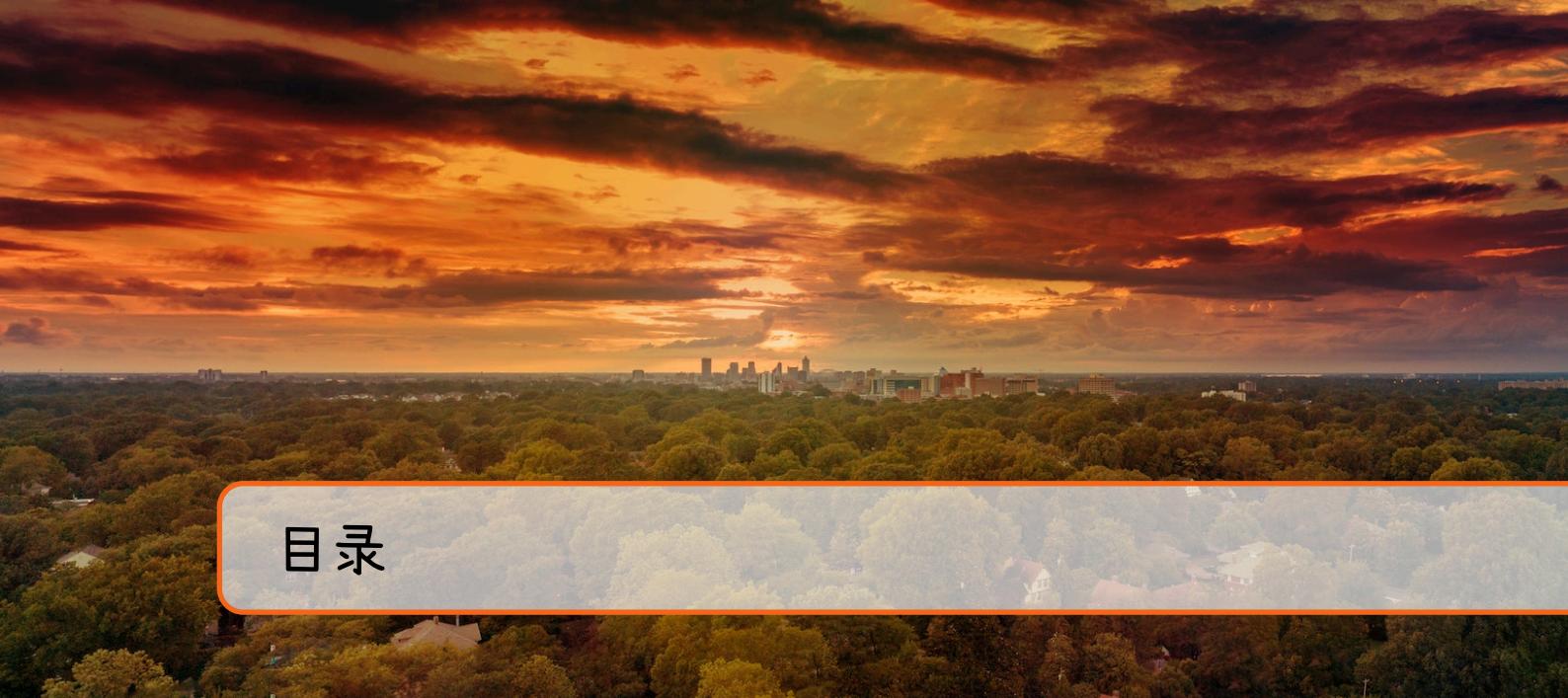


深度学习

理论与实践

左元



目录

I

深度学习

1	引言	15
1.1	监督学习	16
1.1.1	回归和分类问题	17
1.1.2	输入	18
1.1.3	机器学习模型	18
1.1.4	深度神经网络	19
1.1.5	结构化输出	19
1.2	无监督学习	21
1.2.1	生成模型	21
1.2.2	隐变量 (latent variables)	23
1.2.3	联系监督学习和无监督学习	24
1.3	强化学习	24
2	一元线性回归	27
2.1	玩具数据集	27
2.2	线性回归的理论知识	28
2.3	线性回归的实现	29
2.4	梯度下降法为什么可以找到损失函数的最小值呢？	32
2.5	如何求导？(反向传播算法)	33
2.5.1	数值微分的实现	34
2.5.2	反向传播算法	36
2.6	实现一个微型的自动微分引擎	36
3	分类问题：以手写数字识别为例	37
3.1	准备训练数据集	38
4	卷积神经网络：将手写数字识别准确率拉满！	41

5	损失函数：均方误差损失和交叉熵损失的由来	43
6	神经网络的学习：使用梯度下降法使损失函数最小化	45
7	PyTorch 简介	47
7.1	什么是 PyTorch	47
7.1.1	PyTorch 的三大核心组件	47
7.1.2	定义深度学习	48
7.1.3	安装 PyTorch	50
7.2	理解张量	51
7.2.1	标量、向量、矩阵和张量	51
7.2.2	张量数据类型	52
7.2.3	常见的 PyTorch 张量操作	52
7.3	将模型视为计算图	54
7.4	轻松实现自动微分	55
7.5	实现多层神经网络	59
7.6	设置高效的数据加载器	62
7.7	典型的训练循环	66
7.8	保存和加载模型	69
7.9	使用 GPU 优化训练性能	69
7.9.1	在 GPU 设备上运行 PyTorch	70
7.9.2	单个 GPU 训练	71
7.9.3	使用多个 GPU 训练	72
7.10	小结	73
8	自然语言处理：从零实现大语言模型	75
9	数学基础	77
9.1	矩阵微积分	77
9.1.1	引言	77
9.2	概率论	78
10	监督学习	79
10.1	监督学习概述	79
10.2	线性回归示例	80
10.2.1	一维 (1D) 线性回归模型	80
10.2.2	损失	80
10.2.3	训练	82
10.2.4	测试	83
10.2.5	最小二乘法	83
10.3	多项式拟合示例	87
10.3.1	合成数据	87
10.3.2	线性模型	88
10.3.3	误差函数	88
10.3.4	模型复杂度	89

11	梯度下降法	91
11.1	简介	91
11.2	对函数的要求	91
11.3	梯度	94
11.4	梯度下降法	95
11.5	示例 1——二次函数	96
11.6	示例 2——包含鞍点的函数	97



List of Figures

1.1 机器学习是人工智能中的一个子领域，将数学模型拟合到观测数据上。机器学习大致可分为监督学习、无监督学习和强化学习。深度神经网络对这些领域的每一个都有贡献	16
1.2 回归和分类问题	17
1.3 机器学习模型。该模型代表了一系列关系，将输入（儿童的年龄）与输出（儿童的身高）关联在一起。具体关系是通过训练数据集来选择的，这些训练数据由输入/输出对（橙色点）组成。当我们训练模型时，会在可能的关系中搜索一个能很好地描述数据的关系。这里，训练后的模型是青色曲线，可用来计算任何年龄的儿童的身高	19
1.4 具有结构化输出的监督学习任务。在每个案例中，输出都有复杂的内部结构或语法。某些情况下，许多输出和输入兼容	20
1.5 针对图像的生成模型。左边两张图像是由训练有猫图片的模型生成的。这些不是真实的猫，而是概率模型的样本。右边两张图像是由训练有建筑物图像的模型生成的。	21
1.6 文本数据生成模型合成的短篇故事。该模型描述了一个概率分布，为每个输出字符串分配一个概率。从模型中抽样可以创建遵循训练数据（这里是短篇故事）统计特征但之前从未见过的字符串	22
1.7 图像修复。在原始图像（左）中，男孩被金属电缆遮挡。这些不希望出现的区域（中）被移除，生成模型合成了一个新图像（右），在这个过程中保持剩余像素不变	22
1.8 条件文本合成。给定一段初始文本（黑色部分），文本的生成模型可以通过合成“缺失”的剩余部分来可信地延续字符串。由 GPT3 生成	22
1.9 人脸的变化。人脸大约有 42 块肌肉，因此可用大约 42 个数字来描述同一个人在相同光照下的图像中的大部分变化。一般来说，图像，音乐和文本的数据集可以用较少的隐变量来描述，尽管将这些变量与特定的物理机制联系起来通常更困难。	23
1.10 隐变量。许多生成模型使用深度学习模型来描述低维“潜”变量与观察到的高维数据之间的关系。隐变量具有简单的概率分布。因此，可通过从隐变量的简单分布中采样，然后使用深度学习模型样本映射到观察数据空间来生成新的样本	23
1.11 图像插值。在每一行中，左右的图像是真实的，中间的三个图像代表由生成模型创建的一系列插值。这些插值的基础生成模型已经学会所有图像都可以由一组隐变量创建。通过找到这两个真实图像的隐变量，在它们之间插值，然后使用这些中间变量创建新图像，从而生成视觉上可信又混合了两个原始图像特征的中间结果	24
1.12 由标题“时代广场上玩滑板的泰迪熊”生成的多幅图像。由 DALL-E 2 生成	24
1.13 强化学习的策略网络。将深度神经网络融入强化学习的一种方法是使用它们定义从状态（棋盘上的位置）到动作（可能的移动）。这种映射被称为策略	25
2.1 使用的玩具数据集	28

2.2 线性回归的示例	29
2.3 梯度下降法	30
2.4 训练后的模型	32
2.5 梯度下降法示意图	33
2.6 曲线 $y = f(x)$ 和通过其两点的直线	33
2.7 比较真的导数、前向差分近似和中心差分近似	34
3.1 mnist 数据集的前 5 行, 形状为 5x785	38
3.2 第一行的图片	39
7.1 PyTorch 的三大核心组件包括作为计算基础构建块的张量库、用于模型优化的自动微分引擎以及深度学习工具函数, 这使得实现和训练深度神经网络模型更加容易	48
7.2 深度学习是机器学习的一个子类别, 专注于实现深度神经网络。机器学习是人工智能的一个子类别, 涉及从数据中学习的算法。人工智能是一个更广泛的概念, 指的是机器能够执行通常需要人类智能水平的任务	49
7.3 监督学习的预测建模工作流程包括一个训练阶段, 在该阶段中, 模型在训练数据集中带标签的示例上进行训练。训练好的模型随后可用于预测新观测数据的标签	50
7.4 不同秩的张量。这里零维对应于秩 0, 一维对应于秩 1, 二维对应于秩 2。一个由 3 个元素组成的三维向量仍然是秩为 1 的张量	51
7.5 逻辑回归的前向传播作为一个计算图。输入特征 x_1 与模型权重 w_1 相乘, 并在加上偏置后通过激活函数 σ 传递。损失是通过比较模型输出 a 与给定标签 y 来计算的	55
7.6 在计算图中计算损失梯度的最常见方法是从右向左应用链式法则, 这也称为“反向模型自动求导”或“反向传播”。我们从输出层(或损失本身)开始, 向后通过网络一直到输入层。这么做是为了计算损失相对于网络中每个参数(权重和偏置)的梯度, 从而为训练过程中如何更新这些参数提供信息	56
7.7 一个具有两个隐藏层的多层感知机。每个节点表示各自层中的一个单元。为了方便展示, 这里每层都只有几个节点	59
7.8 PyTorch 实现了 Dataset 类和 DataLoader 类。Dataset 类用于实例化定义如何加载每条数据记录的对象。DataLoader 类负责处理数据的打乱和组装成批次	63
9.1 人工神经元	77
10.1 线性回归模型。对于给定的参数 $\Phi = [\phi_0, \phi_1]^T$, 模型根据输入(x 轴)对输出(y 轴)进行预测。不同的截距 ϕ_0 和斜率 ϕ_1 的选择会改变这些预测结果(青色、橙色和灰色直线)。线性回归模型(式(10.4))定义了一组输入/输出关系(直线), 而参数决定了该组中的具体成员(特定的直线)	80
10.2 线性回归的训练数据、模型和损失。图(b)~(d)分别展示了具有不同参数的线性回归模型。根据截距和斜率参数 $\Phi = [\phi_0, \phi_1]^T$ 的选择, 模型误差(橙色虚线)可能更大或者更小。损失 \mathcal{L} 是这些误差平方的总和	81
10.3 针对图 10.2(a)的数据集的线性回归模型的损失函数	82
10.4 线性回归训练。训练目标是找到对应于最小损失的截距和斜率参数	83
10.5 一个由 $N = 10$ 个数据点组成的训练集, 以蓝色圆点显示, 其中每个数据点包含了输入变量 x 及其对应的目标变量 t 的观测值。绿色曲线显示了用来生成数据的函数 $\sin(2\pi x)$ 。我们的目标是在不知道绿色曲线的情况下, 预测新的输入变量 x 所对应的目标变量 t 的值	88
10.6 平方和误差函数的几何解释[该误差函数对应来自函数 $y(x, w)$ 的每个数据点的位移(如垂直的绿色箭头所示)平方和的一半]	89
10.7 具有不同阶数的多项式图示。多项式如红色曲线所示。这里通过最小化平方和误差函数来拟合训练数据集	89
10.8 由式 1.3 定义的均方根误差图(在训练集和独立的测试集上对 M 的各个值进行评估)	90
11.1 可微函数示例	92
11.2 不可微函数示例	92
11.3 凸函数与非凸函数示例图	93

11.4 具有鞍点的半凸 (semi-convex) 函数	94
11.5 $z = x^2 - y^2$ 的鞍点示意图	94
11.6 $f(x) = 0.5x^2 + y^2$ 示意图	95
11.7 梯度下降法步骤	96
11.8 不同学习率的对比	97
11.9 梯度下降法尝试逃离鞍点示意图	98
11.10 没有逃离鞍点	98



List of Tables

9.1 标量导数规则	78
----------------------	----

深度学习

I

1.3	强化学习	24
2	一元线性回归	27
2.1	玩具数据集	27
2.2	线性回归的理论知识	28
2.3	线性回归的实现	29
	梯度下降法为什么可以找到损失函数的最小值	
2.4	呢?	32
2.5	如何求导? (反向传播算法)	33
2.6	实现一个微型的自动微分引擎	36
3	分类问题: 以手写数字识别为例 .	37
3.1	准备训练数据集	38
	卷积神经网络: 将手写数字识别准确率拉	
4	满!	41
	损失函数: 均方误差损失和交叉熵损失的	
5	由来	43
	神经网络的学习: 使用梯度下降法使损失	
6	函数最小化	45
7	PyTorch 简介	47
7.1	什么是 PyTorch	47
7.2	理解张量	51
7.3	将模型视为计算图	54
7.4	轻松实现自动微分	55
7.5	实现多层神经网络	59
7.6	设置高效的数据加载器	62
7.7	典型的训练循环	66
7.8	保存和加载模型	69
7.9	使用 GPU 优化训练性能	69
7.10	小结	73
	自然语言处理: 从零实现大语言模型. .	
8	75	
9	数学基础	77
9.1	矩阵微积分	77
9.2	概率论	78
10	监督学习	79
10.1	监督学习概述	79
10.2	线性回归示例	80
10.3	多项式拟合示例	87
11	梯度下降法	91
11.1	简介	91
11.2	对函数的要求	91
11.3	梯度	94
11.4	梯度下降法	95
11.5	示例 1——二次函数	96
11.6	示例 2——包含鞍点的函数	97



1. 引言

在学术界，深度学习的发展历史极不寻常。一小群科学家坚持不懈地在一个看似没有前途的领域工作了 25 年，最终使得一个领域发生了技术革命并极大地影响了人类社会。研究者持续地探究学术界或者工程界中深奥且难以解决的问题，通常情况下这些问题无法得到根本性的解决。但深度学习领域是个例外，尽管广泛的怀疑仍然存在，但 **Yoshua Bengio**、**Geoff Hinton** 和 **Yann LeCun** 等人的系统性努力最终取得了成效！

🔥 深度学习历史

以 **ChatGPT** 为代表的人工神经网络的逆袭之旅，在整个科技史上也算得上跌宕起伏。它曾经在流派众多的人工智能界内部屡受歧视和打击。不止一位天才先驱以悲剧结束一生：1943 年，沃尔特·皮茨（Walter Pitts）在与沃伦·麦卡洛克（Warren McCulloch）共同提出神经网络的数学表示时才 20 岁，后来因为与导师维纳失和而脱离学术界，最终因饮酒过度于 46 岁辞世；1958 年，30 岁的弗兰克·罗森布拉特（Frank Rosenblatt）通过感知机实际实现了神经网络，而 1971 年，他在 43 岁生日那天溺水身亡；反向传播的主要提出者大卫·鲁梅尔哈特（David Rumelhart）则正值盛年（50 多岁）就罹患了罕见的不治之症，1998 年开始逐渐失智，最终在与病魔斗争十多年来离世……

一些顶级会议以及明斯基这样的学术巨人都曾毫不客气地反对甚至排斥神经网络，逼得辛顿等人不得不先后采用“关联记忆”“并行分布式处理”“卷积网络”“深度学习”等中性或者晦涩的术语为自己赢得一隅生存空间。

辛顿自己从 20 世纪 70 年代开始，坚守冷门方向几十年。从英国到美国，最后立足曾经的学术边陲加拿大，他在资金支持匮乏的情况下努力建立起一个人数不多但精英辈出的学派。

直到 2012 年，他的博士生伊尔亚·苏茨克维等在 **ImageNet** 比赛中用新方法一飞冲天，深度学习才开始成为 AI 的显学，并广泛应用于各个产业。2020 年，他又在 OpenAI 带队，通过千亿参数的 **GPT-3** 开启了大模型时代。

人工智能（**artificial intelligence**, AI）是一种旨在构建模拟人类智能行为的系统。它涵盖了多种方法，包括基于逻辑、搜索和概率推理的方法。机器学习是 AI 的一个子集，通过将数学模型拟合到观察到的数据上来学习做出决策。这一领域已经经历了爆炸性增长，导致现在机器学习几乎被误认为 AI 的同义词了。

深度神经网络是一种机器学习模型，当它拟合到数据时，被称为深度学习。目前，深度学习是最强大、最实用的机器学习模型，经常出现在日常生活中。例如，使用自然语言处理算法翻译文本、使用计算机视觉系统在互联网上搜索特定物体的图像，或者通过语音识别界面与数字助手对话，都是司空见惯的事情。所有这些应用都是由深度学习驱动的。

机器学习方法大致可以分为三个领域：监督学习、无监督学习和强化学习。目前，这三个领域的前沿方法都依赖于深度学习（图 1.1）。

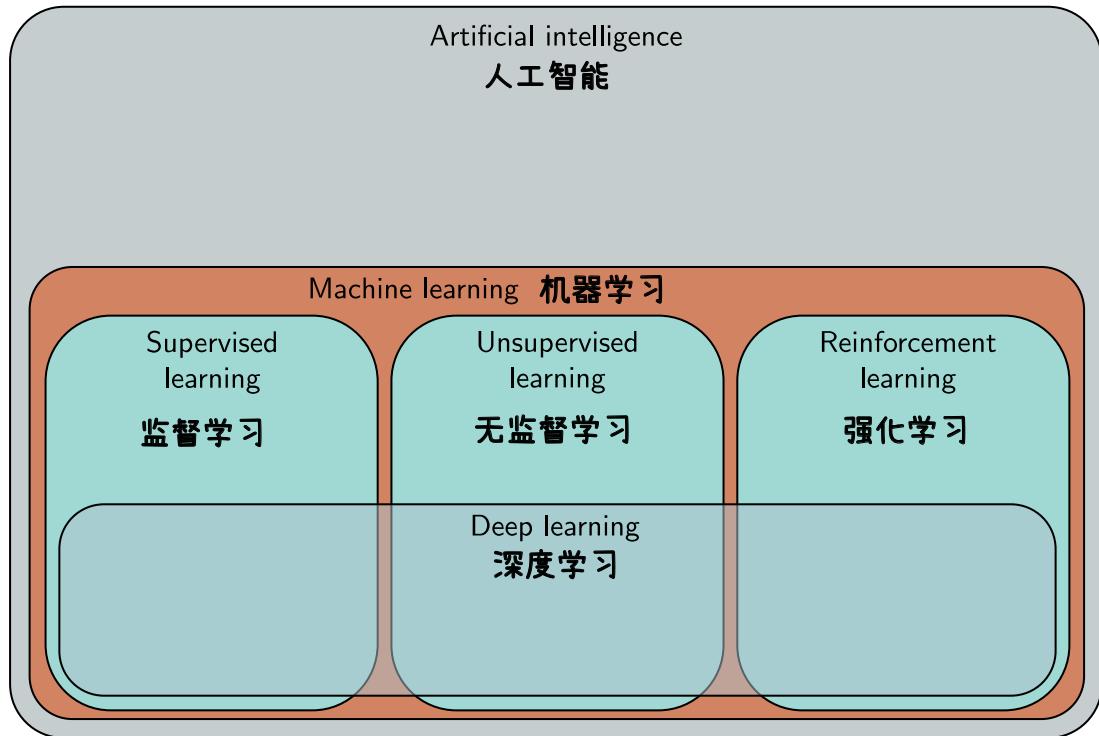


图 1.1 机器学习是人工智能中的一个子领域，将数学模型拟合到观测数据上。机器学习大致可分为监督学习、无监督学习和强化学习。深度神经网络对这些领域的每一个都有贡献

1.1 监督学习

监督学习模型定义了从输入数据到输出预测的映射。接下来将讨论输入、输出、模型本身，以及“学习”这个动作对一个模型究竟意味着什么。

1.1.1 回归和分类问题

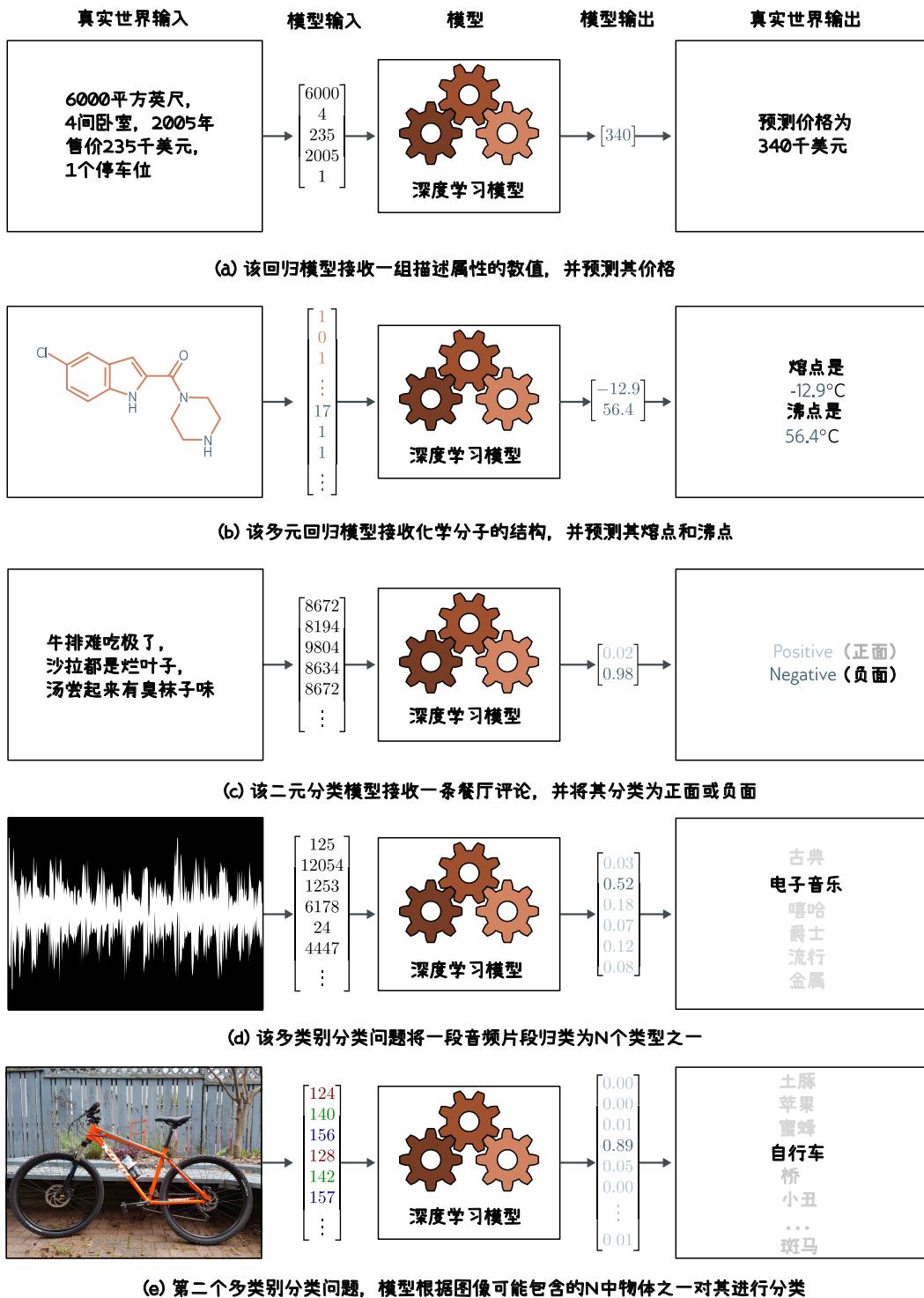


图 1.2 回归和分类问题

图 1.2 展示了几个回归和分类问题。在每个案例中，都有一个有意义的实际输入（句子、声音文件、图片等），并将其编码为一个数值向量。这个向量构成了模型的输入。模型将输入映射到一个输出向量，然后将其“转换”并返回一个有意义的实际预测。目前，我们专注于输入和输出，并将模型视为黑盒，它接收一个数值向量并返回另一个数值向量。

图 1.2(a)中的模型基于输入特征(如房屋的平方英尺数和卧室数量)预测房价。因为模型返回一个连续数值(而不是一个类别分配),所以这是回归问题。相比之下,图 1.2(b)中的模型以一个分子的化学结构为输入,并预测它的熔点和沸点。因为它预测不止一个数值,所以是二元回归问题。

图 1.2(c)中的模型接收一个包含餐厅评论的文本字符串作为输入,并预测评论是正面的还是负面的。因为模型试图将输入分配到两个类别中的一个,所以这是二元分类问题。输出向量包含输入属于每个类别的概率。图 1.2(d)和图 1.2(e)展示了多分类问题。这里,模型将输入分配到 $N > 2$ 个类别中的一个。在图 1.2(d)的案例中,输入是一个音频文件,模型预测它包含的音乐类型。在图 1.2(e)的案例中,输入是一张图片,模型预测它包含的物体。每种情况下,模型都返回一个大小为 N 的向量,其中包含 N 个类别的概率。

1.1.2 输入

图 1.2 中的输入数据的类型差异很大。在房价预测例子中,输入的是包含了表征房屋特征值的固定长度的向量。这是一组表格数据,因为这组数据没有内部结构;如果我们改变输入的顺序并构建一个新模型,我们希望模型的预测结果保持不变。

相反,在餐厅评论的例子中,输入是一段文字。输入的长度可能根据评论中的单词数量而变化,在该例中输入顺序就很重要:“我的妻子吃了鸡肉”与“鸡肉吃了我的妻子”的含义完全不同。文本必须在传递给模型之前编码成数值形式。在该例中,使用大小为 10000 的固定词汇表,并简单地将单词的索引串联起来。

对于音乐分类的例子,输入向量可能是固定大小的(如 10 秒的片段),但维度非常高(包含许多条目)。数字音频通常以 44.1kHz 的频率采样并用 16 位整数表示,因此 10 秒的片段将由 441000 个整数组成。显然,监督学习模型必须能够处理相当庞大的输入数据。在图像分类的例子中的输入(由每个像素处的 RGB 值组成)也很庞大。此外,它的结构自然是二维的;即使在输入向量中不相邻,上下相邻的两个像素也是密切相关的。

最后,考虑预测分子的熔点和沸点的模型输入。一个分子可能包含不同数量的原子。另外,这些原子还可通过不同方式进行连接。所以这种情况下,模型必须同时将分子的几何结构和组成分子的原子作为输入。

1.1.3 机器学习模型

到目前为止,我们可将机器学习模型视作黑盒,它接收输入向量并返回输出向量。但这个黑盒里面究竟是什么呢?联想一下根据年龄预测孩子身高的模型(图 1.3)。机器学习模型是一个数学函数,它描述了平均身高如何随年龄变化(图 1.3 中的青色曲线)。将年龄代入这个函数时,它就会返回身高。例如,如果年龄是 10 岁,预测身高将是 139 厘米。

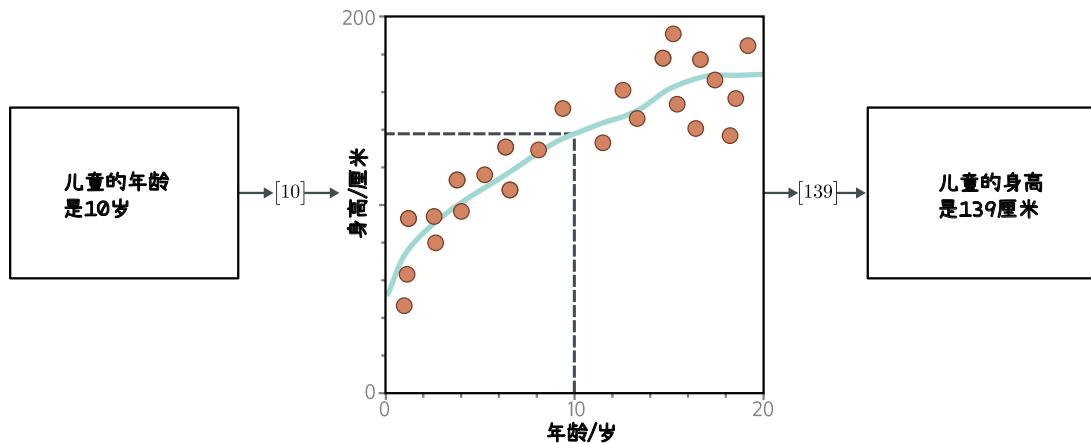


图 1.3 机器学习模型。该模型代表了一系列关系，将输入（儿童的年龄）与输出（儿童的身高）关联在一起。具体关系是通过训练数据集来选择的，这些训练数据由输入/输出对（橙色点）组成。当我们训练模型时，会在可能的关系中搜索一个能很好地描述数据的关系。这里，训练后的模型是青色曲线，可用来计算任何年龄的儿童的身高

更准确地说，该模型表示一系列将输入映射到输出的函数（即，不同的青色曲线族）。特定的函数（曲线）是使用训练数据（输入/输出对）选择的。在图 1.3 中，这些对用橙色点表示，可以看到用该模型（即青线）来描述这些数据就非常合理。当谈论训练或拟合一个模型时，我们的意思是在可能的函数（可能的青色曲线）之间搜索，来找到那个可以最准确地描述训练数据的函数。

因此，图 1.2 中的模型需要用标记的“输入/输出对”进行训练。例如，音乐分类模型需要大量音频片段，其中人类专家已经标记了每个片段的流派。这些输入/输出对在训练过程中扮演教师或监督者的角色，由此产生了“监督学习”这个术语。

1.1.4 深度神经网络

本教程将重点介绍深度神经网络，这是一种特别实用的机器学习模型，也是函数，可以表示输入和输出之间极其广泛的关系族，并可通过遍历这个关系族找到训练数据之间的关系。

深度神经网络可以处理非常大，长度可变且包含各种内部结构的输入。它们可以输出单个实数值（回归），多个数值（多元回归）或两个及更多类别的概率（分别为二元分类和多元分类）。正如我们将在下一节看到的，它们的输出也可能非常大，长度可变且包含内部结构。想象具有这些性质的函数可能十分困难，但你现在应该尝试暂时放下怀疑。

1.1.5 结构化输出

图 1.4(a)描绘了一个用于语义分割的多元分类模型。这里，输入图像的每个像素都被分配一个二元标签，指示它属于牛本身还是属于背景。图 1.4(b)展示了一个单目深度估计模型，该模型的输入是一幅街景图像，输出是每个像素的深度。这两种情况下，输出都是高维且结构化的。然而，这种结构与输入密切相关，并且可供利用；如果一个像素被标记为“牛”，那么与其具有相似 RGB 值的邻居可能也有相同的标签。

图 1.4(c) ~ (e)描绘了三组具有复杂结构的输出但与输入联系不那么密切的模型。图 1.4(c)展示的模型的输入是音频文件，输出是该文件中语音的文字转录。图 1.4(d)是翻译模型，输入是中文文本，而输出是法文文本。图 1.4(e)描述了一种极具挑战性的任务，其输入是描述性文本，而模型的输出是必须与此描述相符的图像。

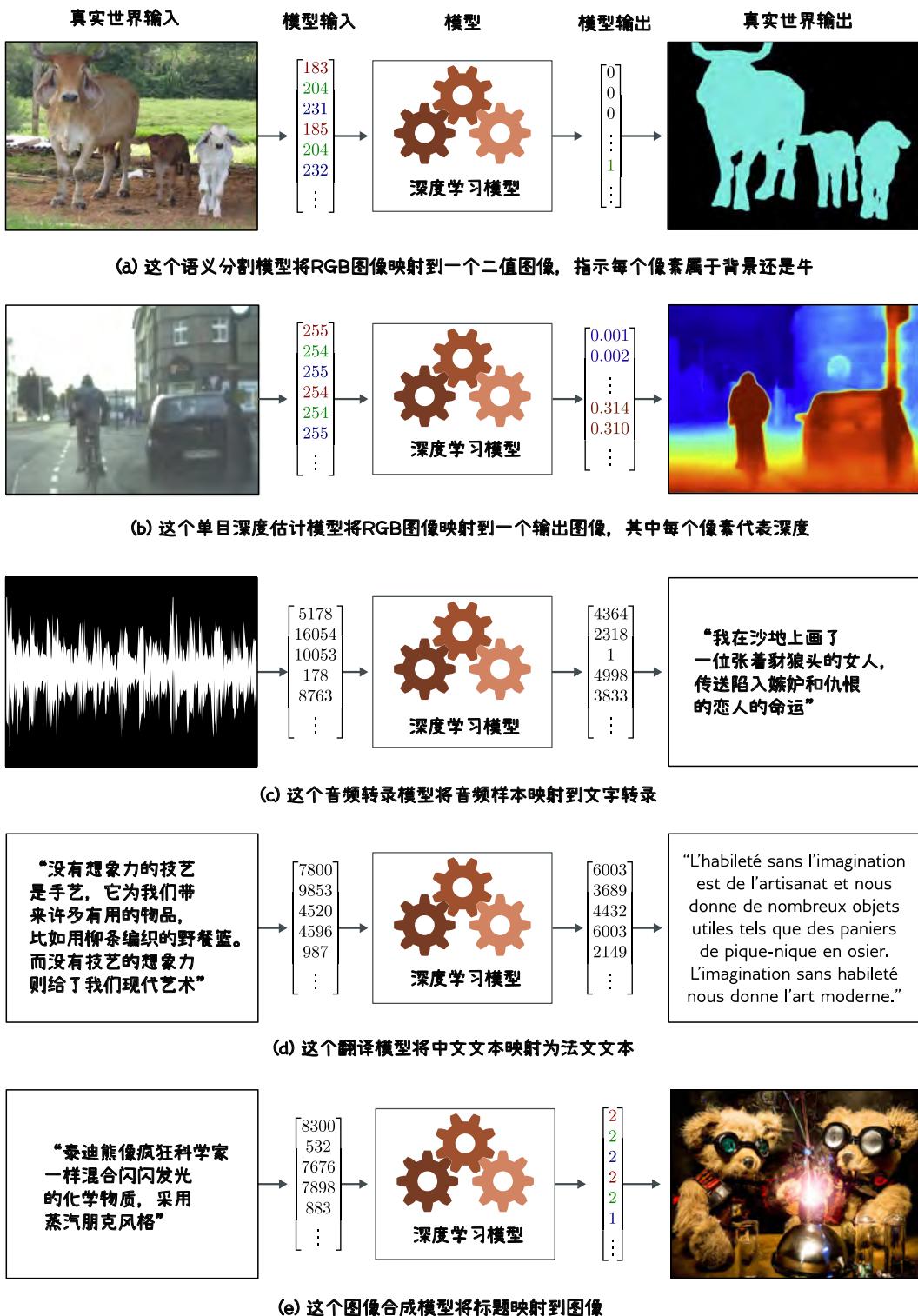


图 1.4 具有结构化输出的监督学习任务。在每个案例中，输出都有复杂的内部结构或语法。某些情况下，许多输出和输入兼容

原则上，后三个任务都可在标准的监督学习框架下实现，但由于下面两个原因使它们变得更加困难。首先，输出可能确实是模糊的；从一个中文句子到法文句子有多种有效的翻译方式，任何标题都可能与多种图像相符。其次，输出包含相当多的结构；并不是所有单词或者字符都能构成有效的中文

和法文句子，也不是所有的 RGB 的组合都能构成合理的图像。除了学习映射，我们还必须遵循输出的“语法”。

幸运的是，这种“语法”可在不需要输出标签的情况下进行学习。例如，可通过学习大量文本数据的统计信息来学习如何形成有效的中文句子。这为后文中的无监督学习模型打下了基础。

1.2 无监督学习

不使用输入数据对应的输出数据标签来构建模型的过程称为无监督学习；没有输出标签意味着没有“监督”。无监督学习的目标并非是学习从输入到输出的映射，而是描述或理解输入数据的结构。就像监督学习一样，数据可能具有非常不同的特征：可能是离散的或连续的，低维的或高维的，长度固定的或可变的。

1.2.1 生成模型

本教程关注的是生成无监督模型，这类模型能合成新的数据样本，这些样本在统计学上与训练数据无法区分。一些生成模型明确描述了输入数据的概率分布，并从这个分布中采样生成新的样本。另一些模型只是学习生成新样本的机制，而不是描述它们的分布。

最先进的生成模型可以合成极其逼真但与训练样本不同的样本。它们在生成图像（图 1.5）和文本（图 1.6）方面特别成功。它们还可以在一定约束条件下合成数据，即预先设定某些输出（称为条件生成），例如图像修复（图 1.7）和文本补全（图 1.8）。事实上，现代文本生成模型如此强大，以至于它们看起来是智能的。给定一段文本然后提出一个问题，模型通常可以通过生成文档最可能的结尾部分来“填补”缺失的答案。然而，实际上，模型只是了解语言的统计信息，并不理解答案的意义。



图 1.5 针对图像的生成模型。左边两张图像是由训练有猫图片的模型生成的。这些不是真实的猫，而是概率模型的样本。右边两张图像是由训练有建筑物图像的模型生成的。

The moon had risen by the time I reached the edge of the forest, and the light that filtered through the trees was silver and cold. I shivered, though I was not cold, and quickened my pace. I had never been so far from the village before, and I was not sure what to expect. I had been walking for hours, and I was tired and hungry. I had left in such a hurry that I had not thought to pack any food, and I had not thought to bring a weapon. I was unarmed and alone in a strange place, and I did not know what I was doing.

I had been walking for so long that I had lost all sense of time, and I had no idea how far I had come. I only knew that I had to keep going. I had to find her. I was getting close. I could feel it. She was nearby, and she was in trouble. I had to find her and help her, before it was too late.

图 1.6 文本数据生成模型合成的短篇故事。该模型描述了一个概率分布，为每个输出字符串分配一个概率。从模型中抽样可以创建遵循训练数据（这里是短篇故事）统计特征但之前从未见过的字符串



图 1.7 图像修复。在原始图像（左）中，男孩被金属电缆遮挡。这些不希望出现的区域（中）被移除，生成模型合成了一个新图像（右），在这个过程中保持剩余像素不变

I was a little nervous before my first lecture at the University of Bath. It seemed like there were hundreds of students and they looked intimidating. I stepped up to the lectern and was about to speak when something bizarre happened.

Suddenly, the room was filled with a deafening noise, like a giant roar. It was so loud that I couldn't hear anything else and I had to cover my ears. I could see the students looking around, confused and frightened. Then, as quickly as it had started, the noise stopped and the room was silent again.

I stood there for a few moments, trying to make sense of what had just happened. Then I realized that the students were all staring at me, waiting for me to say something. I tried to think of something witty or clever to say, but my mind was blank. So I just said, "Well, that was strange," and then I started my lecture.

图 1.8 条件文本合成。给定一段初始文本（黑色部分），文本的生成模型可以通过合成“缺失”的剩余部分来可信地延续字符串。由 GPT3 生成

1.2.2 隐变量 (latent variables)

一些 (但并非全部) 生成模型利用了这样一个观察结果：数据的内在维度可能比观察到的变量维度总数所暗示的要少。例如，有效且有意义的英语句子数量远少于通过随机抽取单词并排列组合生成的字符串数量。同样，真实世界的图像只是通过对每个像素随机抽取 RGB 值而创建的图像中极少的一部分，这是因为图像是由物理过程产生的 (见图 1.9)。



图 1.9 人脸的变化。人脸大约有 42 块肌肉，因此可用大约 42 个数字来描述同一个人在相同光线下图像中的大部分变化。一般来说，图像、音乐和文本的数据集可以用较少的隐变量来描述，尽管将这些变量与特定的物理机制联系起来通常更困难。

这引出了一个观点，即可用更少数量的隐变量来描述每个数据样本。这里，深度学习的作用是描述这些隐变量与数据之间的映射。获得的隐变量可以有一个简单的概率分布。通过从这个分布中抽样并传递给深度学习模型，可以创建新的样本 (图 1.10)。

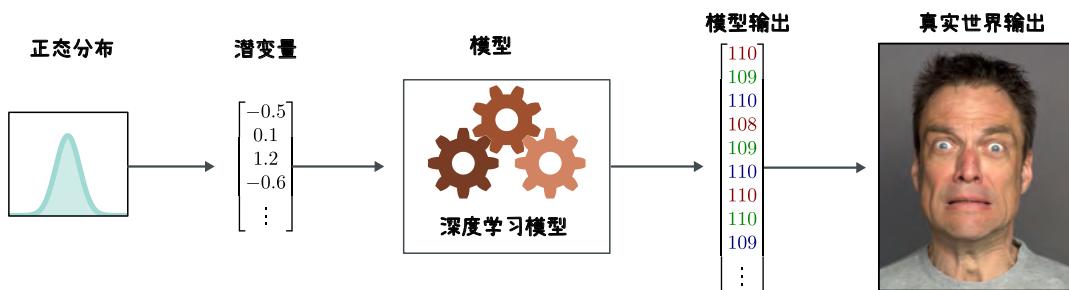


图 1.10 隐变量。许多生成模型使用深度学习模型来描述低维“潜”变量与观察到的高维数据之间的关系。隐变量具有简单的概率分布。因此，可通过从隐变量的简单分布中采样，然后使用深度学习模型样本映射到观察数据空间来生成新的样本。

这些模型开启了操纵真实数据的全新方法。例如，考虑找出支撑两个真实样本的隐变量。可通过在它们的潜在表示之间插值，并将中间位置映射回数据空间来实现这些样本之间的插值 (图 1.11)。



图 1.11 图像插值。在每一行中，左右的图像是真实的，中间的三个图像代表由生成模型创建的一系列插值。这些插值的基础生成模型已经学会所有图像都可以由一组隐变量创建。通过找到这两个真实图像的隐变量，在它们之间插值，然后使用这些中间变量创建新图像，从而生成视觉上可信又混合了两个原始图像特征的中间结果

1.2.3 联系监督学习和无监督学习

具有隐变量的生成模型也可促进监督学习模型的发展，特别是在输出具有结构时（见图 1.4）。例如，考虑学习如何预测与标题对应的图像。可以不直接将文本输入映射到图像上，而是学习解释文本的隐变量与解释图像的隐变量之间的关系。

这样做有三个优点。首先，由于输入和输出的维度较低，我们可能需要更少的文本/图像对来学习这种映射。其次，我们更可能生成看起来合理的图像；任何合理的隐变量值都应该产生像合理样本的东西。最后，如果在两组隐变量之间的映射或从隐变量到图像的映射中引入随机性，那么可生成多个都能被标题很好地描述的图像（见图 1.12）。



图 1.12 由标题“时代广场上玩滑板的泰迪熊”生成的多幅图像。由 DALL-E 2 生成

1.3 强化学习

机器学习的最后一个领域是强化学习。这一范式引入了“智能体”概念，智能体存在于一个世界中，可在每个时间步执行特定的动作。这些动作会改变系统的状态，但这种改变并不总是确定的。执行动作也会产生奖励，强化学习的目标是让“智能体”学会选择能带来高奖励的动作。

一个难点是奖励可能在动作执行后一段时间才出现，因此奖励与动作的关联并不直接。这称为时间信用分配问题。随着智能体的学习，它必须在探索和利用已有知识之间进行权衡；也许智能体已经学会了如何获得适度的奖励；它应该遵循这种策略（利用已知知识），还是应该尝试用不同的动作来检验是否有进一步提升的可能性（探索其他机会）？

两个例子

第一个例子考虑训练一个人形机器人的移动。机器人在任何时候都可执行有限数量的动作（移动各个关节），这些动作会改变机器人的世界状态（它的姿势）。我们可能会因为机器人到达障碍赛中

的检查点奖励它。要到达每个检查点，它必须执行许多动作，且当它收到奖励时，不清楚哪些动作与奖励有关，哪些是无关的。这就是时间信用分配问题的一个例子。

第二个例子是学习下棋。同样，智能体在任何时候都有一组有效的动作（移动棋子）。然而，这些动作以非确定方式改变系统状态；对于选择的任何动作，对手都可能做出许多不同的反馈。这里，可能会在吃掉棋子时设置奖励，或仅在游戏结束时获得单一奖励来设置奖励结构。在后一种情况下，时间信用分配问题是极端的；系统必须学习它所执行的大量动作，并了解哪些对成功或失败起到了关键作用。

探索-利用权衡在这两个例子中也很明显。机器人可能已经发现，它可以通过侧卧并用一条腿推的方式向前移动。这种策略会推动机器人，并带来奖励，但比最佳解决方案（双腿平衡行走）要慢得多。因此，它面临着两种选择：利用它已经知道的东西（如何笨拙地滑过地板）或探索更多的动作空间（可能加快移动速度）。同样，在国际象棋的例子中，智能体可能学会了一系列合理的开局。它应该利用这些知识还是探索不同的开局顺序？

或许，深度学习如何融入强化学习框架并不明显。有几种可能的方法，但一种技术是使用深度网络构建从观察到的世界状态到动作的映射。这称为策略网络。在机器人例子中，策略网络将学习从其传感器测量数据到关节运动的映射。在国际象棋例子中，策略网络将学习从棋盘的当前状态到移动选择的映射（图 1.13）。

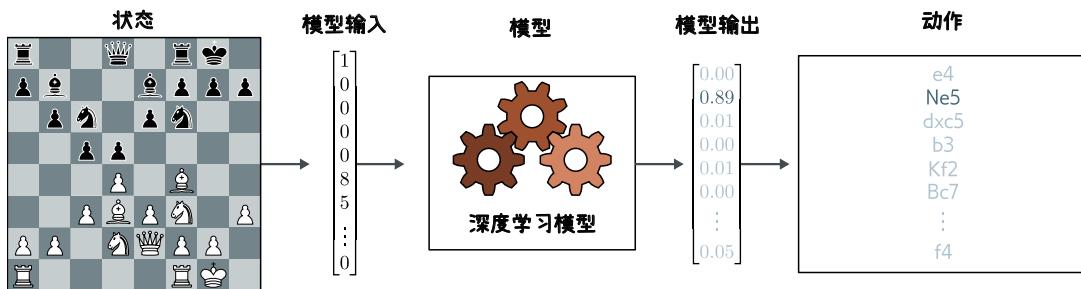


图 1.13 强化学习的策略网络。将深度神经网络融入强化学习的一种方法是使用它们定义从状态（棋盘上的位置）到动作（可能的移动）。这种映射被称为策略

2. 一元线性回归

机器学习使用数据来解决问题。不是由人来思考问题的解决方案，而是让计算机从收集的数据中找到（学习）问题的解决方案。机器学习的本质就是从数据中寻找解决方案。从现在开始，我们将挑战机器学习问题。本章将实现机器学习中最基本的线性回归。

2.1 玩具数据集

在本步骤，我们将创建一个用于实验的小型数据集。这个小型数据集称为玩具数据集(**toy datasets**)。考虑到重现性，我们用固定的随机种子创建数据，具体代码如下。

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 np.random.seed(0) # 固定随机数种子
5 x = np.random.rand(100, 1) # 形状: 100 × 1
6 #  $y = 5 + 2x + \epsilon$ 
7 # 噪声 $\epsilon$ 的形状是100 × 1
8 #  $y$ 的形状也是100 × 1
9 y = 5 + 2 * x + np.random.rand(100, 1)
10
11 # Plot
12 plt.scatter(x, y, s=10)
13 plt.xlabel("x")
14 plt.ylabel("y")
15 plt.show()
```

上面的代码创建了一个由变量 x 和 y 组成的数据集。这些数据点沿直线分布，是在 y 上增加作为噪声的随机数得到的。下图展示了这些 (x, y) 数据点的分布情况。

可视化如下

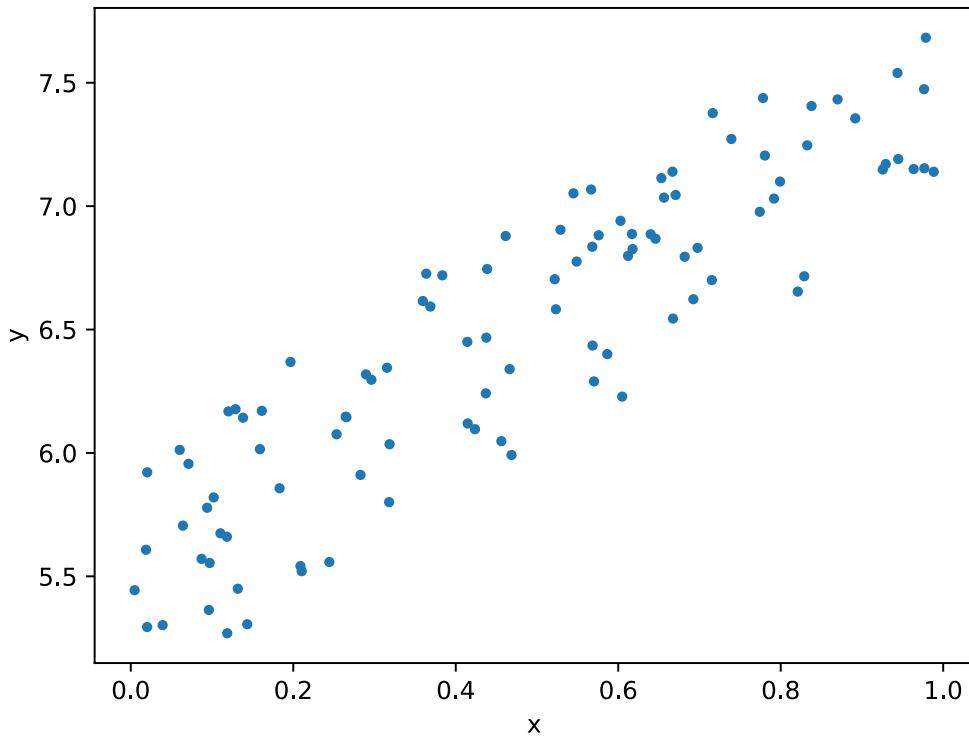


图 2.1 使用的玩具数据集

如图所示，虽然 x 和 y 之间呈线性关系，但数据中存在噪声。我们的目标是创建根据 x 值预测 y 值的模型（式子）。

根据 x 值预测实数值 y 的做法叫作回归（regression）。另外，当预测模型呈线性（直线）时，这种回归分析称为线性回归。

2.2 线性回归的理论知识

接下来的目标是找到拟合给定数据的函数。假设 y 和 x 之间的关系是线性的，函数的式子就可以表示为 $y = Wx + b$ （其中 W 是标量）。 $y = Wx + b$ 这条直线如图所示。

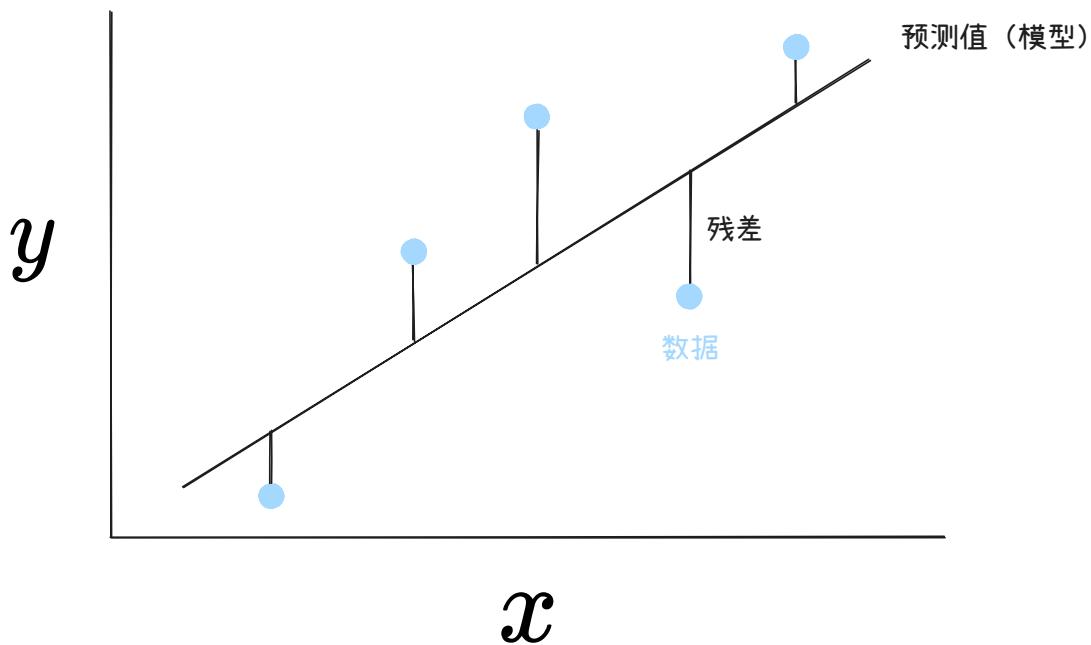


图 2.2 线性回归的示例

如上图所示，我们的目标是找到一条拟合数据的直线 $y = Wx + b$ 。为此，我们需要尽可能地减小数据和预测值之间的差，这个差叫作残差（**residual**）。下面是表示预测值（模型）和数据之间的误差指标的式子。

$$\mathcal{L} = \frac{1}{N} \sum_{i=0}^{N-1} (f(x_i) - y_i)^2 \quad (2.1)$$

在式子中，先求出这 N 个点中的每个点 (x_i, y_i) 的平方误差，然后将它们加起来，之后乘以 $\frac{1}{N}$ 求出平均数。这个式子叫作均方误差（**mean squared error**）。另外，在式子中求平均数时乘的是 $\frac{1}{N}$ ，但在某些情况下，会乘以 $\frac{1}{2N}$ 。但无论哪种情况，在用梯度下降法求解时，都可以通过调整学习率的值来解决同样的问题。

🔥 损失函数

评估模型好坏的函数叫作损失函数（**loss function**）。此时，我们可以说线性回归使用均方误差作为损失函数。

我们的目标是找到使式子表示的损失函数的输出最小的 W 和 b 。这就是函数优化问题。此处使用梯度下降法来找到使式子最小化的参数。

⚡ 梯度下降法

梯度下降法是深度学习中最重要的优化方法，我们会反复讲解！

2.3 线性回归的实现

首先我们来求解损失函数针对参数的梯度 ∇ ，读作“**nabla**”。

$$\nabla = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial W} \\ \frac{\partial \mathcal{L}}{\partial b} \end{bmatrix} \quad (2.2)$$

接下来我们分别求偏导数

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial W} &= \frac{\partial \left\{ \frac{1}{N} \sum_{i=0}^{N-1} (f(x_i) - y_i)^2 \right\}}{\partial W} \\ &= \frac{\partial \left\{ \frac{1}{N} \sum_{i=0}^{N-1} (Wx_i + b - y_i)^2 \right\}}{\partial W} \\ &= \frac{\partial \left\{ \frac{1}{N} \left[(Wx_0 + b - y_0)^2 + (Wx_1 + b - y_1)^2 + \dots + (Wx_{N-1} + b - y_{N-1})^2 \right] \right\}}{\partial W} \\ &= \frac{1}{N} \left[\frac{\partial (Wx_0 + b - y_0)^2}{\partial W} + \dots + \frac{\partial (Wx_{N-1} + b - y_{N-1})^2}{\partial W} \right] \\ &= \frac{1}{N} [2(Wx_0 + b - y_0) \cdot x_0 + \dots + 2(Wx_{N-1} + b - y_{N-1}) \cdot x_{N-1}] \\ &= \frac{2}{N} \sum_{i=0}^{N-1} (Wx_i + b - y_i) \cdot x_i \end{aligned} \quad (2.3)$$

以及

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \left\{ \frac{1}{N} \sum_{i=0}^{N-1} (f(x_i) - y_i)^2 \right\}}{\partial b} \\ &= \frac{\partial \left\{ \frac{1}{N} \sum_{i=0}^{N-1} (Wx_i + b - y_i)^2 \right\}}{\partial b} \\ &= \frac{\partial \left\{ \frac{1}{N} \left[(Wx_0 + b - y_0)^2 + (Wx_1 + b - y_1)^2 + \dots + (Wx_{N-1} + b - y_{N-1})^2 \right] \right\}}{\partial b} \\ &= \frac{1}{N} \left[\frac{\partial (Wx_0 + b - y_0)^2}{\partial b} + \dots + \frac{\partial (Wx_{N-1} + b - y_{N-1})^2}{\partial b} \right] \\ &= \frac{1}{N} [2(Wx_0 + b - y_0) \cdot 1 + \dots + 2(Wx_{N-1} + b - y_{N-1}) \cdot 1] \\ &= \frac{2}{N} \sum_{i=0}^{N-1} (Wx_i + b - y_i) \end{aligned} \quad (2.4)$$

所以损失函数针对参数的梯度为

$$\nabla = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial W} \\ \frac{\partial \mathcal{L}}{\partial b} \end{bmatrix} = \begin{bmatrix} \frac{2}{N} \sum_{i=0}^{N-1} (Wx_i + b - y_i) \cdot x_i \\ \frac{2}{N} \sum_{i=0}^{N-1} (Wx_i + b - y_i) \end{bmatrix} \quad (2.5)$$

使用梯度下降法来寻找损失函数的最小值的算法如下：

$$\begin{aligned} W &\leftarrow W - \alpha \frac{\partial \mathcal{L}}{\partial W} \\ b &\leftarrow b - \alpha \frac{\partial \mathcal{L}}{\partial b} \end{aligned} \quad \begin{array}{l} \text{学习率} \\ \text{学习率} \end{array} \quad (2.6)$$

图 2.3 梯度下降法

在使用梯度下降法时，我们首先需要为参数 W 和 b 选择初始值，我们这里选择 $W = 0$ 以及 $b = 0$ 。

所以有如下代码：

```
1 W = 0
2 b = 0
3
4 def predict(x):
5     """根据输入x做出预测"""
6     y = W * x + b
7     return y
```

 Python

接下来我们使用代码实现损失函数

```
1 def mean_squared_error(x0, x1):
2     diff = x0 - x1
3     return np.sum(diff ** 2) / len(diff)
```

 Python

然后我们使用代码计算参数的梯度

```
1 def gradient(x, y, W, b):
2     N = len(x)
3     W_grad = 2 / N * sum(
4         (W * xi[0] + b - yi[0]) * xi[0] for (xi, yi) in zip(x, y)
5     )
6     b_grad = 2 / N * sum(
7         (W * xi[0] + b - yi[0]) for (xi, yi) in zip(x, y)
8     )
9     return W_grad, b_grad
```

 Python

有了梯度之后，我们使用梯度下降法更新 100 轮参数，学习率设置为 0.1

```
1 lr = 0.1
2 iters = 100
3
4 for i in range(iters):
5     y_pred = predict(x)
6     loss = mean_squared_error(y, y_pred)
7
8     W_grad, b_grad = gradient(x, y, W, b)
9     W = W - lr * W_grad
10    b = b - lr * b_grad
11    print(W, b, loss)
12
13 # 可视化最后拟合出的直线以及训练数据
14 plt.scatter(x, y, s=10)
15 plt.xlabel("x")
16 plt.ylabel("y")
17 y_pred = predict(x)
18 plt.plot(x, y_pred, color="r")
19 plt.show()
```

 Python

可以看到 `loss` 一直在下降，最后得到的参数是

$$\begin{aligned} W &= 2.118073690511974 \\ b &= 5.466089050922982 \end{aligned} \tag{2.7}$$

如果将直线 $y = Wx + b$ 画出来，可视化如下：

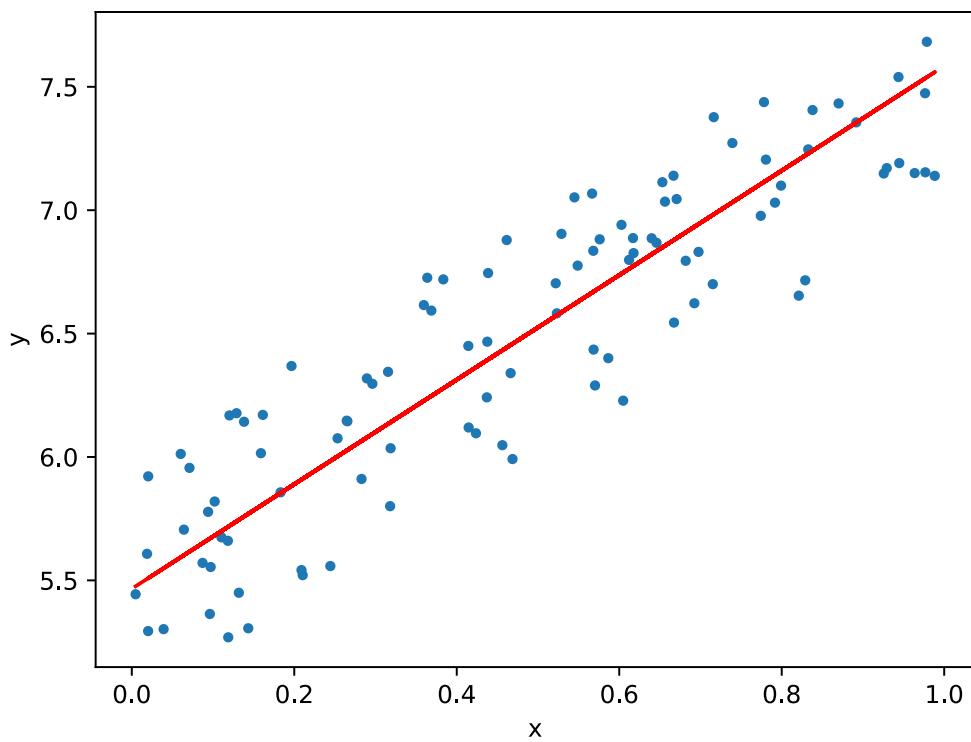


图 2.4 训练后的模型

如图所示，我们已经得到了一个拟合数据的模型。

2.4 梯度下降法为什么可以找到损失函数的最小值呢？

我们都知道损失函数 \mathcal{L} 是参数 W 和 b 的函数。所以我们可以将损失函数可视化出来。

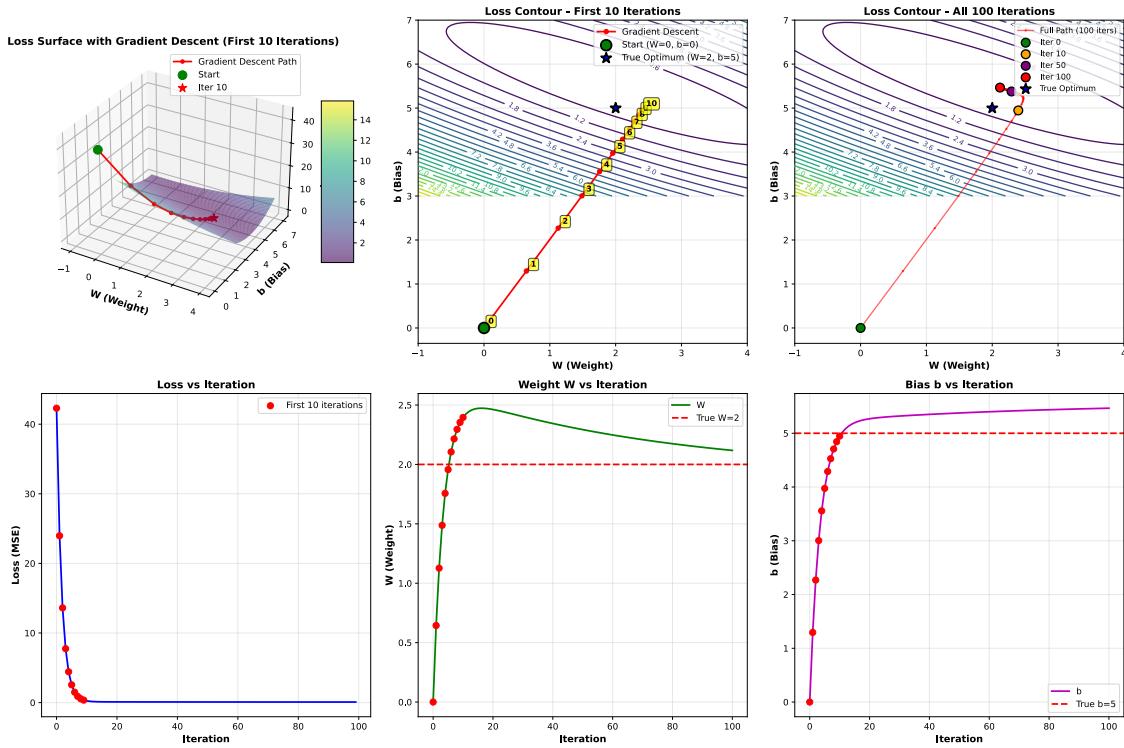


图 2.5 梯度下降法示意图

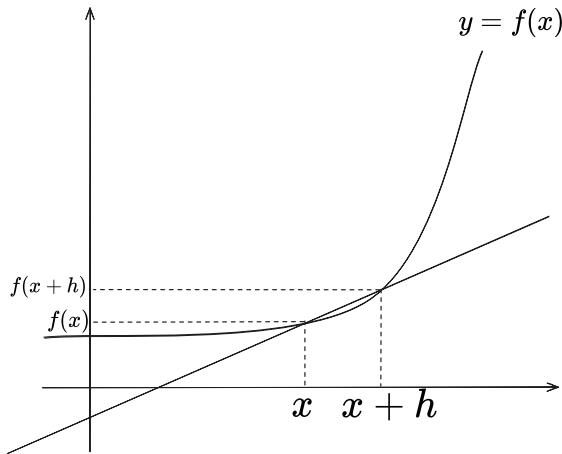
2.5 如何求导? (反向传播算法)

在前面的梯度下降法中，损失函数的梯度是我们手动进行求导得到的。

而在现实中，一个函数的参数是非常多的，每个参数都进行手动求导会非常的繁琐，甚至是不可能的（DeepSeek-V3 的参数数量是 6710 亿）。

所以我们需要寻求能够自动求导（自动微分）的方法，也就是大名鼎鼎的反向传播算法。

我们先来看一下求导。

图 2.6 曲线 $y = f(x)$ 和通过其两点的直线

什么是导数？简单地说，导数是变化率的一种表示方式。比如某个物体的位置相对于时间的变化率就是位置的导数，即速度。速度相对于时间的变化率就是速度的导数，即加速度。像这样，导数表示的是变化率，它被定义为在极短时间内的变化量。函数 $f(x)$ 在 x 处的导数可用下面的式子表示。

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (2.8)$$

上面的式子中 $\lim_{h \rightarrow 0}$ 表示极限，意思是 h 应该尽可能地接近0。 $\frac{f(x+h)-f(x)}{h}$ 表示通过两点的直线的斜率。

如图所示，函数 $f(x)$ 在 x 和 $x + h$ 两点之间的变化率为 $\frac{f(x+h)-f(x)}{h}$ 。让 h 的值尽可能地接近0，就可以求出 x 处的变化率。这就是 $y = f(x)$ 的导数。另外，在 $y = f(x)$ 的可导区间内，对于该区间的任何 x ，上面的式子都成立。因此，式子中的 $f'(x)$ 也是一个函数，我们称之为 $f(x)$ 的导函数。

2.5.1 数值微分的实现

下面根据导数的定义式来实现求导。需要注意的是，计算机不能处理极限值。因此，这里的 h 表示一个近似值。例如我们可以用 $h = 0.0001 (= 1e-4)$ 这种非常小的值来计算求导公式。利用微小的差值获得函数变化量的方法叫作数值微分。

数值微分使用非常小的值 h 求出的是真的导数的近似值。因此，这个值包含误差。中心差分近似是减小近似值误差的一种方法。中心差分近似计算的不是 $f(x)$ 和 $f(x+h)$ 的差，而是 $f(x-h)$ 和 $f(x+h)$ 的差值。下图中的红线表示的就是中心差分近似。

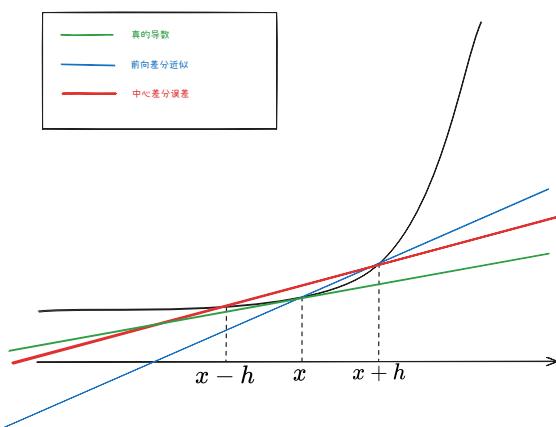


图 2.7 比较真的导数、前向差分近似和中心差分近似

如图所示，计算过 x 和 $x + h$ 这两点的直线的斜率的方法称为前向差分近似，计算 $x - h$ 和 $x + h$ 这两点间斜率的方法称为中心差分近似，中心差分近似实际产生的误差更小。中心差分近似的直线斜率是 $\frac{f(x+h)-f(x-h)}{2h}$ 。

下面我们来实现使用中心差分近似求数值微分的函数，该函数的名称为 `numerical_diff(f, x, eps=1e-4)`。这里的参数 f 是被求导的函数。数值微分可以通过以下代码实现。

```
1 def numerical_diff(f, x, eps=1e-4):
2     x0 = x - eps
3     x1 = x + eps
4     y0 = f(x0)
5     y1 = f(x1)
6     return (y1 - y0) / (2 * eps)
```

Python

例如如果 $f(x) = x^2$ ，并在 $x = 2.0$ 处求导，那么有如下代码：

```
1 def f(x):
```

Python

```

2     return x ** 2
3
4 x = 2.0
5 dy = numerical_diff(f, x)
6 print(dy)

```

运行结果如下

```
1 4.000000000004
```

由上面的运行结果可知, $y = x^2$ 在 $x = 2.0$ 时计算得到的导数之为 4.000000000004。不包含误差的导数值为 4.0, 可以说这个结果大体正确。

🔥 解析解求导数

导数也可以通过解析解的方式求解。解析解的方式求解是指只通过式子的变形推导出答案。在上面的例子中, 根据导数的公式可知, $y = x^2$ 的导数为 $\frac{dy}{dx} = 2x$ 。因此, $y = x^2$ 在 $x = 2.0$ 处的导数为 4.0。这个 4.0 是不包含误差的值。前面的数值微分结果虽然不是正好为 4.0, 但我们可以看出误差是相当小的。

当然复合函数也可以使用数值微分的方式进行求导。例如

$$y = (e^{x^2})^2 \quad (2.9)$$

代码如下:

```

1 def f(x):
2     x1 = x ** 2
3     x2 = math.exp(x1)
4     y = x2 ** 2
5     return y
6
7 x = 0.5
8 dy = numerical_diff(f, x)
9 print(dy)

```

Python

运行结果是: 3.2974426293330694。

现在我们已经成功实现了“自动”求导。只要用代码来定义要完成的计算(前面的例子定义了函数 f), 程序就会自动求出导数。使用这种方法, 无论多么复杂的函数组合, 程序都能自动求出导数。今后函数的种类越来越多的话, 不管是什么计算, 只要是可微函数, 程序就能求出它的导数。不过遗憾的是, 数值微分的方法存在一些问题。

⚡ 数值微分存在的问题

1. 数值微分的结果包含误差。在多数情况下，这个误差非常小，但在一些情况下，计算产生的误差可能会很大。数值微分的结果中容易包含误差的主要原因是“精度丢失”。中心差分近似等求差值的方法计算的是相同量级数值之间的差，但由于精度丢失，计算结果中会出现有效位数减少的情况。以有效位数为 4 的情况为例，在计算两个相近的值之间的差时，比如 $1.234 - 1.233$ ，其结果为 0.001，有效位数只有 1 位。本来可能是 $1.234\dots - 1.233\dots = 0.001434\dots$ 之类的结果，但由于精度丢失，结果变成 0.001。同样的情况也会发生在数值微分的差值计算中，精度丢失使结果更容易包含误差。
2. 数值微分更严重的问题是计算成本高。具体来说，在求多个变量的导数时，程序需要计算每个变量的导数。有些神经网络包含几百万个以上的变量（参数），通过数值微分对这么多的变量求导是不现实的。这时，反向传播就派上了用场。

另外，数值微分可以轻松实现，并能计算出大体正确的数值。而反向传播是一种复杂的算法，实现时容易出现 bug。我们可以使用数值微分的结果检查反向传播的实现是否正确。这种做法叫作梯度检验（gradient checking），它是一种将数值微分的结果与反向传播的结果进行比较的方法。

2.5.2 反向传播算法

我们以 $y = (e^{x^2})^2$ 在 $x = 0.5$ 求导为例子来说明一下反向传播算法。反向传播算法分为两个过程，前向过程（forward）和反向过程（backward）。

1. 前向过程：保存中间计算结果

$$\begin{aligned} v_0 &= x = 0.5 \\ v_1 &= v_0^2 = 0.5^2 = 0.25 \\ v_2 &= e^{v_1} = 1.2840254166877414 \\ v_3 &= v_2^2 = 1.648721270700128 \\ y &= v_3 \end{aligned} \tag{2.10}$$

2. 反向过程：利用前向过程保存的中间结果，以及链式求导法则来计算导数。

根据链式求导法则

$$\begin{aligned} \frac{dy}{dx} &= \frac{dy}{dv_3} \cdot \frac{dv_3}{dv_2} \cdot \frac{dv_2}{dv_1} \cdot \frac{dv_1}{dv_0} \cdot \frac{dv_0}{dx} \\ &= 1 \cdot 2v_2 \cdot e^{v_1} \cdot 2v_0 \cdot 1 \\ &= 1 \times (2 \times 1.2840254166877414) \times e^{0.25} \times (2 \times 0.5) \times 1 \\ &= 3.297442541400256 \end{aligned} \tag{2.11}$$

2.6 实现一个微型的自动微分引擎

3. 分类问题：以手写数字识别为例

从本章开始，我们开始从代码和数学层面研究一下神经网络。

首先，神经网络是一个有很多个参数的函数。既然是函数，那么就有输入和输出，完成手写数字识别任务的神经网络的函数是什么呢？

$$\text{5} \rightarrow f(\bullet) \rightarrow \begin{bmatrix} 0.01(\text{预测为手写数字 0 的概率}) \\ 0.01(\text{预测为手写数字 1 的概率}) \\ 0.01(\text{预测为手写数字 2 的概率}) \\ 0.01(\text{预测为手写数字 3 的概率}) \\ 0.01(\text{预测为手写数字 4 的概率}) \\ 0.90(\text{预测为手写数字 5 的概率}) \\ 0.01(\text{预测为手写数字 6 的概率}) \\ 0.01(\text{预测为手写数字 7 的概率}) \\ 0.01(\text{预测为手写数字 8 的概率}) \\ 0.02(\text{预测为手写数字 9 的概率}) \end{bmatrix} \quad (3.1)$$

也就是说，我们要构造一个函数，能够接收手写数字图片为输入，而输出是手写数字属于每一个分类的概率所组成的向量！

所以我们需要找到一个函数 $f(\bullet)$ ，使得当函数接收到一张手写数字图片 5 时，输出的概率向量中，索引为 5 的概率是最大的。

这个函数的寻找历经了很多年的演变，函数结构可以是 MLP（多层感知机），可以是 CNN（卷积神经网络），还可以是大杀器 Transformer。这都是我们未来要仔细学习的重点。

深度学习就是在固定函数结构的情况下，寻找使得预测准确率最高的函数参数。

以线性回归为例，我们固定了函数结构 $y = Wx + b$ ，想要寻找参数 W 和 b ，使得给定某一个输入 x ，输出的 y 能够比较准确。

所以神经网络的训练流程如下：

1. 设计模型结构（固定函数结构）
2. 收集训练数据集
3. 设计损失函数（用于度量预测准确率）
4. 使用最优化算法（例如梯度下降法）寻找使得损失函数最小的参数。

其中最优化算法需要求梯度（求导），所以我们需要类似于 PyTorch 这种能够自动求导的框架。

本章使用的数据集为 **MNIST** 数据集。**MNIST** 数据集是机器学习领域的经典基准数据集。它包含 28×28 个像素点（784 个像素点）的手写数字（0-9）灰度图像及其对应标签。我们的目标是构建一个能够根据像素值准确分类这些数字的神经网络。

3.1 准备训练数据集

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4
5 data = pd.read_csv("mnist.csv")
6 print(data.head())
```

可以看到我们将数据集的前 5 行打印了出来。

	label	1x1	1x2	1x3	1x4	1x5	1x6	1x7	1x8	1x9	...	28x19	28x20	28x21	28x22	28x23	28x24	28x25	28x26	28x27	28x28
0	5	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	
2	4	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	
3	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	
4	9	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	

5 rows × 785 columns

图 3.1 mnist 数据集的前 5 行，形状为 5×785

“label”这一列表示的是手写数字的分类，例如第一列的 `label` 是 5，所以后面的像素点渲染出来是手写数字 5。

从列 `1x1` 到 `28x28` 共 784 个像素点，是手写数字图片的每个像素点的值。可以看到很多 0，也就是手写数字图片中的很多像素点都是黑色。我们可以尝试将第一行的 784 个像素点渲染成图片。

```
1 first_row = data.iloc[0]
2 label = first_row.iloc[0]
3 pixels = first_row.iloc[1:].values
4
5 # 将一维数组重塑为28x28的图像
6 image = pixels.reshape(28, 28)
7
8 # 创建图形
9 plt.figure(figsize=(6, 6))
10 plt.imshow(image, cmap='gray')
11 plt.title(f'Label: {int(label)}', fontsize=16)
12 plt.axis('off')
13 plt.colorbar(label='Pixel Intensity')
14 plt.tight_layout()
15 plt.show()
```

可以得到图如下：

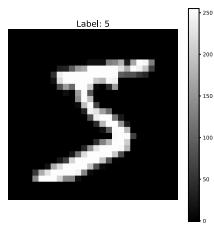


图 3.2 第一行的图片

我们数据集中一共有 60000 张图片，以及对应的 60000 个标签。我们先将数据转换成 numpy 的 ndarray 格式的数据，然后将数据打乱。

```
1 data = np.array(data) # 转换成ndarray格式
2 print(data.shape) # 打印形状
3 m, n = data.shape
4 np.random.shuffle(data) # 打乱数据
```

Python

可以看到数据的形状是(60000, 785)。

我们接下来要将数据集切分为训练数据集和验证数据集。

🔥 训练数据集和验证数据集

- 训练数据集：用来训练我们的神经网络
- 验证数据集：用来验证训练好的神经网络表现怎么样

我们选择前 1000 行数据作为验证集

```
1 data_dev = data[0: 1000].T # 取前1000行，并转置
2 print(data_dev.shape)
```

Python

可以看到 data_dev 的形状是：(785, 1000)。也就是说，数据的每一列都是“1 个标签+784 个像素点”，方便我们后续处理。

然后我们将验证集数据的标签和像素点数据拆分开，先来提取标签数据

```
1 Y_dev = data_dev[0] # 提取标签数据
2 print(Y_dev.shape) # 形状: (1000,)
```

Python

然后提取标签对应的像素点数据，需要注意的是灰度图的每个像素点的范围是 0 ~ 255，像素点的类型是整型，为了方便处理，我们进行归一化，也就是将所有像素点的值归一化到 0 ~ 1 之间。

```
1 X_dev = data_dev[1 : n] # 提取像素点数据
2 X_dev = X_dev / 255.0 # 将像素点归一化
3 print(X_dev.shape) # 形状: (784, 1000)
```

Python

剩下的 59000 条数据我们作为训练模型用的训练数据集。处理方式和上面的验证集基本相同

```
1 data_train = data[1000 : m].T # 取后面的59000行数据并转置
2 print(data_train.shape) # 形状: (785, 59000)
3 Y_train = data_train[0] # 提取标签，形状为(59000,)
4 X_train = data_train[1 : n] # 提取像素点的数据，形状为(784, 59000)
5 X_train = X_train / 255.0 # 将像素点进行归一化
```

Python

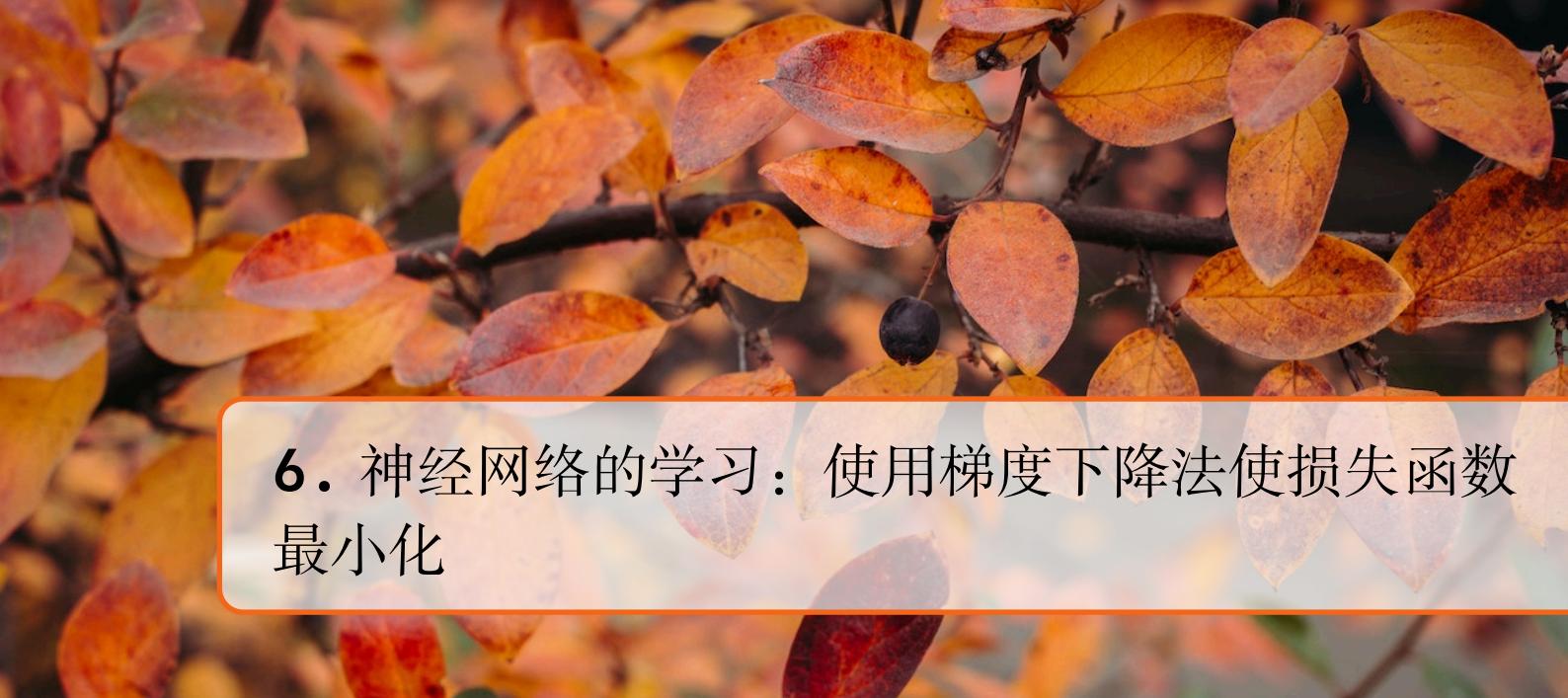
这样我们的数据集就准备好了。



4. 卷积神经网络：将手写数字识别准确率拉满！



5. 损失函数：均方误差损失和交叉熵损失的由来



6. 神经网络的学习：使用梯度下降法使损失函数最小化



7. PyTorch 简介

本附录旨在为你提供必要的技能和知识，以便将深度学习付诸实践并从零开始实现大语言模型。我们将使用 **PyTorch** 作为本书的主要工具，这是一个广泛应用的 **Python** 深度学习库。

首先，我们将指导你搭建一个支持 **PyTorch** 和 **GPU** 的深度学习工作台。然后，我们将介绍张量的基本概念及其在 **PyTorch** 中的用法。接下来，我们将深入探讨 **PyTorch** 的自动微分引擎，这一特性使我们能够方便且高效地使用反向传播，这也是神经网络训练的重要环节。

本附录的目标是为那些刚接触 **PyTorch** 深度学习的读者提供入门资料。虽然我们会从零开始讲解 **PyTorch**，但不会覆盖其所有功能，而是聚焦于实现大语言模型所需的基本概念。如果你对深度学习已有一定了解，那么可以跳过本附录，直接往下阅读。

7.1 什么是 **PyTorch**

PyTorch 是一个开源的基于 **Python** 的深度学习库。根据 **Papers With Code** 这个跟踪和分析研究论文平台的数据，自 2019 年以来，**PyTorch** 已成为研究领域使用最广泛的深度学习库，并且领先优势显著。此外，根据 2022 年 **Kaggle** 数据科学与机器学习调查，大约 40% 的受访者正在使用 **PyTorch**，并且这一比例每年都在增长。

PyTorch 之所以如此受欢迎，原因之一在于其用户友好的界面和高效性。它不仅易于使用，还保留了高度的灵活性，允许专业用户深入修改模型的底层组件，以实现个性化和优化。总之，对许多从业者和研究人员而言，**PyTorch** 在可用性和特性之间提供了恰到好处的平衡。

7.1.1 **PyTorch** 的三大核心组件

PyTorch 是一个相对全面的库，我们可以通过关注其三大核心组件来理解它，如下图所示。

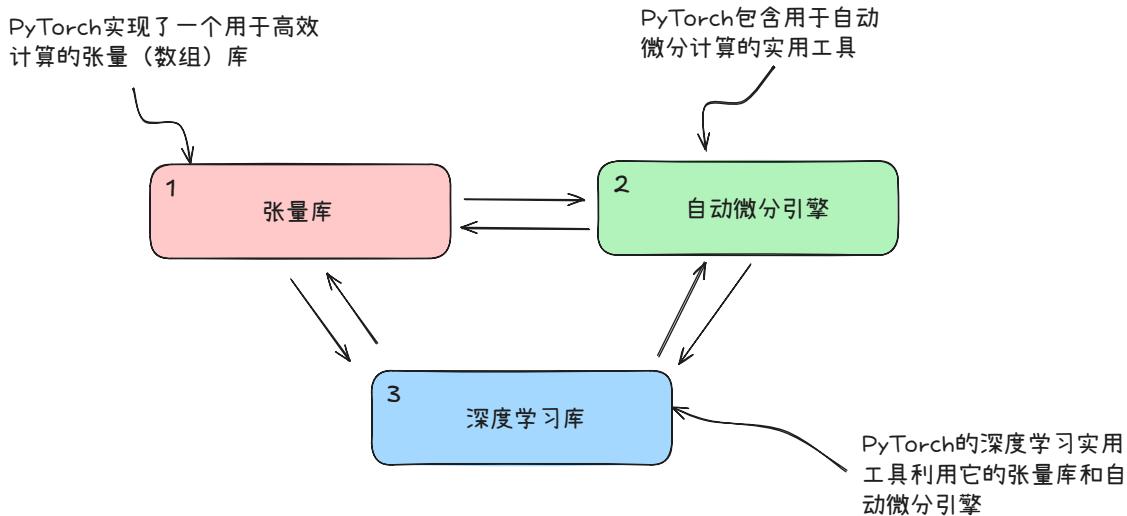


图 7.1 PyTorch 的三大核心组件包括作为计算基础构建块的张量库、用于模型优化的自动微分引擎以及深度学习工具函数，这使得实现和训练深度神经网络模型更加容易

首先，PyTorch 是一个张量库，它扩展了 NumPy 基于数组的编程功能，增加了 GPU 加速特性，从而实现了 CPU 和 GPU 之间的无缝计算切换。其次，PyTorch 是一个自动微分引擎，也称为 `autograd`，它能够自动计算张量操作的梯度，从而简化反向传播和模型优化。最后，PyTorch 是一个深度学习库，它提供了模块化、灵活且高效的构建块（包括预训练模型、损失函数和优化器），能够帮助研究人员和开发人员轻松设计和训练各种深度学习模型。

7.1.2 定义深度学习

在新闻中，大语言模型通常被称为“人工智能模型”。然而，大语言模型实际上也是一种深度神经网络，而 PyTorch 是一个深度学习库。是不是听起来有些困惑？在继续之前，让我们简要总结一下这些术语之间的关系。

人工智能的基本目标是创建能够执行通常需要人类智能水平的任务的计算机系统。这些任务包括自然语言理解、模式识别和决策制定。（尽管取得了显著进展，但人工智能仍远未达到这种通用智能的水平。）

如下图所示，机器学习是人工智能的一个子领域，其专注于学习算法的开发和改进。机器学习背后的主要理念是使计算机能够从数据中学习，并在没有被明确编程的情况下进行预测或决策。这涉及能够识别模式、从历史数据中学习，并随着时间的推移通过更多数据和反馈提升性能的算法。

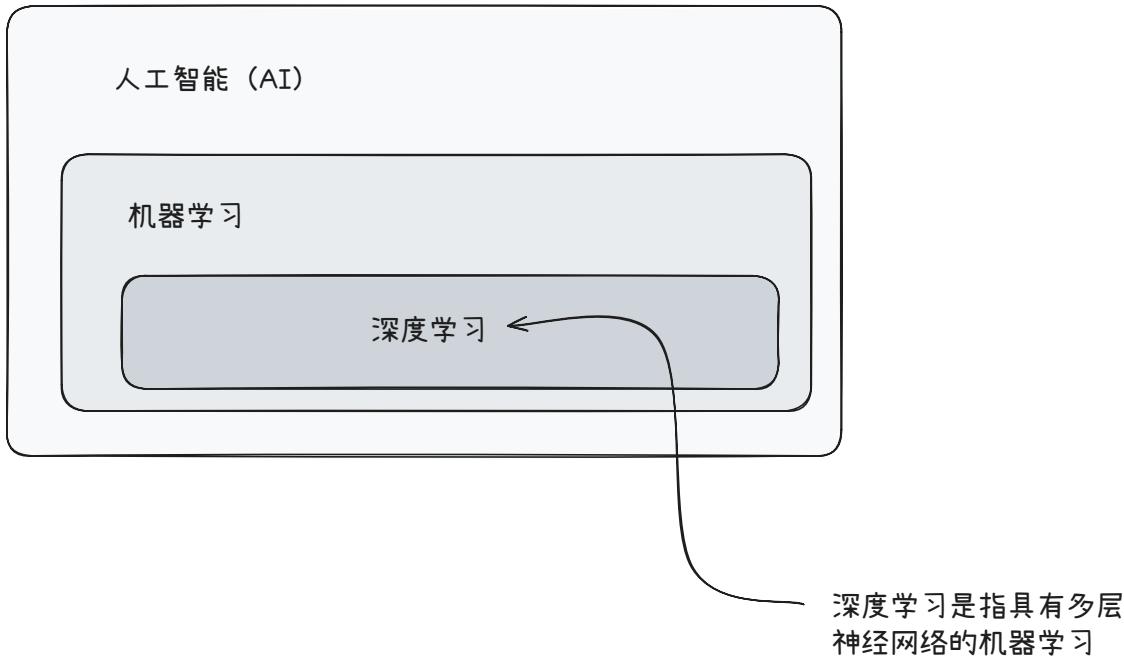


图 7.2 深度学习是机器学习的一个子类别，专注于实现深度神经网络。机器学习是人工智能的一个子类别，涉及从数据中学习的算法。人工智能是一个更广泛的概念，指的是机器能够执行通常需要人类智能水平的任务

机器学习在人工智能的演变中发挥了重要作用，为我们今天所看到的许多进展（包括大语言模型）提供了动力。机器学习还支持在线零售商和流媒体服务使用的推荐系统、垃圾邮件过滤、虚拟助手中的语音识别，甚至自动驾驶汽车等技术。机器学习的引入和发展显著增强了人工智能的能力，使其超越传统的基于规则的系统，并能够适应新的输入或变化的环境。

深度学习是机器学习的一个子类别，专注于深度神经网络的训练和应用。这些深度神经网络最初受到人脑工作原理（特别是许多神经元之间的相互连接）的启发。深度学习中的“深度”指的是人工神经元或节点的多个隐藏层，这些层使它们能够对数据中的复杂非线性关系进行建模。与传统机器学习技术擅长简单模式识别不同，深度学习擅长处理诸如图像、音频、文本之类的非结构化数据，因此特别适合用于大语言模型。

机器学习和深度学习中典型的预测建模工作流程（也称为监督学习）如下图所示。

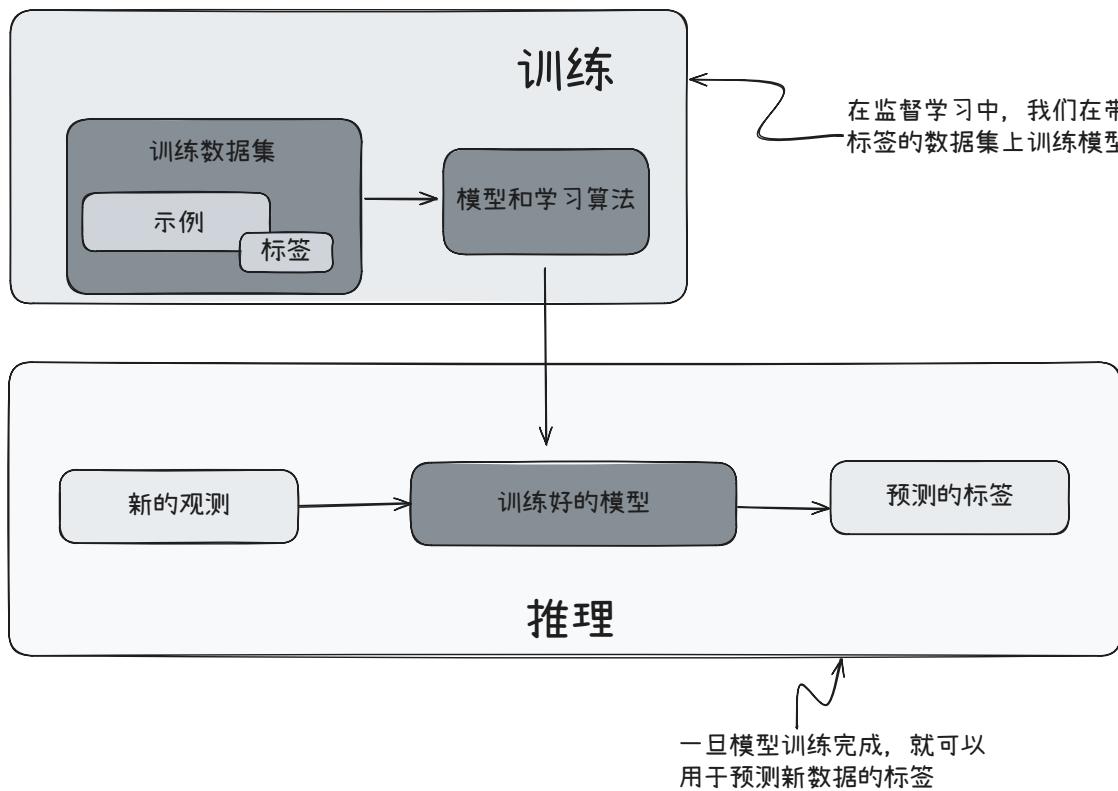


图 7.3 监督学习的预测建模工作流程包括一个训练阶段，在该阶段中，模型在训练数据集中带标签的示例上进行训练。训练好的模型随后可用于预测新观测数据的标签

通过使用学习算法，模型可以在由示例和相应标签组成的训练数据集上进行训练。例如，在垃圾邮件分类器的案例中，训练数据集由电子邮件及其“垃圾消息”和“非垃圾消息”标签组成，这些标签是由人类标注的。然后，训练好的模型可以在新的样本（新的电子邮件）上使用，以预测这些样本的未知标签（“垃圾消息”或“非垃圾消息”）。当然，我们还希望在训练阶段和推理阶段之间添加模型评估，以确保模型在实际应用之前满足性能标准。

如果想要训练大语言模型来对文本进行分类，那么训练和使用大语言模型的工作流程与图 A-3 中描述的类似。即使你关注的是训练大语言模型来生成文本（这也是我们的主要关注点），图 A-3 仍然适用。在这种情况下，预训练期间的标签可以从文本本身获取（第 1 章介绍的下一单词预测任务）。在推理时，大语言模型将在给定输入提示词的情况下生成全新的文本（而不是预测标签）。

7.1.3 安装 PyTorch

PyTorch 可以像其他任何 Python 库或包一样进行安装。然而，由于 PyTorch 是一个包含 CPU 和 GPU 兼容代码的综合性库，安装过程可能需要额外说明。

🔥 Python 版本

许多科学计算库不会立即支持最新版本的 Python。因此，在安装 PyTorch 时，建议使用比最新版本旧一到两个版本的 Python。如果最新的 Python 版本是 Python 3.13，那么推荐使用 Python 3.11 或 Python 3.12。

例如，PyTorch 有两个版本：一个是仅支持 CPU 计算的精简版，另一个是支持 CPU 和 GPU 计算的完整版。如果你的机器有一个兼容 CUDA 的 GPU（理想情况下是 NVIDIA T4、RTX 2080 Ti 或更新的型号），那么推荐安装 GPU 版本。以下是在代码终端中安装 PyTorch 的默认命令：

```
1 $ pip install torch
2 # 或者指定版本安装
3 $ pip install torch==2.7.0
```

Shell

假设你的计算机支持兼容 CUDA 的 GPU。在这种情况下，如果你正在使用的 Python 环境已安装必要的依赖项（如 pip），那么系统将自动安装支持 CUDA 加速的 PyTorch 版本。

安装 PyTorch 后，可以通过在 Python 中运行以下代码来检查安装是否识别了内置的 NVIDIA GPU：

```
1 import torch
2 print(torch.__version__) # 打印torch版本
3 print(torch.cuda.is_available()) # 打印torch是否支持cuda
```

Python

7.2 理解张量

张量表示一个数学概念，它可以将向量和矩阵推广到潜在的更高维度。换句话说，张量是可以通过其阶数（或秩）来表征的数学对象，其中阶数提供了维度的数量。例如，标量（仅是一个数值）是秩为 0 的张量，向量是秩为 1 的张量，矩阵是秩为 2 的张量，如下图所示。

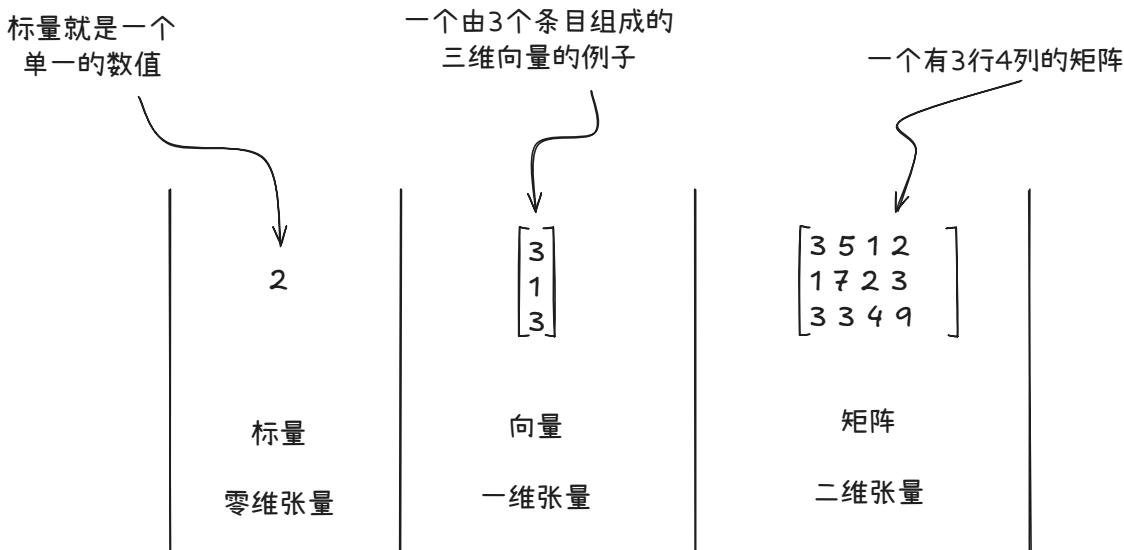


图 7.4 不同秩的张量。这里零维对应于秩 0，一维对应于秩 1，二维对应于秩 2。一个由 3 个元素组成的三维向量仍然是秩为 1 的张量

从计算的角度来看，张量是一种数据容器。举例来说，它们存储多维数据，其中每个维度表示一个不同的特征。像 PyTorch 这样的张量库能够高效地创建、操作和计算这些数组。在这个上下文中，张量库的功能类似于数组库。

PyTorch 张量类似于 NumPy 数组，但具有几个对深度学习至关重要的附加功能。例如，PyTorch 添加了一个自动微分引擎，简化了梯度计算（求导）。PyTorch 张量还支持 GPU 计算，以加速深度神经网络的训练。

7.2.1 标量、向量、矩阵和张量

如前所述，PyTorch 张量是用于与数组类似结构的数据容器。标量是零维张量（例如，仅一个数值），向量是一维张量，矩阵是二维张量。对于更高维的张量没有特定的术语，因此通常将三维张量

称为“3D 张量”，以此类推。可以使用 `torch.tensor()` 函数创建 PyTorch 的 `Tensor` 类对象，如代码所示。

```
创建PyTorch张量
1 import torch
2 # 从Python整数创建一个零维张量（标量）
3 tensor0d = torch.tensor(1)
4 # 从Python列表创建一个一维张量（向量）
5 tensor1d = torch.tensor([1, 2, 3])
6 # 从嵌套的Python列表创建一个二维张量
7 tensor2d = torch.tensor([[1, 2],
8                         [3, 4]])
9 # 从嵌套的Python列表创建一个三维张量
10 tensor3d = torch.tensor([[[1, 2], [3, 4]],
11                          [[5, 6], [7, 8]]])
```

7.2.2 张量数据类型

PyTorch 采用 Python 默认的 64 位整数数据类型。可以通过张量的 `.dtype` 属性来访问张量的数据类型，如下所示：

```
1 tensor1d = torch.tensor([1, 2, 3])
2 print(tensor1d.dtype)
```

输出如下所示：

```
1 torch.int64
```

如果使用 Python 浮点数创建张量，那么 PyTorch 默认会创建具有 32 位精度的张量：

```
1 floatvec = torch.tensor([1.0, 2.0, 3.0])
2 print(floatvec.dtype)
```

输出如下所示：

```
1 torch.float32
```

这种选择主要是为了在精度和计算效率之间取得平衡。32 位浮点数在大多数深度学习任务中提供了足够的精度，同时其消耗的内存和计算资源比 64 位浮点数更少。此外，GPU 架构对 32 位计算进行了优化，使用这种数据类型可以显著加快模型训练和推理速度。还可以使用张量的 `.to` 方法更改精度。以下代码演示了如何将 64 位整数张量更改为 32 位浮点张量：

```
1 floatvec = tensor1d.to(torch.float32)
2 print(floatvec.dtype)
```

这将返回以下内容。

```
1 torch.float32
```

7.2.3 常见的 PyTorch 张量操作

本书无法全面覆盖所有 PyTorch 张量操作和命令。然而，我们将在介绍它们时简要描述相关操作。我们已经介绍了创建新张量的 `torch.tensor()` 函数：

```
1 tensor2d = torch.tensor([[1, 2, 3],
2                           [4, 5, 6]])
```

```
3 print(tensor2d)
```

这将打印以下内容：

```
1 tensor([[1, 2, 3],  
2         [4, 5, 6]])
```

Python

此外，`.shape` 属性允许我们访问张量的形状：

```
1 print(tensor2d.shape)
```

Python

输出如下所示：

```
1 torch.Size([2, 3])
```

Python

如你所见，`.shape` 返回的是`[2, 3]`，这意味着该张量有 2 行 3 列。要将该张量变为 3×2 的形状，可以使用`.reshape` 方法：

```
1 print(tensor2d.reshape(3, 2))
```

Python

这将打印以下内容：

```
1 tensor([[1, 2],  
2         [3, 4],  
3         [5, 6]])
```

Python

然而，请注意，在 PyTorch 中，重塑张量更常用的命令是`.view()`：

```
1 print(tensor2d.view(3, 2))
```

Python

输出如下所示：

```
1 tensor([[1, 2],  
2         [3, 4],  
3         [5, 6]])
```

Python

类似于`.reshape` 和`.view`，在某些情况下，PyTorch 提供了多种语法选项来执行相同的计算。PyTorch 最初遵循了原始 Lua 版本 Torch 的语法约定，但后来应用户的要求，添加了与 NumPy 类似的语法。`(.view()和.reshape())` 的微妙区别在于它们对内存布局的处理方式：`.view()` 要求原始数据是连续的，如果不是，它将无法工作，而`.reshape()` 会工作，如有必要，它会复制数据以确保所需的形状。)

接下来，可以使用`.T` 来转置张量，这意味着将其沿对角线翻转。请注意，这与重塑张量类似，你可以从以下结果中看到这一点：

```
1 print(tensor2d.T)
```

Python

输出如下所示：

```
1 tensor([[1, 4],  
2         [2, 5],  
3         [3, 6]])
```

Python

最后，PyTorch 中常用的矩阵相乘方法是`.matmul` 方法：

```
1 print(tensor2d.matmul(tensor2d.T))
```

Python

输出如下所示：

```
1 tensor([[14, 32],  
2         [32, 77]])
```

Python

然而，也可以使用`@`运算符，它能够更简洁地实现相同的功能：

```
1 print(tensor2d @ tensor2d.T)
```

 Python

输出如下所示：

```
1 tensor([[14, 32],
2         [32, 77]])
```

 Python

如前所述，我们会在需要时介绍额外的操作。

7.3 将模型视为计算图

现在让我们来了解一下 PyTorch 的自动微分引擎，也称为 `autograd`。PyTorch 的 `autograd` 系统能够在动态计算图中自动计算梯度。

计算图是一种有向图，主要用于表达和可视化数学表达式。在深度学习的背景下，计算图列出了计算神经网络输出所需的计算顺序——我们需要用它来计算反向传播所需的梯度，这是神经网络的主要训练算法。

让我们通过一个具体的例子来说明计算图的概念。下面代码实现了一个简单逻辑回归分类器的前向传播（预测步骤），我们可以将其看作一个单层神经网络。它会返回一个介于 0 和 1 之间的分数，当计算损失时，这个分数会与真实的类别标签（0 或 1）进行比较。

逻辑回归的前向传播

```
1 import torch.nn.functional as F
2
3 y = torch.tensor([1.0]) # 真实特征
4 x1 = torch.tensor([1.1]) # 输入特征
5 w1 = torch.tensor([2.2]) # 权重参数
6 b = torch.tensor([0.0]) # 偏置单元
7 z = x1 * w1 + b # 网络输入
8 a = torch.sigmoid(z) # 激活和输出
9 loss = F.binary_cross_entropy(a, y)
```

 Python

🔥 代码对应的数学表达式

线性变换

$$z = w_1 x_1 + b \quad (7.1)$$

激活函数（Sigmoid）：

$$a = \sigma(z) = \frac{1}{1 + e^{-z}} \quad (7.2)$$

损失函数（loss）：

$$\text{loss} = -[y \log(a) + (1 - y) \log(1 - a)] \quad (7.3)$$

即使没有完全理解上述代码中的所有部分，也不要担心。这个例子的重点不是实现一个逻辑回归分类器，而是为了说明如何将一系列计算看作一个计算图，如下图所示。

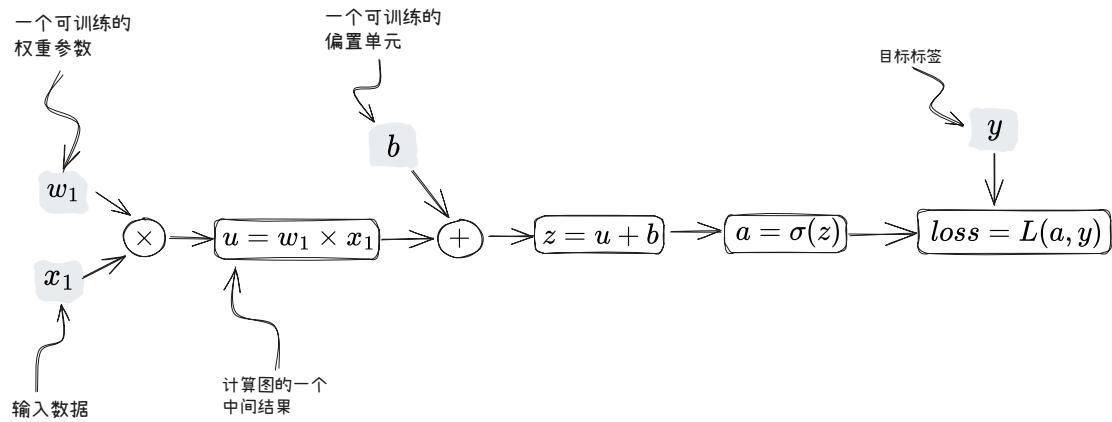


图 7.5 逻辑回归的前向传播作为一个计算图。输入特征 x_1 与模型权重 w_1 相乘，并在加上偏置后通过激活函数 σ 传递。损失是通过比较模型输出 a 与给定标签 y 来计算的

实际上，PyTorch 在后台构建了这样一个计算图，我们可以利用它来计算损失函数相对于模型参数（这里是 w_1 和 b ）的梯度，从而训练模型。

7.4 轻松实现自动微分

如果在 PyTorch 中进行计算，那么只要其终端节点之一的 `requires_grad` 属性被设置为 `True`，PyTorch 默认就会在内部构建一个计算图。这在我们想要计算梯度时非常有用。在训练神经网络时，需要使用反向传播算法计算梯度。反向传播可以被视为微积分中链式法则在神经网络中的应用，如下图所示。

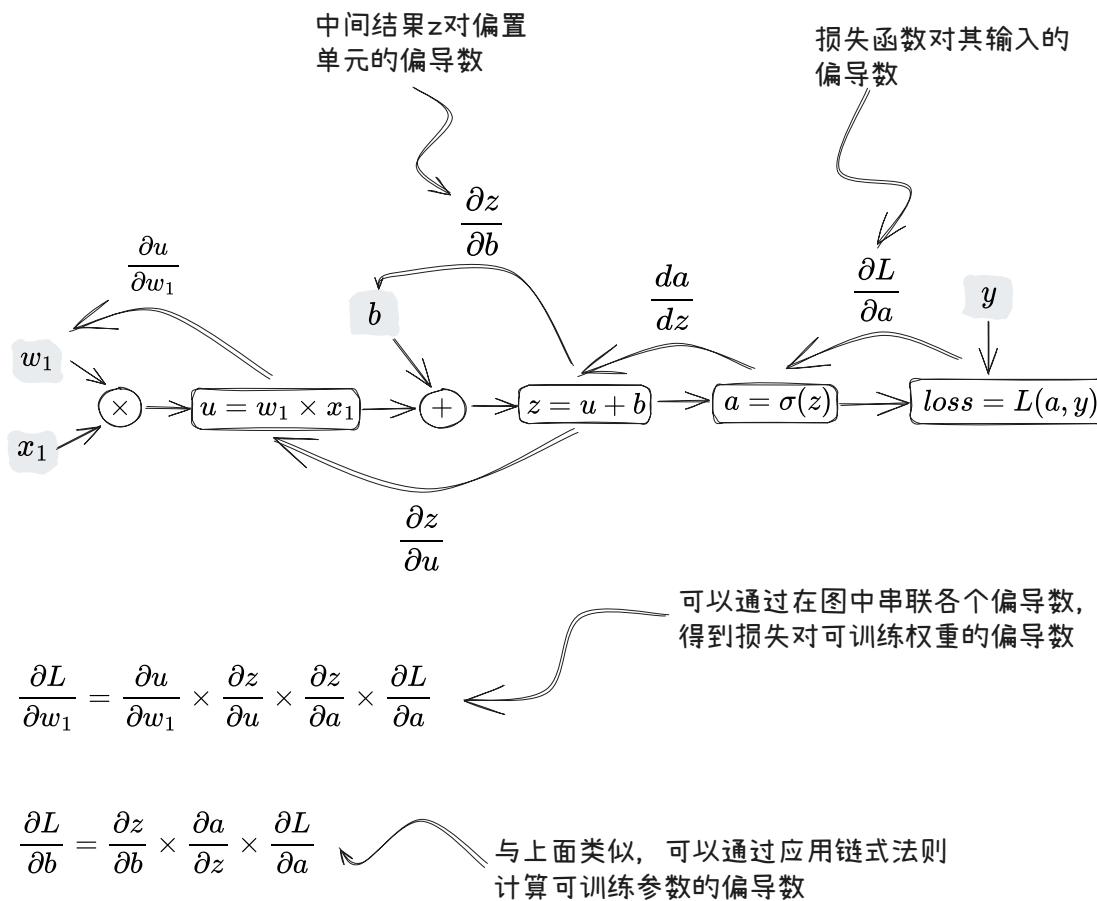


图 7.6 在计算图中计算损失梯度的最常见方法是从右向左应用链式法则，这也称为“反向模型自动求导”或“反向传播”。我们从输出层（或损失本身）开始，向后通过网络一直到输入层。这么做是为了计算损失相对于网络中每个参数（权重和偏置）的梯度，从而为训练过程中如何更新这些参数提供信息

偏导数和梯度

上图展示了偏导数，它测量的是一个函数相对于其中一个变量变化的速率。梯度是一个向量，包含了一个多变量函数（输入变量超过一个的函数）的所有偏导数。

如果你不太熟悉或记不清微积分中的偏导数、梯度或链式法则，不用担心。从高层次来看，你只需要知道链式法则如何在计算图中根据模型参数来计算损失函数的梯度即可。这提供了更新每个参数以最小化损失函数所需的信息，而这个损失函数作为衡量模型性能的代理，可以通过诸如梯度下降之类的方法来实现。我们将在后面重新审视在 PyTorch 中实现这一训练循环的计算过程。

那么，这一切与之前提到的 PyTorch 库的第二个组件——自动微分（`autograd`）引擎有何关联？PyTorch 的 `autograd` 引擎在后台通过跟踪在张量上执行的每个操作来构建计算图。然后，通过调用 `grad` 函数，可以计算损失相对于模型参数 w_1 的梯度，如下面的代码所示。

通过 `autograd` 计算梯度

Python

```

1 import torch.nn.functional as F
2 from torch.autograd import grad
3
4 y = torch.tensor([1.0])
5 x1 = torch.tensor([1.1])
6 w1 = torch.tensor([2.2], requires_grad=True)

```

```

7 b = torch.tensor([0.0], requires_grad=True)
8
9 z = x1 * w1 + b
10 a = torch.sigmoid(z)
11
12 loss = F.binary_cross_entropy(a, y)
13
14 # 默认情况下, PyTorch在计算梯度后会销毁计算图以释放内存。然而, 由于我们即将再次使用这个计算图, 因此可以设置`retain_graph=True`, 使其保留在内存中
15 grad_L_w1 = grad(loss, w1, retain_graph=True)
16 grad_L_b = grad(loss, b, retain_graph=True)

```

给定模型参数的损失结果值如下所示:

```

1 print(grad_L_w1)
2 print(grad_L_b)

```

这将打印如下内容:

```

1 (tensor([-0.0898]),)
2 (tensor([-0.0817]),)

```

这里我们手动使用了 `grad` 函数, 这在实验、调试和概念演示中很有用。但是, 在实际操作中, PyTorch 提供了更高级的工具来自动化这个过程。例如, 我们可以对损失函数调用 `.backward` 方法, 随后 PyTorch 将计算计算图中所有叶节点的梯度, 这些梯度将通过张量的 `.grad` 属性进行存储:

```

1 loss.backward()
2 print(w1.grad)
3 print(b.grad)

```

输出如下所示:

```

1 (tensor([-0.0898]),)
2 (tensor([-0.0817]),)

```

我给你提供了很多信息, 你可能会被微积分的概念弄得有些不知所措, 但不用担心。虽然这些微积分术语是为了解释 PyTorch 的 `autograd` 组件, 但你需要记住的仅仅是 PyTorch 通过 `.backward` 方法为我们处理了微积分问题——我们不需要手动计算任何导数或梯度。

手动计算一下梯度验证

- 前向过程, 将中间计算结果都保存下来

$$\begin{aligned}
 z &= w_1 x_1 + b = 2.2 \times 1.1 + 0.0 \\
 a &= \sigma(z) = \sigma(2.42) = \frac{1}{1 + e^{-2.42}} \approx 0.9183 \\
 \mathcal{L} &= -[y \log(a) + (1 - y) \log(1 - a)] \\
 &= -[1.0 \times \log(0.9183) + (1 - 1.0) \times \log(1 - 0.9183)] \\
 &= -[1.0 \times \log(0.9183) + 0] \\
 &= -\log(0.9183) \approx 0.0853
 \end{aligned} \tag{7.4}$$

- 反向过程, 利用前向过程保存的中间结果, 计算 w_1 和 b 的梯度 (偏导数)

🔥 用到的求导公式

对数求导法则

$$\begin{aligned}\frac{d \log x}{dx} &= \frac{1}{x} \\ \frac{d \log(f(x))}{dx} &= \frac{f'(x)}{f(x)}\end{aligned}\tag{7.5}$$

商法则：若 $f(x) = \frac{u(x)}{v(x)}$

$$f'(x) = \frac{u'(x)v(x) - u(x)v'(x)}{(v(x))^2}\tag{7.6}$$

自然对数

$$\frac{de^x}{dx} = e^x\tag{7.7}$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_1}\tag{7.8}$$

其中

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial a} &= -\left[\frac{y}{a} + (1-y)\frac{-1}{1-a}\right] \quad (\frac{d \log(x)}{dx} = \frac{1}{x}) \\ &= \frac{1-y}{1-a} - \frac{y}{a} \quad (\text{y}=1.0) \\ &= -\frac{1.0}{0.9183} \quad (\text{利用中间计算结果 } a=0.9183)\end{aligned}\tag{7.9}$$

而

$$\begin{aligned}\frac{\partial a}{\partial z} &= \frac{\partial \sigma(z)}{\partial z} = \frac{\partial \frac{1}{1+e^{-z}}}{\partial z} \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \frac{1}{1+e^{-z}} \cdot \left(1 - \frac{1}{1+e^{-z}}\right) \\ &= \sigma(z)(1-\sigma(z)) \\ &= a(1-a) \quad (\text{利用中间计算结果 } a=0.9183) \\ &= 0.9183 \times (1 - 0.9183)\end{aligned}\tag{7.10}$$

显然

$$\frac{\partial z}{\partial w_1} = x_1 = 1.1\tag{7.11}$$

所以

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_1} &= \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w_1} \\ &= \left(-\frac{1.0}{0.9183}\right) \times 0.9183 \times (1 - 0.9183) \times 1.1 \\ &= -0.0898\end{aligned}\tag{7.12}$$

🔥 作业

按照上面的过程手动计算一下 $\frac{\partial \mathcal{L}}{\partial b}$

7.5 实现多层神经网络

接下来，我们将 PyTorch 视为实现深度神经网络的库来进行重点探讨。为了提供一个具体的例子，我们来看看多层感知机（multilayer perceptron），即全连接神经网络，如下图所示。

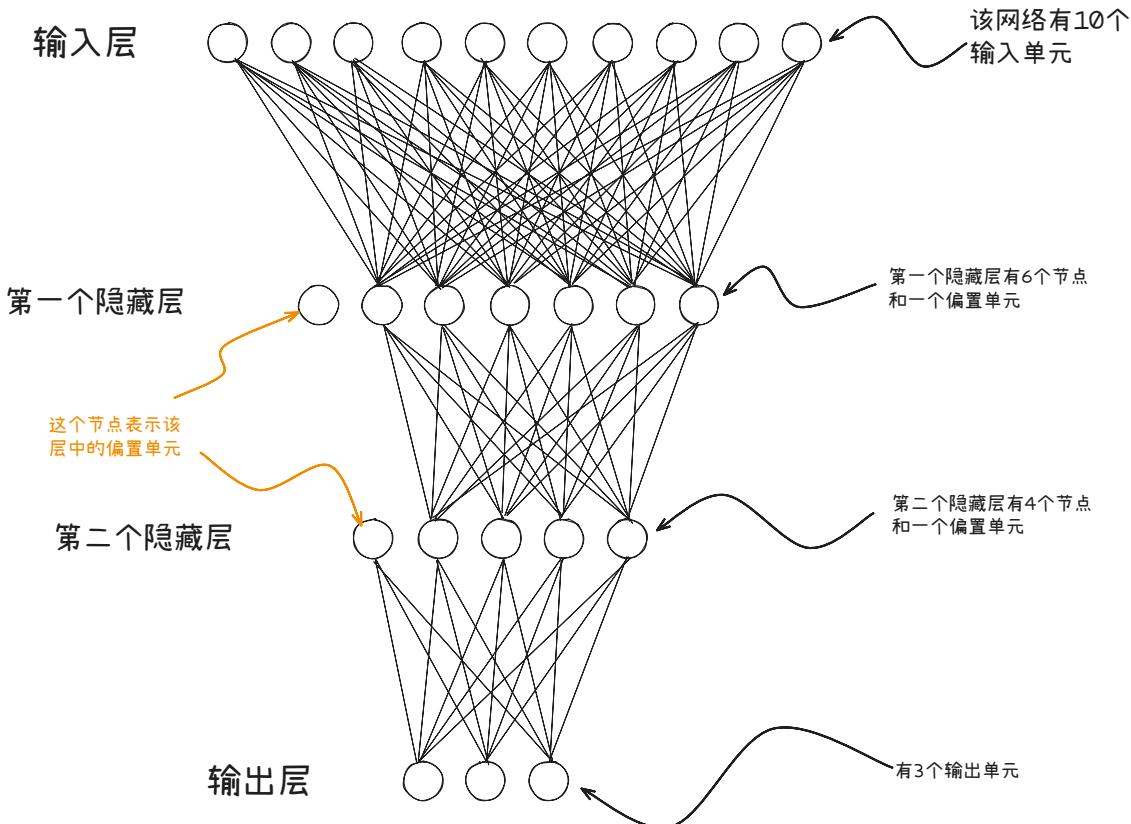


图 7.7 一个具有两个隐藏层的多层感知机。每个节点表示各自层中的一个单元。为了方便展示，这里每层都只有几个节点

在 PyTorch 中实现神经网络时，可以通过子类化 `torch.nn.Module` 类来定义我们自己的自定义网络架构。这个 `Module` 基类提供了很多功能，使得构建和训练模型变得更加容易。例如，它允许我们封装层和操作，并跟踪模型的参数。

在这个子类中，我们在 `__init__` 构造函数中定义网络层，并在 `forward` 方法中指定层与层之间的交互。`forward` 方法描述了输入数据如何通过网络传递，并形成计算图。相比之下，`backward` 方法通常不需要我们自己实现，它在训练期间用于计算给定模型参数的损失函数的梯度。下面的代码通过实现一个具有两个隐藏层的经典的多层感知机展示了 `Module` 类的典型用法。

一个具有两个隐藏层的多层感知机

Python

```

1  class NeuralNetwork(torch.nn.Module):
2      # 将输入和输出的数量编码为变量，使我们可以在具有不同特征数量和类别数量的数据集上重复使用相同的
3      # 代码
4      def __init__(self, num_inputs, num_outputs):
5          super().__init__()

```

```

5
6     self.layers = torch.nn.Sequential(
7         # 第一个隐藏层
8         torch.nn.Linear(num_inputs, 30), # 线性层将输入节点和输出节点的数量作为参数
9         torch.nn.ReLU(), # 非线性激活函数被放置在隐藏层之间
10        # 第二个隐藏层
11        torch.nn.Linear(30, 20), # 一个隐藏层的输出节点数量必须与下一层的输入节点数量相
12        # 匹配
13        torch.nn.ReLU(),
14        # 输出层
15        torch.nn.Linear(20, num_outputs),
16    )
17
18    def forward(self, x):
19        logits = self.layers(x)
20
21    return logits # 最后一层的输出称为logits

```

然后，可以像下面这样实例化一个新的神经网络对象：

```
1 model = NeuralNetwork(50, 3)
```

Python

在使用这个新模型对象之前，可以调用 `print` 函数来查看模型结构的摘要：

```
1 print(model)
```

Python

这将打印以下内容：

```

1 NeuralNetwork(
2     (layers): Sequential(
3         (0): Linear(in_features=50, out_features=30, bias=True)
4         (1): ReLU()
5         (2): Linear(in_features=30, out_features=20, bias=True)
6         (3): ReLU()
7         (4): Linear(in_features=20, out_features=3, bias=True)
8     )
9 )

```

Python

请注意，在实现 `NeuralNetwork` 类时，我们使用了 `Sequential` 类。虽然 `Sequential` 并非必需，但如果有一系列想要按特定顺序执行的层（正如本例中的情况），那么使用它可以让我们的工作更轻松。因此，在 `__init__` 构造函数中实例化 `self.layers = Sequential(...)` 后，只需在 `NeuralNetwork` 的 `forward` 方法中调用 `self.layers`，而无须单独调用每个层。

接下来，检查一下该模型的可训练参数总数：

```
1 num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
2 print("Total number of trainable model parameters:", num_params)
```

Python

这将打印以下内容：

```
1 Total number of trainable model parameters: 2213
```

Python

每一个 `requires_grad=True` 的参数都会被视为可训练参数，并在训练期间进行更新。

对于前面我们提到的具有两个隐藏层的神经网络模型，这些可训练参数包含在 `torch.nn.Linear` 层中。`Linear` 层会将输入与权重矩阵相乘，并加上一个偏置向量。这有时被称为前馈层或全连接层。

基于这里执行的 `print(model)` 调用，可以看到第一个 `Linear` 层在 `layers` 属性中的索引位置是 0。可以通过以下方式访问对应的权重参数矩阵：

```
1 print(model.layers[0].weight)
```

这将打印以下内容：

```
1 Parameter containing:
2 tensor([[ 0.1174, -0.1350, -0.1227, ... , 0.0275, -0.0520, -0.0192],
3         [-0.0169,  0.1265,  0.0255, ... , -0.1247,  0.1191, -0.0698],
4         [-0.0973, -0.0974, -0.0739, ... , -0.0068, -0.0892,  0.1070],
5         ... ,
6         [-0.0681,  0.1058, -0.0315, ... , -0.1081, -0.0290, -0.1374],
7         [-0.0159,  0.0587, -0.0916, ... , -0.1153,  0.0700,  0.0770],
8         [-0.1019,  0.1345, -0.0176, ... ,  0.0114, -0.0559, -0.0088]],,
9         requires_grad=True)
```

由于这个大矩阵未完全显示出来，因此我们使用 `.shape` 属性来查看其维度：

```
1 print(model.layers[0].weight.shape)
```

结果如下所示：

```
1 torch.Size([30, 50])
```

(同样，可以通过 `model.layers[0].bias` 访问偏置向量。)

这里的权重矩阵是一个 30×50 的矩阵，可以看到 `requires_grad` 被设置为 `True`（意味着该矩阵是可训练的）——这是 `torch.nn.Linear` 中权重和偏置的默认设置。

如果你在自己的计算机上执行前面的代码，那么权重矩阵中的数值可能会与本书展示的有所不同。模型权重会用小的随机数进行初始化，每次实例化网络时这些数值都会不同。在深度学习中，使用小的随机数初始化模型权重是为了在训练过程中打破对称性。否则，各节点将执行相同的操作并在反向传播过程中进行相同的更新，导致网络无法学习从输入到输出的复杂映射关系。

然而，虽然我们希望继续使用小的随机数作为层权重的初始值，但可以通过 `manual_seed` 来为 PyTorch 的随机数生成器设定种子，从而使随机数初始化可重复：

```
1 torch.manual_seed(123)
2 model = NeuralNetwork(50, 3)
3 print(model.layers[0].weight)
```

结果如下所示：

```
1 Parameter containing:
2 tensor([[-0.0577,  0.0047, -0.0702, ... , 0.0222,  0.1260,  0.0865],
3         [ 0.0502,  0.0307,  0.0333, ... , 0.0951,  0.1134, -0.0297],
4         [ 0.1077, -0.1108,  0.0122, ... , 0.0108, -0.1049, -0.1063],
5         ... ,
6         [-0.0787,  0.1259,  0.0803, ... , 0.1218,  0.1303, -0.1351],
7         [ 0.1359,  0.0175, -0.0673, ... , 0.0674,  0.0676,  0.1058],
8         [ 0.0790,  0.1343, -0.0293, ... , 0.0344, -0.0971, -0.0509]],,
9         requires_grad=True)
```

现在我们已经花了一些时间检查 `NeuralNetwork` 实例，接下来简单看看如何通过前向传播使用它：

```
1 torch.manual_seed(123)
2 X = torch.rand((1, 50))
3 out = model(X)
```

```
4 print(out)
```

结果如下所示：

```
1 tensor([[-0.1262, 0.1080, -0.1792]], grad_fn=<AddmmBackward0>)
```

 Python

在上述代码中，我们生成了一个单一的随机训练样本 X 作为示例输入（注意，我们的网络期望接收 50 维的特征向量），并将其输入模型，从而得到了 3 个分数。当我们调用 `model(x)` 时，它会自动执行模型的前向传播。

前向传播是指从输入张量开始到计算获得输出张量的过程。这一过程包括将输入数据从输入层开始，经由隐藏层，最后传递至输出层，贯穿整个神经网络的所有层次。

结果中返回的 3 个数值对应于分配给每个输出节点的分数。注意输出张量还包含了一个 `grad_fn` 值。

这里，`grad_fn=<AddmmBackward0>` 表示计算图中用于计算某个变量的最后一个函数。具体来说，`grad_fn=<AddmmBackward0>` 意味着我们正在查看的张量是通过矩阵乘法和加法操作创建的。`PyTorch` 会在反向传播期间使用这些信息来计算梯度。`grad_fn=<AddmmBackward0>` 中的 `<AddmmBackward0>` 指定了执行的操作。在这种情况下，它执行的是一个 `Addmm` 操作。`Addmm` 代表的是矩阵乘法（`mm`）后接加法（`Add`）的组合运算。

如果只想使用网络进行预测而不进行训练或反向传播（比如在训练之后使用它进行预测），那么为反向传播构建这个计算图可能会浪费资源，因为它会执行不必要的计算并消耗额外的内存。因此，当使用模型进行推理（比如做出预测）而不是训练时，最好的做法是使用 `torch.no_grad()` 上下文管理器。这会告诉 `PyTorch` 无须跟踪梯度，从而可以显著节省内存和计算资源：

```
1 with torch.no_grad():
2     out = model(X)
3     print(out)
```

 Python

结果如下所示：

```
1 tensor([[-0.1262, 0.1080, -0.1792]])
```

 Python

在 `PyTorch` 中，通常的做法是让模型返回最后一层的输出（`logits`），而不将这些输出传递给非线性激活函数。这是因为 `PyTorch` 常用的损失函数会将 `softmax`（或二分类时的 `sigmoid`）操作与负对数似然损失结合在一个类中。这样做是为了提高数值计算的效率和稳定性。因此，如果想为预测结果计算类别成员概率，那么就需要显式调用 `softmax` 函数：

```
1 with torch.no_grad():
2     out = torch.softmax(model(X), dim=1)
3     print(out)
```

 Python

这将打印以下内容：

```
1 tensor([[0.3113, 0.3934, 0.2952]]))
```

 Python

现在这些值可以解释为类别成员的概率，并且它们的总和大约为 1。对于这个随机输入，这些值大致相等，这是未经过训练的随机初始化模型的预期结果。

7.6 设置高效的数据加载器

在我们能够训练模型之前，必须简要讨论如何在 `PyTorch` 中创建高效的数据加载器，这些加载器将在训练过程中被迭代使用。`PyTorch` 中数据加载的整体思路如图所示。

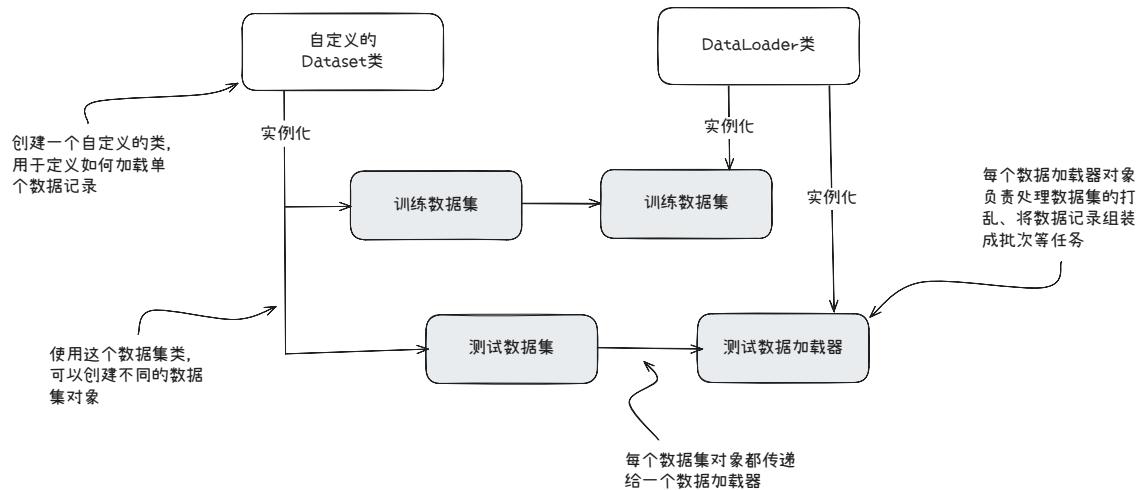


图 7.8 PyTorch 实现了 `Dataset` 类和 `DataLoader` 类。`Dataset` 类用于实例化定义如何加载每条数据记录的对象。`DataLoader` 类负责处理数据的打乱和组装成批次

根据上图，我们将实现一个自定义的 `Dataset` 类，用于创建训练数据集和测试数据集，然后再用这些数据集创建数据加载器。我们首先创建一个简单的示例数据集，其中包含 5 个训练示例，每个示例有两个特征。与训练示例一起，我们还创建了一个包含相应类别标签的张量：3 个示例属于类别标签 0，两个示例属于类别标签 1。此外，我们还构建了一个包含两个样本的测试集。创建此数据集的代码如代码所示。

```
创建一个小型示例数据集
1 X_train = torch.tensor([
2     [-1.2, 3.1],
3     [-0.9, 2.9],
4     [-0.5, 2.6],
5     [2.3, -1.1],
6     [2.7, -1.5]
7 ])
8 y_train = torch.tensor([0, 0, 0, 1, 1])
9
10 X_test = torch.tensor([
11     [-0.8, 2.8],
12     [2.6, -1.6],
13 ])
14 y_test = torch.tensor([0, 1])
```

注意

PyTorch 要求类别标签从标签 0 开始，并且最大的类别标签值不得超过输出节点数减 1（因为 Python 的索引从 0 开始）。因此，如果我们有类别标签 0、1、2、3 和 4，那么神经网络的输出层应包含 5 个节点。

接下来，我们通过继承 PyTorch 的 `Dataset` 父类来创建一个自定义数据集类 `ToyDataset`，如代码所示。

```
定义一个自定义的Dataset类
1 from torch.utils.data import Dataset
```

```

2
3  class ToyDataset(Dataset):
4      def __init__(self, X, y):
5          self.features = X
6          self.labels = y
7
8      def __getitem__(self, index):
9          """检索一条数据记录及其对应标签的说明"""
10         one_x = self.features[index]
11         one_y = self.labels[index]
12         return one_x, one_y
13
14     def __len__(self):
15         """返回数据集总长度的说明"""
16         return self.labels.shape[0]
17
18 train_ds = ToyDataset(X_train, y_train)
19 test_ds = ToyDataset(X_test, y_test)

```

这个自定义的 `ToyDataset` 类的目的是实例化一个 PyTorch `DataLoader`。在进行这一步之前，让我们先简要回顾一下 `ToyDataset` 代码的一般结构。

在 PyTorch 中，自定义的 `Dataset` 类的 3 个主要组成部分是 `__init__` 方法、`__getitem__` 方法和 `__len__` 方法。在 `__init__` 方法中，我们设置一些可以在 `__getitem__` 方法和 `__len__` 方法中访问的属性。这些属性可以是文件路径、文件对象、数据库连接器等。由于我们创建了一个位于内存中的张量数据集，因此只需将 `X` 和 `y` 分配给这些代表张量对象的占位符属性即可。

在 `__getitem__` 方法中，我们定义了通过索引返回数据集中单个项目的具体指令。这指的是与单个训练示例或测试实例对应的特征和类别标签。（数据加载器将提供这个索引，稍后我们会介绍。）

最后，`__len__` 方法包含了检索数据集长度的指令。在这里，我们使用张量的 `.shape` 属性来返回特征数组中的行数。就训练数据集而言，我们有 5 行数据，下面可以再次确认一下：

```
1 print(len(train_ds))
```

结果如下所示。

```
1 5
```

现在我们已经定义了一个可用于示例数据集的 PyTorch `Dataset` 类，我们可以使用 PyTorch 的 `DataLoader` 类从中进行采样，如代码所示。

```

实例化数据加载器
1 from torch.utils.data import DataLoader
2
3 torch.manual_seed(123)
4
5 # 之前创建的示例数据集实例作为数据加载器的输入
6 train_loader = DataLoader(
7     dataset=train_ds,
8     batch_size=2,
9     shuffle=True, # 是否打乱数据
10    num_workers=0 # 后台进程的数量
11 )
12
13 test_loader = DataLoader(

```

```

14     dataset=test_ds,
15     batch_size=2,
16     shuffle=False, # 测试数据集无须打乱顺序
17     num_workers=0
18 )

```

在实例化训练数据加载器后，可以对其进行迭代。对 `test_loader` 的迭代与之类似，但为简洁起见，这里省略了具体细节：

```

1 for idx, (x, y) in enumerate(train_loader):
2     print(f"Batch {idx+1}:", x, y)

```

结果如下所示：

```

1 Batch 1: tensor([[-1.2000, 3.1000],
2                   [-0.5000, 2.6000]]) tensor([0, 0])
3 Batch 2: tensor([[ 2.3000, -1.1000],
4                   [-0.9000, 2.9000]]) tensor([1, 0])
5 Batch 3: tensor([[ 2.7000, -1.5000]]) tensor([1])

```

根据前面的输出，可以看到 `train_loader` 迭代了训练数据集，每个训练示例正好访问一次。这被称为一个训练轮次。由于我们使用 `torch.manual_seed(123)` 设置了随机数生成器，因此你应该得到完全相同的训练示例打乱顺序。然而，当你再次迭代数据集时，你会发现打乱的顺序已经发生变化。这是为了防止深度神经网络在训练过程中陷入重复更新循环。

我们在这里指定的批次大小为 2，但第三批次仅包含一个示例。这是因为我们有 5 个训练示例，而 5 不能被 2 整除。

在实践中，如果一个训练轮次的最后一个批次显著小于其他批次，那么可能会影响训练过程中的收敛。为此，可以设置 `drop_last=True`，这将在每轮中丢弃最后一个批次，如代码所示。

一个丢弃最后一个批次的训练加载器

```

1 train_loader = DataLoader(
2     dataset=train_ds,
3     batch_size=2,
4     shuffle=True,
5     num_workers=0,
6     drop_last=True
7 )

```

现在，迭代训练加载器，可以看到最后一个批次被省略了：

```

1 for idx, (x, y) in enumerate(train_loader):
2     print(f"Batch {idx+1}:", x, y)

```

结果如下所示。

```

1 Batch 1: tensor([[-0.9000, 2.9000],
2                   [ 2.3000, -1.1000]]) tensor([0, 1])
3 Batch 2: tensor([[ 2.7000, -1.5000],
4                   [-0.5000, 2.6000]]) tensor([1, 0])

```

最后，我们来讨论 `DataLoader` 中的 `num_workers=0` 设置。这个参数在 PyTorch 的 `DataLoader` 函数中对于并行加载和预处理数据至关重要。当 `num_workers` 设置为 0 时，数据加载将在主进程中而不是单独的工作进程中进行。这看起来似乎没有问题，但在使用 GPU 训练较大的网络时，这可能会导致模型训练显著减慢。这是因为 CPU 不仅要处理深度学习模型，还要花时间加载和预处理数据。

因此，GPU 在等待 CPU 完成这些任务时可能会闲置。相反，当 `num_workers` 设置为大于 0 的数值时，会启动多个工作进程并行加载数据，从而释放主进程专注于训练模型，并更好地利用系统资源。

然而，如果你处理的是非常小的数据集，那么可能并不需要将 `num_workers` 设置为 1 或更大的数值，因为总训练时间只需几秒。因此，如果你使用的是小型数据集或交互式环境（如 Jupyter Notebook），那么增加 `num_workers` 可能不会显著提高速度，反而会导致一些问题。一个潜在的问题是，启动多个工作进程的开销可能会比实际数据加载所需的时间更长，尤其是数据集很小的时候。

此外，对于 Jupyter Notebook，将 `num_workers` 设置为大于 0 有时可能会导致不同进程之间资源共享的问题，从而引发错误或导致笔记本崩溃。因此，理解这种权衡并对 `num_workers` 参数进行合理设置是非常重要的。如果使用得当，这可以成为一个有益的工具，但应根据你的特定数据集大小和计算环境进行调整，以获得最佳效果。

根据我的经验，设置 `num_workers=4` 通常会在许多真实世界数据集上获得最佳性能，但最佳设置取决于你的硬件和用于加载 `Dataset` 类中训练示例的代码。

7.7 典型的训练循环

现在让我们在示例数据集上训练一个神经网络。代码展示了训练过程。

在PyTorch中进行神经网络训练

Python

```

1 import torch.nn.functional as F
2
3 torch.manual_seed(123)
4 model = NeuralNetwork(num_inputs=2, num_outputs=2)
5 optimizer = torch.optim.SGD(
6     model.parameters(), lr=0.5
7 )
8
9 num_epochs = 3
10 for epoch in range(num_epochs):
11     model.train()
12     for batch_idx, (features, labels) in enumerate(train_loader):
13         logits = model(features)
14         loss = F.cross_entropy(logits, labels)
15         optimizer.zero_grad()
16         loss.backward()
17         optimizer.step()
18
19     ### LOGGING
20     print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
21          f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
22          f" | Train Loss: {loss:.2f}")
23
24 model.eval()
25 # 插入可选的模型评估代码

```

运行上述代码会产生以下输出：

Python

```

1 Epoch: 001/003 | Batch 000/002 | Train Loss: 0.75
2 Epoch: 001/003 | Batch 001/002 | Train Loss: 0.65
3 Epoch: 002/003 | Batch 000/002 | Train Loss: 0.44
4 Epoch: 002/003 | Batch 001/002 | Train Loss: 0.13
5 Epoch: 003/003 | Batch 000/002 | Train Loss: 0.03
6 Epoch: 003/003 | Batch 001/002 | Train Loss: 0.00

```

如你所见，损失在 3 轮后降至 0，这表明模型已经在训练集上收敛。这里初始化了一个具有两个输入和两个输出的模型，因为我们的示例数据集有两个输入特征和两个类别标签需要预测。我们使用了一个学习率 (`lr`) 为 0.5 的随机梯度下降 (SGD) 优化器。学习率是一个超参数，意味着这是可调的设置，我们必须根据观察到的损失进行实验。理想情况下，我们希望选择一个学习率，使得损失在一定轮数后收敛——轮数是另一个需要选择的超参数。

🔥 练习

代码中介绍的神经网络有多少个参数？

在实际操作中，我们通常会使用第三个数据集，即所谓“验证数据集”，来找到最优的超参数设置。验证集类似于测试集。然而，虽然我们只想精确地使用一次测试集以避免评估偏差，但通常会多次使用验证集来调整模型设置。

我们还引入了新的设置：`model.train()` 和 `model.eval()`。顾名思义，这些设置用于将模型置于训练模式或评估模式。这对于在训练和推断过程中具有不同行为的组件（如 `dropout` 或批归一化层）是必要的。由于我们的 `NeuralNetwork` 类中没有受到这些设置影响的 `dropout` 或其他组件，因此在之前的代码中并未使用 `model.train()` 和 `model.eval()`。然而，最好还是包含这些设置，以避免在更改模型架构或重用代码训练其他模型时出现意外行为。

正如之前讨论的那样，我们直接将 `logits` 传递给 `cross_entropy` 损失函数，后者会在内部应用 `softmax` 函数，以提高效率并增强数值稳定性。接下来，调用 `loss.backward()` 会计算由 PyTorch 在后台构建的计算图中的梯度。`optimizer.step()` 方法会利用这些梯度来更新模型参数以最小化损失。对 SGD 优化器而言，这意味着将梯度与学习率相乘，然后将缩放后的负梯度加到参数上。

⚡ 危险

为了避免不必要的梯度累积，确保在每次更新中调用 `optimizer.zero_grad()` 来将梯度重置为 0，这很重要。否则，梯度会逐渐累积起来，这往往是我们不愿意见到的。

在训练好模型后，可以使用它进行预测：

```
1 model.eval()  
2 with torch.no_grad():  
3     outputs = model(X_train)  
4 print(outputs)
```

Python

结果如下所示：

```
1 tensor([[ 2.8569, -4.1618],  
2         [ 2.5382, -3.7548],  
3         [ 2.0944, -3.1820],  
4         [-1.4814,  1.4816],  
5         [-1.7176,  1.7342]])
```

Python

为了获得类别成员概率，可以使用 PyTorch 的 `softmax` 函数：

```
1 torch.set_printoptions(sci_mode=False)  
2 probas = torch.softmax(outputs, dim=1)  
3 print(probas)
```

Python

输出如下所示：

```
1 tensor([[ 0.9991,    0.0009],  
2         [ 0.9982,    0.0018],  
3         [ 0.9949,    0.0051],
```

Python

```
4      [    0.0491,    0.9509],
5      [    0.0307,    0.9693]])
```

来看一下上面代码输出的第 1 行。在这里，第一个值（列）表示该训练示例属于类别标签 0 的概率为 99.91%，属于类别标签 1 的概率为 0.09%。（这里使用 `set_printoptions` 是为了让输出更加易读。）

可以使用 PyTorch 的 `argmax` 函数将这些概率值转换为类别标签预测。如果设置 `dim=1`，它将返回每行中最大值的索引位置（设置 `dim=0` 则返回每列中最大值的索引位置）：

```
1 predictions = torch.argmax(probas, dim=1)
2 print(predictions)
```

这将打印如下内容：

```
1 tensor([0, 0, 0, 1, 1])
```

请注意，为了获得类别标签，计算 `softmax` 概率并非必需步骤，也可以直接对 `logits`（输出）应用 `argmax` 函数：

```
1 predictions = torch.argmax(outputs, dim=1)
2 print(predictions)
```

输出如下所示：

```
1 tensor([0, 0, 0, 1, 1])
```

在这里，我们计算了训练数据集的预测标签。鉴于训练数据集相对较小，我们可以通过肉眼将其与真实的训练标签进行比较，结果显示模型预测的准确率为 100%。可以使用比较运算符 `=` 来再次确认这一点：

```
1 predictions = y_train
```

结果如下所示：

```
1 tensor([True, True, True, True, True])
```

使用 `torch.sum` 可以计算正确预测的数量：

```
1 torch.sum(predictions == y_train)
```

输出如下所示：

```
1 5
```

由于数据集由 5 个训练示例组成，我们的 5 个预测全部正确，准确率为 $\frac{5}{5} \times 100\% = 100\%$ 。

为了使预测准确率的计算更加通用，让我们实现一个 `compute_accuracy` 函数，如代码所示。

一个计算预测准确率的函数

```
1 def compute_accuracy(model, dataloader):
2     model = model.eval()
3     correct = 0.0
4     total_examples = 0
5
6     for idx, (features, labels) in enumerate(dataloader):
7         with torch.no_grad():
8             logits = model(features)
9
10            predictions = torch.argmax(logits, dim=1)
11            compare = labels == predictions # 根据标签是否匹配，返回一个True/False值的张量
```

```
12     correct += torch.sum(compare) # 求和操作计算True值的数量
13     total_examples += len(compare)
14     # 正确预测的比例是一个介于0和1之间的值。调用`.item()`会将张量的值以Python浮点数的形式返回
15     return (correct / total_examples).item()
```

这段代码通过迭代数据加载器来计算正确预测的数量和比例。在处理大规模数据集时，由于内存限制，通常我们只能对数据集的一小部分调用模型。这里的 `compute_accuracy` 函数是一种通用方法，适用于任意大小的数据集，因为在每次迭代中，模型所接收的数据集块的大小与训练期间的批次大小相同。`compute_accuracy` 函数的内部逻辑类似于我们之前将 `logits` 转换为类别标签时使用的方法。

接下来，可以将该函数应用于训练数据：

```
1 print(compute_accuracy(model, train_loader))
```

Python

结果如下所示：

```
1 1.0
```

类似地，可以在测试集上应用这个函数：

```
1 print(compute_accuracy(model, test_loader))
```

Python

这将打印如下内容。

```
1 1.0
```

7.8 保存和加载模型

现在我们的模型已经训练好了，接下来看看如何保存它，以便以后可以重用。下面是在 PyTorch 中保存和加载模型的推荐方法：

```
1 torch.save(model.state_dict(), "model.pth")
```

Python

模型的 `state_dict` 是一个 Python 字典对象，它可以将模型中的每一层映射到其可训练参数（权重和偏置）。`model.pth` 是保存到磁盘的模型文件的任意文件名，我们可以使用任何名称和文件后缀，不过 `.pth` 和 `.pt` 是最常见的约定。

保存模型后，可以从磁盘中恢复它：

```
1 model = NeuralNetwork(2, 2)
2 model.load_state_dict(torch.load("model.pth"))
```

Python

`torch.load("model.pth")` 函数读取文件 `model.pth`，并重建包含模型参数的 Python 字典对象，`model.load_state_dict()` 则将这些参数应用到模型中，有效地恢复了我们保存模型时模型的学习状态。

在同一会话中执行此代码时，`model = NeuralNetwork(2, 2)` 这一行并不是严格必需的。然而，这里包含它是为了说明我们需要在内存中拥有一个模型的实例，这样才能应用保存的参数。此外，`NeuralNetwork(2, 2)` 的架构必须与最初保存的模型完全匹配。

7.9 使用 GPU 优化训练性能

接下来，让我们探讨如何利用 GPU 来加速深度神经网络的训练。（相较于普通 CPU，GPU 能够显著提升训练速度。）首先，我们将了解 PyTorch 中 GPU 计算的主要概念。然后，我们将在单个 GPU 上训练模型。最后，我们将讨论如何使用多个 GPU 进行分布式训练。

7.9.1 在 GPU 设备上运行 PyTorch

修改训练循环使其便于在 GPU 上运行相对简单，只需更改 3 行代码即可。在进行这些修改之前，理解 PyTorch 中 GPU 计算的主要概念非常重要。在 PyTorch 中，设备是执行计算和存储数据的地方。CPU 和 GPU 是设备的示例。如果一个 PyTorch 张量存放在某个设备上，那么其操作也会在同一个设备上执行。

来看一下这一过程是如何进行的。假设你已经安装了兼容 GPU 的 PyTorch 版本，可以使用以下代码再次检查一下你的运行环境是否真的支持 GPU 计算：

```
1 print(torch.cuda.is_available())
```

结果如下所示：

```
1 True
```

现在，假设我们有两个张量可以相加。默认情况下，这个计算将在 CPU 上执行：

```
1 tensor_1 = torch.tensor([1., 2., 3.])
2 tensor_2 = torch.tensor([4., 5., 6.])
3 print(tensor_1 + tensor_2)
```

输出如下所示：

```
1 tensor([5., 7., 9.])
```

现在可以使用`.to()`方法。这个方法与我们用来更改张量数据类型的方法相同，它能够将这些张量转移到 GPU 上并在那里执行加法操作：

```
1 tensor_1 = tensor_1.to("cuda")
2 tensor_2 = tensor_2.to("cuda")
3 print(tensor_1 + tensor_2)
```

输出如下所示：

```
1 tensor([5., 7., 9.], device='cuda:0')
```

生成的张量现在包含了设备信息`device='cuda:0'`，这意味着这些张量位于第一个 GPU 上。如果你的机器有多个 GPU，那么可以指定要将张量转移到哪个 GPU 上。这可以通过在传输命令中指定设备 ID 来实现，比如使用`.to("cuda:0")`、`.to("cuda:1")`等命令。

然而，所有的张量必须位于同一个设备上。否则，如果一个张量位于 CPU，另一个张量位于 GPU，计算就会失败：

```
1 tensor_1 = tensor_1.to("cpu")
2 print(tensor_1 + tensor_2)
```

结果如下所示：

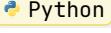
```
1 RuntimeError      Traceback (most recent call last)
2 <ipython-input-7-4ff3c4d20fc3> in <cell line: 2>()
3     1 tensor_1 = tensor_1.to("cpu")
4 ----> 2 print(tensor_1 + tensor_2)
5 RuntimeError: Expected all tensors to be on the same device, but found at
6 least two devices, cuda:0 and cpu!
```

总之，只需要将张量传输到同一个 GPU 设备上，PyTorch 会处理其余的工作。

7.9.2 单个 GPU 训练

我们已经熟悉了将张量传输到 GPU 的过程，现在可以修改训练循环以在 GPU 上运行。这一步仅需要更改 3 行代码，如代码所示。

GPU 上的训练循环



```

1 torch.manual_seed(123)
2 model = NeuralNetwork(num_inputs=2, num_outputs=2)
3
4 device = torch.device("cuda") # 定义一个默认使用GPU的设备变量
5 model = model.to(device) # 将模型转移到GPU上
6
7 optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
8
9 num_epochs = 3
10
11 for epoch in range(num_epochs):
12     model.train()
13     for batch_idx, (features, labels) in enumerate(train_loader):
14         # 将数据转移到GPU上
15         features, labels = features.to(device), labels.to(device)
16         logits = model(features)
17         loss = F.cross_entropy(logits, labels) # Loss function
18
19         optimizer.zero_grad()
20         loss.backward()
21         optimizer.step()
22
23     ### LOGGING
24     print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
25           f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
26           f" | Train/Val Loss: {loss:.2f}")
27
28 model.eval()
29 # 插入可选的模型评估代码

```

运行上述代码将输出以下内容，类似于在 CPU 上获得的结果：

```

1 Epoch: 001/003 | Batch 000/002 | Train/Val Loss: 0.75
2 Epoch: 001/003 | Batch 001/002 | Train/Val Loss: 0.65
3 Epoch: 002/003 | Batch 000/002 | Train/Val Loss: 0.44
4 Epoch: 002/003 | Batch 001/002 | Train/Val Loss: 0.13
5 Epoch: 003/003 | Batch 000/002 | Train/Val Loss: 0.03
6 Epoch: 003/003 | Batch 001/002 | Train/Val Loss: 0.00

```

可以使用`.to("cuda")`来代替`device = torch.device("cuda")`。将张量传输到 "cuda" 而不是`torch.device("cuda")`也可以工作，并且更简洁。还可以修改该语句，这样即使没有 GPU，代码也能在 CPU 上执行。这被认为是分享 PyTorch 代码时的最佳实践：

```
1 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



在当前修改后的训练循环中，由于从 CPU 转移到 GPU 的内存传输成本，我们可能不会看到速度的提升。然而，我们可以期待在训练深度神经网络，尤其是大语言模型时，会有显著的速度提升。

🔥 练习

比较矩阵乘法在 CPU 和 GPU 上的运行时间。在多大尺寸的矩阵上，你开始看到 GPU 上的矩阵乘法比 CPU 上的矩阵乘法更快？（提示：在 Jupyter Notebook 中使用 `%timeit` 命令来比较运行时间。例如，对于矩阵 `a` 和 `b`，在新的笔记本单元中运行命令 `%timeit a @ b。`）

7.9.3 使用多个 GPU 训练

分布式训练的概念是将模型训练分配到多个 GPU 和机器上。为什么要这样做？虽然在单个 GPU 或机器上训练模型是可行的，但这个过程可能会非常耗时。通过将训练过程分布到多台机器上（每台机器可能有多个 GPU），可以显著减少训练时间。这在模型开发的实验阶段尤为重要，因为可能需要进行大量训练迭代来微调模型参数和架构。

让我们从分布式训练最基础的案例开始：`PyTorch` 的分布式数据并行（`DistributedDataParallel`, `DDP`）策略。`DDP` 通过将输入数据分割到可用设备上并同时处理这些数据子集来实现并行化。

这是如何工作的呢？`PyTorch` 会在每个 GPU 上启动一个独立的进程，每个进程都会接收并保存一份模型副本，这些副本在训练过程中会进行同步。假设有两个 GPU，我们想要用它们来训练一个神经网络，如图所示。

每个 GPU 都会接收到一份模型副本。然后，在每次训练迭代中，每个模型都会从数据加载器中接收一个小批次（或简称“批次”）数据。可以使用 `DistributedSampler` 来确保在使用 `DDP` 时，每个 GPU 接收到的批次不同且不重叠。

由于每个模型副本会看到不同的训练数据样本，因此模型副本在反向传播时将返回不同的 `logits` 并计算出不同的梯度。然后，这些梯度在训练过程中会被平均和同步，以便更新模型。通过这种方式，可以确保模型不会出现分歧，如图所示。

使用 `DDP` 的好处在于，与单个 GPU 相比，它能够更快地处理数据集。除去设备之间由于使用 `DDP` 而产生的少量通信开销，理论上使用两个 GPU 可以将训练一轮的时间缩短一半。时间效率会随着 GPU 数量的增加而提高，如果有 8 个 GPU，那么可以将一轮的处理速度提高 8 倍，以此类推。

现在让我们看看这在实践中是如何工作的。为简洁起见，我们将专注于需要为 `DDP` 训练调整的核心代码部分。

首先，导入一些用于分布式训练的 `PyTorch` 附加子模块、类和函数，如代码所示。

用于分布式训练的 `PyTorch` 工具

Python

```
1 import torch.multiprocessing as mp
2 from torch.utils.data.distributed import DistributedSampler
3 from torch.nn.parallel import DistributedDataParallel as DDP
4 from torch.distributed import init_process_group, destroy_process_group
```

在深入讨论使训练与 `DDP` 兼容的更改之前，先简要回顾一下与 `DistributedDataParallel` 类一起使用的这些新导入的工具的原理和用途。

`PyTorch` 的 `multiprocessing` 子模块包含诸如 `multiprocessing.spawn` 之类的函数，我们将使用这些函数来生成多个进程，然后再并行地将一个函数应用于多个输入。我们将为每个 GPU 生成一个训练进程。如果想要为训练生成多个进程，则需要用一种方法将数据集划分给这些不同的进程。为此，可以使用 `DistributedSampler`。

`init_process_group` 和 `destroy_process_group` 用于初始化和退出分布式训练模式。`init_process_group` 函数应在训练脚本开始时调用，以初始化分布式设置中每个进程的进程组，而 `destroy_process_group` 应在训练脚本结束时调用，以销毁给定的进程组并释放其资源。代码展示了这些新组件如何用于实现我们之前实现的 `NeuralNetwork` 模型的 `DDP` 训练。

使用 `DistributedDataParallel` 策略进行模型训练

Python

```
1 def ddp_setup(rank, world_size):
```

```
2     os.environ["MASTER_ADDR"] = "localhost"
3     os.environ["MASTER_PORT"] = "12345"
4     init_process_group(
5         backend="nccl",
6         rank=rank,
7         world_size=world_size
8     )
9     torch.cuda.set_device(rank)
10
11 def prepare_dataset():
12     # 插入数据集准备代码
13     train_loader = DataLoader(
14         dataset=train_ds,
15         batch_size=2,
16         shuffle=False,
17         pin_memory=True,
18         drop_last=True,
19         sampler=DistributedSampler(train_ds)
20     )
21     return train_loader, test_loader
22
23 def main(rank, world_size, num_epochs):
24     ddp_setup(rank, world_size)
25     train_loader, test_loader = prepare_dataset()
26     model = NeuralNetwork(num_inputs=2, num_outputs=2)
27     model.to(rank)
28     optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
29     model = DDP(model, device_ids=[rank])
30     for epoch in range(num_epochs):
```

7.10 小结

- PyTorch 是一个开源库，包含 3 个核心组件：张量库、自动微分函数和深度学习工具。
- PyTorch 的张量库类似于 NumPy 等数组库。
- 在 PyTorch 中，张量是表示标量、向量、矩阵和更高维数组的类数组数据结构。
- PyTorch 张量可以在 CPU 上执行，但 PyTorch 张量格式的一个主要优势是它支持 GPU 加速计算。
- PyTorch 中的自动微分 (autograd) 功能使我们能够方便地使用反向传播训练神经网络，而无须手动推导梯度。
- PyTorch 的深度学习工具提供了创建自定义深度神经网络的构建块。
- PyTorch 提供了 Dataset 类和 DataLoader 类来建立高效的数据加载流水线。
- 在 CPU 或单个 GPU 上训练模型是最简单的。
- 如果有多个 GPU 可用，那么使用 DistributedDataParallel 是 PyTorch 中加速训练的最简单方式。



8. 自然语言处理：从零实现大语言模型

9. 数学基础

9.1 矩阵微积分

9.1.1 引言

我们大多数人上一次接触微积分还是在学生时代，但导数（微分）对于机器学习尤其是深度神经网络至关重要——这类模型通过优化损失函数进行训练。翻开任何机器学习论文或 PyTorch 等库的文档，微积分就如同节假日不期而至的远房亲戚般尖叫着闯入你的生活。这可不是寻常的标量微积分，你需要的是微分矩阵演算——那是线性代数与多元微积分的闪电联姻。

借助现代深度学习库内置的自动微分功能，只需掌握最基础的标量微积分就能成为世界级的深度学习实践者。但如果你真想真正理解这些库背后的运行机制，并读懂讨论最新模型训练技术的学术论文，就需要掌握矩阵微积分领域的某些核心内容。

例如，神经网络中单个计算单元的激活值通常通过边权重向量 \mathbf{w} 与输入向量 \mathbf{x} 的点积（来自线性代数）加上标量偏置（阈值）来计算： $z(\mathbf{x}) = \sum_{i=1}^n w_i x_i + b$ 。函数 $z(\mathbf{x})$ 被称为该单元的仿射函数，随后经过一个将负值截断为零的修正线性单元： $\max(0, z(\mathbf{x}))$ 。这样的计算单元有时被称为“人工神经元”，其结构如图所示：

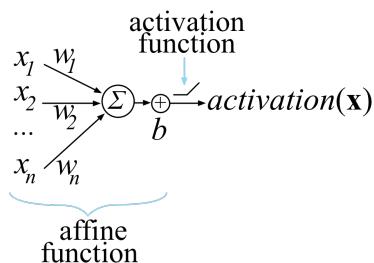


图 9.1 人工神经元

神经网络由许多这样的单元组成，这些单元被组织成多个被称为层的神经元集合。某一层单元的激活成为下一层单元的输入。最后一层中一个或多个单元的激活被称为网络输出。

训练这个神经元意味着选择权重 \mathbf{w} 和偏置 b ，以便对于所有 N 个输入 \mathbf{x} 得到期望的输出。为此，我们最小化一个损失函数，该函数比较网络对所有输入 \mathbf{x} 向量的最终 $activation(\mathbf{x})$ 与 $target(\mathbf{x})$ （ \mathbf{x} 的期望输出）。为了最小化损失，我们使用梯度下降的某种变体，例如普通的随机梯度下降（SGD）、带

动量的 SGD 或 Adam。所有这些方法都需要 $\text{activation}(\mathbf{x})$ 相对于模型参数 \mathbf{w} 和 b 的偏导数（梯度）。我们的目标是逐渐调整 \mathbf{w} 和 b ，使得在所有 \mathbf{x} 输入上的整体损失函数不断减小。

如果小心处理，我们可以通过微分常见损失函数的标量版本（均方误差）来推导梯度：

$$\frac{1}{N} \sum_{\mathbf{x}} (\text{target}(\mathbf{x}) - \text{activation}(\mathbf{x}))^2 = \frac{1}{N} \sum_{\mathbf{x}} \left(\text{target}(\mathbf{x}) - \max \left(0, \sum_i^{|\mathbf{x}|} w_i x_i + b \right) \right)^2 \quad (9.1)$$

但这只是单个神经元的情况，神经网络需要同时训练所有层中所有神经元的权重和偏置。由于存在多个输入以及（可能的）多个网络输出，我们确实需要关于向量函数导数的通用规则，甚至需要向量值函数对向量导数的规则。

本文逐步推导出计算向量偏导数的一些重要规则，这些规则对训练神经网络尤为有用。这一领域被称为矩阵微积分，好消息是我们只需要该领域的一小部分内容，本文将对此进行介绍。虽然关于多元微积分和线性代数的在线资料很多，但它们通常在本科阶段作为两门独立的课程教授，因此大多数资料都将其分开处理。那些讨论矩阵微积分的页面往往只是罗列规则而缺乏解释，或者只是零散的内容。由于使用密集的符号表示且对基础概念讨论甚少，这些内容往往让除少数数学家外的广大读者感到晦涩难懂。

与此相反，我们将重新推导和重新发现一些关键的矩阵微积分规则，以期解释它们。事实证明，矩阵微积分其实并不那么难！没有一大堆新规则需要学习；只有几个关键概念。我们希望这篇短文能让你快速入门矩阵微积分的世界，尤其是在它与神经网络训练相关的情况下。我们假设你已经熟悉了神经网络架构和训练的基础知识。（请注意，与许多更学术的方法不同，我们强烈建议首先在实践中学习训练和使用神经网络，然后再研究其背后的数学。在有实践背景的情况下，数学会更加容易理解；此外，要成为一名有效的实践者，并不需要完全掌握所有这些微积分知识。）

关于符号的说明：本文包含大量数学符号，因为本文的目标之一是帮助您理解深度学习论文和书籍中出现的符号。在文章末尾，您会发现一个简短的符号使用表，其中包括可用于搜索更多细节的单词或短语。

规则	$f(x)$	相对于标量 x 的导数记法	例子
常数	c	0	$\frac{d}{dx} 99 = 0$
乘以常数	$cf(x)$	$c \frac{df(x)}{dx}$	$\frac{d}{dx} 3x = 3$
幂法则	x^n	nx^{n-1}	$\frac{d}{dx} x^3 = 3x^2$
求和法则	$f + g$	$\frac{df}{dx} + \frac{dg}{dx}$	$\frac{d}{dx} (x^2 + 3x) = 2x + 3$
差值法则	$f - g$	$\frac{df}{dx} - \frac{dg}{dx}$	$\frac{d}{dx} (x^2 - 3x) = 2x - 3$
乘积法则	$f \cdot g$	$f \frac{dg}{dx} + g \frac{df}{dx}$	$\frac{d}{dx} (x^2 \cdot x) = x^2 + x \cdot 2x = 3x^2$
链式求导法则	$f(g(x))$	令 $u = g(x)$, $\frac{df(u)}{du} \cdot \frac{du}{dx}$	$\frac{d}{dx} \ln(x^2) = \frac{1}{x^2} \cdot 2x = \frac{2}{x}$

表 9.1 标量导数规则

9.2 概率论



10. 监督学习

监督学习模型定义了从一个或多个输入到一个或多个输出的映射。例如，输入可以是二手丰田普锐斯的车龄和里程数，输出可以是汽车的估价（美元）。

这个模型只是一个数学函数；当输入通过这个函数时，它会计算输出，这称为推理。模型函数也包含参数。不同的参数值会改变计算结果；模型函数描述了输入和输出之间的一组可能的关系，而模型参数则指定了特定的关系。

当训练模型时，目标是寻找能描述输入和输出之间真实关系的参数。学习算法会取一个输入/输出对作为训练集，并调整参数，直到输入尽可能准确地预测其对应的输出。如果模型在这些训练对的数据上表现良好，那么我们希望它能对新输入数据（即真实输出未知的情况）做出良好的预测。

本章的目标是扩展这些概念。首先，将更正式地描述这个框架并引入一些符号。然后，通过一个简单示例演示如何用一条直线来描述输入与输出之间的关系。这个线性模型既为人熟知又易于可视化，但仍能很好地阐明监督学习的所有主要思想。

10.1 监督学习概述

在监督学习中，我们的目标是构建一个模型，它可以接收一个输入 x 并输出一个预测 y 。为简单起见，假设输入 x 和输出 y 都是内容已预先确定且大小固定的向量，并且每个向量的元素始终按照相同的方式排列；在前述的普锐斯示例中，输入 x 总会按顺序包含汽车的车龄，然后是里程数。这类数据被称为结构化数据或表格数据。

为了做出预测，我们需要一个模型 $f[\bullet]$ ，它接收输入 x 并返回输出 y ，所以：

$$y = f(x) \tag{10.1}$$

我们把用输入 x 计算预测结果 y 的过程称为推理。

模型只是一个固定形式的数学函数，代表了输入和输出之间一系列不同的关系。模型也包含参数。参数的选择决定了输入和输出之间的具体关系，所以应该更准确地写成：

$$y = f[x, \Phi] \tag{10.2}$$

当谈论学习或训练模型时，我们的目的是试图找到能够通过输入做出合理输出预测的参数。我们使用一个包含对输入和输出示例的训练数据集 $\{(x_i, y_i)\}$ 来学习获得这些参数。目标是选择参数，尽可能准确地将每个训练输入数据映射到其相应的输出数据。用损失函数 \mathcal{L} 来量化这种映射的不匹配程度。损失是一个标量值，概括了在给定参数的情况下，模型从对应的输入中预测训练输出的误差。

可将损失 \mathcal{L} 视为参数的函数 $\mathcal{L}[\Phi]$ 。当训练模型时，我们就是在寻找能使这个损失函数值最小的参数 $\hat{\Phi}$ ：

$$\hat{\Phi} = \underset{\Phi}{\operatorname{argmin}} [\mathcal{L}[\Phi]] \quad (10.3)$$

提示

更准确地说，损失函数还依赖于训练数据 $\{x_i, y_i\}$ ，所以应该写作 $\mathcal{L}[\{x_i, y_i\}, \Phi]$ ，但这样表示会显得相当繁杂。

如果在完成这一最小化过程后损失函数的值很小，我们就已经找到了能准确预测输入 x_i 到输出 y_i 的模型参数。

训练完一个模型后，必须评估其性能；在独立的测试数据上运行模型，观测它在训练期间未观察到的样本上的泛化能力。如果性能令人满意，就可以准备部署模型了。

10.2 线性回归示例

现在，让我们通过一个简单例子来具体化这些概念。设想一个模型 $y = f[x, \Phi]$ ，它通过一个输入 x 预测一个输出 y 。接下来，我们会设计一个损失数，最后讨论模型的训练过程。

10.2.1 一维（1D）线性回归模型

一维线性回归模型用一条直线描述输入 x 和输出 y 之间的关系：

$$\begin{aligned} y &= f[x, \Phi] \\ &= \phi_0 + \phi_1 x \end{aligned} \quad (10.4)$$

该模型有两个参数 $\Phi = [\phi_0, \phi_1]^T$ ，其中 ϕ_0 是直线的 y 轴截距， ϕ_1 是斜率。不同的 y 轴截距和斜率选择会决定输入和输出之间的不同关系。因此，上面的式子定义了一组可能的输入/输出关系（所有可能的直线），参数的选择决定了这一组中的具体成员（特定的直线）。

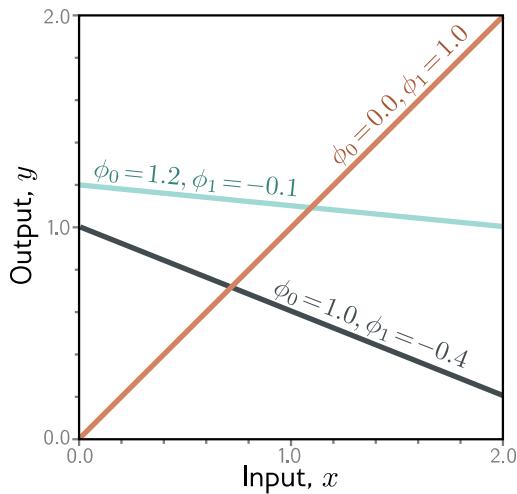


图 10.1 线性回归模型。对于给定的参数 $\Phi = [\phi_0, \phi_1]^T$ ，模型根据输入（ x 轴）对输出（ y 轴）进行预测。不同的截距 ϕ_0 和斜率 ϕ_1 的选择会改变这些预测结果（青色、橙色和灰色直线）。线性回归模型（式（10.4））定义了一组输入/输出关系（直线），而参数决定了该组中的具体成员（特定的直线）

10.2.2 损失

对于这个模型，训练数据集（图 10.2(a)）由 I 个输入/输出对 $\{x_i, y_i\}$ 构成。图 10.2(b) ~ (d) 显示了由三组参数定义的三条直线。图 10.2(d) 中的绿色直线比其他两条线更准确地描述了数据，

因为它更接近数据点。然而，我们需要一个有依据的方法来决定哪个参数比其他参数更好。为此，给每个参数分配一个数值，用该数值来量化模型与数据之间的不匹配程度。将这个值称为损失；更低的损失意味着更好的拟合效果。

这种不匹配由模型的预测 $f[x, \Phi]$ （在 x 处线上方的高度）和实际输出 y 之间的偏差所捕获。这些偏差在图 10.2(b) ~ (d) 中以橙色虚线表示。将所有 I 个训练对的偏差的平方和称为总的不匹配，训练误差或损失：

$$\begin{aligned}\mathcal{L}[\Phi] &= \sum_{i=1}^I (f[x_i, \Phi] - y_i)^2 \\ &= \sum_{i=1}^I (\phi_0 + \phi_1 x_i - y_i)^2\end{aligned}\quad (10.5)$$

由于最佳参数会最小化这个式子的值，所以我们称之为最小二乘损失。平方计算意味着偏差的方向（即直线是在数据上方还是下方）无关重要。后面我们会讲解这样做的理论依据。

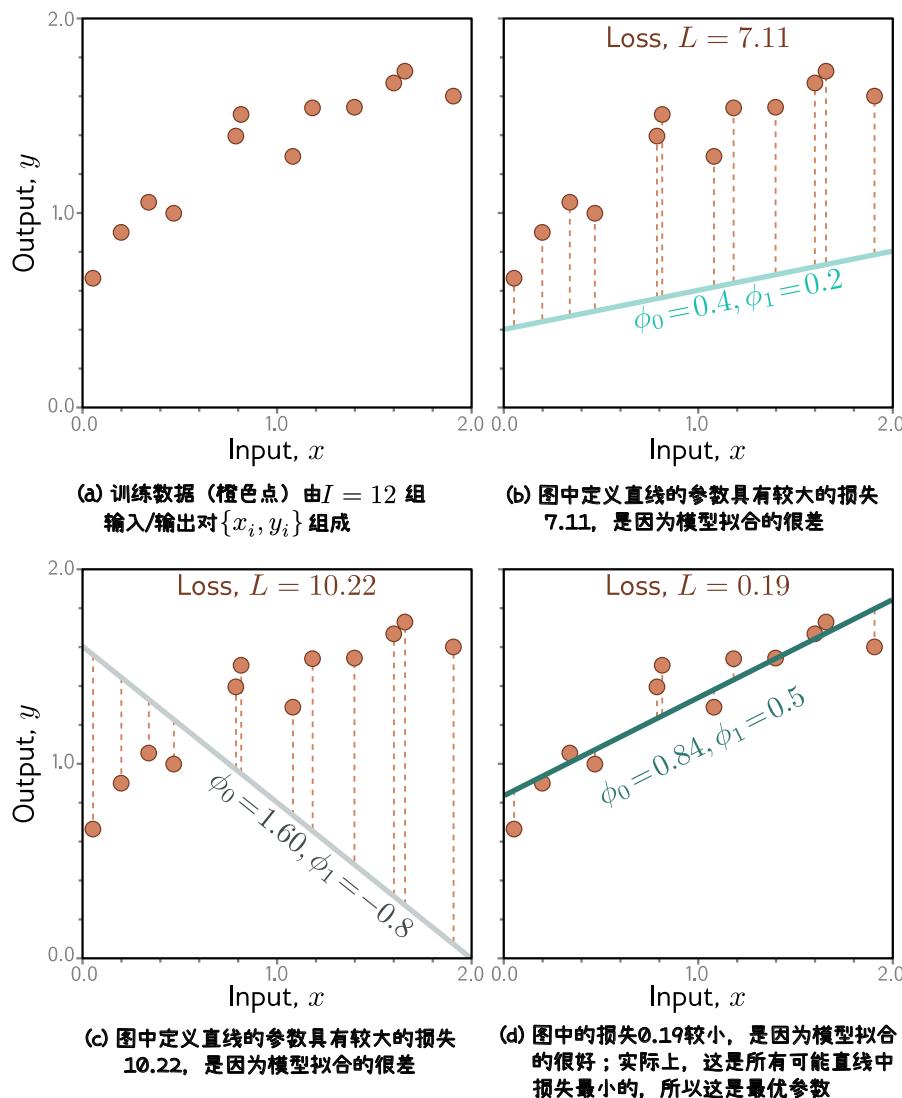


图 10.2 线性回归的训练数据、模型和损失。图(b) ~ (d) 分别展示了具有不同参数的线性回归模型。根据截距和斜率参数 $\Phi = [\phi_0, \phi_1]^T$ 的选择，模型误差（橙色虚线）可能更大或者更小。损失 \mathcal{L} 是这些误差平方的总和

损失 \mathcal{L} 是参数 Φ 的函数；当模型拟合较差时（图 10.2(b) 和图 10.2(c)）损失更大，当拟合良好时（图 10.2(d)）损失更小。从这个角度看，将 $\mathcal{L}[\Phi]$ 称为损失函数或者代价函数。训练模型的目标是找到最小化这个值的参数 $\hat{\Phi}$ ：

$$\begin{aligned}\hat{\Phi} &= \operatorname{argmin}_{\Phi} [\mathcal{L}[\Phi]] \\ &= \operatorname{argmin}_{\Phi} \left[\sum_{i=1}^I (f[x_i, \Phi] - y_i)^2 \right] \\ &= \operatorname{argmin}_{\Phi} \left[\sum_{i=1}^I (\phi_0 + \phi_1 x_i - y_i)^2 \right]\end{aligned}\quad (10.6)$$

只有两个参数（ y 轴截距 ϕ_0 和斜率 ϕ_1 ），因此我们可以计算每种参数值组合的损失，并将损失函数可视化为一个曲面。“最佳”参数位于该曲面的最低点。

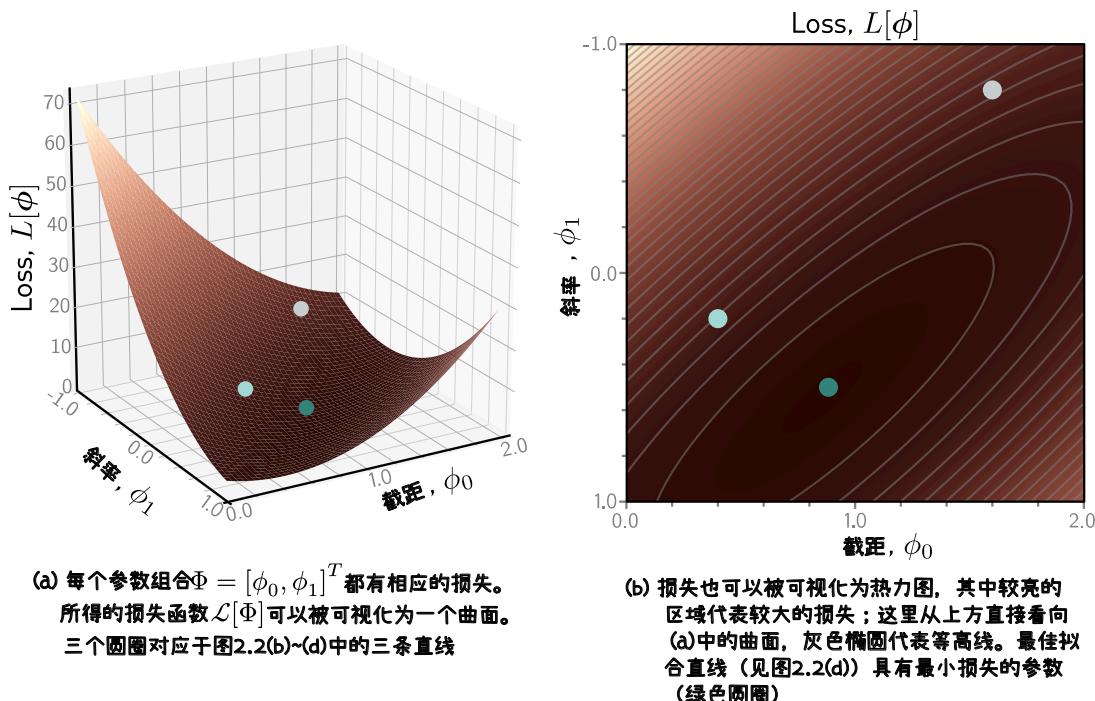


图 10.3 针对图 10.2(a)的数据集的线性回归模型的损失函数

10.2.3 训练

寻找使损失最小化的参数的过程称为模型拟合、训练或者学习。基本方法是随机选择初始参数，然后“沿着”损失函数“下降”的方向改进它们，直至到达最低点。一种方法是测量当前点所在曲面的梯度，并向最陡峭的下坡方向迈出一步。此后重复这个过程，直到梯度变得平坦，无法进一步改进为止。

⚡ 危险

对于线性回归模型来说，这种迭代方法实际上并不是必要的。在这里，我们可以找到参数的解析解表达式。然而，梯度下降法适用于更复杂的模型，这些模型中没有解析解，并且参数特别多，无法评估每种参数组合的损失。

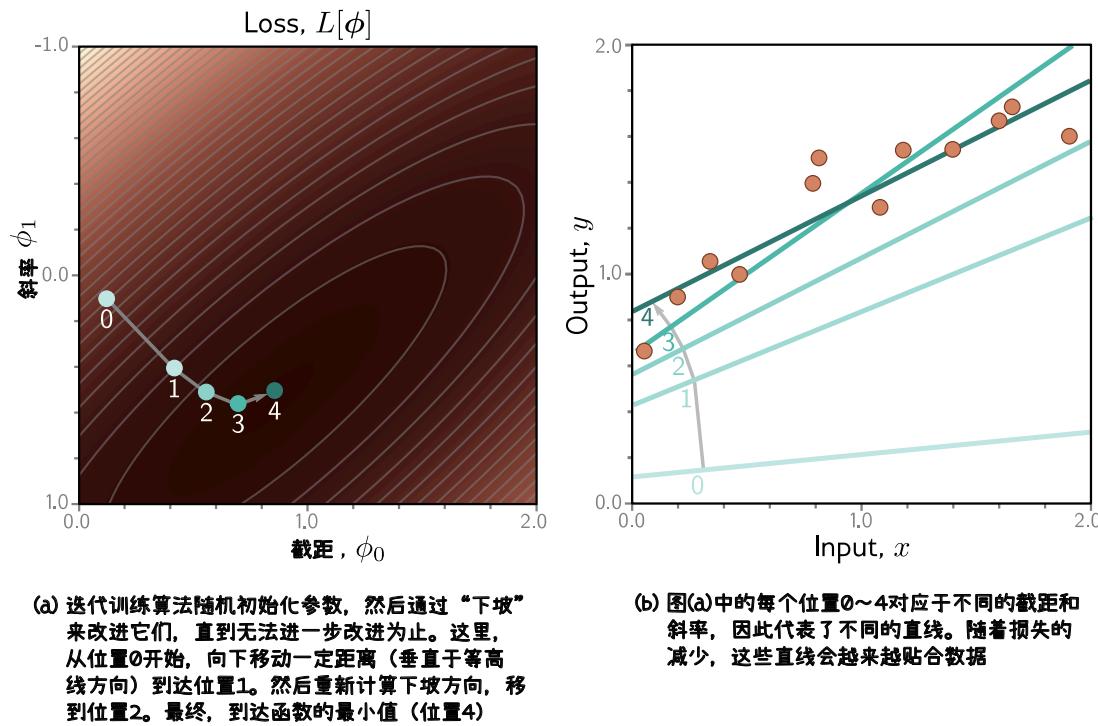


图 10.4 线性回归训练。训练目标是找到对应于最小损失的截距和斜率参数

10.2.4 测试

完成模型训练之后，我们需要知道它在现实世界中的表现如何。通过在单独的测试数据集上面计算损失来评估这一点。预测准确性会在何种程度上泛化到测试数据部分取决于训练数据的代表性和完整性，也取决于模型的表达能力。一个简单模型（如一条直线）可能无法捕捉输入和输出之间的真正关系，这称为欠拟合。相反，一个表达力很强的模型可能描述训练数据的一些非典型的统计特性，同时会引起异常的预测。这被称为过拟合。

10.2.5 最小二乘法

我们还记得一元线性回归的损失函数是

$$\mathcal{L}[\Phi] = \sum_{i=1}^I (\phi_0 + \phi_1 x_i - y_i)^2 \quad (10.7)$$

而我们的目标是想让损失函数 \mathcal{L} 最小化。而通过观察损失函数的图像，我们可以得知这个损失函数是存在最小值的。那么怎么找出这个最小值呢？由于我们有两个参数：截距 ϕ_0 和斜率 ϕ_1 。根据微积分的知识，损失函数对这两个参数求导，导数为 0 的点就是最小值。

所以损失函数对 ϕ_0 的导数如下推导：

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \phi_0} &= \frac{\partial \sum_{i=1}^I (\phi_0 + \phi_1 x_i - y_i)^2}{\partial \phi_0} \quad (\text{求导的线性法则}) \\
&= \sum_{i=1}^I \frac{\partial (\phi_0 + \phi_1 x_i - y_i)^2}{\partial \phi_0} \quad (\text{链式法则}) \\
&= \sum_{i=1}^I \frac{\partial (\phi_0 + \phi_1 x_i - y_i)^2}{\partial (\phi_0 + \phi_1 x_i - y_i)} \cdot \frac{\partial (\phi_0 + \phi_1 x_i - y_i)}{\partial \phi_0} \\
&= \sum_{i=1}^I \{2(\phi_0 + \phi_1 x_i - y_i) \cdot 1\} \\
&= 2 \sum_{i=1}^I \{\phi_0 + \phi_1 x_i - y_i\} \\
&= 2 \cdot I \cdot \phi_0 + 2(x_1 + x_2 + \dots + x_I) \phi_1 - 2(y_1 + y_2 + \dots + y_I) \\
&= 0
\end{aligned} \tag{10.8}$$

化简可以得到如下

$$I \cdot \phi_0 + \left[\sum_{i=1}^I x_i \right] \phi_1 = \sum_{i=1}^I y_i \tag{10.9}$$

所以损失函数对 ϕ_1 的导数如下推导：

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \phi_1} &= \frac{\partial \sum_{i=1}^I (\phi_0 + \phi_1 x_i - y_i)^2}{\partial \phi_1} \quad (\text{求导的线性法则}) \\
&= \sum_{i=1}^I \frac{\partial (\phi_0 + \phi_1 x_i - y_i)^2}{\partial \phi_1} \quad (\text{链式法则}) \\
&= \sum_{i=1}^I \frac{\partial (\phi_0 + \phi_1 x_i - y_i)^2}{\partial (\phi_0 + \phi_1 x_i - y_i)} \cdot \frac{\partial (\phi_0 + \phi_1 x_i - y_i)}{\partial \phi_1} \\
&= \sum_{i=1}^I \{2(\phi_0 + \phi_1 x_i - y_i) \cdot x_i\} \\
&= 2 \sum_{i=1}^I \{(\phi_0 + \phi_1 x_i - y_i) \cdot x_i\} \\
&= 2(x_1 + x_2 + \dots + x_I) \phi_0 + 2(x_1^2 + x_2^2 + \dots + x_I^2) \phi_1 - 2(x_1 y_1 + x_2 y_2 + \dots + x_I y_I) \\
&= 0
\end{aligned} \tag{10.10}$$

化简可以得到如下

$$\left[\sum_{i=1}^I x_i \right] \phi_0 + \left[\sum_{i=1}^I x_i^2 \right] \phi_1 = \sum_{i=1}^I x_i y_i \tag{10.11}$$

也就是说我们要求解的

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial \phi_0} = 0 \\ \frac{\partial \mathcal{L}}{\partial \phi_1} = 0 \end{cases} \tag{10.12}$$

最终化简得到了一个二元一次方程组

$$\begin{cases} I \cdot \phi_0 + \left[\sum_{i=1}^I x_i \right] \phi_1 = \sum_{i=1}^I y_i \\ \left[\sum_{i=1}^I x_i \right] \phi_0 + \left[\sum_{i=1}^I x_i^2 \right] \phi_1 = \sum_{i=1}^I x_i y_i \end{cases} \tag{10.13}$$

这样就可以求解得到 ϕ_0 和 ϕ_1 的数值，也就是最优参数就可以被求解出来了。

如果写成矩阵的表示法，可以看到如下

$$\begin{bmatrix} I & \sum_{i=1}^I x_i \\ \sum_{i=1}^I x_i & \sum_{i=1}^I x_i^2 \end{bmatrix} \cdot \begin{bmatrix} \phi_0 \\ \phi_1 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^I y_i \\ \sum_{i=1}^I x_i y_i \end{bmatrix} \quad (10.14)$$

根据矩阵的性质

$$A \cdot B = C \Rightarrow A^{-1} \cdot A \cdot B = A^{-1}C \Rightarrow B = A^{-1}C \quad (10.15)$$

可以得到如下

$$\begin{bmatrix} \phi_0 \\ \phi_1 \end{bmatrix} = \begin{bmatrix} I & \sum_{i=1}^I x_i \\ \sum_{i=1}^I x_i & \sum_{i=1}^I x_i^2 \end{bmatrix}^{-1} \cdot \begin{bmatrix} \sum_{i=1}^I y_i \\ \sum_{i=1}^I x_i y_i \end{bmatrix} \quad (10.16)$$

2×2 逆矩阵的求解对于计算机来说是很容易的，但如果参数很多呢？那么求解逆矩阵可能就是一件根本无法做到的事情了。

例如，如果我们的训练数据是 $\{(x_{i,1}, x_{i,2}, \dots, x_{i,N}), y_i\}$ 数据对呢？如果我们还是用线性回归的话，我们需要用多元线性回归来解决这个问题。此时，我们的损失函数变成了

$$\begin{aligned} \mathcal{L}[\Phi] &= \sum_{i=1}^I (\phi_0 + \phi_1 x_{i,1} + \phi_2 x_{i,2} + \dots + \phi_N x_{i,N} - y_i)^2 \\ &= \sum_{i=1}^I \left(\phi_0 + \sum_{j=1}^N \phi_j x_{i,j} - y_i \right)^2 \end{aligned} \quad (10.17)$$

那么损失函数需要对 $\{\phi_0, \phi_1, \phi_2, \dots, \phi_N\}$ 共 $N+1$ 个参数求导数为0的点。

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \phi_0} &= 2 \sum_{i=1}^I \left[\left(\phi_0 + \sum_{j=1}^N \phi_j x_{i,j} - y_i \right) \cdot 1 \right] \\ \frac{\partial \mathcal{L}}{\partial \phi_1} &= 2 \sum_{i=1}^I \left[\left(\phi_0 + \sum_{j=1}^N \phi_j x_{i,j} - y_i \right) \cdot x_{i,1} \right] \\ \frac{\partial \mathcal{L}}{\partial \phi_2} &= 2 \sum_{i=1}^I \left[\left(\phi_0 + \sum_{j=1}^N \phi_j x_{i,j} - y_i \right) \cdot x_{i,2} \right] \\ &\vdots \\ \frac{\partial \mathcal{L}}{\partial \phi_N} &= 2 \sum_{i=1}^I \left[\left(\phi_0 + \sum_{j=1}^N \phi_j x_{i,j} - y_i \right) \cdot x_{i,N} \right] \end{aligned} \quad (10.18)$$

如果 N 很大，例如 6710 亿参数（DeepSeek-V3 的参数数量），求解上面这个方程组是做不到的。

所以我们才会使用梯度下降法来去寻找损失函数的极小值点。

我们可以将上面的式子整理成矩阵表示法。

设计矩阵

$$\mathbf{X} = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,N} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{I,1} & x_{I,2} & \cdots & x_{I,N} \end{bmatrix} \quad (10.19)$$

参数向量

$$\Phi = \begin{bmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \vdots \\ \phi_N \end{bmatrix} \in \mathbb{R}^{N+1} \quad (10.20)$$

目标向量

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \in \mathbb{R}^I \quad (10.21)$$

预测值

$$\hat{\mathbf{y}} = \mathbf{X}\Phi = \begin{bmatrix} \phi_0 + \sum_{j=1}^N \phi_j x_{1,j} \\ \phi_0 + \sum_{j=1}^N \phi_j x_{2,j} \\ \vdots \\ \phi_0 + \sum_{j=1}^N \phi_j x_{I,j} \end{bmatrix} \quad (10.22)$$

残差（误差）向量

$$\mathbf{e} = \hat{\mathbf{y}} - \mathbf{y} = \mathbf{X}\Phi - \mathbf{y} = \begin{bmatrix} \phi_0 + \sum_{j=1}^N \phi_j x_{1,j} - y_1 \\ \phi_0 + \sum_{j=1}^N \phi_j x_{2,j} - y_2 \\ \vdots \\ \phi_0 + \sum_{j=1}^N \phi_j x_{I,j} - y_I \end{bmatrix} \quad (10.23)$$

梯度的矩阵形式

观察偏导数的结构

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \phi_2} &= 2 \sum_{i=1}^I \left[\left(\phi_0 + \sum_{j=1}^N \phi_j x_{i,j} - y_i \right) \cdot x_{i,2} \right] \\ &= 2 \sum_{i=1}^I (\mathbf{e}_i \cdot x_{i,2}) \\ &= 2 [\mathbf{e}_1 \ \mathbf{e}_2 \ \cdots \ \mathbf{e}_I] \cdot \begin{bmatrix} x_{1,2} \\ x_{2,2} \\ x_{3,2} \\ \vdots \\ x_{I,2} \end{bmatrix} \\ &= 2 [x_{1,2} \ x_{2,2} \ x_{3,2} \ \cdots \ x_{I,2}] \cdot \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \\ \vdots \\ \mathbf{e}_I \end{bmatrix} \end{aligned} \quad (10.24)$$

这实际上是残差向量与设计矩阵第 2 列的内积！

所以

$$\nabla_{\Phi} \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \phi_0} \\ \frac{\partial \mathcal{L}}{\partial \phi_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial \phi_N} \end{bmatrix} = 2 \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_{1,1} & x_{2,1} & x_{3,1} & \cdots & x_{I,1} \\ x_{1,2} & x_{2,2} & x_{3,2} & \cdots & x_{I,2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{1,N} & x_{2,N} & x_{3,N} & \cdots & x_{I,N} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \\ \vdots \\ \mathbf{e}_I \end{bmatrix} = 2\mathbf{X}^T \mathbf{e} \quad (10.25)$$

所以损失函数的梯度是

$$\nabla_{\Phi} \mathcal{L} = 2\mathbf{X}^T \mathbf{e} = 2\mathbf{X}^T (\mathbf{X}\Phi - \mathbf{y}) \quad (10.26)$$

如果我们要求解梯度为 $\mathbf{0}$ 的解，也就是

$$\nabla_{\Phi} \mathcal{L} = 2\mathbf{X}^T \mathbf{e} = 2\mathbf{X}^T (\mathbf{X}\Phi - \mathbf{y}) = 0 \quad (10.27)$$

先将上面的式子把 2 消去。然后分配率展开得到

$$\begin{aligned} \mathbf{X}^T (\mathbf{X}\Phi - \mathbf{y}) &= 0 \\ \Downarrow \\ \mathbf{X}^T \mathbf{X}\Phi - \mathbf{X}^T \mathbf{y} &= 0 \\ \Downarrow \\ \mathbf{X}^T \mathbf{X}\Phi &= \mathbf{X}^T \mathbf{y} \quad \text{(正规方程)} \\ \Downarrow \\ \Phi &= (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \quad \text{(计算机算不动)} \end{aligned} \quad (10.28)$$

主要是上面的矩阵求逆运算计算机无法完成。

10.3 多项式拟合示例

我们再举一个简单的例子——用多项式拟合一个小型合成数据集。这也是一个监督学习问题，在这个问题中，我们希望根据输入的变量值，对目标变量进行预测。

10.3.1 合成数据

我们用 x 表示输入变量，用 t 表示目标变量，并假设这两个变量在实数轴取值连续。给定一个训练集，其中包含 N 个 x 的观测值，记作 x_1, \dots, x_N ，还包含相应 t 的观测值，记作 t_1, \dots, t_N 。我们的目标是根据 x 的某个新值来预测相应的 t 的值。机器学习的一个关键目标是对以前没有见过的输入进行准确预测，这种能力成为泛化能力（generalization）。

我们可以通过从正弦函数采样生成的合成数据集来说明这一点。下图展示了由 $N = 10$ 个数据点组成的训练数据集，其中输入值 $x_n (n = 1, \dots, N)$ 是通过在区间 $[0, 1]$ 上均匀采样生成的。对应的目标值 t_n 则是先计算每个 x 所对应的函数 $\sin(2\pi x)$ 的值，然后向每个数据点添加少量随机噪声（由高斯分布控制）得到的。通过这种方式生成数据，我们可以捕获许多现实世界数据集的一个重要特性——它们具有我们希望了解的潜在规律，但个别观测值会被随机噪声干扰。这种噪声可能源于它们固有的随机过程（例如放射性衰变），但更常见的原因是存在未被观测到的变异源。

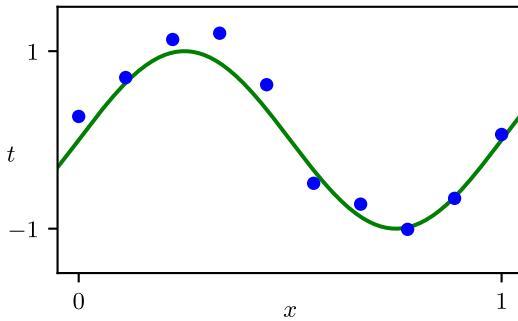


图 10.5 一个由 $N = 10$ 个数据点组成的训练集, 以蓝色圆点显示, 其中每个数据点包含了输入变量 x 及其对应的目标变量 t 的观测值。绿色曲线显示了用来生成数据的函数 $\sin(2\pi x)$ 。我们的目标是在不知道绿色曲线的情况下, 预测新的输入变量 x 所对应的目标变量 t 的值

在这个示例中, 我们事先知道真正生成数据是通过一个正弦函数。在机器学习的实际应用中, 我们的目标是在有限的训练数据集中发现隐藏的规律。不过, 了解数据的生成过程有助于我们阐明机器学习中的一些重要概念。

10.3.2 线性模型

我们的目标是利用这个训练数据集来预测输入变量的新值 \hat{x} 所对应的目标变量的值 \hat{t} , 这涉及到隐式地尝试发现潜在的函数 $\sin(2\pi x)$ 。这本质上是一个十分困难的问题, 因为我们必须从有限的数据集推广到整个函数。此外, 观测数据受到噪声干扰, 因此对于给定的 \hat{x} , \hat{t} 的适当取值存在不确定性。概率论提供了一个以精确和定量的方式来表达这种不确定性的框架。

从数据中学习概率是机器学习的核心!

现在让我们先从一种相对非正式的方式出发, 考虑一种基于曲线拟合的简单方法。我们将使用多项式函数来拟合数据, 其形式如下:

$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \dots + w_M x^M = \sum_{j=0}^M w_j x^j \quad (10.29)$$

其中 M 是多项式的阶数 (`order`), x^j 表示 x 的 j 次幂。多项式系数 w_0, \dots, w_M 统称为向量 \mathbf{w} 。注意, 尽管多项式函数 $y(x, \mathbf{w})$ 是关于 x 的非线性函数, 但它也是系数 \mathbf{w} 的线性函数。在上面的式子中, 像这个多项式一样, 关于未知参数呈线性的函数具有重要的特性, 同时也存在明显的局限性, 它们被称为线性模型 (`linear model`)。

10.3.3 误差函数

多项式系数的值将通过拟合训练数据来确定, 这可以通过最小化误差函数 (`error function`) 来实现, 该误差函数度量了对于任意给定的 \mathbf{w} , 函数 $y(x, \mathbf{w})$ 与训练数据集中数据点之间的拟合误差。有一个使用广泛的简单误差函数, 即每个数据点 x_n 的预测值 $y(x_n, \mathbf{w})$ 与相应目标值 t_n 之间的差的平方和的二分之一:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \{y(x_n, \mathbf{w}) - t_n\}^2 \quad (10.30)$$

其中引入了系数 $1/2$ 是为了后续计算方便。后面我们同样会推导这个误差函数。注意, 这个误差函数是非负的, 当且仅当函数 $y(x, \mathbf{w})$ 正好通过每个训练数据点时, 其值等于 0。平方和误差函数的几何解释如下图所示。

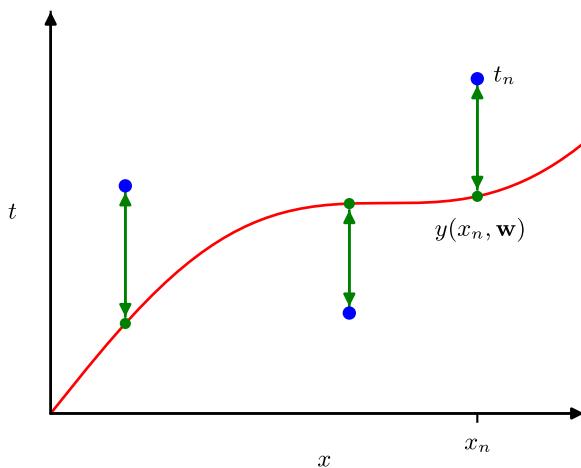


图 10.6 平方和误差函数的几何解释 [该误差函数对应来自函数 $y(x, \mathbf{w})$ 的每个数据点的位移 (如垂直的绿色箭头所示) 平方和的一半]

我们可以通过选择能够使 $E(\mathbf{w})$ 尽可能小的 \mathbf{w} 值来解决曲线拟合问题。因为平方和误差函数是系数 \mathbf{w} 的二次函数，其对系数的导数是系数 \mathbf{w} 的线性函数，所以该误差函数的最小化有一个唯一解，记作 \mathbf{w}^* ，可以通过解析形式求得解析解。最终的多项式由函数 $y(x, \mathbf{w}^*)$ 给出。

10.3.4 模型复杂度

我们还面临选择多项式的阶数 M 的问题，这将引出模型比较或者模型选择这一重要概念。在下图中，我们展示了 4 个拟合实例，它们分别使用阶数 $M = 0, 1, 3, 9$ 的多项式来拟合训练数据集的数据。

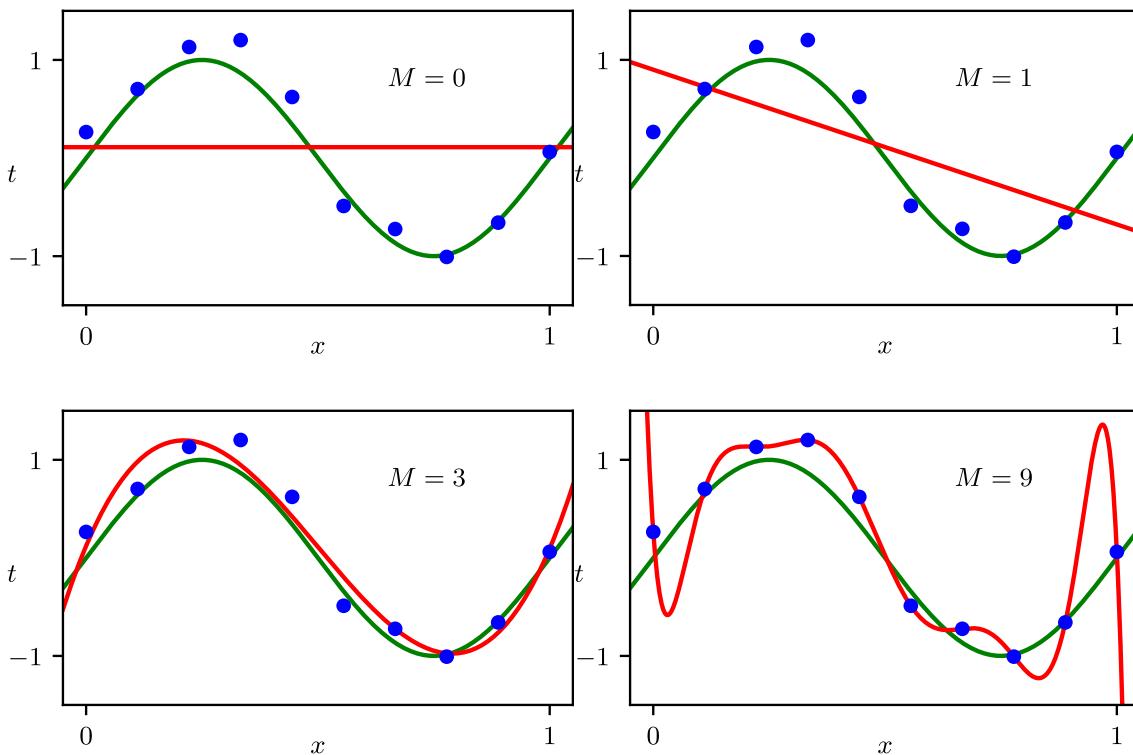


图 10.7 具有不同阶数的多项式图示。多项式如红色曲线所示。这里通过最小化平方和误差函数来拟合训练数据集

可以发现，常数($M=0$)和一阶($M=1$)多项式对数据的拟合较差，因此对函数 $\sin(2\pi x)$ 的表示较差。三阶($M=3$)多项式似乎对函数 $\sin(2\pi x)$ 给出了最佳拟合。高阶($M=9$)多项式得到了一个对训练数据完美的拟合。事实上，这个多项式曲线精确地穿过了每一个数据点，使得误差 $E(\mathbf{w}^*)=0$ 。然而，拟合出的曲线却出现了剧烈的波动，完全不能反映出函数 $\sin(2\pi x)$ 的真实形态。这种现象称为过拟合(**over-fitting**)。

我们的目标是让模型获得良好的泛化能力，使其能够对新的数据做出准确的预测。为了定量地探究泛化性能与模型复杂度 M 之间的依赖关系，我们可以引入一个独立的测试集。该测试集包含100个数据点，其生成方式与训练集相同。针对每一个 M 值，我们不仅可以计算出模型在训练集上的残差 $E(\mathbf{w}^*)$ [式(1.2)]，还可以计算出其在测试集上的残差 $E(\mathbf{w}^*)$ 。与评估误差函数 $E(\mathbf{w})$ 相比，有时使用均方根(Root Mean Square, RMS)误差更为方便，均方根误差定义如下：

$$E_{\text{RMS}} = \sqrt{\frac{1}{N} \sum_{n=1}^N \{y(x_n, \mathbf{w}) - t_n\}^2} \quad (10.31)$$

公式中的 $1/N$ 是为了让不同大小的数据集能够在相同的基准下进行比较，而求平方根则是为了确保 E_{RMS} 是在与目标变量相同的尺度上(以相同的单位)进行测量的。下图展示了不同 M 值的训练集和测试集的RMS误差图。测试集上的RMS误差反映了我们根据新观测数据 x 预测其对应 t 值的能力。从下图中可以看出，当 M 值较小时测试集误差较大，这是因为此时的多项式模型灵活性不足，无法捕捉函数 $\sin(2\pi x)$ 中的振荡。当 M 取值在[3, 8]时，测试集误差较小，同时这些模型也能合理地表示出数据的生成函数 $\sin(2\pi x)$ ，如上图中 $M=3$ 时所示。

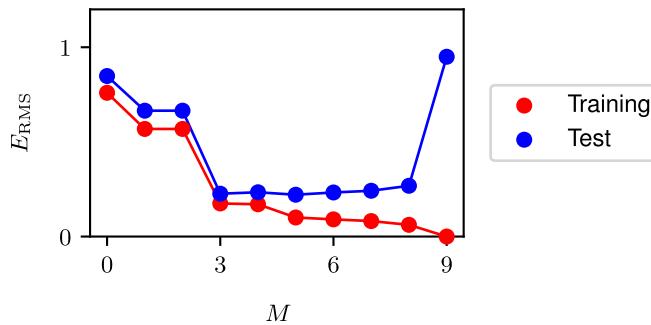


图 10.8 由式 1.3 定义的均方根误差图(在训练集和独立的测试集上对 M 的各个值进行评估)

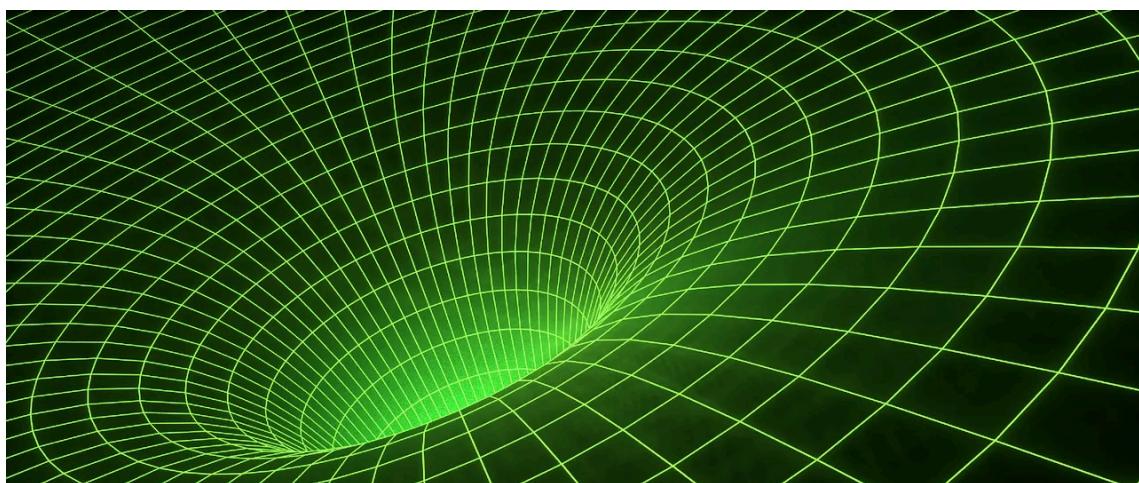
当 $M=9$ 时，训练集误差变为0。这是符合预期的，因为该多项式包含10个自由度(对应10个系数 w_0, \dots, w_9)，所以可以精确地调整到训练集中的10个数据点。然而，如图1.7和图1.8所示，测试集误差变得极大，函数 $y(x, \mathbf{w}^*)$ 表现出剧烈振荡。

这可能看起来很矛盾，因为一个给定阶数的多项式包含了所有更低阶的多项式作为特例。因此， $M=9$ 的多项式理应能够产生至少与 $M=3$ 的多项式一样好的结果。此外，我们或许会认为，预测新数据的最佳模型就应该是生成这些数据的真实函数 $\sin(2\pi x)$ 本身(我们后续将验证这一点)。同时，我们知道 $\sin(2\pi x)$ 的幂级数展开式中包含了所有阶数的项，所以我们会很自然地推断，随着模型复杂度 M 的增加，预测效果应该会持续提升。

通过观察表1.1中不同阶数多项式拟合得到的系数 \mathbf{w}^* ，我们可以更深入地了解这个问题。我们注意到，随着 M 的增加，系数的幅度越来越大。特别是当 $M=9$ 时，为了让对应的多项式曲线能精准地穿过每一个数据点，这些系数被精细地调整到了很大的正值或负值。但在数据点之间，尤其是在数据范围的两端附近，曲线却出现了大幅度的摆动，正如我们在图1.7中看到的那样。直观地看，当多项式模型具有较大的 M 值时，它变得更加灵活，从而更容易受到目标值上随机噪声的影响，并过度拟合了这些噪声。

11. 梯度下降法

梯度下降法奠定了机器学习和深度学习技术的基础。让我们探索它的工作原理、适用场景以及在不同函数中的表现特性。



11.1 简介

梯度下降是一种迭代式一阶优化算法，用于寻找给定函数的局部最小值/最大值。这种方法在机器学习与深度学习领域广泛用于最小化损失函数（例如线性回归场景）。

我们将会深入探讨一阶梯度下降算法的数学原理、实现方式和行为特性。我们将直接引导自定义的函数来寻找其最小值。

梯度下降法由柯西在 1847 年提出，远早于现代计算机时代。自那时起，计算机科学与数值方法领域取得了长足发展，由此衍生出众多改进版的梯度下降算法。

11.2 对函数的要求

梯度下降法并不适用于所有函数，有两个特定的要求。函数必须是：

- 可微函数（可以求导的）
- 凸函数

首先，可微是什么意思？如果一个函数是可微的，那么在其定义域内的每个点都有导数——并非所有函数都满足这个标准。首先，我们来看一些满足这个标准的函数示例：

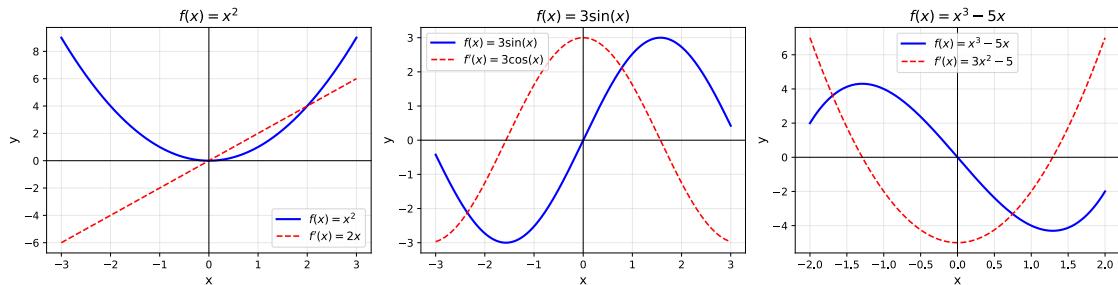


图 11.1 可微函数示例

典型的不可微函数具有阶梯、尖点或不连续点：

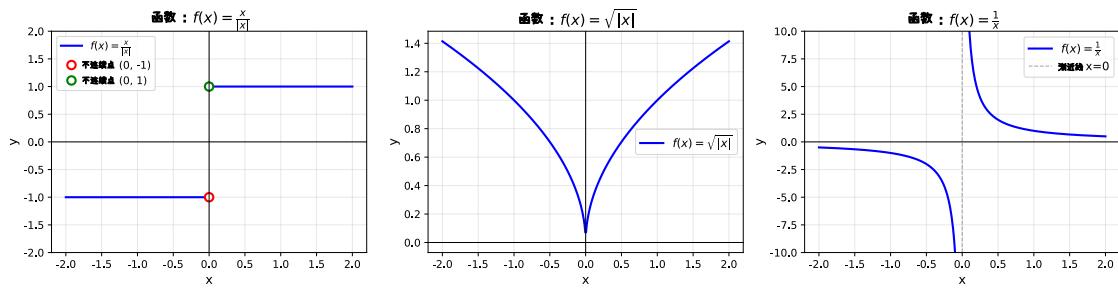


图 11.2 不可微函数示例

下一个要求——函数必须是凸函数。对于一元函数而言，这意味着连接函数上任意两点的线段均位于曲线之上或与之相切（不会穿越曲线）。若线段穿越曲线，则表明函数存在局部最小值而非全局最小值。

数学上，对于位于函数曲线上的两点 x_1 和 x_2 ，一个函数是凸函数的条件可表示为：

$$f(\lambda x_1 + (1 - \lambda)x_2) \leq \lambda f(x_1) + (1 - \lambda)f(x_2) \quad (11.1)$$

其中 λ 表示点在截线上的位置，其值必须在0（左侧点）和1（右侧点）之间，例如 $\lambda = 0.5$ 表示中点位置。

以下是两个带有示例截线的函数。

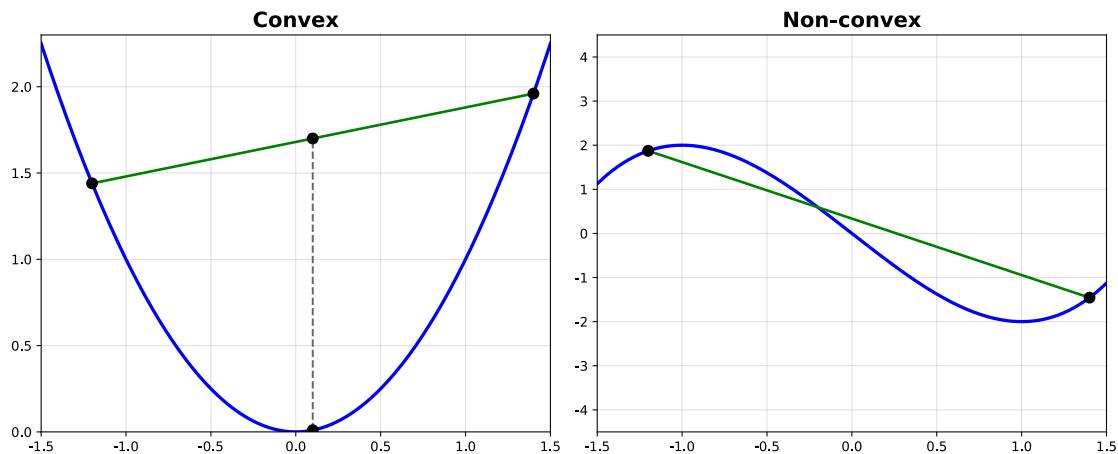


图 11.3 凸函数与非凸函数示例图

判断单变量函数是否为凸函数的另一种数学方法是计算其二阶导数，并检查其值是否始终大于0。

$$\frac{d^2 f(x)}{dx^2} > 0, \text{ 对于定义域内所有的 } x \quad (11.2)$$

我们研究一个由以下公式给出的简单二次函数：

$$f(x) = x^2 - x + 3 \quad (11.3)$$

其一阶导数和二阶导数分别为：

$$\begin{aligned} \frac{df(x)}{dx} &= 2x - 1 \\ \frac{d^2 f(x)}{dx^2} &= 2 \end{aligned} \quad (11.4)$$

由于二阶导数始终大于0，该函数为严格凸函数。

梯度下降法同样可应用于拟凸函数 (**quasi-convex functions**)。然而这类函数常存在所谓鞍点 (**saddle points**)，梯度下降法可能在此陷入停滞。一个拟凸函数的示例如下：

$$\begin{aligned} f(x) &= x^4 - 2x^3 + 2 \\ \frac{df(x)}{dx} &= 4x^3 - 6x^2 = x^2(4x - 6) \\ \frac{d^2 f(x)}{dx^2} &= 12x^2 + 12x = 12x(x - 1) \end{aligned} \quad (11.5)$$

我们注意到一阶导数在 $x = 0$ 和 $x = 1.5$ 处为0，这些位置是函数极值点（极小值或极大值）的候选点——该处斜率为零。但首先需要验证二阶导数的情况。

在 $x = 0$ 和 $x = 1$ 处，二阶导数的值为0。这些位置被称为拐点——即曲率改变符号的地方——意味着函数从凸变为凹，或反之亦然。通过分析这个方程，我们得出以下结论：

- 当 $x < 0$ 时：函数是凸的
- 当 $0 < x < 1$ 时：函数是凹的（二阶导数 < 0 ）
- 当 $x > 1$ 时：函数再次变为凸的

现在我们看到点 $x = 0$ 处的一阶导数和二阶导数均为0，这表明此处是鞍点，而点 $x = 1.5$ 则是全局最小值点。

我们来看一下这个函数的图像。如前计算，鞍点位于 $x = 0$ 处，最小值点位于 $x = 1.5$ 处。

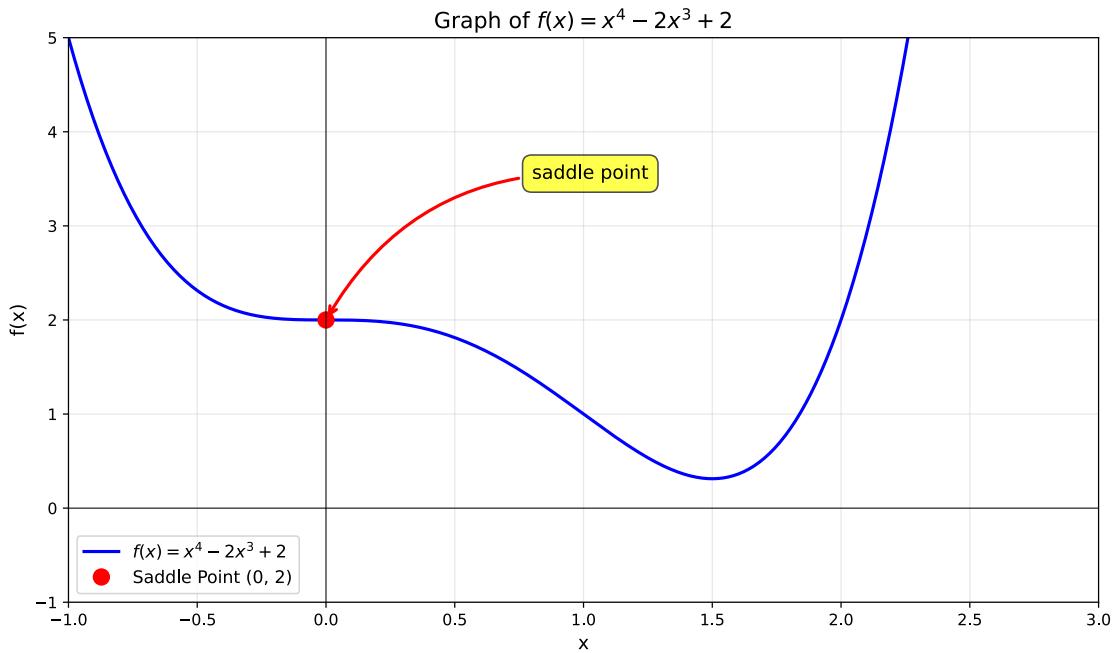


图 11.4 具有鞍点的半凸 (semi-convex) 函数

对于多变量函数，判断某点是否为鞍点的最合适方法是计算 **Hessian** 矩阵，我们后面再讲。一个双变量函数 $z = x^2 - y^2$ 的鞍点的示例如下图所示。

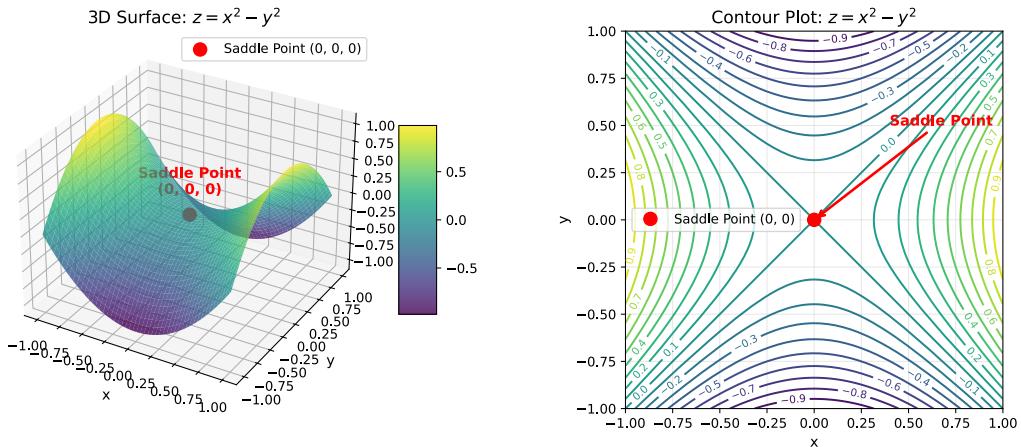


图 11.5 $z = x^2 - y^2$ 的鞍点示意图

11.3 梯度

直观地说，梯度代表了在指定方向上某一点处曲线的斜率。

对于单变量函数来说，它就是选定点的一阶导数。对于多变量函数而言，梯度是沿各主方向（顺变量坐标轴）的导数向量。因为我们只关心沿某一坐标轴的斜率，而不在意其他方向的变化，所以这些导数被称为偏导数。

n 维函数 $f(x)$ 在给定点 p 处的梯度定义如下：

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix} \quad (11.6)$$

倒三角形 ∇ 就是所谓的 **nabla** 符号，读作“**del**”。为了更好地理解如何计算梯度，让我们为下面这个二维示例函数 $f(x) = 0.5x^2 + y^2$ 进行手动计算。

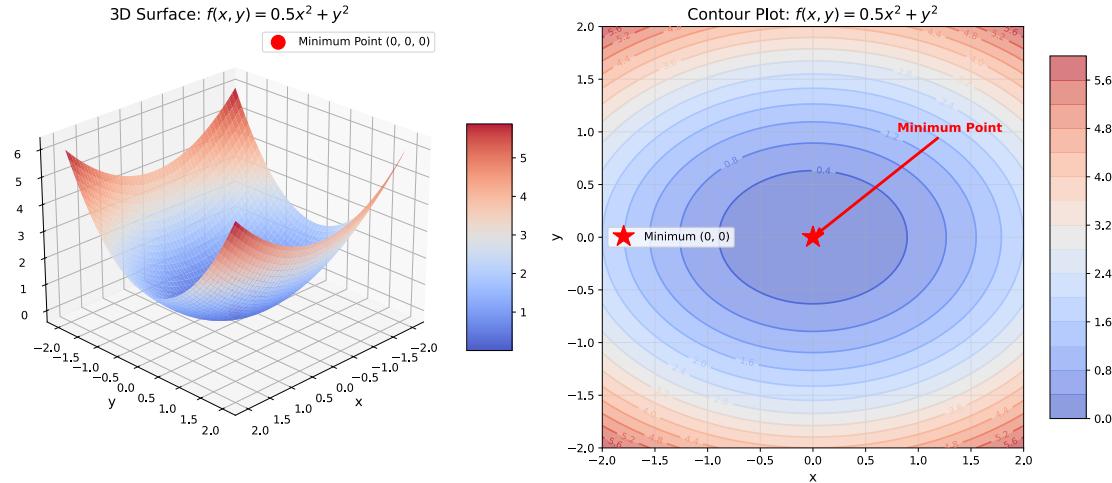


图 11.6 $f(x) = 0.5x^2 + y^2$ 示意图

假设我们关注点 $p(10, 10)$ 处的梯度：

$$\begin{aligned} \frac{\partial f(x, y)}{\partial x} &= x \\ \frac{\partial f(x, y)}{\partial y} &= 2y \end{aligned} \quad (11.7)$$

因此可得：

$$\begin{aligned} \nabla f(x, y) &= \begin{bmatrix} x \\ 2y \end{bmatrix} \\ \nabla f(10, 10) &= \begin{bmatrix} 10 \\ 20 \end{bmatrix} \end{aligned} \quad (11.8)$$

观察这些数值可知，沿 y 轴方向的斜率是 x 轴方向的两倍。

11.4 梯度下降法

梯度下降法迭代地利用当前位置的梯度计算下一个点，通过学习率进行缩放，并从当前位置减去所得值（即执行一步移动）。之所以减去该值，是因为我们想要最小化函数（若要最大化则应相加）。这一过程可以写作：

$$p_{n+1} = p_n - \eta \nabla f(p_n) \quad (11.9)$$

存在一个关键参数 η ，它通过缩放梯度来控制步长大小。在机器学习中，该参数被称为学习率，对算法性能具有重要影响。

- 学习率越小，梯度下降收敛所需时间越长，甚至可能在达到最优解前触及最大迭代次数限制。
- 若学习率过大，算法可能无法收敛至最优点（持续震荡），甚至可能完全发散。

总而言之，梯度下降法的步骤包括：



图 11.7 梯度下降法步骤

梯度下降法代码实现 Python

```

1 import numpy as np
2 from typing import Callable
3
4 def gradient_descent(start: float, gradient: Callable[[float], float],
5                       learn_rate: float, max_iter: int, tol: float = 0.01):
6     x = start
7     steps = [start] # 历史跟踪
8
9     for _ in range(max_iter):
10        diff = learn_rate * gradient(x)
11        if np.abs(diff) < tol:
12            break
13        x = x - diff
14        steps.append(x) # 历史跟踪
15
16    return steps, x

```

这个函数接收 5 个参数：

1. 起始点 [float]——我们这里手动定义了起始点，但在实践中，起始点通常是随机初始化的。
2. 梯度函数 [object]——计算梯度的函数（需要实现定义好然后传给上面的函数）
3. 学习率 [float]——步长的缩放因子
4. 最大迭代次数 [int]
5. 阈值 [float]——算法停止的一个条件（这里默认是 0.01）

11.5 示例 1——二次函数

我们的二次函数为

$$f(x) = x^2 - 4x + 1 \quad (11.10)$$

由于是单变量函数，所以梯度函数为

$$\frac{df(x)}{dx} = 2x - 4 \quad (11.11)$$

写成代码如下

```

1 def func1(x: float):
2     return x ** 2 - 4 * x + 1
3
4 def gradient_func1(x: float):
5     return 2 * x - 4

```

Python

当选择起始点 $x = 9$ 以及学习率为0.1时，我们可以手动计算一下每一步的过程。例如前三步如下：

$$\begin{aligned}
 x_0 &= 9 \\
 x_1 &= 9 - 0.1 \times (2 \times 9 - 4) = 7.6 \\
 x_2 &= 7.6 - 0.1 \times (2 \times 7.6 - 4) = 6.48 \\
 x_3 &= 6.48 - 0.1 \times (2 \times 6.48 - 4) = 5.584
 \end{aligned} \tag{11.12}$$

代码如下

```
1 history, result = gradient_descent(9, gradient_func1, 0.1, 100)
```

Python

如图所示，对于较小的学习率，随着算法逼近最小值，步长逐渐变小。而较大的学习率则在收敛前在两侧来回跳跃。

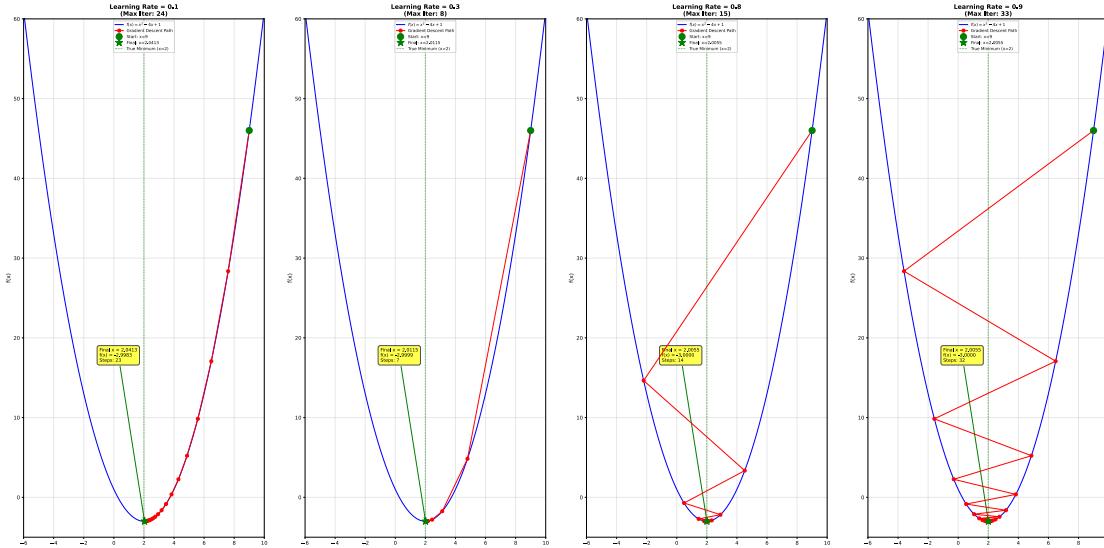


图 11.8 不同学习率的对比

11.6 示例 2——包含鞍点的函数

现在让我们看看算法将如何处理我们先前进行数学分析的半凸函数。

$$f(x) = x^4 - 2x^3 + 2 \tag{11.13}$$

下方展示了两种学习率与两种不同起始点的运算结果。

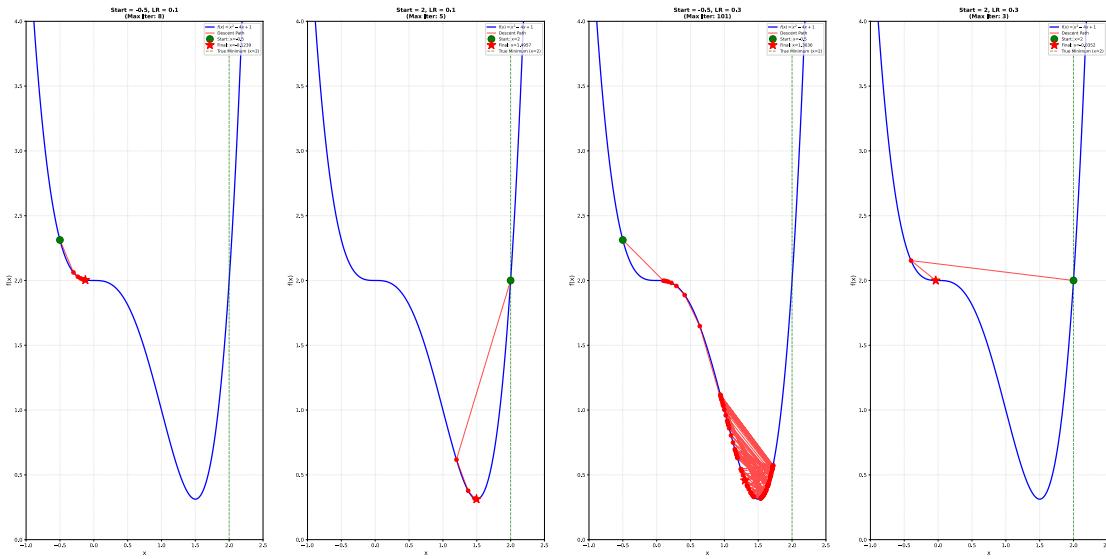


图 11.9 梯度下降法尝试逃离鞍点示意图

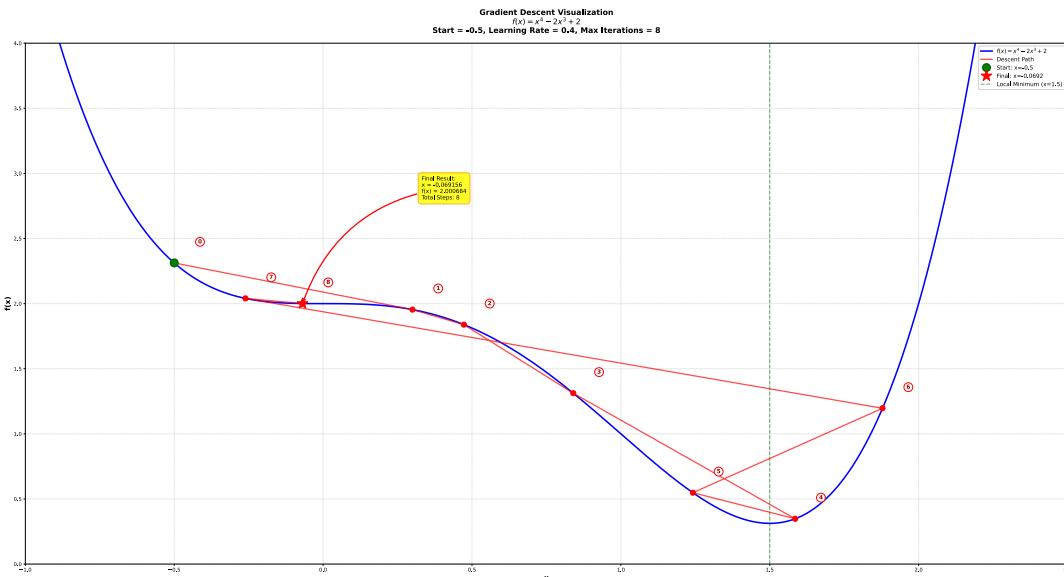


图 11.10 没有逃离鞍点

现在可以看到，鞍点的存在确实给一阶梯度下降法带来了严峻挑战，且无法保证最终能达到全局最小值。二阶优化算法（如牛顿法、拟牛顿法）在此类情况下的表现则更为出色。

我们探讨了梯度下降法的运作机制、适用场景以及使用过程中常见的挑战。后面我们会进一步探索更先进的基于梯度的优化方法，例如动量法、Nesterov 加速梯度下降、RMSprop、Adam，或是牛顿法等二阶优化方法。

IT

反向传播算法

线性层的求导公式推导

$$\begin{aligned}
 y &= w_1x_1 + w_2x_2 + w_3x_3 + b \\
 &= \underbrace{\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}}_{1 \times 3} \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}}_{3 \times 1} + \underbrace{[b]}_{1 \times 1}
 \end{aligned} \tag{11.14}$$

假设损失函数是 \mathcal{L} 。损失函数是一个标量值！

① 求解 $\frac{d\mathcal{L}}{dw}$

$$\begin{aligned}
 \frac{d\mathcal{L}}{dw} &= \frac{d\mathcal{L}}{dy} \cdot \frac{dy}{dw} \\
 &= \underbrace{\begin{bmatrix} \frac{dy}{dw} \\ \frac{dy}{dw} \\ \frac{dy}{dw} \end{bmatrix}}_{3 \times 1} \underbrace{\frac{d\mathcal{L}}{dy}}_{1 \times 1} \\
 &= \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \frac{d\mathcal{L}}{dy} \\
 &= x^T \cdot \underbrace{\frac{d\mathcal{L}}{dy}}_{\text{一个 } 1 \times 1 \text{ 的矩阵}}
 \end{aligned} \tag{11.15}$$

② 求解 $\frac{d\mathcal{L}}{db}$

$$\frac{d\mathcal{L}}{db} = \frac{d\mathcal{L}}{dy} \cdot \underbrace{\frac{d\mathcal{L}}{db}}_1 = \frac{d\mathcal{L}}{dy} \tag{11.16}$$

③ 求解 $\frac{d\mathcal{L}}{dx}$

$$\begin{aligned}
 \frac{d\mathcal{L}}{dx} &= \underbrace{\frac{d\mathcal{L}}{dy}}_{1 \times 1} \cdot \underbrace{\frac{dy}{dx}}_{1 \times 3} \\
 &= \frac{d\mathcal{L}}{dy} [w_1 \ w_2 \ w_3] \\
 &= \frac{d\mathcal{L}}{dy} w^T
 \end{aligned} \tag{11.17}$$

带批次的梯度

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \underbrace{\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{bmatrix}}_{2 \times 3} \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}}_{3 \times 1} + \underbrace{[b]}_{\text{广播!}} \tag{11.18}$$

$$\frac{d\mathcal{L}}{dw} = \begin{bmatrix} \frac{d\mathcal{L}}{dw_1} \\ \frac{d\mathcal{L}}{dw_2} \\ \frac{d\mathcal{L}}{dw_3} \end{bmatrix} \tag{11.19}$$

①

$$\begin{aligned}
\frac{d\mathcal{L}}{dw_1} &= \frac{d\mathcal{L}}{dy} \cdot \frac{dy}{dw_1} \\
&= \frac{d\mathcal{L}}{dy_1} \cdot \frac{dy_1}{dw_1} + \frac{d\mathcal{L}}{dy_2} \cdot \frac{dy_2}{dw_1} \\
&= \frac{d\mathcal{L}}{dy_1} \cdot x_{11} + \frac{d\mathcal{L}}{dy_2} \cdot x_{21}
\end{aligned} \tag{11.20}$$

所以

$$\begin{aligned}
\frac{d\mathcal{L}}{dw} &= \begin{bmatrix} \frac{d\mathcal{L}}{dy_1} \cdot x_{11} + \frac{d\mathcal{L}}{dy_2} \cdot x_{21} \\ \frac{d\mathcal{L}}{dy_1} \cdot x_{12} + \frac{d\mathcal{L}}{dy_2} \cdot x_{22} \\ \frac{d\mathcal{L}}{dy_1} \cdot x_{13} + \frac{d\mathcal{L}}{dy_2} \cdot x_{23} \end{bmatrix} \\
&= \begin{bmatrix} x_{11} & x_{21} \\ x_{12} & x_{22} \\ x_{13} & x_{23} \end{bmatrix} \begin{bmatrix} \frac{d\mathcal{L}}{dy_1} \\ \frac{d\mathcal{L}}{dy_2} \end{bmatrix} \\
&= x^T \cdot \frac{d\mathcal{L}}{dy}
\end{aligned} \tag{11.21}$$

(2)

$$\frac{d\mathcal{L}}{db} = \frac{d\mathcal{L}}{dy_1} \frac{dy_1}{db} + \frac{d\mathcal{L}}{dy_2} \frac{dy_2}{db} = \frac{d\mathcal{L}}{dy_1} + \frac{d\mathcal{L}}{dy_2} \tag{11.22}$$

(3)

$$\frac{d\mathcal{L}}{dx} = \frac{d\mathcal{L}}{dy} w^T \tag{11.23}$$

softmax 函数的求导

$$\vec{o} = [o_1 \ o_2 \ o_3 \ \cdots \ o_N] \tag{11.24}$$

$$\text{softmax}(\vec{o}) = [s_1 \ s_2 \ s_3 \ \cdots \ s_N] \tag{11.25}$$

其中

$$s_i = \frac{e^{o_i}}{\sum_{k=1}^N e^{o_k}} \tag{11.26}$$

如果 $i = j$, 那么有如下:

$$\begin{aligned}
\frac{ds_i}{do_j} &= \frac{(\sum_{k=1}^N e^{o_k}) e^{o_i} - e^{o_i} e^{o_j}}{(\sum_{k=1}^N e^{o_k})^2} \\
&= \frac{e^{o_i} (\sum_{k=1}^N e^{o_k} - e^{o_j})}{(\sum_{k=1}^N e^{o_k})^2} \\
&= \frac{e^{o_i}}{\sum_{k=1}^N e^{o_k}} \cdot \frac{\sum_{k=1}^N e^{o_k} - e^{o_j}}{\sum_{k=1}^N e^{o_k}} \\
&= s_i (1 - s_j)
\end{aligned} \tag{11.27}$$

如果 $i \neq j$, 那么有如下:

$$\begin{aligned}
\frac{ds_i}{do_j} &= \frac{\left(\sum_{k=1}^N e^{o_k}\right)0 - e^{o_i}e^{o_j}}{\left(\sum_{k=1}^N e^{o_k}\right)^2} \\
&= -\frac{e^{o_i}}{\sum_{k=1}^N e^{o_k}} \frac{e^{o_j}}{\sum_{k=1}^N e^{o_k}} \\
&= -s_i s_j
\end{aligned} \tag{11.28}$$

所以

$$\begin{aligned}
J &= \begin{bmatrix} \frac{ds_1}{do_1} & \frac{ds_1}{do_2} & \frac{ds_1}{do_3} \\ \frac{ds_2}{do_1} & \frac{ds_2}{do_2} & \frac{ds_2}{do_3} \\ \frac{ds_3}{do_1} & \frac{ds_3}{do_2} & \frac{ds_3}{do_3} \end{bmatrix} \\
&= \begin{bmatrix} s_1(1-s_1) & -s_1s_2 & -s_1s_3 \\ -s_2s_1 & s_2(1-s_2) & -s_2s_3 \\ -s_3s_1 & -s_3s_2 & s_3(1-s_3) \end{bmatrix}
\end{aligned} \tag{11.29}$$

softmax 的数值稳定性

$$\frac{e^{x-c}}{\sum e^{x-c}} = \frac{e^x/e^c}{(\sum e^x)/e^c} \tag{11.30}$$

如果引入损失函数，也就是说，我们想要求解 $\frac{d\mathcal{L}}{do_j}$ ，而 o_j 对所有 s_1, s_2, s_3, \dots 都有梯度。那么有如下：

$$\begin{aligned}
\frac{d\mathcal{L}}{do_j} &= \sum_i \frac{d\mathcal{L}}{ds_i} \cdot \frac{ds_i}{do_j} \\
&= \sum_i \frac{d\mathcal{L}}{ds_i} (s_i(\delta_{ij} - s_j)) \\
&= \frac{d\mathcal{L}}{ds_j} (s_j(1 - s_j)) + \sum_{i \neq j} \frac{d\mathcal{L}}{ds_i} (-s_i s_j) \\
&= \frac{d\mathcal{L}}{ds_j} s_j - \frac{d\mathcal{L}}{ds_j} s_j s_j - \sum_{i \neq j} \frac{d\mathcal{L}}{ds_i} (s_i s_j) \\
&= \frac{d\mathcal{L}}{ds_j} s_j - \sum_i \frac{d\mathcal{L}}{ds_i} (s_i s_j) \\
&= s_j \left(\frac{d\mathcal{L}}{ds_j} - \underbrace{\sum_i \frac{d\mathcal{L}}{ds_i} s_i}_{\text{点积}} \right)
\end{aligned} \tag{11.31}$$

其中 $\delta_{ij} = 1$ if $i = j$ else 0
写成向量表示如下

$$s \cdot \left(\frac{d\mathcal{L}}{ds} - \left(\frac{d\mathcal{L}}{ds} \cdot s \right) \right) \tag{11.32}$$

ReLU 的梯度

$$\begin{bmatrix} -2 \\ 3 \\ 8 \end{bmatrix} \rightarrow \text{ReLU}(x) \rightarrow \begin{bmatrix} 0 \\ 3 \\ 8 \end{bmatrix} \tag{11.33}$$

$$\frac{dy}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \quad (11.34)$$

LayerNorm 的梯度

- 数据: (B, S, E)
- 沿着 E 进行归一化

$$\begin{aligned} \mu_{b,s} &= \frac{1}{E} \sum_{e=1}^E x_{b,s,e} \\ \text{var}_{b,s} &= \frac{1}{E} \sum_{e=1}^E (x_{b,s,e} - \mu_{b,s})^2 \\ \tilde{x}_{b,s,e} &= \frac{x_{b,s,e} - \mu_{b,s}}{\sqrt{\text{var}_{b,s} + \epsilon}} \\ y_{b,s,e} &= \underbrace{\gamma_e}_{\text{learnable scale param}} \cdot \tilde{x}_{b,s,e} + \underbrace{\beta_e}_{\text{learnable shift param}} \end{aligned} \quad (11.35)$$

给定上游的梯度 $\frac{dL}{dy_{b,s,e}}$, 我们需要如下:

$$\frac{dL}{d\gamma_e}, \frac{dL}{d\beta_e}, \frac{dL}{dx_{b,s,e}}$$

还记得我们的 B, S 索引都是独立的, 所以梯度将会累积。为了简单, 我们会忽略它们。

$$\begin{aligned} \mu &= \frac{1}{E} \sum_{k=1}^E x_k \\ \mu_k &\triangleq x_k - \mu \\ \sigma^2 &= \frac{1}{E} \sum_{k=1}^E \mu_k^2 \\ s &= \sqrt{\sigma^2 + \epsilon} \\ \hat{x}_j &= \frac{\mu_j}{s} = \frac{x_j - \mu}{s} \quad s \text{ 和 } \mu \text{ 都有 } x \text{ 里面} \\ y_j &= \gamma_j \hat{x}_j + \beta_j \\ \nabla_j &= \frac{dL}{dy_j} \leftarrow \text{上游的梯度} \quad (11.36) \\ \frac{dL}{dx_i} &= \sum_{j=1}^E \frac{dL}{dy_j} \cdot \frac{dy_j}{dx_i} \\ &= \sum_{j=1}^E \nabla_j \underbrace{\frac{dy_j}{d\hat{x}_j}}_{\gamma_j} \cdot \frac{d\hat{x}_j}{dx_i} \\ &= \sum_{j=1}^E \nabla_j \cdot \gamma_j \underbrace{\frac{d\hat{x}_j}{dx_i}}_{\text{solve this}} \end{aligned}$$

$$\mu = \frac{1}{E} \sum_{k=1}^E x_k \rightarrow \frac{d\mu}{dx_i} = \frac{1}{E} \quad (11.37)$$

$$\mu_j = x_j - \mu \rightarrow \frac{d\mu_j}{dx_i} = \delta_{ij} - \frac{1}{E} \quad \mu_{ij} = 1 \text{ if } i = j \text{ else } 0 \quad (11.38)$$

$$\begin{aligned}
\sigma^2 &= \frac{1}{E} \sum_{k=1}^E \mu_k^2 \\
\frac{d\sigma^2}{dx_i} &= \frac{1}{E} \sum_{k=1}^E 2\mu_k \cdot \frac{d\mu_k}{dx_i} \\
&= \frac{2}{E} \sum_{k=1}^E \mu_k \left(\delta_{ik} - \frac{1}{E} \right) \\
&= \frac{2}{E} \left(\sum_{\substack{k=1 \\ 0 \text{ if } i \neq k}}^E \mu_k - \sum_{k=1}^E \mu_k \frac{1}{E} \right) \\
&= \frac{2}{E} \left(\mu_i - \frac{1}{E} \sum_{k=1}^E \mu_k \right)
\end{aligned} \tag{11.39}$$

其中 $\frac{1}{E} \sum_{k=1}^E \mu_k = \frac{1}{E} \sum_{k=1}^E x_k - \mu = 0$

$$\therefore \frac{d\sigma^2}{dx_i} = \frac{2}{E} \mu_i \tag{11.40}$$

$$s = \sqrt{\sigma^2 + \varepsilon} = (\sigma^2 + \varepsilon)^{\frac{1}{2}} \tag{11.41}$$

$$\begin{aligned}
\frac{ds}{dx_i} &= \frac{1}{2} (\sigma^2 + \varepsilon)^{-\frac{1}{2}} \cdot \frac{d(\sigma^2 + \varepsilon)}{dx_i} \text{ 是常数} \\
&= \frac{1}{2\sqrt{\sigma^2 + \varepsilon}} \frac{2}{E} \mu_i \\
&= \frac{1}{s} \frac{1}{E} \mu_i \\
\therefore \frac{ds}{dx_i} &= \frac{\mu_i}{sE}
\end{aligned} \tag{11.42}$$

$$\hat{x}_j = \frac{\mu_j}{s} = \mu_j \frac{1}{s} \tag{11.43}$$

所以根据求导的乘法公式

$$\frac{d\hat{x}_j}{dx_i} = \underbrace{\mu_j \cdot \frac{d(\frac{1}{s})}{dx_i}}_{\textcircled{1}} + \underbrace{\frac{d\mu_j}{dx_i} \frac{1}{s}}_{\textcircled{2}} \tag{11.44}$$

①

$$\begin{aligned}
\mu_j \cdot \frac{d(\frac{1}{s})}{dx_i} &= \mu_j \frac{ds^{-1}}{dx_i} = -s^{-2} \frac{ds}{dx_i} \mu_j \\
&= -\frac{1}{s^2} \mu_j \frac{\mu_i}{sE} = -\frac{\mu_i \mu_j}{s^3 E}
\end{aligned} \tag{11.45}$$

②

$$\begin{aligned}
\frac{d\mu_j}{dx_i} \frac{1}{s} &= \left(\delta_{ij} - \frac{1}{E} \right) \left(\frac{1}{s} \right) \\
\therefore \frac{d\hat{x}_j}{dx_i} &= -\frac{\mu_i}{s^3 E} + \left(\delta_{ij} - \frac{1}{E} \right) \left(\frac{1}{s} \right) \\
&= \frac{1}{s} \left(\delta_{ij} - \frac{1}{E} - \frac{\mu_i \mu_j}{s^2 E} \right) \quad \hat{x}_i = \frac{\mu_i}{s} \\
&= \frac{1}{s} \left(\delta_{ij} - \frac{1}{E} - \frac{\hat{x}_i \hat{x}_j}{E} \right)
\end{aligned} \tag{11.46}$$

我们还记得

$$\begin{aligned}
\frac{dL}{dx_i} &= \sum_{j=1}^E \nabla_j \gamma_j \frac{d\hat{x}_j}{dx_i} \\
&= \sum_{j=1}^E \nabla_j \gamma_j \frac{1}{s} \left[\delta_{ij} - \frac{1}{E} - \frac{\hat{x}_i \hat{x}_j}{E} \right] \\
&= \frac{1}{s} \left[\sum_{j=1}^E \nabla_j \gamma_j \delta_{ij} - \frac{1}{E} \sum_{j=1}^E \nabla_j \gamma_j - \frac{1}{E} \sum_{j=1}^E \nabla_j \gamma_j \hat{x}_i \hat{x}_j \right] \\
&= \frac{1}{s} \left[\nabla_i \gamma_i - \frac{1}{E} \sum_{j=1}^E \nabla_j \gamma_j - \frac{\hat{x}_i}{E} \sum_{j=1}^E \nabla_j \gamma_j \hat{x}_j \right]
\end{aligned} \tag{11.47}$$

我们还记得

$$\nabla = \frac{dL}{dy} \text{ and } \frac{dy}{d\hat{x}} = \gamma \tag{11.48}$$

所以为了简单，我们上游的针对 \hat{x} 的梯度（不仅是 y ）是：

$$\begin{aligned}
\frac{dL}{dy} \cdot \frac{dy}{d\hat{x}} &= \nabla \gamma \triangleq \tilde{\nabla} \\
&= \frac{1}{s} \left[\tilde{\nabla} - \frac{1}{E} \sum_{j=1}^E \tilde{\nabla} - \frac{\hat{x}_i}{E} \sum_{j=1}^E \tilde{\nabla} \hat{x}_j \right] \text{ 点积} \\
&= \frac{1}{\sqrt{\sigma^2 + \varepsilon}} [\tilde{\nabla} - \text{mean}(\tilde{\nabla}) - \hat{x} \cdot \text{mean}(\tilde{\nabla} \cdot \hat{x})]
\end{aligned} \tag{11.50}$$

所以最终有

$$\begin{aligned}
\frac{dL}{d\beta_j} &= \sum_{B,S} \frac{dL}{dy} \cdot \frac{dy}{d\beta_j} = \sum_{B,S} \nabla \\
\frac{dL}{d\gamma_j} &= \sum_{B,S} \frac{dL}{dy} \cdot \frac{dy}{d\gamma_j} = \sum_{B,S} \nabla \hat{x}_j
\end{aligned} \tag{11.51}$$

Multihed Attention 的梯度