05.《自然语言处理实战·第2版》第5章 词脑：神经网络

本章内容

- 为神经网络构建基础层
- 使用反向传播训练神经网络
- 用 Python 实现基础神经网络
- 用 PyTorch 实现可扩展神经网络
- 堆叠网络层以获得更好的数据表示
- 调优神经网络以提升性能

当你阅读本章标题中的 *word brain* 时，你大脑中的神经元可能已经开始激活，试图提醒你以前在哪儿听过类似的说法。而当你读到 *heard* 这个词时，你的神经元或许正在把标题中的词与处理单词**声音**的大脑区域联系起来。也许，你听觉皮层中的神经元开始把短语 *word brain* 与押韵的常用短语，如 *bird brain*，联系在一起。

即便我们的大脑并不能很好地预测你的大脑反应，你也即将构建一个能够处理单个单词并预测其含义的小型“大脑”。这个 word brain 将在某些特别困难的 NLP 任务上远胜于人类集体大脑。神经网络甚至能在处理的词是人名而对人类来说毫无意义时，对其含义做出预测。

别担心这些关于大脑、预测和单词的讨论让你困惑。你将从一个简单的人工神经元开始，用 Python 编写，并使用 PyTorch 处理连接神经元所需的所有复杂数学，进而构建人工神经网络。一旦你理解了神经网络，就会开始理解**深度学习（deep learning）**，并能够将其应用于现实世界，以获得积极的社会影响，甚至（如果你坚持的话）获利。

5.1 为什么选择神经网络?

当你在机器学习中使用深度神经网络时，它被称为**深度学习**。过去几年里，深度学习在许多棘手的 NLP 问题上打破了准确率和智能上限，例如：

- 问答系统
- 阅读理解
- 摘要生成
- 自然语言推理

最近，深度学习（深度神经网络）实现了一些过去难以想象的应用：

- 长时间且引人入胜的对话
- 伴侣式交互

- 编写软件

最后一个——编写软件——尤其有趣，因为 NLP 神经网络正被用来为——等等，什么？！——NLP 编写软件。这意味着 AI 和 NLP 算法正日益接近自我复制并自我改进的那一天。这重新点燃了人们将神经网络视作通往**通用人工智能（AGI）**之路（或至少更通用智能机器）的希望与兴趣。人们已经直接用 NLP 生成软件，以提升这些 NLP 算法的智能。[1]

自 2012 年以来，随着 Geoffrey Hinton 对神经网络架构的改进，模型复杂度出现明显拐点，并迅速流行。自 2012 年起，用于最大 AI 训练的计算量呈指数级增长，翻倍周期约为 3.4 个月。

神经网络之所以能实现这一切，是因为它们：

- 能够用少量样本完成更好的泛化
- 可以自动从原始数据中学习特征
- 能在几乎没有标签的文本上轻松训练

5.1.1 词语的神经网络

使用神经网络，你无需猜测是否应把专有名词或平均词长、手工制作的词语情感分数作为模型所需特征；也无需使用可读性分数或情感分析器来降低数据维度，更不必依赖停用词过滤、词干提取、词形还原、LDA、PCA、t-SNE 或聚类等无监督降维方法。神经网络这个“小脑”会根据词语与目标之间的统计关系为你自动、最优地完成这些工作。

> **WARNING**
> 除非你确信能够提升性能，否则请避免在深度学习流水线中使用词干提取、词形还原或其他基于关键词的预处理；它们常会“剥夺”模型的重要信息，减少模型可利用的模式与特征。

如果你确实在做词干提取、词形还原或关键词分析，请尝试去掉这些过滤器后再比较效果；无论你用 NLTK、Stanford Core NLP 还是 spaCy，这些手工规则算法往往弊大于利，因为它们受限于人工标注词汇和规则。以下预处理算法很可能令你的神经网络失效：

- Porter 词干算法
- Penn Treebank 词形还原器
- Flesch-Kincaid 可读性分析器
- VADER 情感分析器

在高度互联、快速演化的现代机器学习与深度学习世界里，自然语言变化过快，这些算法难以跟上。词形还原器、词干器和情感分析器常会对诸如 *hyperconnected* 与 *overfit* 等新词处理不当。[2]

深度学习改变了 NLP 的游戏规则。过去像 Julie Beth Lovins 这样的语言学家需要手工编写算法来抽取词干、词形和关键词。[3]（她的一次遍历词干与词形还原算法后来被 Martin Porter 等人改良。[4]）深度神经网络如今让所有这些繁琐工作变得不必要；它们直接根据统计学意义理解词语，并在无需刻意编写规则算法的情况下完成任务。

即便功能强大的特征工程方法，如第 4 章讨论的潜在语义分析（LSA），也无法与神经网络的自然语言理解（NLU）能力相匹敌。

Tip
以下两本关于自然语言与神经网络的优秀教材可帮助你掌握 NLP 术语，并可直接用于训练深度学习流水线：

- *A Primer on Neural Network Models for Natural Language Processing*，Yoav Goldberg（https://archive.is/BNeqK）
- *CS224d: Deep Learning for Natural Language Processing*，Richard Socher（https://web.stanford.edu/class/cs224d/lectures/）
  你也可以参考 Manning 的 *Deep Learning for Natural Language Processing*（https://www.manning.com/books/deep-learning-for-natural-language-processing）。

即使是强大的自动特征工程方法，如决策树、随机森林、梯度提升树，也不具备神经网络那样的深度语言理解能力。传统的机器学习算法造就了全文检索与开放获取知识库，而神经网络的深入学习让人工智能与智能助手成为现实，并已融入众多数字产品，正以你数年前难以想象的方式驱动你的思考。

你在前几章学到的 NLP 技术即将变得更加强大。为了让算法造福社会而非带来破坏，你需要深入了解神经网络的工作原理，并理解它们为何在许多 NLP 问题上表现出色，却在另一些问题上失败。[5]

如果错误地使用神经网络，你可能会陷入只在测试数据上有效、却在真实世界灾难性的过拟合陷阱。首先，你需要对单个神经元的工作原理建立直觉。

5.1.2 神经元作为特征工程师（feature engineer）

线性回归（linear regression）、逻辑回归（logistic regression）和朴素贝叶斯（naive Bayes）模型的一个主要局限是：它们都要求你逐一手动设计（engineer）特征。你必须在所有可能的文本

数值表示方法中找出最优方案，然后再对一个接收这些人工特征表示并输出预测的函数进行参数化。只有完成这些步骤，优化器（optimizer）才能开始搜索最能预测输出变量的参数值。

> **NOTE** 在某些情况下，你仍需要为 NLP 流水线手动设计阈值型特征，尤其当你需要一个可解释模型，方便与团队讨论并映射到现实现象时。若要在极少特征的情况下创建一个更简单的可解释模型，需要你检查每个特征的残差图（residual plot）。当残差在某个特征值处出现不连续或非线性时，这就是一个适合加入管道的阈值。当运气好时，你甚至能在工程阈值与现实现象之间发现关联。

例如，第 3 章中使用的 TF–IDF 向量表示非常适合信息检索（information retrieval）和全文搜索（full-text search）。然而，在真实世界中，单词经常被拼错或以含糊方式使用，而 TF–IDF 向量往往无法很好泛化用于语义搜索或自然语言理解（NLU）；第 4 章介绍的 PCA（主成分分析，Principal Component Analysis）或 LSA（潜在语义分析，Latent Semantic Analysis）也可能无法给出适合你具体问题的主题向量表示。它们适合可视化，但并非 NLU 的最佳选择。多层神经网络会为你执行这种特征工程，并以某种意义上的最优方式完成——神经网络在更广泛的可能特征函数空间中搜索。

应对多项式特征爆炸（polynomial feature explosion）

另一个神经网络可为你优化的特征工程示例是多项式特征抽取（polynomial feature extraction；想想你上次使用 `sklearn.preprocessing.PolynomialFeatures` 时的经历）。在特征工程过程中，你可能猜测输入与输出间的关系是二次的，于是将输入特征平方，然后用这些新特征重新训练模型，看看测试集准确率是否提高。基本做法是：如果某特征的残差（预测减测试标签）不像以零为中心的白噪声，那么就可使用某些非线性函数（如平方 `(**2)`、立方 `(**3)`、平方根 `sqrt`、对数 `log` 或指数 `exp`）变换该特征，以进一步减少误差。你可以随意构造函数，并逐渐培养出直觉，帮助你确定哪种函数能提升准确率。如果你不知道哪些交互作用至关重要，就不得不把所有特征两两相乘。

你知道这只“兔子洞”的深度和广度：可能的四阶多项式特征数量几乎无限。如果你试图将 TF–IDF 向量的维度从数万降到数百，再引入四阶多项式特征会使维度呈指数级膨胀，甚至超过 TF–IDF 本身的维度。

即便拥有数百万个可能的多项式特征，仍然还有数百万个阈值特征。随机森林（random forest）的决策树和提升树（boosted tree）在自动化特征工程方面已有长足进步，但仍难以找到合适的阈值特征，这基本仍是个未解决的问题。

功能强大的特征表示往往难以解释，有时在现实中并不具备良好泛化性，这正是神经网络能够提供帮助的地方。特征工程的“圣杯”是寻找能够解释现实世界物理规律的表示；如果特征按照真实世

界现象可解释，你就可以开始相信它们不仅仅是预测性的，而是可能揭示了某种因果关系。

Peter Woit 指出，现代物理学中可能模型数量的爆炸大多 **Not Even Wrong**[7]；当你使用 `sklearn.preprocessing.PolynomialFeatures` 时创造的模型正是如此——这是真正的问题。数以百万计的多项式特征中，极少数在物理上可实现；换言之，大多数多项式特征只是噪声。[8] 因此，如果你在预处理阶段使用 `PolynomialFeatures`，请将 `degree` 参数限制为 2 或更低。

NOTE 对于任何机器学习流水线，务必确保你的多项式特征不包含三个以上物理量的乘积。如果你决定尝试次数大于 2 的多项式特征，可以通过过滤掉不现实（幻想型）的三元交互特征来避免麻烦。例如，`x1 * x2 ** 2` 是合法的三次多项式特征，而 `x1 * x2 * x3` 则不合法。涉及三个以上特征相互作用的多项式特征在物理上无法实现。去掉这些“幻想特征”能提升 NLP 流水线的稳健性，并帮助你减少生成模型产生的幻觉。

我们希望此刻的你已对神经网络带来的各种可能性感到振奋。接下来，让我们正式踏入神经网络世界，从构建看起来很像逻辑回归的小型单神经元开始。最终，你将能够将这些神经元按层组合与堆叠，以最优方式完成特征工程。

5.1.3 生物神经元（Biological neurons）

Frank Rosenblatt 基于自己对人脑中生物神经元工作方式的理解，提出了第一个人工神经网络。他把它称为**感知机（perceptron）**，因为他用它来帮助机器感知周围环境，将传感器数据作为输入。[9] 他希望借此革新机器学习，免去为数据手工制作滤波器以抽取特征的需求，并自动化为任何问题寻找最佳函数组合的过程。

Rosenblatt 的目标是让工程师在构建 AI 系统时不必为每个问题设计专门模型。当时，工程师常使用线性回归、多项式回归、逻辑回归和决策树来帮助机器人做决策。Rosenblatt 的感知机是一种全新的机器学习算法，能够近似任何函数，而不仅仅是直线、逻辑函数或多项式。[10] 他以生物神经元的工作方式为基础，在已有大量成功逻辑回归模型的历史上，稍微修改了优化算法，使其更贴近神经科学家关于生物神经元如何随时间调整对环境响应的研究成果。

电信号通过**树突（dendrite）流入你大脑中的生物神经元（见图 5–1），进入细胞核（nucleus）**。细胞核会积累电荷，并随时间累积。当核内累积电荷达到该神经元的激活水平时，它会通过**轴突（axon）发射电信号。一个神经元的轴突与另一神经元树突相接之处称为突触（synapse）**；不过神经元并非生而平等。你大脑中某些神经元的树突对某些输入比对其他输入更“敏感”，而细胞核本身的激活阈值也因其在大脑中的功能不同而差异较大。因此，对某些更敏感的神经元而言，仅需更小的输入信号即可触发输出信号沿轴突传递。

你可以想象神经科学家如何通过真实神经元实验来测量单个树突和神经元的敏感度，并用数值来表示这种敏感度。Rosenblatt 的感知机将这种生物神经元抽象为人工神经元，并为每个输入（树突）赋予一个**权重（weight）**。在感知机等人工神经元中，你用数值权重或增益表示各树突对输入信号的敏感度。生物细胞在决定输出轴突发射强度和频率时，会增强或抑制输入信号；权重越高，表示对输入中微小变化越敏感，对应给定输入的输出信号越强。

生物神经元会在决策过程中动态改变这些权重，在其一生中学习在特定情境下哪些输出会得到奖励。你即将使用称为**反向传播（backpropagation）**的机器学习过程模拟这种生物学习机制。但在学习如何将权重变化反向传播到网络之前，请先查看图 5–2，理解单个神经元中的前向传播如何工作。前向传播的数学是否让你想起了线性回归?

AI 研究人员希望用神经网络这种更具模糊性与泛化能力的逻辑——一群“小型大脑”——取代逻辑回归、线性回归和多项式特征抽取的刻板数学。Rosenblatt 的人工神经元甚至能处理三角函数等高度非线性函数。每个神经元解决问题的一部分，并可与其他神经元组合学习更复杂的函数（不过并非所有函数——即使简单如 XOR 门，单层感知机也无法解决）。这些人工神经元的集合就是他所谓的感知机。

Rosenblatt 当时并未意识到，他的人工神经元可以像生物神经元那样层层连接。在现代深度学习中，我们把一组神经元的预测连接到另一组神经元，以进一步精炼预测。这使我们能够创建**分层网络**以建模任何函数；只要数据和时间足够，它们几乎可以解决所有机器学习问题。图 5–3 展示了神经网络如何堆叠多层以产生更复杂的输出。

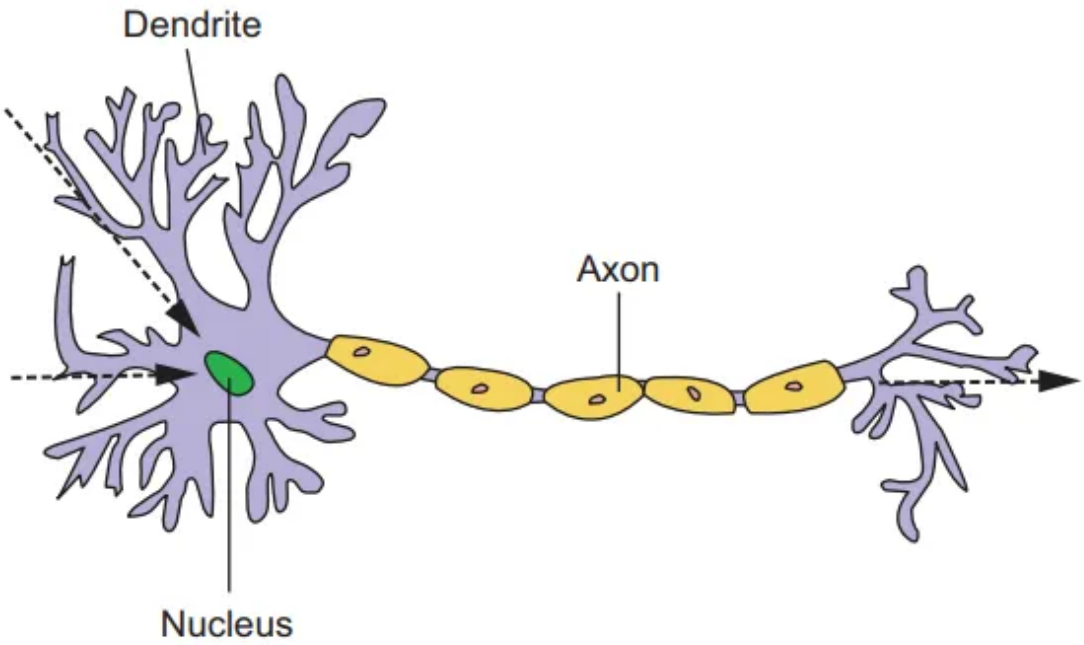

图 5–1　一种生物神经元细胞

（标注：树突 Dendrite，轴突 Axon，细胞核 Nucleus）

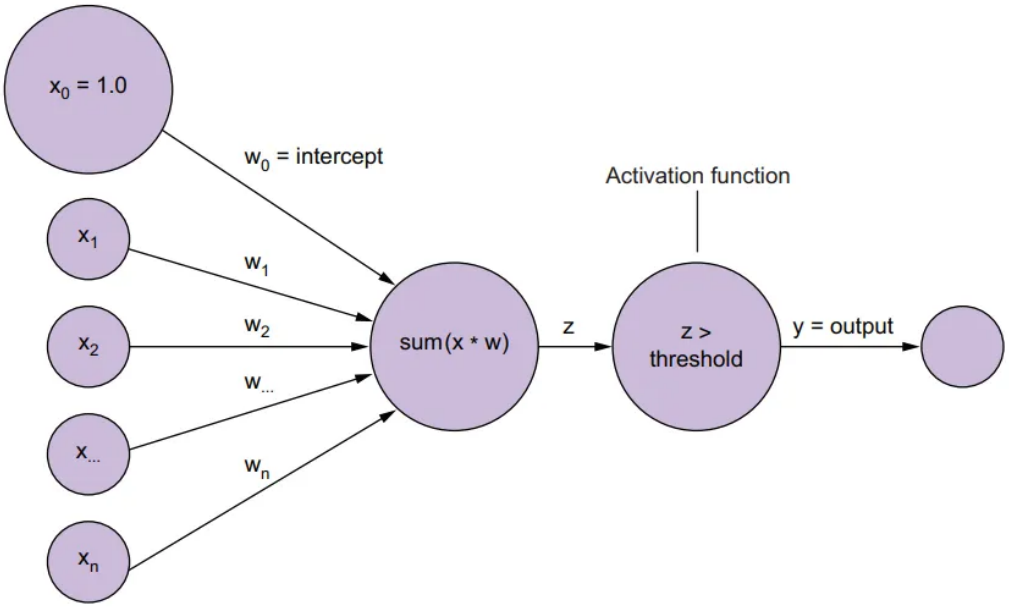

图 5–2　基本感知机

（标注示例：x0=1.0、w0=截距、sum(x * w)、激活函数、z > threshold、y=输出）

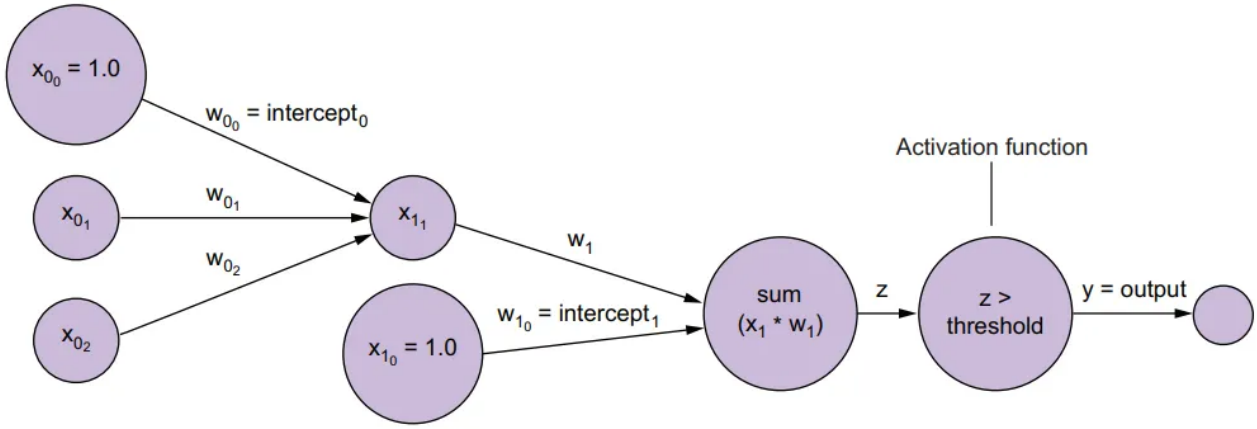

图 5–3　神经网络层

5.1.4 感知机（Perceptron）

神经元最复杂的工作之一是处理语言。思考一下，感知机如何用于处理自然语言文本。图 5.2 中的数学公式是否让你想起之前用过的某些机器学习模型？在你熟悉的模型中，哪些会把输入特征与一组权重或系数相乘？答案是线性回归。但如果在线性回归输出上再套用一个 sigmoid 激活函数或逻辑函数，就开始很像逻辑回归了。

感知机中使用的 **sigmoid 激活函数** 与逻辑回归中的逻辑函数完全相同——sigmoid 一词仅表示 S 形曲线。逻辑函数的形状非常适合构造软阈值或逻辑二元输出，因此，此处神经元的行为等价于在输入上执行逻辑回归。

Python 中实现逻辑函数的公式如下：

```
def logistic(x, w=1., phase=0, gain=1):
    return gain / (1. + np.exp(-w * (x - phase)))
```

下面演示逻辑函数的外观，以及系数（权重）和相位（截距）如何影响其形状：

```
import pandas as pd
import numpy as np
import seaborn as sns
sns.set_style()

xy = pd.DataFrame(np.arange(-50, 50) / 10., columns=['x'])
for w, phase in zip([1, 3, 1, 1, .5], [0, 0, 2, -1, 0]):
    kwargs = dict(w=w, phase=phase)
    xy[f'{kwargs}'] = logistic(xy['x'], **kwargs)
xy.plot(grid="on", ylabel="y")
```

在前几章进行逻辑回归时，你的输入是什么？你先用关键词检测器 `CountVectorizer` 或 `TfidfVectorizer` 处理文本；这些模型使用分词器（第 2 章介绍过）将文本切分成单词并计数。因此，在 NLP 中常用 BOW 计数或 TF-IDF 向量作为模型输入，对神经网络亦然。

Rosenblatt 的每个输入权重（生物树突）都可调节该信号的权重或敏感度。他用电位计（类似老式收音机的音量旋钮）来实现权重，让研究人员可以逐一手动调整神经元对各输入的敏感度。通过调节这个“旋钮”，可以使感知机对 BOW 或 TF-IDF 中某个词的计数更敏感或更迟钝。

当某个词的信号根据权重被放大或减弱后，信号就进入了感知机“本体”（在生物神经元中亦是如此），所有输入信号在此相加。随后，此信号会经过软阈值函数（如 sigmoid），再沿轴突输出。生物神经元仅当信号超过某阈值才会触发；感知机中的 sigmoid 方便地把阈值设在最小一最大范围的 50%。若某组输入组合不足以触发神经元，则表示负类匹配。

5.1.5 Python 版感知机

机器可以通过将数值特征乘以“权重”并求和来模拟一个极其简单的神经元，从而做出预测或决策。这些数值特征把对象表示成机器可理解的向量。举例来说，假设某服务希望根据房屋的自然语言描述预测房价，把描述表示成数值向量后即可实现。仅用 NLP 模型该怎么做？

你可以像第 2、3 章那样，把房屋描述中的每个单词计数作为特征；或者像第 4 章那样，用主成分分析（PCA）之类的变换把上千维压缩成主题向量。但这些方法只是基于方差等指标猜测哪些特征重要。实际上，描述中的关键词也许正是房屋面积和卧室数量这种数值，而词向量与主题向量会完全漏掉这些信息。

在“常规”机器学习任务（如房价预测）中，你通常有结构化数值数据——例如面积、上一笔成交价、卧室数量、经纬度或邮编等列。但对于自然语言问题，我们希望模型能够处理非结构化文本：模型必须自行确定哪些词，以及它们以何种组合或顺序，对目标变量具有预测性。模型要阅读房屋描述，并像人类大脑那样猜测房价；神经网络是最接近模拟人类直觉的机器。

深度学习的美妙之处在于：你可以把能想到的**所有**特征都作为输入。这意味着你可以直接输入完整文本描述，由转换器生成高维 TF-IDF 向量——神经网络也能处理；甚至可以输入更高维的表示，或干脆把未过滤的文本作为 one-hot 单词序列传入。还记得第 2 章提到的钢琴卷帘图吗？神经网络天生适合处理这类自然语言数据的原始表示。

浅层学习

对于你的第一个深度学习 NLP 问题，我们先保持“浅层”。要理解深度学习的神奇之处，最好先看看**单个神经元**是如何工作的。

你需要为模型输入的每个特征找到一个**权重**。可以把这些权重想象成信号进入神经元时被保留的百分比。如果你熟悉线性回归，会发现图示里的斜率就是线性回归的系数；如果再加上逻辑函数（sigmoid），这些权重就对应逻辑回归在训练数据中学到的系数。换言之，一个单神经元输入的权重在数学上等价于多元线性回归或逻辑回归中的斜率。

> **TIP**
> 与 scikit-learn 里的机器学习模型一样，单个特征记作 x_i，在 Python 中写作 `x[i]`。i 表示特征在输入向量中的位置，某个样本的所有特征组成向量
> $x = x_1, x_2, \ldots, x_i, \ldots, x_n$.
> 相应地，每个特征的权重写作 w_i，权重向量为
> $w = w_1, w_2, \ldots, w_i, \ldots, w_n$.

有了特征，就把每个特征 x_i 乘以对应权重 w_i 并求和：

$$y = (x_1 \times w_1) + (x_2 \times w_2) + \cdots + (x_i \times w_i).$$

下面用一个简单示例加深理解。假设输入 BOW 向量来自短语 *green egg egg ham ham ham spam spam spam spam spam*：

```
>>> from collections import Counter

>>> np.random.seed(451)
>>> tokens = "green egg egg ham ham ham spam spam spam spam spam".split()
>>> bow = Counter(tokens)
>>> x = pd.Series(bow)
>>> x
green    1
egg      2
ham      3
spam     4

>>> x1, x2, x3, x4 = x
>>> x1, x2, x3, x4
(1, 2, 3, 4)

>>> w0 = np.round(.1 * np.random.randn(), 2)
>>> w0
0.07
```

```
>>> w1, w2, w3, w4 = (.1 * np.random.randn(len(x))).round(2)
>>> w1, w2, w3, w4
(0.12, -0.16, 0.03, -0.18)

>>> x = np.array([1., x1, x2, x3, x4])   # 额外的 1 对应截距
>>> w = np.array([w0, w1, w2, w3, w4])   # w0 为截距权重
>>> y = np.sum(w * x)
>>> y
-0.76
```

这个四输入一输出、尚未训练的单神经元网络输出了 −0.76。

接下来要在输出 y 上应用**非线性函数**，使其不再只是线性回归。常用做法是阈值（threshold）或裁剪（clipping）函数：若加权和超过阈值，感知机输出 1，否则输出 0；这在图 5.2 中标为 *Activation function*。

```
>>> threshold = 0.0
>>> y = int(y > threshold)
```

若希望模型输出连续的概率或似然而不是二元 0/1，可使用本章前面定义的逻辑激活函数[11]：

```
>>> y = logistic(x)
```

神经网络与其他机器学习模型一样 —— 你提供输入（特征向量）和输出（预测），网络通过试错找到最佳权重。**损失函数（loss function）** 衡量模型的预测误差。

请注意，我们这里给出的 Python 代码仅实现了一个神经元的 **前向传播（feed forward）**。它的数学形式与你在 scikit-learn 中 `LogisticRegression.predict()` 函数（四输入一输出的逻辑回归）里看到的几乎相同[12]。

> NOTE
> *Loss function* 输出分数，表示模型预测有多“差”；*objective function* 衡量模型有多“好”，通常误差越小越好。就像考试中的错误率与得分率：一个衡量错了多少题，另一个衡量得了多少分。两者都能帮助模型学到正确答案并不断改进。

为什么有个额外的权重参数

你注意到还有一个额外的权重 w_0 吗？输入中并没有标记为 X_0 的项，可为什么要有 w_0 ？能猜到为什么总要给神经元一个常量值为 1.0 的输入并把它命名为 X_0 吗？回想你以前写过的线性回归和逻辑回归公式。还记得单变量线性回归中的那个额外系数吗？

$$y = m \times x + b$$

这里 y 是模型输出或预测，x 是唯一的自变量特征，而 m 代表斜率。那 b 是什么？

$$y = \text{slope} \times x + \text{intercept}$$

现在，你能猜到额外的权重 w_0 是做什么用的吗？并且为什么它总是与值 1.0 的输入相乘？

$$w_0 \times 1.0 + w_1 \times x_1 + \cdots + (x_n \times w_n)$$

这就是线性回归里的 **截距（intercept）**，在神经网络这一层中则“换名”叫 **偏置（bias）权重** w_0 。

图 5.2 中的 w_0 示例引用了 *bias*。偏置是什么？它是神经元的“常开”输入。神经元为它专门保留一个权重，与其他输入的权重一同训练。文献里有两种写法：要么把 1 视为输入向量的基准元素、在向量前或末尾追加一个 1（让向量维度变为 n+1），要么在示意图里省略 1，但将独立的偏置权重乘以隐含的 1 后加到点积中。两种方法本质相同。

引入偏置权重的原因是让神经元对全零输入保持鲁棒。网络也许需要在全零输入时输出 0，有了偏置权重就不会出现 0 × 权重 = 0 的问题。若需要输出 0，神经元可以把偏置权重调低，使点积低于阈值即可。

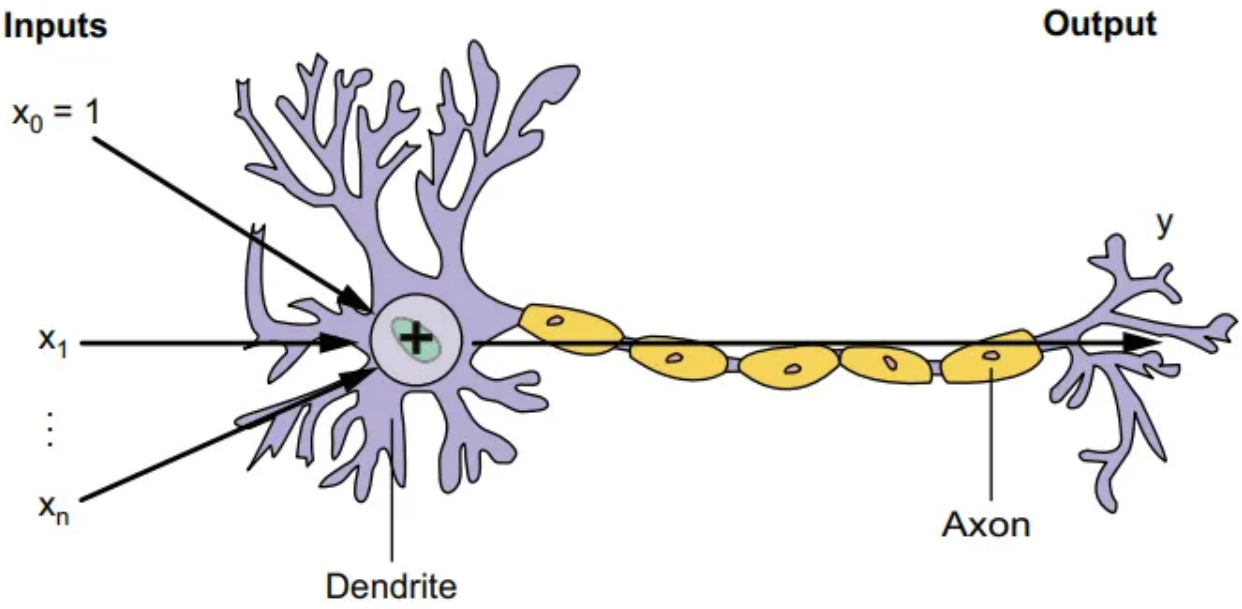

图5.4 感知器和生物神经元

图 5.4 展示了生物神经元与感知机在输入–输出过程上的对应关系。

```
>>> def neuron(x, w):
...     z = sum(wi * xi for xi, wi in zip(x, w))     # x 和 w 必须是数值向量
...     return z > 0                                 # 这里的表达式即 w.dot(x)
```

若你更习惯使用 NumPy 的向量化运算，可写成：

```
>>> def neuron(x, w):
...     z = np.array(w).dot(x)
...     return z > 0
```

NOTE 任何 Python 条件表达式都会返回布尔值 `True/False` 。在数学运算中， `True` 会被强制转换成 1 或 1.0， `False` 转成 0，这样你就可以把布尔值与数字相乘或相加。

变量 *w* 保存模型的权重向量；这些值会在训练过程中根据目标输出调整。变量 *x* 保存输入信号向量，如 NLP 模型中的 TF–IDF 向量；对生物神经元而言，输入信号就是树突上的电脉冲速率。

TIP 输入 *x* 与权重 *w* 逐项相乘再求和，与向量点积 `w.dot(x)` 完全相同。这正是**线性代数**为何对神经网络如此有用；GPU 能并行完成大量点积乘加运算，速度可比四核 CPU 快 250 倍以上。

如果你更喜欢数学符号，式 5.1 用求和表示同一激活阈值：

$$f(\vec{x}) = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} x_i w_i > \text{threshold} \\ 0 & \text{else} \end{cases} \qquad \text{(Equation 5.1)}$$

激活阈值方程与 Python 表达式 `int(sum(x_i * w_i) > threshold)` 等价，使分类器能基于输入特征作二元决策。由于会丢弃加权和的幅度信息，这种阈值激活函数只应放在网络的最后输出层。加权和的大小仍可用来衡量分类器对该样本决策的置信度。

你的感知机目前还没**学习**任何东西，但你已经完成了重要一步：把数据输入模型并得到输出。虽然输出很可能是错的（因为权重还未学习），但接下来就会变得有趣。

> **TIP** 神经网络的基本单元是**神经元（neuron）**。感知机只是神经元的一种特殊情况。暂把“感知机”当作神经元来看待；当不再适用时，我们会回到更通用的术语。

5.2 一个逻辑神经元示例

当你把逻辑函数用作神经元的**激活函数（activation function）**时，本质上就已经创建了一个逻辑回归模型。单个神经元如果采用逻辑函数作为激活函数，在数学上等同于 scikit-learn 中的 `LogisticRegression` 模型——唯一的区别只是训练方式不同。本节首先训练一个逻辑回归模型，然后将其与在同一数据集上训练的单神经元网络进行比较。

5.2.1 点击诱饵的逻辑学

软件（以及人类）经常需要基于逻辑条件做决策。举例来说，你每天可能要多次决定是否点击某个链接或标题。有时这些链接会把你带到假新闻文章，于是大脑在点击之前会遵循一些逻辑规则：

- 这是你感兴趣的话题吗?
- 这个链接看起来像推广或垃圾信息吗?
- 它来自可信来源或你喜欢的来源吗?
- 它看起来是真实还是虚假?

每个决定都可以在机器中建模为一个人工神经元，并据此在电路板中实现逻辑门，或在软件里实现条件表达式（if 语句）。如果用人工神经元来做，这四个决定最小只需四个逻辑回归门，就能构建出处理它们的“微型大脑”。

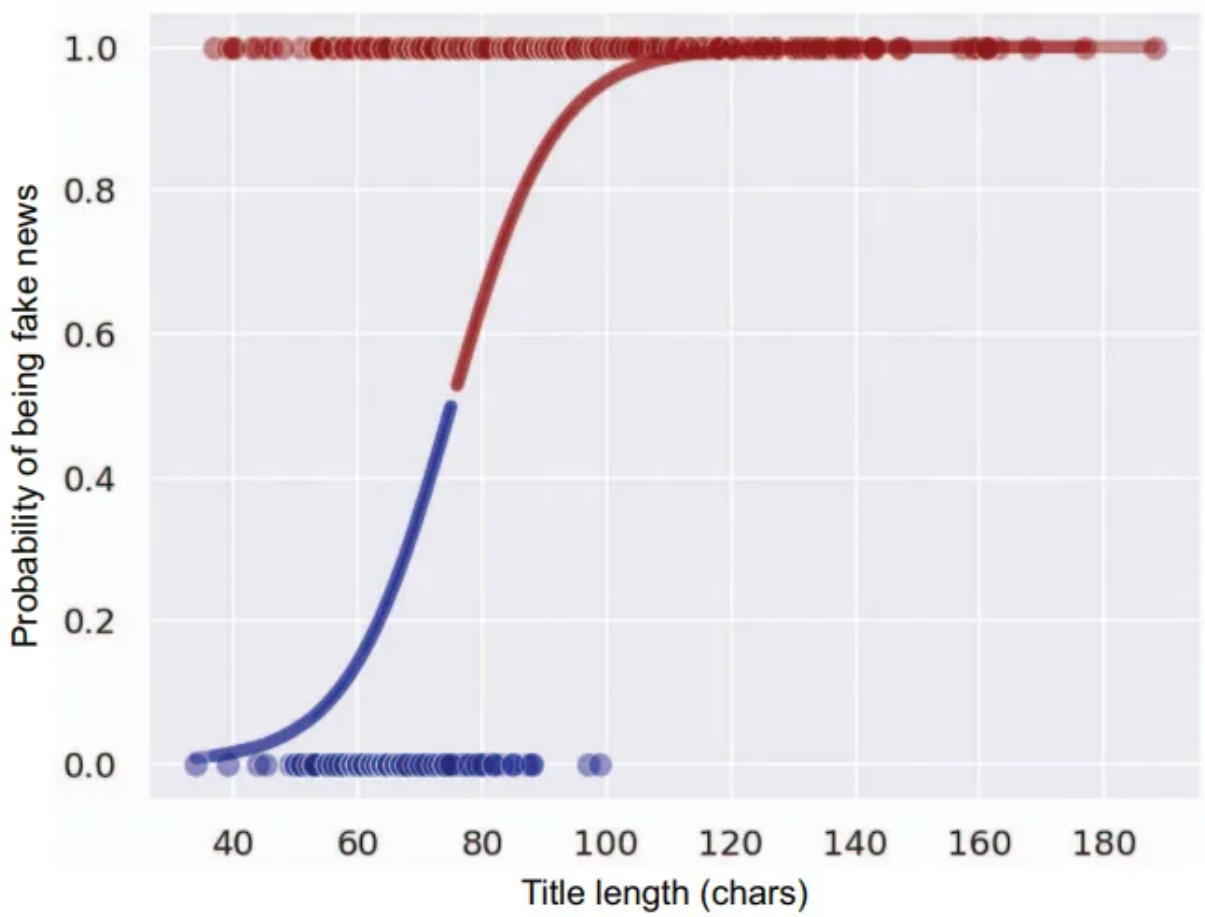

图5.5 逻辑回归：假新闻的标题长度

为了模仿大脑的“clickbait 过滤器”，你可能决定用标题长度训练逻辑回归模型——也许你猜标题越长越可能耸人听闻、夸张。图 5.5 展示了假新闻与真实新闻标题在字符长度上的散点图以及单特征逻辑回归曲线，其中神经元输入权重相当于曲线中段的最大斜率。

5.2.2 性别教育

（看到小节标题“性别教育”是否像点击诱饵？）由于假新闻（点击诱饵）数据集已在 Kaggle 被充分利用，本节转向更有趣且更实用的数据集：用感知机（人工神经元）预测名字的性别。

要解决的问题是大脑每天要处理的 NLU 日常任务：在人际社交媒体互动中，大脑乐于猜测对方出生时被登记的性别[13]。使用一个人工神经元，仅凭名字中的字母就能以约 80% 准确率完成这一挑战。本示例将使用跨越 100 多年的美国 317 万张出生证明数据中的一个样本。

在美国，婴儿出生证上会记录姓名、性别和出生日期。**sex at birth** 这一类别由填写并签署出生证的人基于外生殖器、染色体或激素判断，大部分新生儿被标记为 **male** 或 **female**，但这种简单二分无法覆盖现实[14]。

male 和 **female** 并非“出生性别”分类的终点。美国疾病控制与预防中心（CDC）建议在 USCDI 标准中为临床/医学用途加入若干非二元性别类别，除 *female* 和 *male* 外，还包括 *unknown* 和 *something not listed (specify)*[15]。

测试集中的姓名不应出现在训练集中，并且每个姓名只应有一个“正确”性别标签。事实上，没有任何姓名有唯一正确的二元性别标签；对于任一姓名，都存在一个“男性/女性”概率分值，该值会随着数据添加而变化。

5.2.3 代词、性别与社会性别

我们若只谈“出生时指定的性别（sex）”而不讨论“社会性别（gender）”就无法深入性别分类这一主题。性别是一个比出生性别更宏大、更复杂的概念，它涵盖社会建构的角色、行为、表达与认同，并且可能与个人的出生性别不同。一个人的**性别认同（gender identity）**并不局限于男性/女性二元范畴，且可随时间变化。**代词（pronoun）**是人们用来表达与确认其性别认同的重要方式之一。

讨论性别是一件细腻的事务，具有重大的法律与社会影响。在许多文化——尤其是规范严格的文化——中，讨论与表达性别认同可能带来严重后果，有时甚至威胁到个人安全。正因如此，我们曾犹豫是否在本书中加入这一节；然而，性别、社会性别与性别认同三者之间存在人们常常假设的隐性联系，值得深入探讨。自然语言处理（Natural Language Processing, **NLP**）技术既可能生成带有性别偏见并加剧社会性别歧视与跨性别恐惧的系统，也可以用来识别并减轻数据集、模型与系统中的性别偏见，从而影响塑造我们生活的工具。

NLP 中有一个重要挑战叫**核心指代消解（coreference resolution）**：算法需识别文本中代词所指涉的对象或词语[16]。例如，下列句子中的代词：“Maria was born in Ukraine. Her father was a physicist. 15 years later she left there for Israel.”你可能未察觉，但在眨眼之间已解决了三个指代：判断“Her”指向“Maria”，并推断“there”指代“Ukraine”。

认识到大脑对姓名附加的偏见（这种偏见往往影响现实生活中的行为）[17]有助于你设计出能够抵消并补偿该偏见的算法。因此，了解文本中姓名所隐含的“男性化”或“女性化”信息，对于构建自然语言理解（Natural Language Understanding, **NLU**）流水线至关重要。即使某个名字所对应的出生性别并不能很好地指示文中人物的呈现性别，这种信息也依然有用。作者往往期望读者根据名字对性别作出假设；在性别颠覆题材的科幻小说中，像 William Gibson 这样的前瞻性作家正是利用这一点来激发读者思考并拓宽视野[18]。本章使用了简化的二元性别数据集，为你打下从零开始学习自然语言处理技能的基础。

TIP

为确保 NLP 流水线和聊天机器人对所有人友好、包容，可在处理文本数据时对任何性别信息进行规范化（normalize），以消除算法偏见。下一章你将看到性别如何影响算法决策，以及如何影响企业和雇主的每日决策。

5.2.4 性别分类的后勤准备（Sex logistics）

首先，导入 pandas，并将 `max_rows` 设置得小一些，以便 DataFrame 只显示几行：

```
>>> import pandas as pd
>>> import numpy as np
>>> pd.options.display.max_rows = 7
```

接着，从 nlpia2 仓库下载原始数据，并随机抽样 10 000 行，以保证在任何电脑上都能快速运行：

```
>>> np.random.seed(451)
>>> URL = 'https://gitlab.com/tangibleai/nlpia2/'\
...         '-/raw/main/src/nlpia2/data/baby-names-us-10k.csv.gz'
>>> df = pd.read_csv(URL) # 如果你已经从 GitLab 克隆了 nlpia2 源码,可以直接加载较小的 baby-names-us-10k.csv.gz
>>> df = df.sample(10_000)    # 在接下来的示例中，我们只需要出生证明数据的一个小样本
>>> df.shape
(10000, 6)
```

```

 该数据跨越 100 多年的美国出生证记录，但只包含婴儿的**名字**：

 | region | sex | year | name     | count | freq |
 |-------|-----|------|----------|-------|------|
 | 6139665 | WV | F | 1987 | Brittani | 10 | 0.000003 |
 | 2565339 | MD | F | 1954 | Ida      | 18 | 0.000005 |
 | 22297   | AK | M | 1988 | Maxwell  | 5 | 0.000001 |
 | … | … | … | … | … | … |
 | 44757894 | OK | F | 1950 | Leah     | 9 | 0.000003 |
 | 5744351  | VA | F | 2007 | Carley   | 11 | 0.000003 |
 | 5583882  | TX | M | 2019 | Kartier  | 10 | 0.000003 |
```

可以先忽略 `region`（地区）和 `birth year`（出生年份）字段。仅用名字这一自然语言特征，就可以在可接受的精度范围内预测性别。目标变量是已登记的性别 `sex`（M 或 F）。本数

据集中只有 male 和 female 两种性别类别。

不妨动手探索数据集，看看父母给孩子取名时，你的直觉究竟有多准。机器学习和 NLP 是打破刻板印象与误解的绝佳方式：

```
>>> df.groupby(['name', 'sex'])['count'].sum()[('Timothy',)]
sex
F       5
M    3538
```

这正是 NLP 和数据科学的乐趣所在：它能让我们跳出有限的主观视角，看到更广阔的世界。你或许从未遇到过名叫 Timothy 的女性，但在美国，至少 0.1 %的 Timothy 在出生证上登记为 female。

为了加快模型训练速度，如果你不打算让模型预测 region 和 year 的影响，可以把这些维度聚合（combine）起来。可借助 pandas 的 `DataFrame.groupby()` 实现：

```
>>> df = df.set_index(['name', 'sex'])
>>> groups  = df.groupby(['name', 'sex'])
>>> counts  = groups['count'].sum()
>>> counts
name    sex
Aaden   M      51
Aahana  F      26
Aahil   M       5
..      ..    ...
Zvi     M       5
Zya     F       8
Zylah   F       5
```

由于把数值列 `count` 聚合后， `counts` 对象成为 pandas **Series**（而非 `DataFrame`）。这是因为我们在 name 和 sex 上建立了多层索引。

现在，这个数据集已是训练逻辑回归的高效样本集合；若只想预测数据库中已知名字的性别，用最大 count （最常用性别）即可。但本书讨论 NLP 和自然语言理解（NLU），希望模型能**理解**名字文本本身，并能处理数据库中从未出现的奇怪名字——例如 *Carlana*（Carl + Ana 的混成词）或独一无二的 *Cason*。训练/测试集中未出现的例子称为 **out of distribution**。现实世界总会遇到新词新句；模型若能外推到这些 OOD 样本，就是具备 **generalization** 能力。

那么，如何对单个单词（名字）分词，使模型能泛化到完全原创、从未见过的名字？一种做法是使用字符 *n-gram*：把名字拆成字符片段并统计其频率。可用 `TfidfVectorizer` 按字符和字符 *n-gram* 计数。3-gram 通常是基于 TF-IDF 的信息检索与 NLU 算法的最佳折中；例如业界数据

库 PostgreSQL 的全文索引默认使用字符 3-gram。后续章节还会用到 word-piece 或 sentence-piece 分词，它们能自动挑选最佳字符序列作 token。

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> vectorizer = TfidfVectorizer(
...     use_idf=False,             # 让向量化器不要用逆文档频率归一化
...     analyzer='char',           # 以字符为单位
...     ngram_range=(1, 3)         # PostgreSQL 等全文检索系统内部使用的“三元索引”即由生成 1-、2-、3-gram
... )
>>> vectorizer
TfidfVectorizer(analyzer='char', ngram_range=(1, 3), use_idf=False)
# use_idf=False 防止向量化器对每一行再除以逆文档频率
```

是否应该像文档频率那样对 token 计数做归一化？在名字的 TF-IDF 向量中，我们希望使用出生证数量（`count`）作为 **document frequency**，从而让向量表示反映名字在语料外的真实普遍度。

现在你已按 `name` 和 `sex` 索引了 `names` 序列，跨 state 和 year 聚合后，Series 行数比原始 DataFrame 少得多。在计算字符 *n-gram* TF-IDF 权重前，可以“去重”名字，避免名字重复造成的计数偏移。

别忘了记录出生证明的数量——这里把它作为文档频率：

```
df = pd.DataFrame([list(tup) for tup in counts.index.values],
                  columns=['name', 'sex'])
df['count'] = counts.values
df
```

```
[4238 rows x 3 columns]
```

你已将 10 000 条 *name-sex* 记录聚合成 4 238 条唯一的 *name-sex* 组合。接下来，将数据拆分为训练集与测试集：

```
df['istrain'] = np.random.rand(len(df)) < .9
df
```

为了避免不小心把某些名字的性别标签搞反，重新创建 *name, sex* 的多重索引（multi-index）：

```
df.index = pd.MultiIndex.from_tuples(
    zip(df['name'], df['sex']), names=['name_', 'sex_'])
df
```

如前所示，该数据集中许多名字对应冲突标签：现实里，同一个名字可同时用于男性和女性（以及其他人类性别类别）。像所有机器学习分类问题一样，数学把这视作回归问题——模型实际上预测的是连续值，而非离散二元类别。线性代数和真实世界只处理实数；在机器学习中，一切二分法都是假的。[19] 机器不会将词语和概念视为硬边界的类别，你也不应如此：

```
df_most_common = {}                                    # 使用字典逐步构建 Series 的最快方法
for name, group in df.groupby('name'):
    row_dict = group.iloc[group['count'].argmax()].to_dict()   # 如果同名出现两条不同性别记录，取计数更大的那条
    df_most_common[(name, row_dict['sex'])] = row_dict
df_most_common = pd.DataFrame(df_most_common).T   # 字典嵌套字典生成的 DataFrame 只有一行，转置成列
```

由于存在重复项，可按 `~istrain` 创建测试集标志：

```
df_most_common['istest'] = ~df_most_common['istrain'].astype(bool)
df_most_common
```

```
[4025 rows x 5 columns]
```

现在，将 `istest` 与 `istrain` 标志复制到原 DataFrame，并为训练集与测试集中的 NaN 填补 False：

```
df['istest']  = df_most_common['istest']
df['istest']  = df['istest'].fillna(False)
df['istrain'] = ~df['istest']
istrain = df['istrain']

df['istrain'].sum() / len(df)   # 约 91% 用于训练
df['istest'].sum()  / len(df)   # 约  9% 用于测试
```

现在，可用训练集来拟合 `TfidfVectorizer`，避免重复名字扭曲 n-gram 计数：

```
unique_names = df['name'][istrain].unique()
vectorizer.fit(unique_names)
```

```
vecs = vectorizer.transform(df['name'])
vecs
```

```
<4238x2855 sparse matrix of type '<class 'numpy.float64'>'
        with 59599 stored elements in Compressed Sparse Row format>
```

处理稀疏数据结构时需小心：若将其 `.todense()` 转为普通稠密数组，可能耗尽计算机内存。不过此稀疏矩阵仅含约 1 700 万元素，大多数笔记本都能应付。可用 `toarray()` 将稀疏矩阵转换为 DataFrame，并为行列添加易读标签：

```
vecs = pd.DataFrame(vecs.toarray())
vecs.columns = vectorizer.get_feature_names_out()
vecs.index = df.index
vecs.iloc[:, :7]
```

注意列标签（字符 n-gram）全部小写——看起来 `TfidfVectorizer` 自动将字符转为小写。大写信息可能对模型有益，因此重新向量化名字并禁用小写折叠：

```
vectorizer = TfidfVectorizer(analyzer='char',
                             ngram_range=(1, 3),
                             use_idf=False,
                             lowercase=False)
vectorizer = vectorizer.fit(unique_names)
vecs = vectorizer.transform(df['name'])
vecs = pd.DataFrame(vecs.toarray())
vecs.columns = vectorizer.get_feature_names_out()
vecs.index = df.index
vecs.iloc[:, :5]
```

这样就好多了！这些 1-、2-、3-gram 字符特征应足以帮助神经网络在该出生证数据库中推测名字对应的性别。

选择神经网络框架（Choosing a neural network framework）

逻辑回归是处理高维特征向量（例如 TF-IDF 向量）的完美机器学习模型。若要把逻辑回归“变成”一个神经元，你只需把它与其他神经元连接起来即可。你需要一个神经元能够学习预测其他神经元的输出，并且要把学习分散开，避免单个神经元包揽全部工作。每当神经网络得到一条带有正确答案的数据样本时，它就能计算出自身错得有多离谱——即损失或误差。然而，当不仅仅一个神经元共同参与预测时，每个神经元都必须知道该把自身的权重改动多少，才能让整体输出更接近正确答案。为此，你需要知道每个权重对输出有多大影响——也就是误差相对于权重的梯度（斜率）。计算梯度并通知所有神经元如何调整权重以减小损失的过程称为**反向传播（backpropagation）**或 backprop。

深度学习框架会自动为你处理这些细节。撰写本书第一版时，TensorFlow[20] 及其易于使用的人体工程版 Keras 是构建机器学习库和应用的最流行工具。然而自那以后，人们用 TensorFlow 创建新模型的热情持续下降[21]，而另一个框架 **PyTorch** 的使用量稳步上升。虽然 TensorFlow 仍被许多公司采用（因其适合大规模部署并支持各类硬件），从业者却越来越倾向于 PyTorch，因其灵

活、直观且“更像 Python”。TensorFlow 生态系统的式微与 PyTorch 人气的迅速增长正是本书推出第二版时的主要原因。那么，PyTorch 究竟好在哪？

维基百科提供了一份公正而详细的深度学习框架比较表；借助 pandas，你可以直接把这张表从网页加载为 DataFrame：

```
>>> import pandas as pd
>>> import re
>>> dfs = pd.read_html('https://en.wikipedia.org/wiki/'
...                    + 'Comparison_of_deep_learning_software')
>>> tabl = dfs[0]
```

下面用一点 NLP 技巧，对维基百科列出的优缺点给各框架算一个“总分”。当你需要将半结构化文本数据转成 NLP 流水线用的数据时，这段代码很实用：

```
>>> bincols = list(tabl.loc[:, 'OpenMP support':].columns)
>>> bincols += ['Open source', 'Platform', 'Interface']
>>> dfd = {}
>>> for i, row in tabl.iterrows():
...     rowd = row.fillna('No').to_dict()
...     for c in bincols:
...         text = str(rowd[c]).strip().lower()
...         tokens = re.split(r'\W+', text)
...         tokens += ['*']                          # 占位符，方便匹配
...         rowd[c] = 0
...     # “便携性”评分关键词及对应权重
...     for kw, score in zip(
...         'yes via roadmap no linux android python \\*'.split(),
...         [1, .9, .2, 0, 2, 2, .1]):
...         if kw in tokens:
...             rowd[c] = score
...             break
...     dfd[i] = rowd
```

维基百科表格清洗完毕后，就可以为每个深度学习框架计算类似“综合得分”了：

```
>>> tabl = pd.DataFrame(dfd).T
>>> scores = tabl[bincols].T.sum()                # 便携性评分包含了“仍在积极开发”“开源”
>>> tabl['Portability'] = scores                  # “支持 Linux”“拥有 Python API”等因素
>>> tabl = tabl.sort_values('Portability', ascending=False)
>>> tabl = tabl.reset_index()
>>> tabl[['Software', 'Portability']][:10]
      Software  Portability
0      PyTorch         14.9
1  Apache MXNet         14.2
2   TensorFlow         13.2
3  Deeplearning4j         13.1
4        Keras         12.2
5        Caffe         11.2
6      PlaidML         11.2
7   Apache SINGA         11.2
8  Wolfram Mathematica 11.1
9      Chainer         11.0
```

PyTorch 之所以几乎拿到满分，是因为它支持 Linux、Android 以及所有主流深度学习应用。本书选择 PyTorch 的另一个原因是：它能一步步展示深度学习流程，比 Keras 的抽象层更透明、直观。

另一个值得关注的框架是 **ONNX**。严格说来，ONNX 是一个元框架，也是一个开放标准，可在不同深度学习框架之间互转模型。ONNX 还提供量化与剪枝等优化功能，让模型能在资源受限的硬件（如便携设备）上更快地推理。仅作比较，接下来我们会看看 scikit-learn 与 PyTorch 在构建神经网络模型方面的表现（见表 5.1）。

关于框架就说到这儿——你来此是要了解神经元的。PyTorch 恰好满足你的需求，而要探索的内容还有很多，足够让你熟悉这款新的 PyTorch 工具箱。

Table 5.1 Scikit-learn vs. PyTorch

Scikit-learn	PyTorch
For machine learning	For deep learning
Not GPU friendly	Made for GPUs (parallel processing)
`model.predict()`	`model.forward()`
`model.fit()`	Trained with custom `for` loop
Simple, familiar API	Flexible, powerful API

5.2.5 一个时髦的全新 PyTorch 神经元

最后，是时候使用 PyTorch 框架来构建一个神经元了。让我们通过预测本章前面清洗过的名字的性别来将所有内容付诸实践。你可以从使用 PyTorch 实现一个带有逻辑激活函数（logistic activation function）的单神经元开始——就像你在本章开头用来学习玩具示例的那一个一样：

```
>>> import torch
>>> class LogisticRegressionNN(torch.nn.Module):
...     def __init__(self, num_features, num_outputs=1):
...         super().__init__()
...         self.linear = torch.nn.Linear(num_features, num_outputs)
...     def forward(self, X):
...         return torch.sigmoid(self.linear(X))
>>> model = LogisticRegressionNN(num_features=vecs.shape[1], num_outputs=1
)
>>> model
LogisticRegressionNN(
  (linear): Linear(in_features=3663, out_features=1, bias=True)
)
```

让我们看看这里发生了什么。我们的模型是一个扩展了用来定义神经网络的 PyTorch 类 `torch.nn.Module` 的类。与所有 Python 类一样，它有一个名为 `__init__` 的构造函数（constructor）方法。构造函数是你可以定义神经网络所有属性的地方——最重要的就是模型的各个层。在我们的例子中，我们有一个极其简单的架构，只有一层且只有一个神经元，这意味着只

会有一个输出。输入（特征）的数量将等于你的 TF–IDF 向量的长度，也就是特征的维度。在我们的名字数据集中，有 3,663 个独特的 1-gram、2-gram 和 3-gram，所以对于这个单神经元网络，你将有这么多输入。

实现神经网络的第二个关键方法是 `forward()` 方法。该方法定义了输入如何通过各层传播——即前向传播（forward propagation）。至于反向传播（backpropagation）是在何处，你很快就会看到，但它不在构造函数中。我们决定为神经元使用逻辑或 sigmoid 激活函数——因此我们的 `forward()` 方法将使用 PyTorch 内置的 `sigmoid` 函数。

这就是训练模型所需的一切吗？还不是。你的神经元需要学习另外两个关键部分。其一是损失函数（loss function，或称代价函数 cost function），你在本章前面已经见过。均方误差 (MSE) (mean squared error)，在附录 D 中有详细讨论，如果这是一个回归问题，它会是一个很好的错误度量候选。但是对于这个问题，你正在做二分类，所以二元交叉熵 (binary cross-entropy) 更常用作为错误（损失）度量。下面展示了单个分类概率 p 的二元交叉熵公式：

$$\mathrm{BCE} = -\big(y \log p + (1-y)\log(1-p)\big) \qquad \text{(方程 5.2 二元交叉熵)}$$

该函数的对数性质允许它惩罚“自信地错误”的示例：当你的模型以高概率预测某个名字的性别为男性，而实际上该名字更常被标为女性时。我们可以通过利用另一个可用信息来使惩罚更贴近现实——在我们数据集中该名字对应每个性别的出现频率：

```
>>> loss_func_train = torch.nn.BCELoss(
...     weight=torch.Tensor(df[['count']][istrain].values))
>>> loss_func_test = torch.nn.BCELoss( # 损失函数具有状态，因此你需要为训练集和测试集分别创建实例
...     weight=torch.Tensor(df[['count']][~istrain].values))
>>> loss_func_train
BCELoss()
```

我们最后需要选择的是如何基于损失来调整权重——即优化算法。请记住，我们的目标是最小化损失函数。想象自己置身于损失“碗”上的斜坡，试图滑到碗底。实现滑向最低点最常见的方式称为随机梯度下降 (stochastic gradient descent, SGD)。与之前你的 Python 感知器不同，SGD 并非一次性考虑整个数据集，而是基于单个样本或一小批样本计算梯度。我们将在本章末尾深入讲解滑雪类比和 SGD 的机制。

你的优化器需要两个超参数来决定如何或多快地沿着损失斜率滑动：学习率 (learning rate) 和动量 (momentum)。学习率决定了权重对错误的响应幅度——可以把它看作你的“滑雪速度”。增大学习率可以帮助模型更快地收敛到局部最小值，但若过大，每次接近最小值时可能都会越过它。任何在 PyTorch 中使用的优化器都将具有学习率属性。

动量是梯度下降算法的一个属性，它允许当它朝正确方向移动时“加速”，当朝远离目标方向时“减速”。我们如何决定赋予这两个属性哪些值？与本书中看到的其他超参数一样，你需要优化它们以找到最适合你问题的值。现在，你可以为超参数 `momentum` 和 `lr`（学习率）选择一些任意值：

```
>>> from torch.optim import SGD
>>> hyperparams = {'momentum': 0.001, 'lr': 0.02}
>>> optimizer = SGD(
...     model.parameters(), **hyperparams)
>>> optimizer
SGD(
  Parameter Group 0
    dampening: 0
    differentiable: False
    foreach: None
    lr: 0.02
    maximize: False
    momentum: 0.001
    nesterov: False
    weight_decay: 0
)
```

将超参数存储在字典中可以让你更容易记录模型调参结果。将模型参数传递给优化器可让它知道应在每次训练步骤中更新哪些参数。

运行模型训练前的最后一步是将测试集和训练集转换为 PyTorch 模型可处理的格式：

```
>>> X = vecs.values
>>> y = (df[['sex']] == 'F').values
>>> X_train = torch.Tensor(X[istrain])
>>> X_test  = torch.Tensor(X[~istrain])
>>> Y_train = torch.Tensor(y[istrain])
>>> Y_test  = torch.Tensor(y[~istrain])
```

最后，你准备好本章最重要的部分——性别学习！让我们看看它的实现，并理解每一步发生了什么：

```
>>> from tqdm import tqdm
>>> num_epochs   = 200
>>> pbar_epochs  = tqdm(range(num_epochs), desc='Epoch:', total=num_epochs
, ascii='! =')

>>> for epoch in pbar_epochs:
...     optimizer.zero_grad()        # 步骤 1: 将累积的梯度清零
...     outputs = model(X_train)     # 步骤 2: 计算训练损失
...     loss_train = loss_func_train(outputs, Y_train)
...     loss_train.backward()        # 步骤 3: 在训练集上计算梯度
...     optimizer.step()             # 步骤 4: 使用优化器更新权重和偏置

Epoch: 100%|=========================| 200/200 [00:04<00:00, 42.84it/s]
→ 96.26it/s
```

那真快！训练这个单神经元大约 200 个 epoch，每个 epoch 有成千上万个样本，只需几秒钟。

看起来很简单，对吧？我们把它做得尽可能简单，以便你能清晰地看到各个步骤。但我们甚至不知道模型的表现如何！让我们添加一些实用函数，以便查看神经元随时间的改进情况。这称为仪表化(instrumentation)。当然，我们可以查看损失，但用更直观的指标（比如准确率）来评估模型也是很有帮助的。

首先，你需要一个函数将从模块返回的 PyTorch 张量(tensor)转换回 Numpy(NumPy) 数组：

```
>>> def make_array(x):
...     if hasattr(x, 'detach'):
...         return torch.squeeze(x).detach().numpy()
...     return x
```

你使用此实用函数来测量每次迭代时模型输出（预测）的准确率：

```
>>> def measure_binary_accuracy(y_pred, y):
...     y_pred = make_array(y_pred).round()
...     y      = make_array(y).round()
...     num_correct = (y_pred == y).sum()
...     return num_correct / len(y)
```

现在，你可以在训练过程中调用此实用函数，以查看模型在每个 epoch 上的损失和准确率进展：

```
for epoch in range(num_epochs):
    optimizer.zero_grad()       # 步骤 1：将累积的梯度清零，确保不累加到上一个 epoch 的梯度
    outputs       = model(X_train)
    loss_train    = loss_func_train(outputs, Y_train)
    loss_train.backward()       # 步骤 3：在训练集上计算梯度
    epoch_loss_train = loss_train.item()
    optimizer.step()            # 步骤 4：使用优化器更新权重和偏置
    outputs_test  = model(X_test)
    loss_test     = loss_func_test(outputs_test, Y_test).item()
    accuracy_test = measure_binary_accuracy(outputs_test, Y_test)
    if epoch % 20 == 19:
        print(f'Epoch {epoch}:'
              f' loss_train/test: {loss_train.item():.4f}/{loss_test:.4f},'
              f' accuracy_test: {accuracy_test:.4f}')   # 每隔 20 个 epoch 打印一次进度报告（记住 Python 的零起始索引）
```

```
Epoch 19:  loss_train/test: 80.1816/75.3989, accuracy_test: 0.4275
Epoch 39:  loss_train/test: 75.0748/74.4430, accuracy_test: 0.5933
Epoch 59:  loss_train/test: 71.0529/73.7784, accuracy_test: 0.6503
Epoch 79:  loss_train/test: 67.7637/73.2873, accuracy_test: 0.6839
Epoch 99:  loss_train/test: 64.9957/72.9028, accuracy_test: 0.6891
Epoch 119: loss_train/test: 62.6145/72.5862, accuracy_test: 0.6995
Epoch 139: loss_train/test: 60.5302/72.3139, accuracy_test: 0.7073
Epoch 159: loss_train/test: 58.6803/72.0716, accuracy_test: 0.7073
Epoch 179: loss_train/test: 57.0198/71.8502, accuracy_test: 0.7202
Epoch 199: loss_train/test: 55.5152/71.6437, accuracy_test: 0.7280
```

仅凭单神经元的一组权重，你的简单模型就在这个混乱、模糊的真实数据集上实现了超过 70% 的准确率。现在，你可以添加一些来自 Tangible AI 实际世界以及我们部分贡献者的更多示例：

```
>>> X = vectorizer.transform(
...     ['John', 'Greg', 'Vishvesh',
...      'Ruby', 'Carlana', 'Sarah'])
>>> model(torch.Tensor(X.todense()))
tensor([[0.0196],
        [0.1808],
        [0.3729],
        [0.4964],
        [0.8062],
        [0.8199]], grad_fn=<SigmoidBackward0>)
```

之前，我们选择用 1 表示 female（女性），用 0 表示 male（男性）。前三个示例名字 John、Greg 和 Vishvesh 是对那些在开源项目（包括本书中的代码）做出慷慨贡献的男士的致敬。

Vishvesh 这个名字在美国男性新生儿的出生证上没有 John 或 Greg 那么常见。因此，模型对 John 的字符 n-gram 中所体现的男性特征比对 Vishvesh 更有把握。

接下来的三个名字 Sarah、Carlana 和 Ruby，是写作本书时我们脑海中浮现的杰出女性的名字[22,23]。名字 Ruby 的字符 n-gram 可能带有一些男性特征，因为与之相近的名字 Rudy（常用作男孩名）仅与 Ruby 相差一个字符。有趣的是，名字 Carlana 中包含常见的男性名字 Carl，但模型却自信地将其预测为女性名。

5.3 沿误差斜坡滑行

训练神经网络(neural network)的目标是通过寻找最优参数（权重）来最小化损失函数(loss function)。在优化循环的每一步，你的算法都在寻找沿斜坡最陡峭的下坡方向。请记住，此处的误差斜坡(error slope)并非只针对数据集中某一个样本的误差，而是针对一批(batch)数据中所有点的误差均值在减小。将此问题的这一面可视化，有助于你在调整网络权重时建立心智模型。

你可能熟悉均方根误差(RMSE, root mean squared error)，它是回归问题中最常用的代价函数(cost function)。如果你想象将误差作为可能权重的函数绘制出来，给定特定输入和特定期望输出，总有一个点使该函数最接近零；那就是你的最小值(minimum)——模型误差最小的点。

此最小值对应于使给定训练样本输出最优的一组权重。你常会看到它被表示为一个三维的“碗”，其中两个轴代表二维权重向量，第三个轴代表误差（参见图 5.6）。这种描述是对更高维空间（权重超过两个的情况）的巨大简化，但概念相同。

类似地，你可以将误差曲面(error surface)绘制为所有可能权重在整个训练集输入上的聚合误差函数。但你需要对误差函数进行一些调整，需要一个表示给定权重集合在所有输入上的聚合误差的指标。对于本例，你将使用均方误差(mean squared error)作为 z 轴。你将找到误差曲面上这样一个位置，其坐标对应的权重向量能最小化预测与训练集中分类标签之间的平均误差。那组权重将使你的模型尽可能拟合整个训练集。

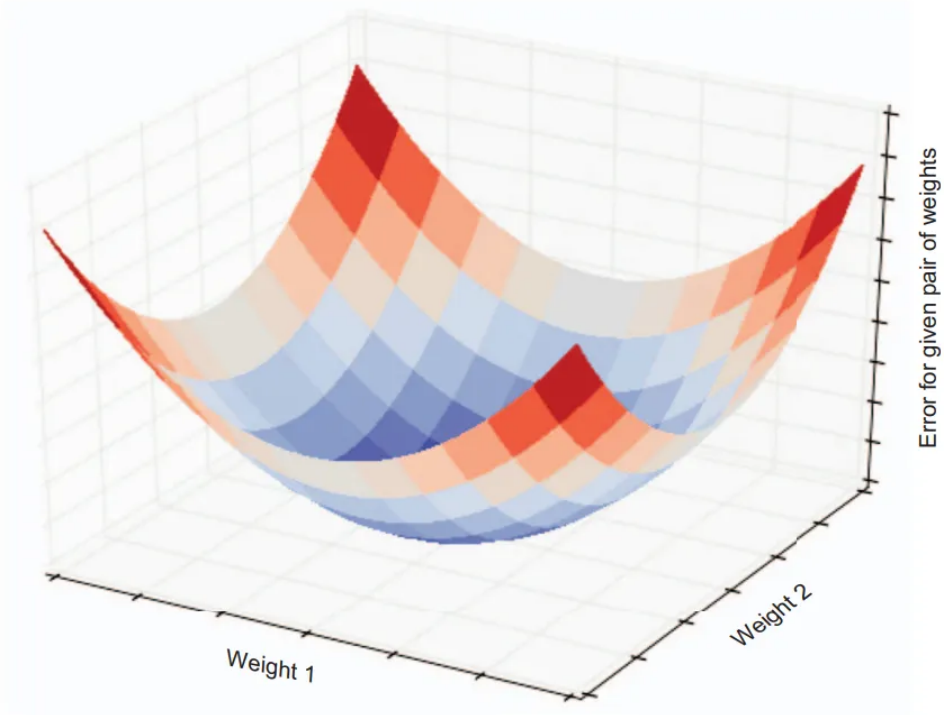

图 5.6 凸错误曲线

5.3.1 离开椅子，踏上斜坡：梯度下降(gradient descent)与局部极小值

每个 epoch 中，算法都在执行梯度下降，以最小化误差。每次你沿某个方向调整权重，希望下次误差能够降低。一个凸的误差曲面(convex error surface)对模型训练大有裨益：站在“滑雪坡”上，环顾四周，找到下坡方向，然后沿着它前进！

但你并不总是足够幸运地拥有如此形状平滑的“碗”；它可能四处散布着坑洞。这种情况被称为非凸误差曲线（nonconvex error curve）。正如滑雪一样，如果这些坑足够大，它们会将你吸入坑中，导致你无法到达斜坡底部。

同样，图示展示的是二维输入的权重，但即使输入是十维、五十维或一千维，其概念也是相同的。在那些高维空间中，可视化就不再有意义了，因此你需要信任数学。一旦开始使用神经网络，对误差曲面的可视化就变得不那么重要。你只需从观察（或绘制）误差或相关指标随训练时间的变化情况中获取相同的信息，看它是否朝着零值趋势即可。这些三维表示有助于你为该过程构建心智模型。

但如果误差空间是非凸的呢？那些坑洞难道不是问题吗？确实如此。取决于你从何处随机初始化权重，你可能会陷入截然不同的局部极小值（local minimum），而训练因此停止，因为没有其他路径可使你从该局部极小值继续下行（见图 5.7）。

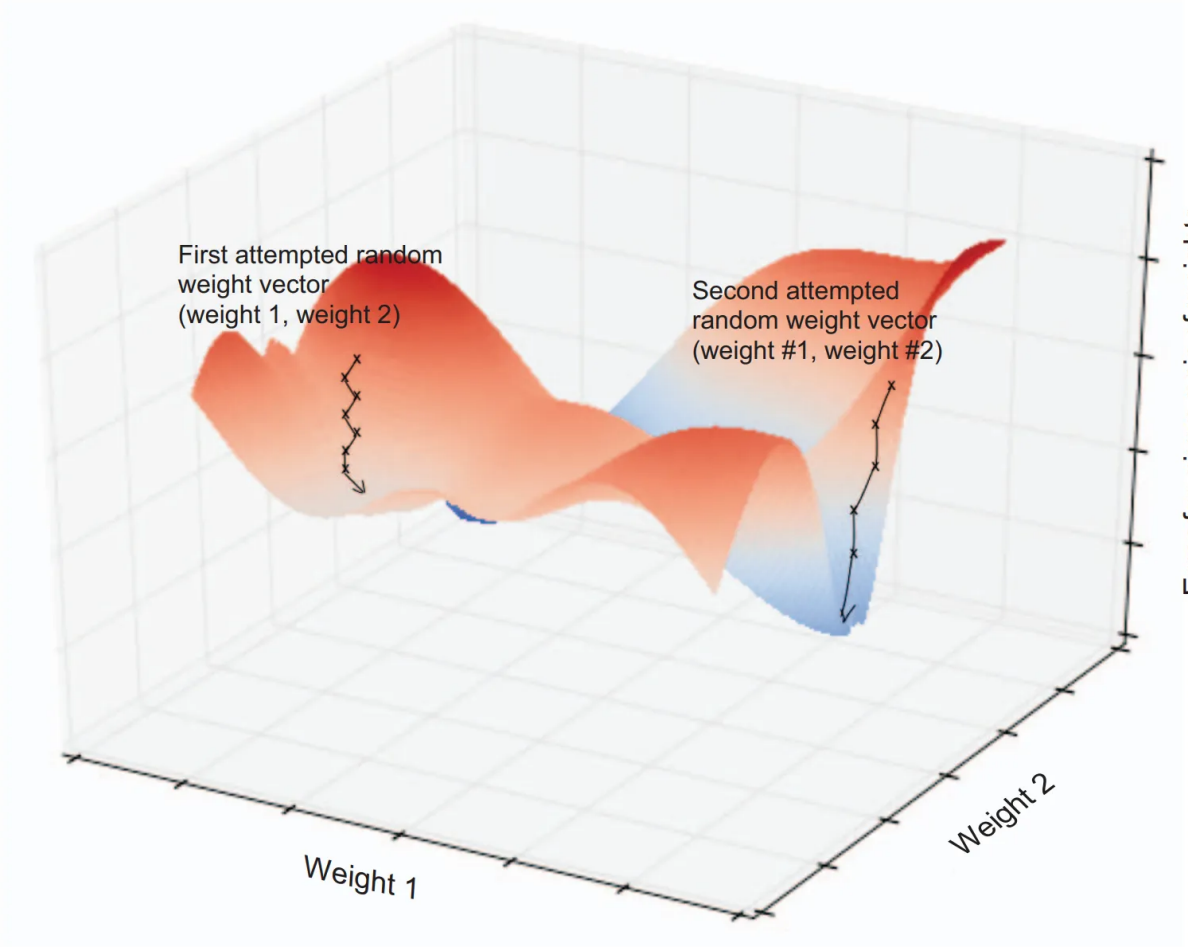

图 5.7 非凸误差曲线

随着进入更高维空间，局部极小值也会随之出现。维度越多，越难区分算法正在搜索的全局最小值和那些局部极小值。

5.3.2 通过“摇晃”来改进：随机梯度下降

到目前为止，你一直在聚合所有训练样本的误差，并以最快速度沿最陡路径滑下斜坡。但一次性在整个训练集上训练（一步一个样本）有些目光短浅——这就像选择雪地公园中下坡的跳台并忽视所有的颠簸。有时，一个恰当的滑雪跳台能帮助你跳过一些崎岖地形。

如果你尝试一次性在整个数据集上训练，可能会耗尽 RAM，从而使训练进入交换状态——在 RAM 和更慢的持久化磁盘存储之间来回交换数据。这个静态的单一误差曲面可能具有陷阱。因为你是从随机起点（初始模型权重）开始的，你可能会盲目地沿斜坡滑入某个局部极小值（凹坑、坑洞或洞穴）。你可能不知道对于权重而言存在更好的选项，并且此时误差曲面是静态的。一旦你到达误差曲面的某个局部极小值，就没有下行斜坡帮助你的模型在山上来回滑行了。

因此，为了“摇晃”这一过程，你需要在训练过程中添加一些随机化，并定期打乱模型学习样本的顺序。通常，你在每次遍历训练数据集后重新洗牌训练样本的顺序。打乱数据会改变模型计算每个

样本预测误差的顺序，从而改变其在搜索全局最小值（即该数据集的最小模型误差）路径上的走向。这种洗牌就是随机梯度下降（stochastic gradient descent）中的“随机”（stochastic）部分。

在梯度下降的“梯度”估计方面仍有提升空间。你可以为优化器加入一点“谦逊”（humility），使其不至于过于自信并盲目地将每一次新猜测都追随到底以寻找全局最小值。实际上，你所在位置的滑雪斜坡很少会指向山底的滑雪小屋（ski lodge）——全然笔直。因此，你的模型应仅沿下坡方向（梯度）前进一小段距离，而非一次性走到底。如此，每个样本的梯度都不会让模型偏离太远，也不会迷失方向。你可以调整 SGD 优化器的学习率（learning rate）超参数，以控制模型对每个样本梯度的“自信”程度。

另一种训练方式是批量学习（batch learning）。一个批次（batch）是训练数据集的一个子集——例如总数据集的 0.1%、1%、10% 或 20%。每个批次都会产生一个新的误差曲面，以便你在“滑行”搜索未知的全局误差曲面最小值时进行实验。你的训练数据仅是现实世界中可能遇到示例的一个样本，因此你的模型不应假设“全局”真实误差曲面的形状与任何部分训练数据的误差曲面相同。

这也引出了大多数 NLP 问题的最佳策略：小批量学习（mini-batch learning）[24]。Geoffrey Hinton 发现，批量大小在 16 到 64 个样本时，对于大多数神经网络训练问题最为理想[25]。这是在 SGD 的“摇晃性”与你希望沿正确方向向全局最小值取得显著进展之间的平衡。随着你朝着不断变化的局部极小值前进，在合适的数据和超参数下，你可以更容易地向全局最小值“漫步”（bumble）。小批量学习是全批量学习（full batch learning）和单样本训练（individual example training）之间的折中，它兼具随机学习（stochastic learning，随机漫步）和梯度下降学习（gradient descent learning，即沿斜坡直奔）两者的优势。

尽管反向传播（backpropagation）的细节很有趣，[26]它们并不简单，我们在此不再赘述。训练模型的一个良好心智图景是，将你的问题的误差曲面想象成某个外星星球的未知地形。优化器只能查看脚下地面的斜率，并利用这些信息向下迈出几步，然后再次检查斜率（梯度）。以这种方式探索“星球”可能需要很长时间，但优秀的优化算法会帮助你的神经网络记住“地图”上的所有优良位置，并利用它们在地图上猜测下一处要探索的位置，以寻找全局最小值。在地球上，该星球表面的最低点是南极丹曼冰川（Denman Glacier）下方的峡谷底部——低于海平面约 3.5 公里[27]。良好的小批量学习策略将帮助你找到滑雪坡或冰川向全局最小值（这幅景象若你恐高会令人不适）最陡的下行路径。希望不久之后，你就会在山脚下的滑雪小屋火堆旁，或丹曼冰川冰洞下的篝火旁与人相聚。

看看能否为你在本章创建的感知器（perceptron）添加额外层，并检验随着网络复杂度增加，结果是否有所改善。更大并非总是更好，尤其对于小型问题而言。

5.4 自测

1. Rosenblatt 的人工神经元无法解决的简单 AI 逻辑“问题”是什么?
2. 对 Rosenblatt 的结构进行什么小改动“修复”了感知器并结束了第一次“AI 寒冬”?
3. PyTorch 模型的 `forward()` 函数在 scikit-learn 模型中相当于哪个部分?
4. 如果你按年份和地区聚合姓名进行性别预测的 LogisticRegression 模型测试集能达到怎样的准确率? 别忘了对测试集进行分层抽样以避免“作弊”。

本章小结

- 最小化代价函数（cost function）是机器逐步学习更多词语信息的方式。
- 反向传播算法（backpropagation algorithm）是网络学习的手段。
- 权重对模型误差的贡献量与该权重需要更新的程度直接相关。
- 神经网络本质上是优化引擎（optimization engines）。
- 在训练过程中，通过监控误差的逐步减少来警惕陷阱（局部极小值）。

本章注释

1. 查看 Dario Amodei 和 Danny Hernandez 的分析[1]：https://openai.com/blog/ai-and-compute/
2. 参见第 3 章中的词形还原（lemmatizing）FAQ 聊天机器人示例，该示例在“过拟合”（overfitting）问题上失败。
3. “Julie Beth Lovins”，维基百科[3]（https://en.wikipedia.org/wiki/Julie_Beth_Lovins）。
4. https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html
5. Stuart Russell 的《Human Compatible AI》一书阐明了人工智能和通用人工智能（AGI）的风险与前景，并包括一些深刻的自然语言处理（Natural Language Processing, NLP）示例[5]（https://people.eecs.berkeley.edu/~russell/hc.html）。
6. 可以通过阅读 Robin Jia 的博士论文《Building Robust NLP Systems》了解如何衡量模型的鲁棒性并加以改进[6]（https://robinjia.GitHub.io/assets/pdf/robinjia_thesis.pdf）。
7. Peter Woit 著，《Not Even Wrong: The Failure of String Theory and the Search for Unity in Physical Law》[7]（Basic Books, 2007）。

8. 参见 Lex Fridman 与 Peter Woit 的访谈[8]（https://lexfridman.com/peter-woit/）。
9. Frank Rosenblatt 著，《The Perceptron: A Perceiving and Recognizing Automaton》，报告 85-460-1，康奈尔航空实验室[9]。
10. “Universal Approximation Theorem”，维基百科[10]（https://en.wikipedia.org/wiki/Universal_approximation_theorem）。
11. 逻辑激活函数（logistic activation function）可用于将线性回归（linear regression）转为逻辑回归（logistic regression）[11]（https://scikit-learn.org/stable/auto_examples/linear_model/plot_logistic.html）。
12. https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
13. 若想了解其原理，Richard McElreath 和 Robert Boyd 合著的《Mathematical Models of Social Evolution: A Guide for the Perplexed》一书很有趣[13]（University of Chicago Press, 2007）。
14. “Sex (Assigned at Birth)”，HealthIT[14]（https://www.healthit.gov/isa/uscdi-data/sex-assigned-birth）。
15. T. S. Eliot 的诗句“When I am pinned and wriggling on the wall”，选自《The Love Song of J. Alfred Prufrock》[15]（https://www.poetryfoundation.org/poetrymagazine/poems/44212/the-love-song-of-j-alfred-prufrock）。
16. “Coreference Resolution”概述，斯坦福 NLP 小组[16]（https://nlp.stanford.edu/projects/coref.shtml）。
17. 参见一位为求职而更改姓名的女性示例[17]（https://archive.is/FEvWz）。
18. William Gibson 著，《The Peripheral》[18]，维基百科（https://en.wikipedia.org/wiki/The_Peripheral）。
19. “False Dilemma”，维基百科[19]（https://en.wikipedia.org/wiki/False_dilemma）。
20. https://www.tensorflow.org/
21. https://archive.is/xLywp
22. “Sarah E. Goode”，维基百科[22]（https://en.wikipedia.org/wiki/Sarah_E._Goode）。
23. “Ruby Bridges”，维基百科[23]（https://en.wikipedia.org/wiki/Ruby_Bridges）。
24. Fischetti 等著，《Faster SGD Training by Minibatch Persistency》[24]（https://arxiv.org/pdf/1806.07353.pdf）。
25. Geoffrey Hinton 著，《Neural Networks for Machine Learning: Overview of Mini-Batch Gradient Descent》[25]

（https://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf）。

26. “Backpropagation”，维基百科[26]（https://en.wikipedia.org/wiki/Backpropagation）。
27. 维基百科列出了低于海平面的地点[27]
（https://en.wikipedia.org/wiki/List_of_places_on_land_with_elevations_below_sea_level）。