型的核心是一个适用于时间序列数据的**一维卷积层**（详情请参见 Keras 卷积层文档）。

```
In [18]: set_seeds()
         model = Sequential()
         model.add(Conv1D(filters=96, kernel_size=5, activation='relu',
                          input_shape=(len(cols), 1)))
         model.add(Flatten())
         model.add(Dense(10, activation='relu'))
         model.add(Dense(1, activation='sigmoid'))

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

In [19]: model.summary()
         Model: "sequential_1"
         _________________________________________________________________
         Layer (type)                 Output Shape              Param #
         =================================================================
```


专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

```
conv1d_1 (Conv1D)            (None, 36, 96)           576
_________________________________________________________________
flatten_1 (Flatten)          (None, 3456)             0
_________________________________________________________________
dense_1 (Dense)              (None, 10)               34570
_________________________________________________________________
dense_2 (Dense)              (None, 1)                11
=================================================================
Total params: 35,157
Trainable params: 35,157
Non-trainable params: 0
_________________________________________________________________

In [20]: %%time
         model.fit(np.atleast_3d(train[cols]), train['d'],
                   epochs=60, batch_size=48, verbose=False,
                   validation_split=0.15, shuffle=False)
         CPU times: user 10.1 s, sys: 1.87 s, total: 12 s
         Wall time: 4.78 s

Out[20]: <keras.callbacks.callbacks.History at 0x7ffe3f32b110>
```

图 C-1 展示了训练数据集和验证数据集在不同训练轮数的性能指标。

```
In [21]: res = pd.DataFrame(model.history.history)

In [22]: res.tail(3)
Out[22]:     val_loss val_accuracy      loss  accuracy
         57  0.699932     0.508361  0.635633  0.597165
         58  0.719671     0.501672  0.634539  0.598937
         59  0.729954     0.505017  0.634403  0.601890

In [23]: res.plot(figsize=(10, 6));
```

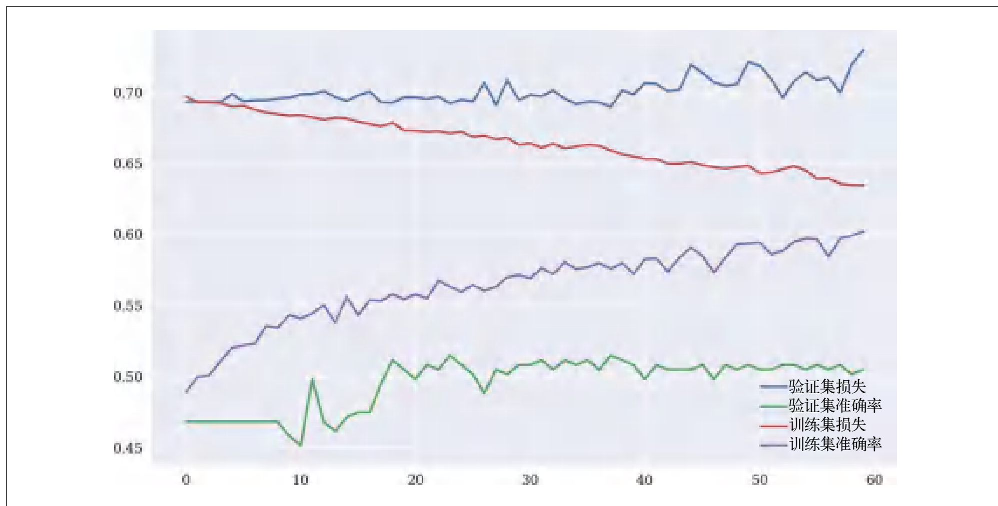


图 C-1：CNN 训练数据集和验证数据集在不同训练轮数的性能指标

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

C.3 测试模型

最后，接下来的 Python 代码会将训练好的模型应用于测试数据集。CNN 模型明显优于被动基准投资。然而，如果考虑以典型（零售）买卖价差形式出现的交易成本，大部分出色表现会被交易成本吞噬。图 C-2 显示了随时间变化的业绩。

```
In [24]: model.evaluate(np.atleast_3d(test[cols]), test['d'])  ❶
         499/499 [==============================] - 0s 25us/step

Out[24]: [0.7364848222665653, 0.5210421085357666]

In [25]: test['p'] = np.where(model.predict(np.atleast_3d(test[cols])) > 0.5, 1, 0)

In [26]: test['p'] = np.where(test['p'] > 0, 1, -1)  ❷

In [27]: test['p'].value_counts()  ❷
Out[27]: -1    478
          1     21
          Name: p, dtype: int64

In [28]: (test['p'].diff() != 0).sum()  ❸
Out[28]: 41

In [29]: test['s'] = test['p'] * test['r']  ❹

In [30]: ptc = 0.00012 / test[symbol]  ❺

In [31]: test['s_'] = np.where(test['p'] != 0, test['s'] - ptc, test['s'])  ❻

In [32]: test[['r', 's', 's_']].sum().apply(np.exp)
Out[32]: r     0.931992
         s     1.086525
         s_    1.031307
         dtype: float64

In [33]: test[['r', 's', 's_']].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ 样本外的准确率。

❷ 基于预测的头寸（多头 / 空头）。

❸ 头寸产生的交易数量。

❹ 给定买卖差价的成比例交易成本。

❺ 考虑交易成本**之前**的策略投资收益。

❻ 考虑交易成本**之后**的策略投资收益。

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

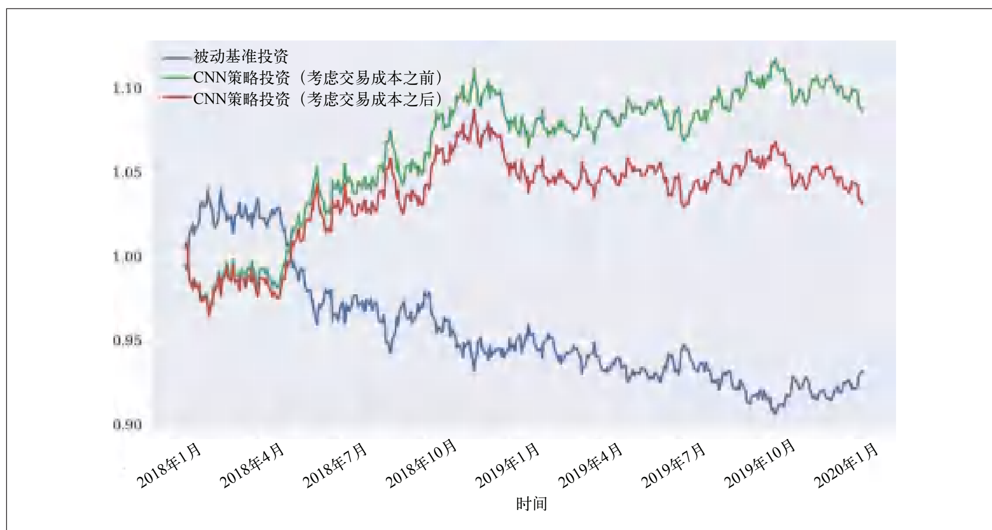


图 C-2：考虑交易成本之前 / 之后的被动基准投资和 CNN 策略投资的总体表现

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

参考文献

[1] Chollet, François. *Deep Learning with Python*. Shelter Island: Manning, 2017.

[2] Hilpisch, Yves. *Python for Finance: Mastering Data-Driven Finance*. 2nd ed. Sebastopol: O'Reilly, 2018.

[3] Hilpisch, Yves. *Python for Algorithmic Trading: From Idea to Cloud Deployment*. Sebastopol: O'Reilly, 2020.

[4] Lee, Justina, Melissa Karsh.“Machine-Learning Hedge Fund Voleon Group Returns 7% in 2019.”2020. *Bloomberg*, January 21, 2020.

[5] López de Prado, Marcos. *Advances in Financial Machine Learning*. Hoboken, NJ: John Wiley & Sons, 2018.

[6] Mnih, Volodymyr, et al.“Playing Atari with Deep Reinforcement Learning.”arXiv. December 19, 2013.

[7] Shiller, Robert.“The Mirage of the Financial Singularity.”Yale Insights. July 16, 2015.

[8] Silver, David, et al.“Mastering the Game of Go with Deep Neural Networks and Tree Search.”2016, *Nature* 529 (January): 484-489.

[9] Alpaydin, Ethem. *Machine Learning*. MIT Press, Cambridge, 2016.

[10] Chollet, Francois. *Deep Learning with Python*. Shelter Island: Manning, 2017.

[11] Goodfellow, Ian, Yoshua Bengio, Aaron Courville. *Deep Learning*. Cambridge: MIT Press, 2016.

[12] Kratsios, Anastasis.“Universal Approximation Theorems.”2019.

[13] Silver, David, et al.“Mastering the Game of Go with Deep Neural Networks and Tree Search.”2016, *Nature* 529 (January): 484-489.

[14] Shanahan, Murray. *The Technological Singularity*. Cambridge: MIT Press, 2015.

[15] Tegmark, Max. *Life 3.0: Being Human in the Age of Artificial Intelligence*. United Kingdom: Penguin Random House, 2017.

[16] VanderPlas, Jake. *Python Data Science Handbook*. Sebastopol: O'Reilly, 2017.

[17] Barrat, James. *Our Final Invention: Artificial Intelligence and The End of the Human Era*. New York: St. Martin's Press, 2013.

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

[18] Bostrom, Nick. *Superintelligence: Paths, Dangers, Strategies*. Oxford: Oxford University Press, 2014.

[19] Chollet, François. *Deep Learning with Python*. Shelter Island: Manning, 2017.

[20] Domingos, Pedro. *The Master Algorithm: How the Quest for the Ultimate Learning Machine will Remake our World*. United Kingdom: Penguin Random House, 2015.

[21] Doudna, Jennifer, Samuel H. Sternberg. *A Crack in Creation: The New Power to Control Evolution*. London: The Bodley Head, 2017.

[22] Gerrish, Sean. *How Smart Machines Think*. Cambridge: MIT Press, 2018.

[23] Harari, Yuval Noah. *Homo Deus: A Brief History of Tomorrow*. London: Harvill Secker, 2015.

[24] Kasparov, Garry. *Deep Thinking: Where Machine Intelligence Ends*. London: John Murray, 2017.

[25] Kurzweil, Ray. *The Singularity Is Near: When Humans Transcend Biology*. New York: Penguin Group, 2005.

[26] Kurzweil, Ray. *How to Create a Mind: The Secret of Human Thought Revealed*. New York: Penguin Group, 2012.

[27] Mnih, Volodymyr, et al. “Playing Atari with Deep Reinforcement Learning.” 2013, arXiv. December 19, 2013.

[28] Murgia, Madhumita, Siddarth Shrikanth. “How Governments Are Beginning to Regulate AI.” 2019, *Financial Times*, May 30, 2019.

[29] Silver, David, et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search.” 2016, *Nature* 529 (January): 484-489.

[30] Silver, David, et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.” 2017a, arXiv. December 5, 2017.

[31] Silver, David, et al. “Mastering the Game of Go without Human Knowledge.” 2017b, *Nature*, 550 (October): 354–359.

[32] Shanahan, Murray. *The Technological Singularity*. Cambridge: MIT Press, 2015.

[33] Tegmark, Max. *Life 3.0: Being Human in the Age of Artificial Intelligence.* United Kingdom: Penguin Random House, 2017.

[34] Vinge, Vernor. “Vernor Vinge on the Singularity.” 1993.

[35] Bender, Jennifer, et al. “Foundations of Factor Investing.” MSCI Research Insight, 2013.

[36] Calvello, Angelo. “Fund Managers Must Embrace AI Disruption.” 2020, Financial Times, January 15, 2020.

[37] Eichberger, Jürgen, Ian R. Harper. *Financial Economics*. New York: Oxford University Press, 1997.

[38] Fishburn, Peter. “Utility Theory.” 1968, *Management Science* 14 (5): 335-378.

[39] Fama, Eugene F., Kenneth R. French. “The Capital Asset Pricing Model: Theory and Evidence.” 2004, *Journal of Economic Perspectives* 18 (3): 25-46.

[40] Halevy, Alon, Peter Norvig, Fernando Pereira. “The Unreasonable Effectiveness of Data.” 2009, *IEEE Intelligent Systems*, Expert Opinion.

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

[41] Hilpisch, Yves. *Derivatives Analytics with Python: Data Analysis, Models, Simulation, Calibration, and Hedging*. Wiley Finance, 2015.

[42] Jacod, Jean, Philip Protter. *Probability Essentials*. 2nd ed. Berlin: Springer, 2004.

[43] Johnstone, David, Dennis Lindley. "Mean-Variance and Expected Utility: The Borch Paradox." 2013, *Statistical Science* 28 (2): 223-237.

[44] Jones, Charles P. *Investments: Analysis and Management*. 12th ed. Hoboken: John Wiley & Sons, 2012.

[45] Karni, Edi. "Axiomatic Foundations of Expected Utility and Subjective Probability." 2014, In *Handbook of the Economics of Risk and Uncertainty*, edited by Mark J. Machina and W. Kip Viscusi, 1-39. Oxford: North Holland.

[46] Lintner, John. "The Valuation of Risk Assets and the Selection of Risky Investments in Stock Portfolios and Capital Budgets." 1965, *Review of Economics and Statistics*47 (1): 13-37.

[47] Markowitz, Harry. "Portfolio Selection." 1952, *Journal of Finance* 7 (1): 77-91.

[48] Pratt, John W. "Risk Aversion in the Small and in the Large." 1964, *Econometrica*32 (1/2): 122-136.

[49] Ross, Stephen A. "Portfolio and Capital Market Theory with Arbitrary Preferences and Distributions: The General Validity of the Mean-Variance Approach in Large Markets." 1971, Working Paper No. 12-72, Rodney L. White Center for Financial Research.

[50] Ross, Stephen A. "The Arbitrage Theory of Capital Asset Pricing." 1976, *Journal of Economic Theory* 13: 341-360.

[51] Rubinstein, Mark. *A History of the Theory of Investments—My Annotated Bibliography*. Hoboken: Wiley Finance, 2006.

[52] Sharpe, William F. "Capital Asset Prices: A Theory of Market Equilibrium under Conditions of Risk." 1964, Journal of Finance 19 (3): 425-442.

[53] Sharpe, William F. "Mutual Fund Performance." 1966, *Journal of Business* 39 (1): 119-138.

[54] Varian, Hal R. *Intermediate Microeconomics: A Modern Approach*. 8th ed. New York & London: W.W. Norton & Company, 2010.

[55] von Neumann, John, Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton: Princeton University Press, 1944.

[56] Allais, M. "Le Comportement de l'Homme Rationnel devant le Risque: Critique des Postulats et Axiomes de l'Ecole Americaine." 1953, *Econometrica* 21 (4): 503-546.

[57] Alexander, Carol. *Quantitative Methods in Finance*. Market Risk Analysis I, West Sussex: John Wiley & Sons, 2008a.

[58] Alexander, Carol. *Practical Financial Econometrics*. Market Risk Analysis II, West Sussex: John Wiley & Sons, 2008b.

[59] Bender, Jennifer, et al. "Foundations of Factor Investing." 2013, *MSCI Research Insight*.

[60] Campbell, John Y. *Financial Decisions and Markets: A Course in Asset Pricing*. Princeton and Oxford: Princeton University Press, 2018.

[61] Ellsberg, Daniel. "Risk, Ambiguity, and the Savage Axioms." 1961, *Quarterly Journal of Economics* 75 (4): 643-669.

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

[62] Fontaine, Philippe, Robert Leonard. *The Experiment in the History of Economics*. London and New York: Routledge, 2005.

[63] Kopf, Dan. "The Discovery of Statistical Regression." 2015, *Priceonomics*, November 6, 2015.

[64] Lee, Kai-Fu. *AI Superpowers: China, Silicon Valley, and the New World Order*. Boston and New York: Houghton Mifflin Harcourt, 2018.

[65] Sapolsky, Robert M. *Behave: The Biology of Humans at Our Best and Worst*. New York: Penguin Books, 2018.

[66] Savage, Leonard J. *The Foundations of Statistics*. 2nd ed. New York: Dover Publications, (1954) 1972.

[67] Wigglesworth, Robin. "How Investment Analysts Became Data Miners." 2019, *Financial Times*, November 28, 2019.

[68] Chollet, François. *Deep Learning with Python*. Shelter Island: Manning, 2017.

[69] Domingos, Pedro. *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. New York: Basic Books, 2015.

[70] Goodfellow, Ian, Yoshua Bengio, Aaron Courville. *Deep Learning*. Cambridge: MIT Press, 2016.

[71] Harari, Yuval Noah. *Homo Deus: A Brief History of Tomorrow*. London: Harvill Secker, 2015.

[72] Mitchell, Tom M. *Machine Learning*. New York: McGraw-Hill, 1997.

[73] VanderPlas, Jake. *Python Data Science Handbook*. Sebastopol: O'Reilly, 2017.

[74] Agrawal, Ajay, Joshua Gans, Avi Goldfarb. *Prediction Machines: The Simple Economics of Arti cial Intelligence*. Boston: Harvard Business Review Press, 2018.

[75] Copeland, Thomas, Fred Weston, Kuldeep Shastri. *Financial Theory and Corporate Policy*. 4th ed. Boston: Pearson, 2005.

[76] Fama, Eugene. "Random Walks in Stock Market Prices." 1965, *Financial Analysts Journal* (September/October): 55-59.

[77] Halevy, Alon, Peter Norvig, Fernando Pereira. "The Unreasonable Effectiveness of Data." 2009, *IEEE Intelligent Systems*, Expert Opinion.

[78] Hilpisch, Yves. *Python for Finance: Mastering Data-Driven Finance*. 2nd ed. Sebastopol: O'Reilly, 2018.

[79] Jensen, Michael. "Some Anomalous Evidence Regarding Market Efficiency." 1978, *Journal of Financial Economics* 6 (2/3): 95-101.

[80] Tegmark, Max. *Life 3.0: Being Human in the Age of Artificial Intelligence*. United Kingdom: Penguin Random House, 2017.

[81] Tsay, Ruey S. *Analysis of Financial Time Series*. Hoboken: Wiley, 2005.

[82] Chollet, Francois. *Deep Learning with Python*. Shelter Island: Manning, 2017.

[83] Goodfellow, Ian, Yoshua Bengio, Aaron Courville. *Deep Learning*. Cambridge: MIT Press, 2016.

[84] Chollet, François. *Deep Learning with Python*. Shelter Island: Manning, 2017.

[85] Goodfellow, Ian, Yoshua Bengio, Aaron Courville. *Deep Learning*. Cambridge: MIT Press, 2016.

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

[86] Sutton, Richard S., Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge and London: MIT Press, 2018.

[87] Watkins, Christopher. *Learning from Delayed Rewards*. Ph.D. thesis, University of Cambridge, 1989.

[88] Watkins, Christopher, Peter Dayan. "Q-Learning." 1992, *Machine Learning* 8 (May): 279-282.

[89] Gibbs Samuel. "Elon Musk: Tesla Cars Will Be Able to Cross Us with No Driver in Two Years." 2016, *The Guardian*. January 11, 2016.

[90] Hilpisch, Yves. *Python for Finance: Mastering Data-Driven Finance*. 2nd ed. Sebastopol: O'Reilly, 2018.

[91] Hilpisch, Yves. *Python for Algorithmic Trading: From Idea to Cloud Deployment*. Sebastopol: O'Reilly, 2020.

[92] Agrawal, Ajay, Joshua Gans, Avi Goldfarb. *Prediction Machines: The Simple Economics of Arti cial Intelligence*. Boston: Harvard Business Review Press, 2018.

[93] Hilpisch, Yves. *Python for Algorithmic Trading: From Idea to Cloud Deployment*. Sebastopol: O'Reilly, 2020.

[94] Khonji, Majid, Jorge Dias, Lakmal Seneviratne. "Risk-Aware Reasoning for Autonomous Vehicles." 2019, arXiv. October 6, 2019.

[95] Hilpisch, Yves. *Python for Algorithmic Trading: From Idea to Cloud Deployment*. Sebastopol: O'Reilly, 2020.

[96] Litman, Todd. "Autonomous Vehicle Implementation Predictions." 2020, *Victoria Transport Policy Institute*.

[97] Babaev, Dmitrii, et al. "E.T.-RNN: Applying Deep Learning to Credit Loan Applications." 2019.

[98] Black, Fischer, Myron Scholes. "The Pricing of Options and Corporate Liabilities." 1973, *Journal of Political Economy* 81 (3): 638–659.

[99] Bradshaw, Tim. "Google chief Sundar Pichai warns against rushing into AI regulation." 2019, *Financial Times*, September 20, 2019.

[100] Bostrom, Nick. *Superintelligence: Paths, Dangers, Strategies*. Oxford: Oxford University Press, 2014.

[101] Buehler, Hans, et al. "Deep Hedging: Hedging Derivatives Under Generic Market Frictions Using Reinforcement Learning." 2019, Finance Institute Research Paper No. 19-80.

[102] Copeland, Thomas, Fred Weston, Kuldeep Shastri. *Financial Theory and Corporate Policy*. 4th ed. Boston: Pearson, 2005.

[103] Cox, John, Stephen Ross, Mark Rubinstein. "Option Pricing: A Simplified Approach." 1979, *Journal of Financial Economics* 7, (3): 229–263.

[104] Fortado, Lindsay. "Data specialist Enigma reels in investment group cash." 2018, *Financial Times*, September 18, 2018.

[105] Golbayani, Parisa, Dan Wang, Ionut Florescu. "Application of Deep Neural Networks to Assess Corporate Credit Rating." 2020.

[106] Huber, Nick. "AI 'Only Scratching the Surface' of Potential in Financial Services." 2020, *Financial Times*, July 1, 2020.

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

[107] Jia, Zhe, et al. "Dissecting the Graphcore IPU Architecture via Microbenchmarking." 2019.

[108] Jones, Charles P. *Investments: Analysis and Management*. 12th ed. Hoboken: John Wiley & Sons, 2012.

[109] Klein, Aaron. "Reducing Bias in AI-based Financial Services." 2020, The Brookings Institution Report, July 10, 2020

[110] López de Prado, Marcos. *Advances in Financial Machine Learning*. Hoboken: Wiley Finance, 2018.

[111] López de Prado, Marcos. *Machine Learning for Asset Managers*. Cambridge: Cambridge University Press, 2020.

[112] Matyus, Allison. "Elon Musk Warns that All A.I. Must Be Regulated, Even at Tesla." 2020, *Digital Trends*, February 18, 2020.

[113] Merton, Robert C. "Theory of Rational Option Pricing." 1973, *Bell Journal of Economics and Management Science* 4 (Spring): 141–183.

[114] Murray, Seb. "Graduates with Tech and Finance Skills in High Demand." 2019, *Financial Times*, June 17, 2019.

[115] Ning, Brian, Franco Ho Ting Lin, Sebastian Jaimungal. "Double Deep Q-Learning for Optimal Execution." 2020.

[116] Noonan, Laura. "JPMorgan's requirement for new staff: coding lessons." 2018, *Financial Times*, October 8, 2018.

[117] Yousefi, Niloofar, Marie Alaghband, Ivan Garibay. "A Comprehensive Survey on Machine Learning Techniques and User Authentication Approaches for Credit Card Fraud Detection." 2019.

[118] Yu, Shi, Yuxin Chen, Hussain Zaidi. "AVA: A Financial Service Chatbot based on Deep Bidirectional Transformers." 2020.

[119] Bostrom, Nick. *Superintelligence: Paths, Dangers, Strategies*. Oxford: Oxford University Press, 2014.

[120] Huber, Nick. "AI 'Only Scratching the Surface' of Potential in Financial Services." 2020, *Financial Times*, July 1, 2020.

[121] Shiller, Robert. "The Mirage of the Financial Singularity." 2015, *Yale Insights* (blog).

[122] Chollet, Francois. *Deep Learning with Python*. Shelter Island: Manning, 2017.

[123] Chollet, François. *Deep Learning with Python*. Shelter Island: Manning, 2017.

[124] Economist, The. "From Not Working to Neural Networking." 2016, *The Economist* Special Report, June 23, 2016.

[125] Gerrish, Sean. *How Smart Machines Think*. Cambridge: MIT Press, 2018.

[126] Goodfellow, Ian, Yoshua Bengio, Aaron Courville. *Deep Learning*. Cambridge: MIT Press, 2016.

专为量化交易设计的实时行情数据API: www.alltick.co

专为量化交易设计的实时行情数据API: www.alltick.co

关于作者

伊夫·希尔皮斯科（Yves Hilpisch），金融数学博士，Python Quants 公司创始人兼 CEO，致力于普及人工智能、算法交易等相关技术在金融中的应用。此外，他还创建了 AI Machine 平台，提供人工智能算法交易策略的标准化部署。

除了本书，他还著有以下图书。

- *Python for Algorithmic Trading*
- 《Python 金融大数据分析》[1]
- 《金融衍生品大数据分析》
- *Listed Volatility and Variance Derivatives*

伊夫是国际量化投资分析师认证（CQF）讲师，讲授计算金融学、机器学习和算法交易。他还是第一批获得 Python 算法交易和 Python 计算金融学大学证书的在线培训项目的负责人。

伊夫编写了金融分析库 DX Analytics，并在伦敦、法兰克福、柏林、巴黎和纽约组织过多场关于 Python 的量化金融和算法交易的聚会、会议和训练营，并多次在美国、欧洲和亚洲的技术会议上发表主题演讲。

关于封面

本书封面上的动物是堤岸田鼠（学名：Myodes Glareolus），这些田鼠遍布欧洲和中亚的森林、河岸和沼泽，在芬兰和英国有大量的种群存在。

堤岸田鼠个头小，体长只有 10~11 厘米，体重平均 17~20 克，眼睛和耳朵都很小。它们的皮毛很厚，通常呈棕色或灰色，覆盖全身。相对于它们的体型，堤岸田鼠的尾巴很短，脑袋也很小。堤岸田鼠每胎 4~8 只，幼崽出生时没有视觉且非常脆弱，但成熟速度相当快，雌性在 2~3 周内成熟，雄性在 6~8 周内成熟。堤岸田鼠的平均寿命反映了这种快速成熟的状态，多数个体寿命在半年到两年之间。

这些小型啮齿类动物主要在黄昏时活跃，但它们也可以昼夜活动。饮食主要是植物性物质，食物种类随季节而变化。在社会属性上，雌性堤岸田鼠比雄性堤岸田鼠更具优势，后者一旦成熟就会分散四处，而雌性堤岸田鼠通常会留在更靠近它们出生的地方。

鉴于它们相对健康的种群数量和广泛的分布，堤岸田鼠目前的保护状态被列为无危（least concern，LC）。O’Reilly 图书封面上的许多动物濒临灭绝，它们对世界很重要。

封面插图来自 Karen Montgomery 的作品，该插图基于 *British Quadrupeds* 的黑白版画。

注 1：此书已由人民邮电出版社出版。——编者注

专为量化交易设计的实时行情数据API: www.alltick.co

微信连接

回复“机器学习”查看相关书单

微博连接

关注@图灵教育 每日分享IT好书

QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

专为量化交易设计的实时行情数据API: www.alltick.co

O'REILLY®

金融人工智能 用Python实现AI量化交易

人工智能和机器学习的广泛应用给当今的许多行业带来了根本性的变革。在金融领域，人工智能技术也已锋芒初露。通过阅读本书，你将了解如何利用神经网络和强化学习等方法，对金融市场的走势做出预测。

作者伊夫·希尔皮斯科博士基于多年开发、回测和部署人工智能算法交易策略的实战经验，展示了将人工智能算法应用于金融场景的实用方法。本书包含大量Python示例，有助于你边学边练，轻松复现书中的所有结果。

- 学习人工智能的主要概念和算法，并了解通用人工智能和超级智能
- 理解机器学习和数据驱动的金融学将如何改变金融理论和实践
- 运用神经网络和强化学习等方法，发掘金融市场的统计失效现象
- 学习向量化回测和算法交易，并掌握人工智能算法交易策略的执行与部署
- 展望金融人工智能的未来，涉及基于人工智能的竞争和金融奇点

伊夫·希尔皮斯科（Yves Hilpisch），金融数学博士，Python Quants公司创始人兼CEO，致力于普及人工智能、算法交易等相关技术在金融数据中的应用。此外，他还创建了AI Machine平台，提供人工智能算法交易策略的标准化部署。伊夫是国际量化投资分析师认证（CQF）讲师，讲授计算金融学、机器学习和算法交易。他另著有《Python金融大数据分析》。

"凭借其全面和直观的方法，这将是金融领域从业人员和学者的主要参考书。"

——Abdullah Karasan
金融数据科学学者

"这是一本优秀的机器学习实践指南，旨在解决量化金融领域的一系列问题。"

——Tim Nugent
路孚特公司研究员

"这本书有助于熟悉人工智能技术在量化投资交易中的应用之道。"

——漆远
复旦大学浩清特聘教授，复旦大学人工智能创新与产业研究院院长、前蚂蚁集团AI科学家

"希尔皮斯科博士将正统金融理论和人工智能有机融合，带领读者一步步走进金融人工智能的世界。"

——梁举
BigQuant人工智能量化平台创始人兼CEO

"投资行业正处于一个加速变革的过程中。这本实用指南是不可多得的工具书。"

——张一
CFA Institute中国区总经理

AI / PYTHON / FINANCE

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn

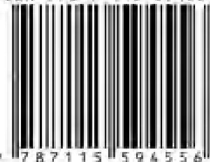
分类建议 计算机/机器学习/Python

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社有限公司出版

此简体中文版仅限于在中华人民共和国境内（不包括香港特别行政区、澳门特别行政区及台湾地区）销售发行
This Authorized Edition for sale only in the People's Republic of China (excluding Hong Kong SAR, Macao SAR and Taiwan)

扫码领取
随书代码资料


ISBN 978-7-115-59455-6

定价：129.80元

专为量化交易设计的实时行情数据API: www.alltick.co