

深度学习简介

(注：此节完全同[原文](#)，为了完整性而搬运过来) 你可能已经接触过编程，并开发过一两款程序。同时你可能读过关于深度学习或者机器学习的铺天盖地的报道，尽管很多时候它们被赋予了更广义的名字：人工智能。实际上，或者说幸运的是，大部分程序并不需要深度学习或者是更广义上的人工智能技术。例如，如果我们要为一台微波炉编写一个用户界面，只需要一点儿工夫我们便能设计出十几个按钮以及一系列能精确描述微波炉在各种情况下的表现的规则。再比如，假设我们要编写一个电子邮件客户端。这样的程序比微波炉要复杂一些，但我们还是可以沉下心来一步一步思考：客户端的用户界面将需要几个输入框来接受收件人、主题、邮件正文等，程序将监听键盘输入并写入一个缓冲区，然后将它们显示在相应的输入框中。当用户点击“发送”按钮时，我们需要检查收件人邮箱地址的格式是否正确，并检查邮件主题是否为空，或在主题为空时警告用户，而后用相应的协议传送邮件。

值得注意的是，在以上两个例子中，我们都不需要收集真实世界中的数据，也不需要系统地提取这些数据的特征。只要有充足的时间，我们的常识与编程技巧已经足够让我们完成任务。

与此同时，我们很容易就能找到一些连世界上最好的程序员也无法仅用编程技巧解决的简单问题。例如，假设我们想要编写一个判定一张图像中有没有猫的程序。这件事听起来好像很简单，对不对？程序只需要对每张输入图像输出“真”（表示有猫）或者“假”（表示无猫）即可。但令人惊讶的是，即使是世界上最优秀的计算机科学家和程序员也不懂如何编写这样的程序。

我们该从哪里入手呢？我们先进一步简化这个问题：若假设所有图像的高和宽都是同样的 400 像素大小，一个像素由红绿蓝三个值构成，那么一张图像就由近 50 万个数值表示。那么哪些数值隐藏着我们需要的信息呢？是所有数值的平均数，还是四个角的数值，抑或是图像中的某一个特别的点？事实上，要想解读图像中的内容，需要寻找仅仅在结合成千上万的数值时才会出现的特征，如边缘、质地、形状、眼睛、鼻子等，最终才能判断图像中是否有猫。

一种解决以上问题的思路是逆向思考。与其设计一个解决问题的程序，不如从最终的需求入手来寻找一个解决方案。事实上，这也是目前的机器学习和深度学习应用共同的核心思想：我们可以称其为“用数据编程”。与其枯坐在房间里思考怎么设计一个识别猫的程序，不如利用人类肉眼在图像中识别猫的能力。我们可以收集一些已知包含猫与不包含猫的真实图像，然后我们的目标就转化成如何从这些图像入手得到一个可以推断出图像中是否有猫的函数。这个函数的形式通常通过我们的知识来针对特定问题选定。例如，我们使用一个二次函数来判断图像中是否有猫，但是像二次函数系数值这样的函数参数的具体值则是通过数据来确定。

通俗来说，机器学习是一门讨论各式各样的适用于不同问题的函数形式，以及如何使用数据来有效地获取函数参数具体值的学科。深度学习是指机器学习中的一类函数，它们的形式通常为多层神经网络。近年来，仰仗着大数据集和强大的硬件，深度学习已逐渐成为处理图像、文本语料和声音信号等复杂高维度数据的主要方法。

我们现在正处于一个程序设计得到深度学习的帮助越来越多的时代。这可以说是计算机科学历史上的一个分水岭。举个例子，深度学习已经在你的手机里：拼写校正、语音识别、认出社交媒体照片里的好朋友们等。得益于优秀的算法、快速而廉价的算力、前所未有的大量数据以及强大的软件工具，如今大多数软件工程师都有能力建立复杂的模型来解决十年前连最优秀的科学家都觉得棘手的问题。

本书希望能帮助读者进入深度学习的浪潮中。我们希望结合数学、代码和样例让深度学习变得触手可及。本书不要求具有高深的数学或编程背景，我们将随着章节的发展逐一解释所需要的知识。更值得一提的是，本书的每一节都是一个可以独立运行的 Jupyter 记事本。读者可以从网上获得这些记事本，并且可以在个人电脑或云端服务器上执行它们。这样读者就可以随意改动书中的代码并得到及时反馈。我们希望本书能帮助和启发新一代的程序员、创业者、统计学家、生物学家，以及所有对深度学习感兴趣的人。

起源

虽然深度学习似乎是最近几年刚兴起的名词，但它所基于的神经网络模型和用数据编程的核心思想已经被研究了数百年。自古以来，人类就一直渴望能从数据中分析出预知未来的窍门。实际上，数据分析正是大部分自然科学的本质，我们希望从日常的观测中提取规则，并找寻不确定性。

早在 17 世纪，[雅各比·伯努利 \(1655–1705\)](#) 提出了描述只有两种结果的随机过程（如抛掷一枚硬币）的伯努利分布。大约一个世纪之后，[卡尔·弗里德里希·高斯 \(1777–1855\)](#) 发明了今日仍广泛用在从保险计算到医学诊断等领域的最小二乘法。概率论、统计学和模式识别等工具帮助自然科学的实验学家们从数据回归到自然定律，从而发现了如欧姆定律（描述电阻两端电压和流经电阻电流关系的定律）这类可以用线性模型完美表达的一系列自然法则。

即使是在中世纪，数学家也热衷于利用统计学来做出估计。例如，在[雅各比·科贝尔 \(1460–1533\)](#) 的几何书中记载了使用 16 名男子的平均脚长来估计男子的平均脚长。

图 1.1 在中世纪，16 名男子的平均脚长被用来估计男子的平均脚长

如图 1.1 所示，在这个研究中，16 位成年男子被要求在离开教堂时站成一排并把脚贴在一起，而后他们脚的总长度除以 16 得到了一个估计：这个数字大约相当于今日的一英尺。这个算法之后又被改进，以应对特异形状的脚：最长和最短的脚不计入，只对剩余的脚长取平均值，即裁剪平均值的雏形。

现代统计学在 20 世纪的真正起飞要归功于数据的收集和发布。统计学巨匠之一罗纳德·费雪（1890–1962）对统计学理论和统计学在基因学中的应用功不可没。他发明的许多算法和公式，例如线性判别分析和费雪信息，仍经常被使用。即使是他在 1936 年发布的 Iris 数据集，仍然偶尔被用于演示机器学习算法。

克劳德·香农（1916–2001）的信息论以及阿兰·图灵（1912–1954）的计算理论也对机器学习有深远影响。图灵在他著名的论文《计算机与智能》中提出了“机器可以思考吗？”这样一个问题 [1]。在他描述的“图灵测试”中，如果一个人在使用文本交互时不能区分他的对话对象到底是人类还是机器的话，那么即可认为这台机器是有智能的。时至今日，智能机器的发展可谓日新月异。

另一个对深度学习有重大影响的领域是神经科学与心理学。既然人类显然能够展现出智能，那么对于解释并逆向工程人类智能机理的探究也在情理之中。最早的算法之一是由唐纳德·赫布（1904–1985）正式提出的。在他开创性的著作《行为的组织》中，他提出神经是通过正向强化来学习的，即赫布理论 [2]。赫布理论是感知机学习算法的原型，并成为支撑今日深度学习的随机梯度下降算法的基石：强化合意的行为、惩罚不合意的行为，最终获得优良的神经网络参数。

来源于生物学的灵感是神经网络名字的由来。这类研究者可以追溯到一个多世纪前的亚历山大·贝恩（1818–1903）和查尔斯·斯科特·谢灵顿（1857–1952）。研究者们尝试组建模仿神经元互动的计算电路。随着时间发展，神经网络的生物学解释被稀释，但仍保留了这个名字。时至今日，绝大多数神经网络都包含以下的核心原则。

- 交替使用线性处理单元与非线性处理单元，它们经常被称为“层”。
- 使用链式法则（即反向传播）来更新网络的参数。

在最初的快速发展之后，自约 1995 年起至 2005 年，大部分机器学习研究者的视线从神经网络上移开了。这是由于多种原因。首先，训练神经网络需要极强的计算力。尽管 20 世纪末内存已经足够，计算力却不够充足。其次，当时使用的数据集也相对小得多。费雪在 1936 年发布的 Iris 数据集仅有 150 个样本，并被广泛用于测试算法的性能。具有 6 万个样本的 MNIST 数据集在当时已经被认为是非常庞大了，尽管它如今已被认为是典型的简单数据集。由于数据和计算力的稀缺，从经验上来说，如核方法、决策树和概率图模型等统计工具更优。它们不像神经网络一样需要长时间的训练，并且在强大的理论保证下提供可以预测的结果。

发展

互联网的崛起、价廉物美的传感器和低价的存储器令我们越来越容易获取大量数据。加之便宜的计算力，尤其是原本为电脑游戏设计的 GPU 的出现，上文描述的情况

改变了许多。一瞬间，原本被认为不可能的算法和模型变得触手可及。这样的发展趋势从如下表格中可见一斑。

年代	数据样本个数	内存	每秒浮点计算数
1970	100 (Iris)	1 KB	100 K (Intel 8080)
1980	1 K (波士顿房价)	100 KB	1 M (Intel 80186)
1990	10 K (手写字符识别)	10 MB	10 M (Intel 80486)
2000	10 M (网页)	100 MB	1 G (Intel Core)
2010	10 G (广告)	1 GB	1 T (NVIDIA C2050)
2020	1 T (社交网络)	100 GB	1 P (NVIDIA DGX-2)

很显然，存储容量没能跟上数据量增长的步伐。与此同时，计算力的增长又盖过了数据量的增长。这样的趋势使得统计模型可以在优化参数上投入更多的计算力，但同时需要提高存储的利用效率，例如使用非线性处理单元。这也相应导致了机器学习和统计学的最优选择从广义线性模型及核方法变化为深度多层神经网络。这样的变化正是诸如多层感知机、卷积神经网络、长短期记忆循环神经网络和 Q 学习等深度学习的支柱模型在过去 10 年从坐了数十年的冷板凳上站起来被“重新发现”的原因。

近年来在统计模型、应用和算法上的进展常被拿来与寒武纪大爆发（历史上物种数量大爆发的一个时期）做比较。但这些进展不仅仅是因为可用资源变多了而让我们得以用新瓶装旧酒。下面的列表仅仅涵盖了近十年来深度学习长足发展的部分原因。

- 优秀的容量控制方法，如丢弃法，使大型网络的训练不再受制于过拟合（大型神经网络学会记忆大部分训练数据的行为）[3]。这是靠在整个网络中注入噪声而达到的，如训练时随机将权重替换为随机的数字 [4]。
- 注意力机制解决了另一个困扰统计学超过一个世纪的问题：如何在不增加参数的情况下扩展一个系统的记忆容量和复杂度。注意力机制使用了一个可学习的指针结构来构建出一个精妙的解决方法 [5]。也就是说，与其在像机器翻译这样的任务中记忆整个句子，不如记忆指向翻译的中间状态的指针。由于生成译文前不需要再存储整句原文的信息，这样的结构使准确翻译长句变得可能。
- 记忆网络 [6] 和神经编码器—解释器 [7] 这样的多阶设计使得针对推理过程的迭代建模方法变得可能。这些模型允许重复修改深度网络的内部状态，这样就能模拟出推理链条上的各个步骤，就好像处理器在计算过程中修改内存一样。
- 另一个重大发展是生成对抗网络的发明 [8]。传统上，用在概率分布估计和生成模型上的统计方法更多地关注于找寻正确的概率分布，以及正确的采样算法。生成

对抗网络的关键创新在于将采样部分替换成了任意的含有可微分参数的算法。这些参数将被训练到使辨别器不能再分辨真实的和生成的样本。生成对抗网络可使用任意算法来生成输出的这一特性为许多技巧打开了新的大门。例如生成奔跑的斑马 [9] 和生成名流的照片 [10] 都是生成对抗网络发展的见证。

- 许多情况下单个 GPU 已经不能满足在大型数据集上进行训练的需要。过去 10 年内我们构建分布式并行训练算法的能力已经有了极大的提升。设计可扩展算法的最大瓶颈在于深度学习优化算法的核心：随机梯度下降需要相对更小的批量。与此同时，更小的批量也会降低 GPU 的效率。如果使用 1,024 个 GPU，每个 GPU 的批量大小为 32 个样本，那么单步训练的批量大小将是 32,000 个以上。近年来李沐 [11]、Yang You 等人 [12] 以及 Xianyan Jia 等人 [13] 的工作将批量大小增至多达 64,000 个样例，并把在 ImageNet 数据集上训练 ResNet-50 模型的时间降到了 7 分钟。与之对比，最初的训练时间需要以天来计算。
- 并行计算的能力也为至少在可以采用模拟情况下的强化学习的发展贡献了力量。并行计算帮助计算机在围棋、雅达利游戏、星际争霸和物理模拟上达到了超过人类的水准。
- 深度学习框架也在传播深度学习思想的过程中扮演了重要角色。[Caffe](#)、[Torch](#)和[Theano](#)这样的第一代框架使建模变得更简单。许多开创性的论文都用到了这些框架。如今它们已经被[TensorFlow](#)（经常是以高层 API [Keras](#)的形式被使用）、[CNTK](#)、[Caffe 2](#) 和[Apache MXNet](#)所取代。第三代，即命令式深度学习框架，是由用类似 NumPy 的语法来定义模型的 [Chainer](#)所开创的。这样的思想后来被 [PyTorch](#)和 MXNet 的[Gluon API](#) 采用，后者也正是本书用来教学深度学习的工具。

系统研究者负责构建更好的工具，统计学家建立更好的模型。这样的分工使工作大大简化。举例来说，在 2014 年时，训练一个逻辑回归模型曾是卡内基梅隆大学布置给机器学习方向的新入学博士生的作业问题。时至今日，这个问题只需要少于 10 行的代码便可以完成，普通的程序员都可以做到。

成功案例

长期以来机器学习总能完成其他方法难以完成的目标。例如，自 20 世纪 90 年代起，邮件的分拣就开始使用光学字符识别。实际上这正是知名的 MNIST 和 USPS 手写数字数据集的来源。机器学习也是电子支付系统的支柱，可以用于读取银行支票、进行授信评分以及防止金融欺诈。机器学习算法在网络上被用来提供搜索结果、个性化推荐和网页排序。虽然长期处于公众视野之外，但是机器学习已经渗透到了我们工作和生

活的方方面面。直到近年来，在此前认为无法被解决的问题以及直接关系到消费者的问题上取得突破性进展后，机器学习才逐渐变成公众的焦点。这些进展基本归功于深度学习。

- 苹果公司的 Siri、亚马逊的 Alexa 和谷歌助手一类的智能助手能以可观的准确率回答口头提出的问题，甚至包括从简单的开关灯具（对残疾群体帮助很大）到提供语音对话帮助。智能助手的出现或许可以作为人工智能开始影响我们生活的标志。
- 智能助手的关键是需要能够精确识别语音，而这类系统在某些应用上的精确度已经渐渐增长到可以与人类比肩 [14]。
- 物体识别也经历了漫长的发展过程。在 2010 年从图像中识别出物体的类别仍是一个相当有挑战性的任务。当年日本电气、伊利诺伊大学香槟分校和罗格斯大学团队在 ImageNet 基准测试上取得了 28% 的前五错误率 [15]。到 2017 年，这个数字降低到了 2.25% [16]。研究人员在鸟类识别和皮肤癌诊断上，也取得了同样惊世骇俗的成绩。
- 游戏曾被认为是人类智能最后的堡垒。自使用时间差分强化学习玩双陆棋的 TD-Gammon 开始，算法和算力的发展催生了一系列在游戏上使用的新算法。与双陆棋不同，国际象棋有更复杂的状态空间和更多的可选动作。“深蓝”用大量的并行、专用硬件和游戏树的高效搜索打败了加里·卡斯帕罗夫 [17]。围棋因其庞大的状态空间被认为是更难的游戏，AlphaGo 在 2016 年用结合深度学习与蒙特卡洛树采样的方法达到了人类水准 [18]。对德州扑克游戏而言，除了巨大的状态空间之外，更大的挑战是游戏的信息并不完全可见，例如看不到对手的牌。而“冷扑大师”用高效的策略体系超越了人类玩家的表现 [19]。以上的例子都体现出了先进的算法是人工智能在游戏上的表现提升的重要原因。
- 机器学习进步的另一个标志是自动驾驶汽车的发展。尽管距离完全的自动驾驶还有很长的路要走，但诸如Tesla、NVIDIA、MobilEye和Waymo这样的公司发布的具有部分自动驾驶功能的产品展示出了这个领域巨大的进步。完全自动驾驶的难点在于它需要将感知、思考和规则整合在同一个系统中。目前，深度学习主要被应用在计算机视觉的部分，剩余的部分还是需要工程师们的大量调试。

以上列出的仅仅是近年来深度学习所取得的成果的冰山一角。机器人学、物流管理、计算生物学、粒子物理学和天文学近年来的发展也有一部分要归功于深度学习。可以看到，深度学习已经逐渐演变成一个工程师和科学家皆可使用的普适工具。

特点

在描述深度学习的特点之前，我们先回顾并概括一下机器学习和深度学习的关系。机器学习研究如何使计算机系统利用经验改善性能。它是人工智能领域的分支，也是实现人工智能的一种手段。在机器学习的众多研究方向中，表征学习关注如何自动找出表示数据的合适方式，以便更好地将输入变换为正确的输出，而本书要重点探讨的深度学习是具有多级表示的表征学习方法。在每一级（从原始数据开始），深度学习通过简单的函数将该级的表示变换为更高级的表示。因此，深度学习模型也可以看作是由许多简单函数复合而成的函数。当这些复合的函数足够多时，深度学习模型就可以表达非常复杂的变换。

深度学习可以逐级表示越来越抽象的概念或模式。以图像为例，它的输入是一堆原始像素值。深度学习模型中，图像可以逐级表示为特定位置和角度的边缘、由边缘组合得出的花纹、由多种花纹进一步汇合得到的特定部位的模式等。最终，模型能够较容易根据更高级的表示完成给定的任务，如识别图像中的物体。值得一提的是，作为表征学习的一种，深度学习将自动找出每一级表示数据的合适方式。

因此，深度学习的一个外在特点是端到端的训练。也就是说，并不是将单独调试的部分拼凑起来组成一个系统，而是将整个系统组建好之后一起训练。比如说，计算机视觉科学家之前曾一度将特征抽取与机器学习模型的构建分开处理，像是 Canny 边缘探测 [20] 和 SIFT 特征提取 [21] 曾占据统治性地位达 10 年以上，但这也就是人类能找到的最好方法了。当深度学习进入这个领域后，这些特征提取方法就被性能更强的自动优化的逐级过滤器替代了。

相似地，在自然语言处理领域，词袋模型多年来都被认为是不二之选 [22]。词袋模型是将一个句子映射到一个词频向量的模型，但这样的做法完全忽视了单词的排列顺序或者句中的标点符号。不幸的是，我们也没有能力来手工抽取更好的特征。但是自动化的算法反而可以从所有可能的特征中搜寻最好的那个，这也带来了极大的进步。例如，语义相关的词嵌入能够在向量空间中完成如下推理：“柏林 - 德国 + 中国 = 北京”。可以看出，这些都是端到端训练整个系统带来的效果。

除端到端的训练以外，我们也正在经历从含参数统计模型转向完全无参数的模型。当数据非常稀缺时，我们需要通过简化对现实的假设来得到实用的模型。当数据充足时，我们就可以用能更好地拟合现实的无参数模型来替代这些含参数模型。这也使我们可以得到更精确的模型，尽管需要牺牲一些可解释性。

相对其它经典的机器学习方法而言，深度学习的不同在于：对非最优解的包容、对非凸非线性优化的使用，以及勇于尝试没有被证明过的方法。这种在处理统计问题上的新经验主义吸引了大量人才的涌入，使得大量实际问题有了更好的解决方案。尽管大部分情况下需要为深度学习修改甚至重新发明已经存在数十年的工具，但是这绝对是一件非常有意义并令人兴奋的事。

最后，深度学习社区长期以来以在学术界和企业之间分享工具而自豪，并开源了许多优秀的软件库、统计模型和预训练网络。正是本着开放开源的精神，本书的内容和基于它的教学视频可以自由下载和随意分享。我们致力于为所有人降低学习深度学习的门槛，并希望大家从中获益。

小结

- 机器学习研究如何使计算机系统利用经验改善性能。它是人工智能领域的分支，也是实现人工智能的一种手段。
- 作为机器学习的一类，表征学习关注如何自动找出表示数据的合适方式。
- 深度学习是具有多级表示的表征学习方法。它可以逐级表示越来越抽象的概念或模式。
- 深度学习所基于的神经网络模型和用数据编程的核心思想实际上已经被研究了数百年。
- 深度学习已经逐渐演变成一个工程师和科学家皆可使用的普适工具。

练习

- 你现在正在编写的代码有没有可以被“学习”的部分，也就是说，是否有可以被机器学习改进的部分？
- 你在生活中有没有这样的场景：虽有许多展示如何解决问题的样例，但缺少自动解决问题的算法？它们也许是深度学习的最好猎物。
- 如果把人工智能的发展看作是新一次工业革命，那么深度学习和数据的关系是否像是蒸汽机与煤炭的关系呢？为什么？
- 端到端的训练方法还可以用在哪里？物理学，工程学还是经济学？
- 为什么应该让深度网络模仿人脑结构？为什么不该让深度网络模仿人脑结构？

参考文献

- [1] Machinery, C. (1950). Computing machinery and intelligence-AM Turing. *Mind*, 59(236), 433.
- [2] Hebb, D. O. (1949). *The organization of behavior; a neuropsychological theory*. A Wiley Book in Clinical Psychology., 62-78.
- [3] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 1929-1958.

- [4] Bishop, C. M. (1995). Training with noise is equivalent to Tikhonov regularization. *Neural computation*, 7(1), 108-116.
- [5] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.
- [6] Sukhbaatar, S., Weston, J., & Fergus, R. (2015). End-to-end memory networks. In *Advances in neural information processing systems* (pp. 2440-2448).
- [7] Reed, S., & De Freitas, N. (2015). Neural programmer-interpreters. arXiv preprint arXiv:1511.06279.
- [8] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672-2680).
- [9] Zhu, J. Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. arXiv preprint.
- [10] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. arXiv preprint arXiv:1710.10196.
- [11] Li, M. (2017). Scaling Distributed Machine Learning with System and Algorithm Co-design (Doctoral dissertation, PhD thesis, Intel).
- [12] You, Y., Gitman, I., & Ginsburg, B. Large batch training of convolutional networks. ArXiv e-prints.
- [13] Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., ... & Chen, T. (2018). Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. arXiv preprint arXiv:1807.11205.
- [14] Xiong, W., Droppo, J., Huang, X., Seide, F., Seltzer, M., Stolcke, A., ... & Zweig, G. (2017, March). The Microsoft 2016 conversational speech recognition system. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on* (pp. 5255-5259). IEEE.
- [15] Lin, Y., Lv, F., Zhu, S., Yang, M., Cour, T., Yu, K., ... & Huang, T. (2010). Imagenet classification: fast descriptor coding and large-scale svm training. Large scale visual recognition challenge.
- [16] Hu, J., Shen, L., & Sun, G. (2017). Squeeze-and-excitation networks. arXiv preprint arXiv:1709.01507, 7.
- [17] Campbell, M., Hoane Jr, A. J., & Hsu, F. H. (2002). Deep blue. *Artificial intelligence*, 134(1-2), 57-83.
- [18] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Dieleman, S. (2016). Mastering the game of Go with deep neural networks and

tree search. *nature*, 529(7587), 484.

[19] Brown, N., & Sandholm, T. (2017, August). Libratus: The superhuman ai for no-limit poker. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence.

[20] Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6), 679-698.

[21] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2), 91-110.

[22] Salton, G., & McGill, M. J. (1986). Introduction to modern information retrieval.

2.1 环境配置

本节简单介绍一些必要的软件的安装与配置，由于不同机器软硬件配置不同，所以不详述，遇到问题请擅用 Google。## 2.1.1 Anaconda Anaconda 是 Python 的一个开源发行版本，主要面向科学计算。我们可以简单理解为，Anaconda 是一个预装了很多我们用的到或用不到的第三方库的 Python。而且相比于大家熟悉的 pip install 命令，Anaconda 中增加了 conda install 命令。当你熟悉了 Anaconda 以后会发现，conda install 会比 pip install 更方便一些。强烈建议先去看看[最省心的 Python 版本和第三方库管理——初探 Anaconda 和初学 Python 者自学 Anaconda 的正确姿势-猴子的回答](#)。

总的来说，我们应该完成以下几步：* 根据操作系统下载并安装 Anaconda（或者 mini 版本 Miniconda）并学会常用的几个 conda 命令，例如如何管理 python 环境、如何安装卸载包等；* Anaconda 安装成功之后，我们需要修改其包管理镜像为国内源，这样以后安装包时就会快一些。

2.1.2 Jupyter

在没有 notebook 之前，在 IT 领域是这样工作的：在普通的 Python shell 或者在 IDE（集成开发环境）如 Pycharm 中写代码，然后在 word 中写文档来说明你的项目。这个过程很反锁，通常是写完代码，再写文档的时候我还的重头回顾一遍代码。最蛋疼的地方在于，有些数据分析的中间结果，还得重新跑代码，然后把结果弄到文档里给客户看。有了 notebook 之后，世界突然美好了许多，因为 notebook 可以直接在代码旁写出叙述性文档，而不是另外编写单独的文档。也就是它可以能将代码、文档等这一切集中到一处，让用户一目了然。如下图所示。

Jupyter Notebook 已迅速成为数据分析，机器学习的必备工具。因为它可以让数据分析师集中精力向用户解释整个分析过程。

我们参考[jupyter notebook-猴子的回答](#)进行 jupyter notebook 及常用包（例如环境自动关联包 nb_conda）的安装。

安装好后，我们使用以下命令打开一个 jupyter notebook：

```
jupyter notebook
```

这时在浏览器打开 <http://localhost:8888> (通常会自动打开)位于当前目录的 jupyter 服务。

2.1.3 PyTorch

由于本文需要用到 PyTorch 框架，所以还需要安装 PyTorch（后期必不可少地会使用 GPU，所以安装 GPU 版本的）。直接去[PyTorch 官网](#)找到自己的软硬件对应的

安装命令即可（这里不得不吹一下PyTorch 的官方文档，从安装到入门，深入浅出，比 tensorflow 不知道高到哪里去了）。安装好后使用以下命令可查看安装的 PyTorch 及版本号。

```
conda list | grep torch
```

2.1.4 其他

此外还可以安装 python 最好用的 IDE PyCharm，专业版的应该是需要收费的，但学生用户可以申请免费使用（[传送门](#)），或者直接用免费的社区版。

如果不喜欢用 IDE 也可以选择编辑器，例如 VSCode 等。

本节与原文有很大不同，[原文传送门](#)

2.2 数据操作

在深度学习中，我们通常会频繁地对数据进行操作。作为动手学深度学习的基础，本节将介绍如何对内存中的数据进行操作。

在 PyTorch 中，`torch.Tensor` 是存储和变换数据的主要工具。如果你之前用过 NumPy，你会发现 `Tensor` 和 NumPy 的多维数组非常类似。然而，`Tensor` 提供 GPU 计算和自动求梯度等更多功能，这些使 `Tensor` 更加适合深度学习。> “tensor” 这个单词一般可译作“张量”，张量可以看作是一个多维数组。标量可以看作是 0 维张量，向量可以看作 1 维张量，矩阵可以看作是二维张量。

2.2.1 创建 Tensor

我们先介绍 `Tensor` 的最基本功能，即 `Tensor` 的创建。

首先导入 PyTorch：

```
import torch
```

然后我们创建一个 5x3 的未初始化的 `Tensor`：

```
x = torch.empty(5, 3)
print(x)
```

输出：

```
tensor([[ 0.0000e+00,  1.5846e+29,  0.0000e+00],
       [ 1.5846e+29,  5.6052e-45,  0.0000e+00],
       [ 0.0000e+00,  0.0000e+00,  0.0000e+00],
       [ 0.0000e+00,  0.0000e+00,  0.0000e+00],
       [ 0.0000e+00,  1.5846e+29, -2.4336e+02]])
```

创建一个 5x3 的随机初始化的 `Tensor`：

```
x = torch.rand(5, 3)
print(x)
```

输出：

```
tensor([[0.4963, 0.7682, 0.0885],
       [0.1320, 0.3074, 0.6341],
       [0.4901, 0.8964, 0.4556],
       [0.6323, 0.3489, 0.4017],
       [0.0223, 0.1689, 0.2939]])
```

创建一个 5x3 的 long 型全 0 的 Tensor:

```
x = torch.zeros(5, 3, dtype=torch.long)
print(x)
```

输出:

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

还可以直接根据数据创建:

```
x = torch.tensor([5.5, 3])
print(x)
```

输出:

```
tensor([5.5000, 3.0000])
```

还可以通过现有的 Tensor 来创建，此方法会默认重用输入 Tensor 的一些属性，例如数据类型，除非自定义数据类型。

```
x = x.new_ones(5, 3, dtype=torch.float64) # 返回的 tensor 默认具有相同的 torch.dtype
print(x)
```

```
x = torch.randn_like(x, dtype=torch.float) # 指定新的数据类型
print(x)
```

输出:

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
tensor([[ 0.6035,  0.8110, -0.0451],
```

```
[ 0.8797,  1.0482, -0.0445],
[-0.7229,  2.8663, -0.5655],
[ 0.1604, -0.0254,  1.0739],
[ 2.2628, -0.9175, -0.2251]])
```

我们可以通过 `shape` 或者 `size()` 来获取 `Tensor` 的形状:

```
print(x.size())
print(x.shape)
```

输出:

```
torch.Size([5, 3])
torch.Size([5, 3])
```

注意: 返回的 `torch.Size` 其实就是一个 `tuple`, 支持所有 `tuple` 的操作。

还有很多函数可以创建 `Tensor`, 去翻翻官方 API 就知道了, 下表给了一些常用的参考。

函数	功能
<code>Tensor(*sizes)</code>	基础构造函数
<code>tensor(data,)</code>	类似 <code>np.array</code> 的构造函数
<code>ones(*sizes)</code>	全 1Tensor
<code>zeros(*sizes)</code>	全 0Tensor
<code>eye(*sizes)</code>	对角线为 1, 其他为 0
<code>arange(s,e,step)</code>	从 s 到 e, 步长为 step
<code>linspace(s,e,steps)</code>	从 s 到 e, 均匀切分成 steps 份
<code>rand/randn(*sizes)</code>	均匀/标准分布
<code>normal(mean,std)/uniform(from,to)</code>	正态分布/均匀分布
<code>randperm(m)</code>	随机排列

这些创建方法都可以在创建的时候指定数据类型 `dtype` 和存放 `device(cpu/gpu)`。

2.2.2 操作

本小节介绍 `Tensor` 的各种操作。### 算术操作在 PyTorch 中, 同一种操作可能有很多种形式, 下面用加法作为例子。*** 加法形式一** `python y = torch.rand(5,`

3) `print(x + y)` * 加法形式二 python `print(torch.add(x, y))` 还可指定输出: python `result = torch.empty(5, 3)` `torch.add(x, y, out=result)` `print(result)` * 加法形式三、inplace python `# adds x to y` `y.add_(x)` `print(y)` > **注: PyTorch 操作 inplace 版本都有后缀“_”, 例如 `x.copy_(y)`, `x.t_()`**

以上几种形式的输出均为:

```
tensor([[ 1.3967,  1.0892,  0.4369],
        [ 1.6995,  2.0453,  0.6539],
        [-0.1553,  3.7016, -0.3599],
        [ 0.7536,  0.0870,  1.2274],
        [ 2.5046, -0.1913,  0.4760]])
```

索引

我们还可以使用类似 NumPy 的索引操作来访问 Tensor 的一部分, 需要注意的是: 索引出来的结果与原数据共享内存, 也即修改一个, 另一个会跟着修改。

```
y = x[0, :]
y += 1
print(y)
print(x[0, :]) # 源 tensor 也被改了
```

输出:

```
tensor([1.6035, 1.8110, 0.9549])
tensor([1.6035, 1.8110, 0.9549])
```

除了常用的索引选择数据之外, PyTorch 还提供了一些高级的选择函数:

函数	功能
<code>index_select(input, dim, index)</code>	在指定维度 dim 上选取, 比如选取某些行、某些列
<code>masked_select(input, mask)</code>	例子如上, <code>a[a>0]</code> , 使用 ByteTensor 进行选取
<code>non_zero(input)</code>	非 0 元素的下标
<code>gather(input, dim, index)</code>	根据 index, 在 dim 维度上选取数据, 输出的 size 与 index 一样

这里不详细介绍, 用到了再查官方文档。### 改变形状用 `view()` 来改变 Tensor

的形状:

```
y = x.view(15)
z = x.view(-1, 5) # -1 所指的维度可以根据其他维度的值推出来
print(x.size(), y.size(), z.size())
```

输出:

```
torch.Size([5, 3]) torch.Size([15]) torch.Size([3, 5])
```

注意 `view()` 返回的新 `tensor` 与源 `tensor` 共享内存（其实是同一个 `tensor`），也即更改其中的一个，另外一个也会跟着改变。（顾名思义，`view` 仅仅是改变了对这个张量的观察角度）

```
x += 1
print(x)
print(y) # 也加了 1
```

输出:

```
tensor([[1.6035, 1.8110, 0.9549],
        [1.8797, 2.0482, 0.9555],
        [0.2771, 3.8663, 0.4345],
        [1.1604, 0.9746, 2.0739],
        [3.2628, 0.0825, 0.7749]])
tensor([1.6035, 1.8110, 0.9549, 1.8797, 2.0482, 0.9555, 0.2771, 3.8663, 0.4345,
       1.1604, 0.9746, 2.0739, 3.2628, 0.0825, 0.7749])
```

所以如果我们想返回一个真正新的副本（即不共享内存）该怎么办呢？Pytorch 还提供了一个 `reshape()` 可以改变形状，但是此函数并不能保证返回的是其拷贝，所以不推荐使用。推荐先用 `clone` 创造一个副本然后再使用 `view`。[参考此处](#)

```
x_cp = x.clone().view(15)
x -= 1
print(x)
print(x_cp)
```

输出:

```
tensor([[ 0.6035,  0.8110, -0.0451],
       [ 0.8797,  1.0482, -0.0445],
       [-0.7229,  2.8663, -0.5655],
       [ 0.1604, -0.0254,  1.0739],
       [ 2.2628, -0.9175, -0.2251]])
tensor([1.6035, 1.8110, 0.9549, 1.8797, 2.0482, 0.9555, 0.2771, 3.8663, 0.4345,
       1.1604, 0.9746, 2.0739, 3.2628, 0.0825, 0.7749])
```

使用 `clone` 还有一个好处是会被记录在计算图中，即梯度回传到副本时也会传到源 `Tensor`。

另外一个常用的函数就是 `item()`，它可以将一个标量 `Tensor` 转换成一个 Python number：

```
x = torch.randn(1)
print(x)
print(x.item())
```

输出：

```
tensor([2.3466])
2.3466382026672363
```

线性代数

另外，PyTorch 还支持一些线性函数，这里提一下，免得用起来的时候自己造轮子，具体用法参考官方文档。如下表所示：

函数	功能
trace	对角线元素之和 (矩阵的迹)
diag	对角线元素
triu/tril	矩阵的上三角/下三角，可指定偏移量
mm/bmm	矩阵乘法，batch 的矩阵乘法
addmm/addbmm/addmv/addr/badbmm..	矩阵运算
t	转置
dot/cross	内积/外积
inverse	求逆矩阵
svd	奇异值分解

PyTorch 中的 `Tensor` 支持超过一百种操作，包括转置、索引、切片、数学运算、线性代数、随机数等等，可参考[官方文档](#)。

2.2.3 广播机制

前面我们看到如何对两个形状相同的 `Tensor` 做按元素运算。当对两个形状不同的 `Tensor` 按元素运算时，可能会触发广播（broadcasting）机制：先适当复制元素使这两个 `Tensor` 形状相同后再按元素运算。例如：

```
x = torch.arange(1, 3).view(1, 2)
print(x)
y = torch.arange(1, 4).view(3, 1)
print(y)
print(x + y)
```

输出：

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

由于 `x` 和 `y` 分别是 1 行 2 列和 3 行 1 列的矩阵，如果要计算 `x + y`，那么 `x` 中第一行的 2 个元素被广播（复制）到了第二行和第三行，而 `y` 中第一列的 3 个元素被广播（复制）到了第二列。如此，就可以对 2 个 3 行 2 列的矩阵按元素相加。

2.2.4 运算的内存开销

前面说了，索引、`view` 是不会开辟新内存的，而像 `y = x + y` 这样的运算是会新开内存的，然后将 `y` 指向新内存。为了演示这一点，我们可以使用 Python 自带的 `id` 函数：如果两个实例的 ID 一致，那么它们所对应的内存地址相同；反之则不同。

```
x = torch.tensor([1, 2])
y = torch.tensor([3, 4])
id_before = id(y)
```

```
y = y + x
print(id(y) == id_before) # False
```

如果想指定结果到原来的 `y` 的内存，我们可以使用前面介绍的索引来进行替换操作。在下面的例子中，我们把 `x + y` 的结果通过 `[:]` 写进 `y` 对应的内存中。

```
x = torch.tensor([1, 2])
y = torch.tensor([3, 4])
id_before = id(y)
y[:] = y + x
print(id(y) == id_before) # True
```

我们还可以使用运算符全名函数中的 `out` 参数或者自加运算符 `+=(`也即 `add_()`) 达到上述效果，例如 `torch.add(x, y, out=y)` 和 `y += x(y.add_(x))`。

```
x = torch.tensor([1, 2])
y = torch.tensor([3, 4])
id_before = id(y)
torch.add(x, y, out=y) # y += x, y.add_(x)
print(id(y) == id_before) # True
```

2.2.5 Tensor 和 NumPy 相互转换

我们很容易用 `numpy()` 和 `from_numpy()` 将 `Tensor` 和 `NumPy` 中的数组相互转换。但是需要注意的一点是：这两个函数所产生的的 `Tensor` 和 `NumPy` 中的数组共享相同的内存（所以他们之间的转换很快），改变其中一个时另一个也会改变!!! > 还有一个常用的将 `NumPy` 中的 `array` 转换成 `Tensor` 的方法就是 `torch.tensor()`，需要注意的是，此方法总是会进行数据拷贝（就会消耗更多的时间和空间），所以返回的 `Tensor` 和原来的数据不再共享内存。

Tensor 转 NumPy

使用 `numpy()` 将 `Tensor` 转换成 `NumPy` 数组：

```
a = torch.ones(5)
b = a.numpy()
print(a, b)
```

```
a += 1  
print(a, b)  
b += 1  
print(a, b)
```

输出：

```
tensor([1., 1., 1., 1., 1.]) [1. 1. 1. 1. 1.]  
tensor([2., 2., 2., 2., 2.]) [2. 2. 2. 2. 2.]  
tensor([3., 3., 3., 3., 3.]) [3. 3. 3. 3. 3.]
```

NumPy 数组转 Tensor

使用 `from_numpy()` 将 NumPy 数组转换成 Tensor:

```
import numpy as np  
a = np.ones(5)  
b = torch.from_numpy(a)  
print(a, b)  
  
a += 1  
print(a, b)  
b += 1  
print(a, b)
```

输出：

```
[1. 1. 1. 1. 1.] tensor([1., 1., 1., 1., 1.], dtype=torch.float64)  
[2. 2. 2. 2. 2.] tensor([2., 2., 2., 2., 2.], dtype=torch.float64)  
[3. 3. 3. 3. 3.] tensor([3., 3., 3., 3., 3.], dtype=torch.float64)
```

所有在 CPU 上的 Tensor（除了 CharTensor）都支持与 NumPy 数组相互转换。

此外上面提到还有一个常用的方法就是直接用 `torch.tensor()` 将 NumPy 数组转换成 Tensor，需要注意的是该方法总是会进行数据拷贝，返回的 Tensor 和原来的数据不再共享内存。

```
c = torch.tensor(a)  
a += 1  
print(a, c)
```

输出

```
[4. 4. 4. 4. 4.] tensor([3., 3., 3., 3., 3.], dtype=torch.float64)
```

2.2.6 Tensor on GPU

用方法 `to()` 可以将 `Tensor` 在 CPU 和 GPU（需要硬件支持）之间相互移动。

```
# 以下代码只有在 PyTorch GPU 版本上才会执行
if torch.cuda.is_available():
    device = torch.device("cuda")           # GPU
    y = torch.ones_like(x, device=device)   # 直接创建一个在 GPU 上的 Tensor
    x = x.to(device)                      # 等价于 .to("cuda")
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))       # to() 还可以同时更改数据类型
```

注: 本文主要参考[PyTorch 官方文档](#)和[此处](#), 与[原书同一节](#)有很大不同。

2.3 自动求梯度

在深度学习中，我们经常需要对函数求梯度（gradient）。PyTorch 提供的 `autograd` 包能够根据输入和前向传播过程自动构建计算图，并执行反向传播。本节将介绍如何使用 `autograd` 包来进行自动求梯度的有关操作。

2.3.1 概念

上一节介绍的 `Tensor` 是这个包的核心类，如果将其属性 `.requires_grad` 设置为 `True`，它将开始追踪（track）在其上的所有操作（这样就可以利用链式法则进行梯度传播了）。完成计算后，可以调用 `.backward()` 来完成所有梯度计算。此 `Tensor` 的梯度将累积到 `.grad` 属性中。> 注意在 `y.backward()` 时，如果 `y` 是标量，则不需要为 `backward()` 传入任何参数；否则，需要传入一个与 `y` 同形的 `Tensor`。解释见 2.3.2 节。

如果不想要被继续追踪，可以调用 `.detach()` 将其从追踪记录中分离出来，这样就可以防止将来的计算被追踪，这样梯度就传不过去了。此外，还可以用 `with torch.no_grad()` 将不想被追踪的操作代码块包裹起来，这种方法在评估模型的时候很常用，因为在评估模型时，我们并不需要计算可训练参数（`requires_grad=True`）的梯度。

`Function` 是另外一个很重要的类。`Tensor` 和 `Function` 互相结合就可以构建一个记录有整个计算过程的有向无环图（DAG）。每个 `Tensor` 都有一个 `.grad_fn` 属性，该属性即创建该 `Tensor` 的 `Function`，就是说该 `Tensor` 是不是通过某些运算得到的，若是，则 `grad_fn` 返回一个与这些运算相关的对象，否则是 `None`。

下面通过一些例子来理解这些概念。

2.3.2 Tensor

创建一个 `Tensor` 并设置 `requires_grad=True`:

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
print(x.grad_fn)
```

输出:

```
tensor([[1., 1.],
       [1., 1.]], requires_grad=True)
None
```

再做一下运算操作:

```
y = x + 2
print(y)
print(y.grad_fn)
```

输出:

```
tensor([[3., 3.],
       [3., 3.]], grad_fn=<AddBackward>)
<AddBackward object at 0x1100477b8>
```

注意 x 是直接创建的，所以它没有 grad_fn，而 y 是通过一个加法操作创建的，所以它有一个为 <AddBackward> 的 grad_fn。

像 x 这种直接创建的称为叶子节点，叶子节点对应的 grad_fn 是 None。

```
print(x.is_leaf, y.is_leaf) # True False
```

再来点复杂度运算操作:

```
z = y * y * 3
out = z.mean()
print(z, out)
```

输出:

```
tensor([[27., 27.],
       [27., 27.]], grad_fn=<MulBackward>) tensor(27., grad_fn=<MeanBackward1>)
```

通过 .requires_grad_() 来用 in-place 的方式改变 requires_grad 属性:

```
a = torch.randn(2, 2) # 缺失情况下默认 requires_grad = False
a = ((a * 3) / (a - 1))
print(a.requires_grad) # False
a.requires_grad_(True)
print(a.requires_grad) # True
b = (a * a).sum()
print(b.grad_fn)
```

输出:

```
False
True
<SumBackward0 object at 0x118f50cc0>
```

2.3.2 梯度

因为 `out` 是一个标量，所以调用 `backward()` 时不需要指定求导变量：

```
out.backward() # 等价于 out.backward(torch.tensor(1.))
```

我们来看看 `out` 关于 `x` 的梯度 $\frac{d(out)}{dx}$:

```
print(x.grad)
```

输出：

```
tensor([[4.5000, 4.5000],
       [4.5000, 4.5000]])
```

我们令 `out` 为 o ，因为

$$o = \frac{1}{4} \sum_{i=1}^4 z_i = \frac{1}{4} \sum_{i=1}^4 3(x_i + 2)^2$$

所以

$$\left. \frac{\partial o}{\partial x_i} \right|_{x_i=1} = \frac{9}{2} = 4.5$$

所以上面的输出是正确的。

数学上，如果有一个函数值和自变量都为向量的函数 $\vec{y} = f(\vec{x})$ ，那么 \vec{y} 关于 \vec{x} 的梯度就是一个雅可比矩阵（Jacobian matrix）：

$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

而 `torch.autograd` 这个包就是用来计算一些雅克比矩阵的乘积的。例如，如果 v 是一个标量函数的 $l = g(\vec{y})$ 的梯度：

$$v = \left(\frac{\partial l}{\partial y_1} \quad \cdots \quad \frac{\partial l}{\partial y_m} \right)$$

那么根据链式法则我们有 l 关于 \vec{x} 的雅克比矩阵就为：

$$vJ = \left(\frac{\partial l}{\partial y_1} \quad \cdots \quad \frac{\partial l}{\partial y_m} \right) \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} = \left(\frac{\partial l}{\partial x_1} \quad \cdots \quad \frac{\partial l}{\partial x_n} \right)$$

注意：grad 在反向传播过程中是累加的 (accumulated)，这意味着每一次运行反向传播，梯度都会累加之前的梯度，所以一般在反向传播之前需把梯度清零。

```
# 再来反向传播一次，注意 grad 是累加的
out2 = x.sum()
out2.backward()
print(x.grad)

out3 = x.sum()
x.grad.data.zero_()
out3.backward()
print(x.grad)
```

输出：

```
tensor([[5.5000, 5.5000],
        [5.5000, 5.5000]])
tensor([[1., 1.],
        [1., 1.]])
```

现在我们解释 2.3.1 节留下的问题，为什么在 `y.backward()` 时，如果 `y` 是标量，则不需要为 `backward()` 传入任何参数；否则，需要传入一个与 `y` 同形的 `Tensor`？简单来说就是为了避免向量（甚至更高维张量）对张量求导，而转换成标量对张量求导。举个例子，假设形状为 `m x n` 的矩阵 `X` 经过运算得到了 `p x q` 的矩阵 `Y`，`Y` 又经过运算得到了 `s x t` 的矩阵 `Z`。那么按照前面讲的规则， dZ/dY 应该是一个 `s x t x p x q` 四维张量， dY/dX 是一个 `p x q x m x n` 的四维张量。问题来了，怎样反向传播？怎样将两个四维张量相乘??? 这要怎么乘??? 就算能解决两个四维张量怎么乘的问题，四维和三维的张量又怎么乘？导数的导数又怎么求，这一连串的问题，感觉要疯掉..... 为了避免这个问题，我们不允许张量对张量求导，只允许标量对张量求导，求导结果是和自变量同形的张量。所以必要时我们要把张量通过将所有张量的元素加权求和的方式转换为标量，举个例子，假设 `y` 由自变量 `x` 计算而来，`w` 是和 `y` 同形的张量，则 `y.backward(w)` 的含义是：先计算 `l = torch.sum(y * w)`，则 `l` 是个标量，然后求 `l` 对自变量 `x` 的导数。[参考](#)

来看一些实际例子。

```
x = torch.tensor([1.0, 2.0, 3.0, 4.0], requires_grad=True)
y = 2 * x
```

```
z = y.view(2, 2)
print(z)
```

输出：

```
tensor([[2., 4.],
       [6., 8.]], grad_fn=<ViewBackward>)
```

现在 `y` 不是一个标量，所以在调用 `backward` 时需要传入一个和 `y` 同形的权重向量进行加权求和得到一个标量。

```
v = torch.tensor([[1.0, 0.1], [0.01, 0.001]], dtype=torch.float)
z.backward(v)
print(x.grad)
```

输出：

```
tensor([2.0000, 0.2000, 0.0200, 0.0020])
```

注意，`x.grad` 是和 `x` 同形的张量。

再来看看中断梯度追踪的例子：

```
x = torch.tensor(1.0, requires_grad=True)
y1 = x ** 2
with torch.no_grad():
    y2 = x ** 3
y3 = y1 + y2

print(x.requires_grad)
print(y1, y1.requires_grad) # True
print(y2, y2.requires_grad) # False
print(y3, y3.requires_grad) # True
```

输出：

```
True
tensor(1., grad_fn=<PowBackward0>) True
tensor(1.) False
tensor(2., grad_fn=<ThAddBackward>) True
```

可以看到，上面的 `y2` 是没有 `grad_fn` 而且 `y2.requires_grad=False` 的，而 `y3` 是有 `grad_fn` 的。如果我们将 `y3` 对 `x` 求梯度的话会是多少呢？

```
y3.backward()  
print(x.grad)
```

输出：

```
tensor(2.)
```

为什么是 2 呢？ $y_3 = y_1 + y_2 = x^2 + x^3$ ，当 $x = 1$ 时 $\frac{dy_3}{dx}$ 不应该是 5 吗？事实上，由于 `y2` 的定义是被 `torch.no_grad()` 包裹的，所以与 `y2` 有关的梯度是不会回传的，只有与 `y1` 有关的梯度才会回传，即 x^2 对 x 的梯度。

上面提到，`y2.requires_grad=False`，所以不能调用 `y2.backward()`，会报错：

```
RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn
```

此外，如果我们想要修改 `tensor` 的数值，但是又不希望被 `autograd` 记录（即不会影响反向传播），那么我么可以对 `tensor.data` 进行操作。

```
x = torch.ones(1, requires_grad=True)  
  
print(x.data) # 还是一个 tensor  
print(x.data.requires_grad) # 但是已经是独立于计算图之外  
  
y = 2 * x  
x.data *= 100 # 只改变了值，不会记录在计算图，所以不会影响梯度传播  
  
y.backward()  
print(x) # 更改 data 的值也会影响 tensor 的值  
print(x.grad)
```

输出：

```
tensor([1.])  
False  
tensor([100.], requires_grad=True)  
tensor([2.])
```

注：本文主要参考[PyTorch 官方文档](#)，与[原书同一节](#)有很大不同。

3.1 线性回归

线性回归输出是一个连续值，因此适用于回归问题。回归问题在实际中很常见，如预测房屋价格、气温、销售额等连续值的问题。与回归问题不同，分类问题中模型的最终输出是一个离散值。我们所说的图像分类、垃圾邮件识别、疾病检测等输出为离散值的问题都属于分类问题的范畴。softmax 回归则适用于分类问题。

由于线性回归和 softmax 回归都是单层神经网络，它们涉及的概念和技术同样适用于大多数的深度学习模型。我们首先以线性回归为例，介绍大多数深度学习模型的基本要素和表示方法。

3.1.1 线性回归的基本要素

我们以一个简单的房屋价格预测作为例子来解释线性回归的基本要素。这个应用的目标是预测一栋房子的售出价格（元）。我们知道这个价格取决于很多因素，如房屋状况、地段、市场行情等。为了简单起见，这里我们假设价格只取决于房屋状况的两个因素，即面积（平方米）和房龄（年）。接下来我们希望探索价格与这两个因素的具体关系。

3.1.1.1 模型定义

设房屋的面积为 x_1 ，房龄为 x_2 ，售出价格为 y 。我们需要建立基于输入 x_1 和 x_2 来计算输出 y 的表达式，也就是模型（model）。顾名思义，线性回归假设输出与各个输入之间是线性关系：

$$\hat{y} = x_1 w_1 + x_2 w_2 + b$$

其中 w_1 和 w_2 是权重（weight）， b 是偏差（bias），且均为标量。它们是线性回归模型的参数（parameter）。模型输出 \hat{y} 是线性回归对真实价格 y 的预测或估计。我们通常允许它们之间有一定误差。

3.1.1.2 模型训练

接下来我们需要通过数据来寻找特定的模型参数值，使模型在数据上的误差尽可能小。这个过程叫作模型训练（model training）。下面我们介绍模型训练所涉及的 3 个要素。

(1) 训练数据

我们通常收集一系列的真实数据，例如多栋房屋的真实售出价格和它们对应的面积和房龄。我们希望在这个数据上面寻找模型参数来使模型的预测价格与真实价格的误

差最小。在机器学习术语里，该数据集被称为训练数据集（training data set）或训练集（training set），一栋房屋被称为一个样本（sample），其真实售出价格叫作标签（label），用来预测标签的两个因素叫作特征（feature）。特征用来表征样本的特点。

假设我们采集的样本数为 n ，索引为 i 的样本的特征为 $x_1^{(i)}$ 和 $x_2^{(i)}$ ，标签为 $y^{(i)}$ 。对于索引为 i 的房屋，线性回归模型的房屋价格预测表达式为

$$\hat{y}^{(i)} = x_1^{(i)}w_1 + x_2^{(i)}w_2 + b$$

(2) 损失函数

在模型训练中，我们需要衡量价格预测值与真实值之间的误差。通常我们会选取一个非负数作为误差，且数值越小表示误差越小。一个常用的选择是平方函数。它在评估索引为 i 的样本误差的表达式为

$$\ell^{(i)}(w_1, w_2, b) = \frac{1}{2} (\hat{y}^{(i)} - y^{(i)})^2$$

其中常数 $\frac{1}{2}$ 使对平方项求导后的常数系数为 1，这样在形式上稍微简单一些。显然，误差越小表示预测价格与真实价格越相近，且当二者相等时误差为 0。给定训练数据集，这个误差只与模型参数相关，因此我们将它记为以模型参数为参数的函数。在机器学习里，将衡量误差的函数称为损失函数（loss function）。这里使用的平方误差函数也称为平方损失（square loss）。

通常，我们用训练数据集中所有样本误差的平均来衡量模型预测的质量，即

$$\ell(w_1, w_2, b) = \frac{1}{n} \sum_{i=1}^n \ell^{(i)}(w_1, w_2, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} (x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)})^2$$

在模型训练中，我们希望找出一组模型参数，记为 w_1^*, w_2^*, b^* ，来使训练样本平均损失最小：

$$w_1^*, w_2^*, b^* = \arg \min_{w_1, w_2, b} \ell(w_1, w_2, b)$$

(3) 优化算法

当模型和损失函数形式较为简单时，上面的误差最小化问题的解可以直接用公式表达出来。这类解叫作解析解（analytical solution）。本节使用的线性回归和平方误差刚好属于这个范畴。然而，大多数深度学习模型并没有解析解，只能通过优化算法有限次迭代模型参数来尽可能降低损失函数的值。这类解叫作数值解（numerical solution）。

在求数值解的优化算法中，小批量随机梯度下降（mini-batch stochastic gradient descent）在深度学习中被广泛使用。它的算法很简单：先选取一组模型参数的初始值，

如随机选取；接下来对参数进行多次迭代，使每次迭代都可能降低损失函数的值。在每次迭代中，先随机均匀采样一个由固定数目训练数据样本所组成的小批量（mini-batch） \mathcal{B} ，然后求小批量中数据样本的平均损失有关模型参数的导数（梯度），最后用此结果与预先设定的一个正数的乘积作为模型参数在本次迭代的减小量。

在训练本节讨论的线性回归模型的过程中，模型的每个参数将作如下迭代：

$$\begin{aligned} w_1 &\leftarrow w_1 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial w_1} \\ w_2 &\leftarrow w_2 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial w_2} \\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial b} \end{aligned}$$

在上式中， $|\mathcal{B}|$ 代表每个小批量中的样本个数（批量大小，batch size）， η 称作学习率（learning rate）并取正数。需要强调的是，这里的批量大小和学习率的值是人为设定的，并不是通过模型训练学出的，因此叫作超参数（hyperparameter）。我们通常所说的“调参”指的正是调节超参数，例如通过反复试错来找到超参数合适的值。在少数情况下，超参数也可以通过模型训练学出。本书对此类情况不做讨论。

3.1.1.3 模型预测

模型训练完成后，我们将模型参数 w_1, w_2, b 在优化算法停止时的值分别记作 $\hat{w}_1, \hat{w}_2, \hat{b}$ 。注意，这里我们得到的并不一定是最小化损失函数的最优解 w_1^*, w_2^*, b^* ，而是对最优解的一个近似。然后，我们就可以使用学出的线性回归模型 $x_1\hat{w}_1 + x_2\hat{w}_2 + \hat{b}$ 来估算训练数据集以外任意一栋面积（平方米）为 x_1 、房龄（年）为 x_2 的房屋的价格了。这里的估算也叫作模型预测、模型推断或模型测试。

3.1.2 线性回归的表示方法

我们已经阐述了线性回归的模型表达式、训练和预测。下面我们解释线性回归与神经网络的联系，以及线性回归的矢量计算表达式。

3.1.2.1 神经网络图

在深度学习中，我们可以使用神经网络图直观地表现模型结构。为了更清晰地展示线性回归作为神经网络的结构，图 3.1 使用神经网络图表示本节中介绍的线性回归模型。神经网络图隐去了模型参数权重和偏差。

图 3.1 线性回归是一个单层神经网络

在图 3.1 所示的神经网络中，输入分别为 x_1 和 x_2 ，因此输入层的输入个数为 2。输入个数也叫特征数或特征向量维度。图 3.1 中网络的输出为 o ，输出层的输出个数为 1。需要注意的是，我们直接将图 3.1 中神经网络的输出 o 作为线性回归的输出，即 $\hat{y} = o$ 。由于输入层并不涉及计算，按照惯例，图 3.1 所示的神经网络的层数为 1。所以，线性回归是一个单层神经网络。输出层中负责计算 o 的单元又叫神经元。在线性回归中， o 的计算依赖于 x_1 和 x_2 。也就是说，输出层中的神经元和输入层中各个输入完全连接。因此，这里的输出层又叫全连接层（fully-connected layer）或稠密层（dense layer）。

3.1.2.2 矢量计算表达式

在模型训练或预测时，我们常常会同时处理多个数据样本并用到矢量计算。在介绍线性回归的矢量计算表达式之前，让我们先考虑对两个向量相加的两种方法。a

下面先定义两个 1000 维的向量。

```
import torch
from time import time

a = torch.ones(1000)
b = torch.ones(1000)
```

向量相加的一种方法是，将这两个向量按元素逐一做标量加法。

```
start = time()
c = torch.zeros(1000)
for i in range(1000):
    c[i] = a[i] + b[i]
print(time() - start)
```

输出：

0.02039504051208496

向量相加的另一种方法是，将这两个向量直接做矢量加法。

```
start = time()
d = a + b
print(time() - start)
```

输出：

0.0008330345153808594

结果很明显，后者比前者更省时。因此，我们应该尽可能采用矢量计算，以提升计算效率。

让我们再次回到本节的房价预测问题。如果我们对训练数据集里的 3 个房屋样本（索引分别为 1、2 和 3）逐一预测价格，将得到

$$\hat{y}^{(1)} = x_1^{(1)}w_1 + x_2^{(1)}w_2 + b,$$

$$\hat{y}^{(2)} = x_1^{(2)}w_1 + x_2^{(2)}w_2 + b,$$

$$\hat{y}^{(3)} = x_1^{(3)}w_1 + x_2^{(3)}w_2 + b.$$

现在，我们将上面 3 个等式转化成矢量计算。设

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \hat{y}^{(3)} \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ x_1^{(3)} & x_2^{(3)} \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

对 3 个房屋样本预测价格的矢量计算表达式为 $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$ ，其中的加法运算使用了广播机制（参见 2.2 节）。例如：

```
a = torch.ones(3)
```

```
b = 10
```

```
print(a + b)
```

输出：

```
tensor([11., 11., 11.])
```

广义上讲，当数据样本数为 n ，特征数为 d 时，线性回归的矢量计算表达式为

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$$

其中模型输出 $\hat{\mathbf{y}} \in \mathbb{R}^{n \times 1}$ 批量数据样本特征 $\mathbf{X} \in \mathbb{R}^{n \times d}$ ，权重 $\mathbf{w} \in \mathbb{R}^{d \times 1}$ ，偏差 $b \in \mathbb{R}$ 。相应地，批量数据样本标签 $\mathbf{y} \in \mathbb{R}^{n \times 1}$ 。设模型参数 $\boldsymbol{\theta} = [w_1, w_2, b]^\top$ ，我们可以重写损失函数为

$$\ell(\boldsymbol{\theta}) = \frac{1}{2n}(\hat{\mathbf{y}} - \mathbf{y})^\top(\hat{\mathbf{y}} - \mathbf{y})$$

小批量随机梯度下降的迭代步骤将相应地改写为

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\boldsymbol{\theta}} \ell^{(i)}(\boldsymbol{\theta}),$$

其中梯度是损失有关 3 个为标量的模型参数的偏导数组成的向量：

$$\nabla_{\boldsymbol{\theta}} \ell^{(i)}(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial w_1} \\ \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial w_2} \\ \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial b} \end{bmatrix} = \begin{bmatrix} x_1^{(i)}(x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)}) \\ x_2^{(i)}(x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)}) \\ x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)} \end{bmatrix} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ 1 \end{bmatrix} (\hat{y}^{(i)} - y^{(i)})$$

小结

- 和大多数深度学习模型一样，对于线性回归这样一种单层神经网络，它的基本要素包括模型、训练数据、损失函数和优化算法。
 - 既可以用神经网络图表示线性回归，又可以用矢量计算表示该模型。
 - 应该尽可能采用矢量计算，以提升计算效率。
-

注：本节除了代码之外与原书基本相同，[原书传送门](#)

3.2 线性回归的从零开始实现

在了解了线性回归的背景知识之后，现在我们可以动手实现它了。尽管强大的深度学习框架可以减少大量重复性工作，但若过于依赖它提供的便利，会导致我们很难深入理解深度学习是如何工作的。因此，本节将介绍如何只利用 `Tensor` 和 `autograd` 来实现一个线性回归的训练。

首先，导入本节中实验所需的包或模块，其中的 `matplotlib` 包可用于作图，且设置成嵌入显示。

```
%matplotlib inline
import torch
from IPython import display
from matplotlib import pyplot as plt
import numpy as np
import random
```

3.2.1 生成数据集

我们构造一个简单的人工训练数据集，它可以使我们能够直观比较学到的参数和真实的模型参数的区别。设训练数据集样本数为 1000，输入个数（特征数）为 2。给定随机生成的批量样本特征 $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$ ，我们使用线性回归模型真实权重 $\mathbf{w} = [2, -3.4]^\top$ 和偏差 $b = 4.2$ ，以及一个随机噪声项 ϵ 来生成标签

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon$$

其中噪声项 ϵ 服从均值为 0、标准差为 0.01 的正态分布。噪声代表了数据集中无意义的干扰。下面，让我们生成数据集。

```
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
features = torch.from_numpy(np.random.normal(0, 1, (num_examples, num_inputs)))
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += torch.from_numpy(np.random.normal(0, 0.01, size=labels.size()))
```

注意，`features` 的每一行是一个长度为 2 的向量，而 `labels` 的每一行是一个长度为 1 的向量（标量）。

```
print(features[0], labels[0])
```

输出：

```
tensor([0.8557, 0.4793]) tensor(4.2887)
```

通过生成第二个特征 `features[:, 1]` 和标签 `labels` 的散点图，可以更直观地观察两者间的线性关系。

```
def use_svg_display():
    # 用矢量图显示
    display.set_matplotlib_formats('svg')

def set_figsize(figsize=(3.5, 2.5)):
    use_svg_display()
    # 设置图的尺寸
    plt.rcParams['figure.figsize'] = figsize

# # 在../d2lzh_pytorch 里面添加上面两个函数后就可以这样导入
# import sys
# sys.path.append("..")
# from d2lzh_pytorch import *

set_figsize()
plt.scatter(features[:, 1].numpy(), labels.numpy(), 1);
```

我们将上面的 `plt` 作图函数以及 `use_svg_display` 函数和 `set_figsize` 函数定义在 `d2lzh_pytorch` 包里。以后在作图时，我们将直接调用 `d2lzh_pytorch.plt`。由于 `plt` 在 `d2lzh_pytorch` 包中是一个全局变量，我们在作图前只需要调用 `d2lzh_pytorch.set_figsize()` 即可打印矢量图并设置图的尺寸。> 原书中提到的 `d2lzh` 里面使用了 `mxnet`，改成 `pytorch` 实现后本项目统一将原书的 `d2lzh` 改为 `d2lzh_pytorch`。

3.2.2 读取数据

在训练模型的时候，我们需要遍历数据集并不断读取小批量数据样本。这里我们定义一个函数：它每次返回 `batch_size`（批量大小）个随机样本的特征和标签。

```
# 本函数已保存在 d2lzh 包中方便以后使用
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    random.shuffle(indices) # 样本的读取顺序是随机的
    for i in range(0, num_examples, batch_size):
        j = torch.LongTensor(indices[i: min(i + batch_size, num_examples)]) # 最后一次
        yield features.index_select(0, j), labels.index_select(0, j)
```

让我们读取第一个小批量数据样本并打印。每个批量的特征形状为 (10, 2)，分别对应批量大小和输入个数；标签形状为批量大小。

```
batch_size = 10
```

```
for X, y in data_iter(batch_size, features, labels):
    print(X, y)
    break
```

输出：

```
tensor([[-1.4239, -1.3788],
       [ 0.0275,  1.3550],
       [ 0.7616, -1.1384],
       [ 0.2967, -0.1162],
       [ 0.0822,  2.0826],
       [-0.6343, -0.7222],
       [ 0.4282,  0.0235],
       [ 1.4056,  0.3506],
       [-0.6496, -0.5202],
       [-0.3969, -0.9951]])
tensor([ 6.0394, -0.3365,  9.5882,  5.1810, -2.7355,  5.3873,  4.9827,  5.7962,
       4.6727,  6.7921])
```

3.2.3 初始化模型参数

我们将权重初始化成均值为 0、标准差为 0.01 的正态随机数，偏差则初始化成 0。

```
w = torch.tensor(np.random.normal(0, 0.01, (num_inputs, 1)), dtype=torch.float32)
b = torch.zeros(1, dtype=torch.float32)
```

之后的模型训练中，需要对这些参数求梯度来迭代参数的值，因此我们要让它们的 `requires_grad=True`。

```
w.requires_grad_(requires_grad=True)  
b.requires_grad_(requires_grad=True)
```

3.2.4 定义模型

下面是线性回归的矢量计算表达式的实现。我们使用 `mm` 函数做矩阵乘法。

```
def linreg(X, w, b): # 本函数已保存在 d2lzh_pytorch 包中方便以后使用  
    return torch.mm(X, w) + b
```

3.2.5 定义损失函数

我们使用上一节描述的平方损失来定义线性回归的损失函数。在实现中，我们需要把真实值 `y` 变形成预测值 `y_hat` 的形状。以下函数返回的结果也将和 `y_hat` 的形状相同。

```
def squared_loss(y_hat, y): # 本函数已保存在 d2lzh_pytorch 包中方便以后使用  
    # 注意这里返回的是向量，另外，pytorch 里的 MSELoss 并没有除以 2  
    return (y_hat - y.view(y_hat.size())) ** 2 / 2
```

3.2.6 定义优化算法

以下的 `sgd` 函数实现了上一节中介绍的小批量随机梯度下降算法。它通过不断迭代模型参数来优化损失函数。这里自动求梯度模块计算得来的梯度是一个批量样本的梯度和。我们将它除以批量大小来得到平均值。

```
def sgd(params, lr, batch_size): # 本函数已保存在 d2lzh_pytorch 包中方便以后使用  
    for param in params:  
        param.data -= lr * param.grad / batch_size # 注意这里更改 param 时用的 param.
```

3.2.7 训练模型

在训练中，我们将多次迭代模型参数。在每次迭代中，我们根据当前读取的小批量数据样本（特征 `X` 和标签 `y`），通过调用反向函数 `backward` 计算小批量随机梯度，并调用优化算法 `sgd` 迭代模型参数。由于我们之前设批量大小 `batch_size` 为 10，每个小批量的损失 `l` 的形状为 $(10, 1)$ 。回忆一下自动求梯度一节。由于变量 `l` 并不是一个

标量，所以我们可以调用`.sum()` 将其求和得到一个标量，再运行`l.backward()` 得到该变量有关模型参数的梯度。注意在每次更新完参数后不要忘了将参数的梯度清零。

在一个迭代周期 (epoch) 中，我们将完整遍历一遍`data_iter` 函数，并对训练数据集中所有样本都使用一次（假设样本数能够被批量大小整除）。这里的迭代周期个数`num_epochs` 和学习率`lr` 都是超参数，分别设 3 和 0.03。在实践中，大多超参数都需要通过反复试错来不断调节。虽然迭代周期数设得越大模型可能越有效，但是训练时间可能过长。而有关学习率对模型的影响，我们会在后面“优化算法”一章中详细介绍。

```
lr = 0.03
num_epochs = 3
net = linreg
loss = squared_loss

for epoch in range(num_epochs):  # 训练模型一共需要 num_epochs 个迭代周期
    # 在每一个迭代周期中，会使用训练数据集中所有样本一次（假设样本数能够被批量大小整除）。
    # 和 y 分别是小批量样本的特征和标签
    for X, y in data_iter(batch_size, features, labels):
        l = loss(net(X, w, b), y).sum()  # l 是有关小批量 X 和 y 的损失
        l.backward()  # 小批量的损失对模型参数求梯度
        sgd([w, b], lr, batch_size)  # 使用小批量随机梯度下降迭代模型参数

    # 不要忘了梯度清零
    w.grad.data.zero_()
    b.grad.data.zero_()
    train_l = loss(net(features, w, b), labels)
    print('epoch %d, loss %f' % (epoch + 1, train_l.mean().item()))
```

输出：

```
epoch 1, loss 0.028127
epoch 2, loss 0.000095
epoch 3, loss 0.000050
```

训练完成后，我们可以比较学到的参数和用来生成训练集的真实参数。它们应该很接近。

```
print(true_w, '\n', w)
print(true_b, '\n', b)
```

输出：

```
[2, -3.4]
tensor([[ 1.9998],
       [-3.3998]], requires_grad=True)
4.2
tensor([4.2001], requires_grad=True)
```

小结

- 可以看出，仅使用 `Tensor` 和 `autograd` 模块就可以很容易地实现一个模型。接下来，本书会在此基础上描述更多深度学习模型，并介绍怎样使用更简洁的代码（见下一节）来实现它们。

注：本节除了代码之外与原书基本相同，[原书传送门](#)

3.3 线性回归的简洁实现

随着深度学习框架的发展，开发深度学习应用变得越来越便利。实践中，我们通常可以用比上一节更简洁的代码来实现同样的模型。在本节中，我们将介绍如何使用 PyTorch 更方便地实现线性回归的训练。

3.3.1 生成数据集

我们生成与上一节中相同的数据集。其中 `features` 是训练数据特征，`labels` 是标签。

```
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
features = torch.tensor(np.random.normal(0, 1, (num_examples, num_inputs)), dtype=torch.float32)
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size()), dtype=torch.float32)
```

3.3.2 读取数据

PyTorch 提供了 `data` 包来读取数据。由于 `data` 常用作变量名，我们将导入的 `data` 模块用 `Data` 代替。在每一次迭代中，我们将随机读取包含 10 个数据样本的小批量。

```
import torch.utils.data as Data

batch_size = 10
# 将训练数据的特征和标签组合
dataset = Data.TensorDataset(features, labels)
# 随机读取小批量
data_iter = Data.DataLoader(dataset, batch_size, shuffle=True)
```

这里 `data_iter` 的使用跟上一节中的一样。让我们读取并打印第一个小批量数据样本。

```
for X, y in data_iter:
    print(X, y)
    break
```

输出：

```
tensor([[-2.7723, -0.6627],  
       [-1.1058,  0.7688],  
       [ 0.4901, -1.2260],  
       [-0.7227, -0.2664],  
       [-0.3390,  0.1162],  
       [ 1.6705, -2.7930],  
       [ 0.2576, -0.2928],  
       [ 2.0475, -2.7440],  
       [ 1.0685,  1.1920],  
       [ 1.0996,  0.5106]])  
tensor([ 0.9066, -0.6247,  9.3383,  3.6537,  3.1283, 17.0213,  5.6953, 17.6279,  
       2.2809,  4.6661])
```

3.3.3 定义模型

在上一节从零开始的实现中，我们需要定义模型参数，并使用它们一步步描述模型是怎样计算的。当模型结构变得更复杂时，这些步骤将变得更繁琐。其实，PyTorch 提供了大量预定义的层，这使我们只需关注使用哪些层来构造模型。下面将介绍如何使用 PyTorch 更简洁地定义线性回归。

首先，导入 `torch.nn` 模块。实际上，“nn”是 neural networks（神经网络）的缩写。顾名思义，该模块定义了大量神经网络的层。之前我们已经用过了 `autograd`，而 `nn` 就是利用 `autograd` 来定义模型。`nn` 的核心数据结构是 `Module`，它是一个抽象概念，既可以表示神经网络中的某个层（layer），也可以表示一个包含很多层的神经网络。在实际使用中，最常见的做法是继承 `nn.Module`，撰写自己的网络/层。一个 `nn.Module` 实例应该包含一些层以及返回输出的前向传播（forward）方法。下面先来看看如何用 `nn.Module` 实现一个线性回归模型。

```
class LinearNet(nn.Module):  
    def __init__(self, n_feature):  
        super(LinearNet, self).__init__()  
        self.linear = nn.Linear(n_feature, 1)  
    # forward 定义前向传播  
    def forward(self, x):  
        y = self.linear(x)  
        return y
```

```
net = LinearNet(num_inputs)
print(net) # 使用 print 可以打印出网络的结构
```

输出：

```
LinearNet(
    (linear): Linear(in_features=2, out_features=1, bias=True)
)
```

事实上我们还可以用 `nn.Sequential` 来更加方便地搭建网络，`Sequential` 是一个有序的容器，网络层将按照在传入 `Sequential` 的顺序依次被添加到计算图中。

写法一

```
net = nn.Sequential(
    nn.Linear(num_inputs, 1)
    # 此处还可以传入其他层
)
```

写法二

```
net = nn.Sequential()
net.add_module('linear', nn.Linear(num_inputs, 1))
# net.add_module .....
```

写法三

```
from collections import OrderedDict
net = nn.Sequential(OrderedDict([
    ('linear', nn.Linear(num_inputs, 1)),
    # .....
]))
```

```
print(net)
print(net[0])
```

输出：

```
Sequential(
    (linear): Linear(in_features=2, out_features=1, bias=True)
```

```
)  
Linear(in_features=2, out_features=1, bias=True)
```

可以通过 `net.parameters()` 来查看模型所有的可学习参数，此函数将返回一个生成器。

```
for param in net.parameters():  
    print(param)
```

输出：

```
Parameter containing:  
tensor([-0.0277,  0.2771], requires_grad=True)  
Parameter containing:  
tensor([0.3395], requires_grad=True)
```

回顾图 3.1 中线性回归在神经网络图中的表示。作为一个单层神经网络，线性回归输出层中的神经元和输入层中各个输入完全连接。因此，线性回归的输出层又叫全连接层。

注意：`torch.nn` 仅支持输入一个 batch 的样本不支持单个样本输入，如果只有单个样本，可使用 `input.unsqueeze(0)` 来添加一维。

3.3.4 初始化模型参数

在使用 `net` 前，我们需要初始化模型参数，如线性回归模型中的权重和偏差。PyTorch 在 `init` 模块中提供了多种参数初始化方法。这里的 `init` 是 `initializer` 的缩写形式。我们通过 `init.normal_` 将权重参数每个元素初始化为随机采样于均值为 0、标准差为 0.01 的正态分布。偏差会初始化为零。

```
from torch.nn import init  
  
init.normal_(net[0].weight, mean=0, std=0.01)  
init.constant_(net[0].bias, val=0) # 也可以直接修改 bias 的 data: net[0].bias.data.f
```

3.3.5 定义损失函数

PyTorch 在 `nn` 模块中提供了各种损失函数，这些损失函数可看作是一种特殊的层，PyTorch 也将这些损失函数实现为 `nn.Module` 的子类。我们现在使用它提供的均方误差损失作为模型的损失函数。

```
loss = nn.MSELoss()
```

3.3.6 定义优化算法

同样，我们也无须自己实现小批量随机梯度下降算法。`torch.optim` 模块提供了很多常用的优化算法比如 SGD、Adam 和 RMSProp 等。下面我们创建一个用于优化 `net` 所有参数的优化器实例，并指定学习率为 0.03 的小批量随机梯度下降（SGD）为优化算法。

```
import torch.optim as optim

optimizer = optim.SGD(net.parameters(), lr=0.03)
print(optimizer)
```

输出：

```
SGD (
Parameter Group 0
  dampening: 0
  lr: 0.03
  momentum: 0
  nesterov: False
  weight_decay: 0
)
```

我们还可以为不同子网络设置不同的学习率，这在 finetune 时经常用到。例：

```
optimizer = optim.SGD([
    # 如果对某个参数不指定学习率，就使用最外层的默认学习率
    {'params': net.subnet1.parameters()}, # lr=0.03
    {'params': net.subnet2.parameters(), 'lr': 0.01}
], lr=0.03)
```

有时候我们不想让学习率固定成一个常数，那如何调整学习率呢？主要有两种做法。一种是修改 `optimizer.param_groups` 中对应的学习率，另一种是更简单也是较为推荐的做法——新建优化器，由于 `optimizer` 十分轻量级，构建开销很小，故而可以构建新的 `optimizer`。但是后者对于使用动量的优化器（如 Adam），会丢失动量等状态信息，可能会造成损失函数的收敛出现震荡等情况。

```
# 调整学习率
for param_group in optimizer.param_groups:
    param_group['lr'] *= 0.1 # 学习率为之前的 0.1 倍
```

3.3.7 训练模型

在使用 Gluon 训练模型时，我们通过调用 `optim` 实例的 `step` 函数来迭代模型参数。按照小批量随机梯度下降的定义，我们在 `step` 函数中指明批量大小，从而对批量中样本梯度求平均。

```
num_epochs = 3
for epoch in range(1, num_epochs + 1):
    for X, y in data_iter:
        output = net(X)
        l = loss(output, y.view(-1, 1))
        optimizer.zero_grad() # 梯度清零，等价于 net.zero_grad()
        l.backward()
        optimizer.step()
        print('epoch %d, loss: %f' % (epoch, l.item()))
```

输出：

```
epoch 1, loss: 0.000457
epoch 2, loss: 0.000081
epoch 3, loss: 0.000198
```

下面我们分别比较学到的模型参数和真实的模型参数。我们从 `net` 获得需要的层，并访问其权重（`weight`）和偏差（`bias`）。学到的参数和真实的参数很接近。

```
dense = net[0]
print(true_w, dense.weight)
print(true_b, dense.bias)
```

输出：

```
[2, -3.4] tensor([[ 1.9999, -3.4005]])
4.2 tensor([4.2011])
```

小结

- 使用 PyTorch 可以更简洁地实现模型。
- `torch.utils.data` 模块提供了有关数据处理的工具，`torch.nn` 模块定义了大量神经网络的层，`torch.nn.init` 模块定义了各种初始化方法，`torch.optim` 模块提供了模型参数初始化的各种方法。

注：本节除了代码之外与原书基本相同，[原书传送门](#)

3.4 softmax 回归

前几节介绍的线性回归模型适用于输出为连续值的情景。在另一类情景中，模型输出可以是一个像图像类别这样的离散值。对于这样的离散值预测问题，我们可以使用诸如 softmax 回归在内的分类模型。和线性回归不同，softmax 回归的输出单元从一个变成了多个，且引入了 softmax 运算使输出更适合离散值的预测和训练。本节以 softmax 回归模型为例，介绍神经网络中的分类模型。

3.4.1 分类问题

让我们考虑一个简单的图像分类问题，其输入图像的高和宽均为 2 像素，且色彩为灰度。这样每个像素值都可以用一个标量表示。我们将图像中的 4 像素分别记为 x_1, x_2, x_3, x_4 。假设训练数据集中图像的真实标签为狗、猫或鸡（假设可以用 4 像素表示出这 3 种动物），这些标签分别对应离散值 y_1, y_2, y_3 。

我们通常使用离散的数值来表示类别，例如 $y_1 = 1, y_2 = 2, y_3 = 3$ 。如此，一张图像的标签为 1、2 和 3 这 3 个数值中的一个。虽然我们仍然可以使用回归模型来进行建模，并将预测值就近定点化到 1、2 和 3 这 3 个离散值之一，但这种连续值到离散值的转化通常会影响到分类质量。因此我们一般使用更加适合离散值输出的模型来解决分类问题。

3.4.2 softmax 回归模型

softmax 回归跟线性回归一样将输入特征与权重做线性叠加。与线性回归的一个主要不同在于，softmax 回归的输出值个数等于标签里的类别数。因为一共有 4 种特征和 3 种输出动物类别，所以权重包含 12 个标量（带下标的 w ）、偏差包含 3 个标量（带下标的 b ），且对每个输入计算 o_1, o_2, o_3 这 3 个输出：

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1, \\ o_2 &= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2, \\ o_3 &= x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3. \end{aligned}$$

图 3.2 用神经网络图描绘了上面的计算。softmax 回归同线性回归一样，也是一个单层神经网络。由于每个输出 o_1, o_2, o_3 的计算都要依赖于所有的输入 x_1, x_2, x_3, x_4 ，softmax 回归的输出层也是一个全连接层。

图 3.2 softmax 回归是一个单层神经网络

既然分类问题需要得到离散的预测输出，一个简单的办法是将输出值 o_i 当作预测类别是 i 的置信度，并将值最大的输出所对应的类作为预测输出，即输出 $\arg \max_i o_i$ 。例

如, 如果 o_1, o_2, o_3 分别为 0.1, 10, 0.1, 由于 o_2 最大, 那么预测类别为 2, 其代表猫。

然而, 直接使用输出层的输出有两个问题。一方面, 由于输出层的输出值的范围不确定, 我们难以直观上判断这些值的意义。例如, 刚才举的例子中的输出值 10 表示“很置信”图像类别为猫, 因为该输出值是其他两类的输出值的 100 倍。但如果 $o_1 = o_3 = 10^3$, 那么输出值 10 却又表示图像类别为猫的概率很低。另一方面, 由于真实标签是离散值, 这些离散值与不确定范围的输出值之间的误差难以衡量。

softmax 运算符 (softmax operator) 解决了以上两个问题。它通过下式将输出值变换成值为正且和为 1 的概率分布:

$$\hat{y}_1, \hat{y}_2, \hat{y}_3 = \text{softmax}(o_1, o_2, o_3)$$

其中

$$\hat{y}_1 = \frac{\exp(o_1)}{\sum_{i=1}^3 \exp(o_i)}, \quad \hat{y}_2 = \frac{\exp(o_2)}{\sum_{i=1}^3 \exp(o_i)}, \quad \hat{y}_3 = \frac{\exp(o_3)}{\sum_{i=1}^3 \exp(o_i)}.$$

容易看出 $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$ 且 $0 \leq \hat{y}_1, \hat{y}_2, \hat{y}_3 \leq 1$, 因此 $\hat{y}_1, \hat{y}_2, \hat{y}_3$ 是一个合法的概率分布。这时候, 如果 $\hat{y}_2 = 0.8$, 不管 \hat{y}_1 和 \hat{y}_3 的值是多少, 我们都知道图像类别为猫的概率是 80%。此外, 我们注意到

$$\arg \max_i o_i = \arg \max_i \hat{y}_i$$

因此 softmax 运算不改变预测类别输出。

3.4.3 单样本分类的矢量计算表达式

为了提高计算效率, 我们可以将单样本分类通过矢量计算来表达。在上面的图像分类问题中, 假设 softmax 回归的权重和偏差参数分别为

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix}, \quad \mathbf{b} = [b_1 \ b_2 \ b_3],$$

设高和宽分别为 2 个像素的图像样本 i 的特征为

$$\mathbf{x}^{(i)} = [x_1^{(i)} \ x_2^{(i)} \ x_3^{(i)} \ x_4^{(i)}],$$

输出层的输出为

$$\mathbf{o}^{(i)} = \begin{bmatrix} o_1^{(i)} & o_2^{(i)} & o_3^{(i)} \end{bmatrix},$$

预测为狗、猫或鸡的概率分布为

$$\hat{\mathbf{y}}^{(i)} = \begin{bmatrix} \hat{y}_1^{(i)} & \hat{y}_2^{(i)} & \hat{y}_3^{(i)} \end{bmatrix}.$$

softmax 回归对样本 i 分类的矢量计算表达式为

$$\begin{aligned}\mathbf{o}^{(i)} &= \mathbf{x}^{(i)}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{y}}^{(i)} &= \text{softmax}(\mathbf{o}^{(i)}).\end{aligned}$$

3.4.4 小批量样本分类的矢量计算表达式

为了进一步提升计算效率，我们通常对小批量数据做矢量计算。广义上讲，给定一个小批量样本，其批量大小为 n ，输入个数（特征数）为 d ，输出个数（类别数）为 q 。设批量特征为 $\mathbf{X} \in \mathbb{R}^{n \times d}$ 。假设 softmax 回归的权重和偏差参数分别为 $\mathbf{W} \in \mathbb{R}^{d \times q}$ 和 $\mathbf{b} \in \mathbb{R}^{1 \times q}$ 。softmax 回归的矢量计算表达式为

$$\begin{aligned}\mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{O}),\end{aligned}$$

其中的加法运算使用了广播机制， $\mathbf{O}, \hat{\mathbf{Y}} \in \mathbb{R}^{n \times q}$ 且这两个矩阵的第 i 行分别为样本 i 的输出 $\mathbf{o}^{(i)}$ 和概率分布 $\hat{\mathbf{y}}^{(i)}$ 。

3.4.5 交叉熵损失函数

前面提到，使用 softmax 运算后可以更方便地与离散标签计算误差。我们已经知道，softmax 运算将输出转换成一个合法的类别预测分布。实际上，真实标签也可以用类别分布表达：对于样本 i ，我们构造向量 $\mathbf{y}^{(i)} \in \mathbb{R}^q$ ，使其第 $y^{(i)}$ （样本 i 类别的离散数值）个元素为 1，其余为 0。这样我们的训练目标可以设为使预测概率分布 $\hat{\mathbf{y}}^{(i)}$ 尽可能接近真实的标签概率分布 $\mathbf{y}^{(i)}$ 。

我们可以像线性回归那样使用平方损失函数 $\|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\|^2/2$ 。然而，想要预测分类结果正确，我们其实并不需要预测概率完全等于标签概率。例如，在图像分类的例子中，如果 $y^{(i)} = 3$ ，那么我们只需要 $\hat{y}_3^{(i)}$ 比其他两个预测值 $\hat{y}_1^{(i)}$ 和 $\hat{y}_2^{(i)}$ 大就行了。即使 $\hat{y}_3^{(i)}$ 值为 0.6，不管其他两个预测值为多少，类别预测均正确。而平方损失则过于严格，例如 $\hat{y}_1^{(i)} = \hat{y}_2^{(i)} = 0.2$ 比 $\hat{y}_1^{(i)} = 0, \hat{y}_2^{(i)} = 0.4$ 的损失要小很多，虽然两者都有同样正确的分类预测结果。

改善上述问题的一个方法是使用更适合衡量两个概率分布差异的测量函数。其中，交叉熵（cross entropy）是一个常用的衡量方法：

$$H(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) = -\sum_{j=1}^q y_j^{(i)} \log \hat{y}_j^{(i)},$$

其中带下标的 $y_j^{(i)}$ 是向量 $\mathbf{y}^{(i)}$ 中非 0 即 1 的元素，需要注意将它与样本 i 类别的离散数值，即不带下标的 $y^{(i)}$ 区分。在上式中，我们知道向量 $\mathbf{y}^{(i)}$ 中只有第 $y^{(i)}$ 个元素 $y_{y^{(i)}}^{(i)}$ 为 1，其余全为 0，于是 $H(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) = -\log \hat{y}_{y^{(i)}}^{(i)}$ 。也就是说，交叉熵只关心对正确类别的预测概率，因为只要其值足够大，就可以确保分类结果正确。当然，遇到一个样本有多个标签时，例如图像里含有不止一个物体时，我们并不能做这一步简化。但即便对于这种情况，交叉熵同样只关心对图像中出现的物体类别的预测概率。

假设训练数据集的样本数为 n ，交叉熵损失函数定义为

$$\ell(\Theta) = \frac{1}{n} \sum_{i=1}^n H(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}),$$

其中 Θ 代表模型参数。同样地，如果每个样本只有一个标签，那么交叉熵损失可以简写成 $\ell(\Theta) = -(1/n) \sum_{i=1}^n \log \hat{y}_{y^{(i)}}^{(i)}$ 。从另一个角度来看，我们知道最小化 $\ell(\Theta)$ 等价于最大化 $\exp(-n\ell(\Theta)) = \prod_{i=1}^n \hat{y}_{y^{(i)}}^{(i)}$ ，即最小化交叉熵损失函数等价于最大化训练数据集所有标签类别的联合预测概率。

3.4.6 模型预测及评价

在训练好 softmax 回归模型后，给定任一样本特征，就可以预测每个输出类别的概率。通常，我们把预测概率最大的类别作为输出类别。如果它与真实类别（标签）一致，说明这次预测是正确的。在 3.6 节的实验中，我们将使用准确率（accuracy）来评价模型的表现。它等于正确预测数量与总预测数量之比。

小结

- softmax 回归适用于分类问题。它使用 softmax 运算输出类别的概率分布。
- softmax 回归是一个单层神经网络，输出个数等于分类问题中的类别个数。
- 交叉熵适合衡量两个概率分布的差异。

注：本节与原书基本相同，[原书此节传送门](#)

3.5 图像分类数据集 (Fashion-MNIST)

在介绍 softmax 回归的实现前我们先引入一个多类图像分类数据集。它将在后面的章节中被多次使用，以方便我们观察比较算法之间在模型精度和计算效率上的区别。图像分类数据集中最常用的是手写数字识别数据集 MNIST[1]。但大部分模型在 MNIST 上的分类精度都超过了 95%。为了更直观地观察算法之间的差异，我们将使用一个图像内容更加复杂的数据集 Fashion-MNIST[2]（这个数据集也比较小，只有几十 M，没有 GPU 的电脑也能吃得消）。

本节我们将使用 torchvision 包，它是服务于 PyTorch 深度学习框架的，主要用来构建计算机视觉模型。torchvision 主要由以下几部分构成：1. `torchvision.datasets`: 一些加载数据的函数及常用的数据集接口；2. `torchvision.models`: 包含常用的模型结构（含预训练模型），例如 AlexNet、VGG、ResNet 等；3. `torchvision.transforms`: 常用的图片变换，例如裁剪、旋转等；4. `torchvision.utils`: 其他的一些有用的方法。

3.5.1 获得数据集

首先导入本节需要的包或模块。

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import time
import sys
sys.path.append("..") # 为了导入上层目录的 d2lzh_pytorch
import d2lzh_pytorch as d2l
```

下面，我们通过 torchvision 的 `torchvision.datasets` 来下载这个数据集。第一次调用时会自动从网上获取数据。我们通过参数 `train` 来指定获取训练数据集或测试数据集 (testing data set)。测试数据集也叫测试集 (testing set)，只用来评价模型的表现，并不用于训练模型。

另外我们还指定了参数 `transform = transforms.ToTensor()` 使所有数据转换为 `Tensor`，如果不进行转换则返回的是 PIL 图片。`transforms.ToTensor()` 将尺寸为 (H x W x C) 且数据位于 [0, 255] 的 PIL 图片或者数据类型为 `np.uint8` 的 NumPy 数组转换为尺寸为 (C x H x W) 且数据类型为 `torch.float32` 且位于 [0.0, 1.0] 的 `Tensor`。
> 注意：由于像素值为 0 到 255 的整数，所以刚好是 `uint8` 所能表示的范围，包括 `transforms.ToTensor()` 在内的一些关于图片的函数就默认输入的是 `uint8` 型，若不

是，可能不会报错但的可能得不到想要的结果。所以，如果用像素值 (0-255 整数) 表示图片数据，那么一律将其类型设置成 `uint8`，避免不必要的 bug。本人就被这点坑过，详见[我的这个博客 2.2.4 节](#)。

```
mnist_train = torchvision.datasets.FashionMNIST(root='~/Datasets/FashionMNIST', train=True)
mnist_test = torchvision.datasets.FashionMNIST(root='~/Datasets/FashionMNIST', train=False)
```

上面的 `mnist_train` 和 `mnist_test` 都是`torch.utils.data.Dataset`的子类，所以我们可以用 `len()` 来获取该数据集的大小，还可以用下标来获取具体的一个样本。训练集中和测试集中的每个类别的图像数分别为 6,000 和 1,000。因为有 10 个类别，所以训练集和测试集的样本数分别为 60,000 和 10,000。

```
print(type(mnist_train))
print(len(mnist_train), len(mnist_test))
```

输出：

```
<class 'torchvision.datasets.mnist.FashionMNIST'>
60000 10000
```

我们可以通过下标来访问任意一个样本：

```
feature, label = mnist_train[0]
print(feature.shape, label) # Channel x Height X Width
```

输出：

```
torch.Size([1, 28, 28]) tensor(9)
```

变量 `feature` 对应高和宽均为 28 像素的图像。由于我们使用了 `transforms.ToTensor()`，所以每个像素的数值为 [0.0, 1.0] 的 32 位浮点数。需要注意的是，`feature` 的尺寸是 (C x H x W) 的，而不是 (H x W x C)。第一维是通道数，因为数据集中是灰度图像，所以通道数为 1。后面两维分别是图像的高和宽。

Fashion-MNIST 中一共包括了 10 个类别，分别为 t-shirt (T 恤)、trouser (裤子)、pullover (套衫)、dress (连衣裙)、coat (外套)、sandal (凉鞋)、shirt (衬衫)、sneaker (运动鞋)、bag (包) 和 ankle boot (短靴)。以下函数可以将数值标签转成相应的文本标签。

```
# 本函数已保存在 d2lzh 包中方便以后使用
def get_fashion_mnist_labels(labels):
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                   'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]
```

下面定义一个可以在一行里画出多张图像和对应标签的函数。

```
# 本函数已保存在 d2lzh 包中方便以后使用
def show_fashion_mnist(images, labels):
    d2l.use_svg_display()
    # 这里的 _ 表示我们忽略（不使用）的变量
    _, figs = plt.subplots(1, len(images), figsize=(12, 12))
    for f, img, lbl in zip(figs, images, labels):
        f.imshow(img.view((28, 28)).numpy())
        f.set_title(lbl)
        f.axes.get_xaxis().set_visible(False)
        f.axes.get_yaxis().set_visible(False)
    plt.show()
```

现在，我们看一下训练数据集中前 9 个样本的图像内容和文本标签。

```
X, y = [], []
for i in range(10):
    X.append(mnist_train[i][0])
    y.append(mnist_train[i][1])
show_fashion_mnist(X, get_fashion_mnist_labels(y))
```

3.5.2 读取小批量

我们将在训练数据集上训练模型，并将训练好的模型在测试数据集上评价模型的表现。前面说过，`mnist_train` 是 `torch.utils.data.Dataset` 的子类，所以我们可以将其传入 `torch.utils.data.DataLoader` 来创建一个读取小批量数据样本的 `DataLoader` 实例。

在实践中，数据读取经常是训练的性能瓶颈，特别当模型较简单或者计算硬件性能较高时。PyTorch 的 `DataLoader` 中一个很方便的功能是允许使用多进程来加速数据读取。这里我们通过参数 `num_workers` 来设置 4 个进程读取数据。

```
batch_size = 256
if sys.platform.startswith('win'):
    num_workers = 0 # 0 表示不用额外的进程来加速读取数据
else:
    num_workers = 4
train_iter = torch.utils.data.DataLoader(mnist_train, batch_size=batch_size, shuffle=True)
test_iter = torch.utils.data.DataLoader(mnist_test, batch_size=batch_size, shuffle=False)
```

我们将获取并读取 Fashion-MNIST 数据集的逻辑封装在 `d2lzh_pytorch.load_data_fashion_mnist` 函数中供后面章节调用。该函数将返回 `train_iter` 和 `test_iter` 两个变量。随着本书内容的不断深入，我们会进一步改进该函数。它的完整实现将在 5.6 节中描述。

最后我们查看读取一遍训练数据需要的时间。

```
start = time.time()
for X, y in train_iter:
    continue
print('%.2f sec' % (time.time() - start))
```

输出：

```
1.57 sec
```

小结

- Fashion-MNIST 是一个 10 类服饰分类数据集，之后章节里将使用它来检验不同算法的表现。
- 我们将高和宽分别为 h 和 w 像素的图像的形状记为 $h \times w$ 或 (h, w) 。

参考文献

-
- [1] LeCun, Y., Cortes, C., & Burges, C. <http://yann.lecun.com/exdb/mnist/>
 - [2] Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747.

注：本节除了代码之外与原书基本相同，[原书传送门](#)

3.6 softmax 回归的从零开始实现

这一节我们来动手实现 softmax 回归。首先导入本节实现所需的包或模块。

```
import torch
import torchvision
import numpy as np
import sys
sys.path.append("..") # 为了导入上层目录的 d2lzh_pytorch
import d2lzh_pytorch as d2l
```

3.6.1 获得和读取数据

我们将使用 Fashion-MNIST 数据集，并设置批量大小为 256。

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

3.6.2 初始化模型参数

跟线性回归中的例子一样，我们将使用向量表示每个样本。已知每个样本输入是高和宽均为 28 像素的图像。模型的输入向量的长度是 $28 \times 28 = 784$: 该向量的每个元素对应图像中每个像素。由于图像有 10 个类别，单层神经网络输出层的输出个数为 10，因此 softmax 回归的权重和偏差参数分别为 784×10 和 1×10 的矩阵。

```
num_inputs = 784
num_outputs = 10
```

```
W = torch.tensor(np.random.normal(0, 0.01, (num_inputs, num_outputs)), dtype=torch.float)
b = torch.zeros(num_outputs, dtype=torch.float)
```

同之前一样，我们需要模型参数梯度。

```
W.requires_grad_(requires_grad=True)
b.requires_grad_(requires_grad=True)
```

3.6.3 实现 softmax 运算

在介绍如何定义 softmax 回归之前，我们先描述一下对如何对多维 Tensor 按维度操作。在下面的例子中，给定一个 Tensor 矩阵 X。我们可以只对其中同一列 (`dim=0`) 或同一行 (`dim=1`) 的元素求和，并在结果中保留行和列这两个维度 (`keepdim=True`)。

```
X = torch.tensor([[1, 2, 3], [4, 5, 6]])
print(X.sum(dim=0, keepdim=True))
print(X.sum(dim=1, keepdim=True))
```

输出：

```
tensor([[5, 7, 9]])
tensor([[ 6,
         15]])
```

下面我们就可以定义前面小节里介绍的 softmax 运算了。在下面的函数中，矩阵 X 的行数是样本数，列数是输出个数。为了表达样本预测各个输出的概率，softmax 运算会先通过 `exp` 函数对每个元素做指数运算，再对 `exp` 矩阵同行元素求和，最后令矩阵每行各元素与该行元素之和相除。这样一来，最终得到的矩阵每行元素和为 1 且非负。因此，该矩阵每行都是合法的概率分布。softmax 运算的输出矩阵中的任意一行元素代表了一个样本在各个输出类别上的预测概率。

```
def softmax(X):
    X_exp = X.exp()
    partition = X_exp.sum(dim=1, keepdim=True)
    return X_exp / partition # 这里应用了广播机制
```

可以看到，对于随机输入，我们将每个元素变成了非负数，且每一行和为 1。

```
X = torch.rand((2, 5))
X_prob = softmax(X)
print(X_prob, X_prob.sum(dim=1))
```

输出：

```
tensor([[0.2206, 0.1520, 0.1446, 0.2690, 0.2138],
       [0.1540, 0.2290, 0.1387, 0.2019, 0.2765]]) tensor([1., 1.])
```

3.6.4 定义模型

有了 softmax 运算，我们可以定义上节描述的 softmax 回归模型了。这里通过 `view` 函数将每张原始图像改成长度为 `num_inputs` 的向量。

```
def net(X):
    return softmax(torch.mm(X.view((-1, num_inputs)), W) + b)
```

3.6.5 定义损失函数

上一节中，我们介绍了 softmax 回归使用的交叉熵损失函数。为了得到标签的预测概率，我们可以使用 `gather` 函数。在下面的例子中，变量 `y_hat` 是 2 个样本在 3 个类别的预测概率，变量 `y` 是这 2 个样本的标签类别。通过使用 `gather` 函数，我们得到了 2 个样本的标签的预测概率。与 3.4 节（softmax 回归）数学表述中标签类别离散值从 1 开始逐一递增不同，在代码中，标签类别的离散值是从 0 开始逐一递增的。

```
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y = torch.LongTensor([0, 2])
y_hat.gather(1, y.view(-1, 1))
```

输出：

```
tensor([[0.1000],
       [0.5000]])
```

下面实现了 3.4 节（softmax 回归）中介绍的交叉熵损失函数。

```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat.gather(1, y.view(-1, 1)))
```

3.6.6 计算分类准确率

给定一个类别的预测概率分布 `y_hat`，我们把预测概率最大的类别作为输出类别。如果它与真实类别 `y` 一致，说明这次预测是正确的。分类准确率即正确预测数量与总预测数量之比。

为了演示准确率的计算，下面定义准确率 `accuracy` 函数。其中 `y_hat.argmax(dim=1)` 返回矩阵 `y_hat` 每行中最大元素的索引，且返回结果与变量 `y` 形状相同。相等条件判断式 (`y_hat.argmax(dim=1) == y`) 是一个类型为 `ByteTensor` 的 `Tensor`，我们用 `float()` 将其转换为值为 0（相等为假）或 1（相等为真）的浮点型 `Tensor`。

```
def accuracy(y_hat, y):  
    return (y_hat.argmax(dim=1) == y).float().mean().item()
```

让我们继续使用在演示 `gather` 函数时定义的变量 `y_hat` 和 `y`，并将它们分别作为预测概率分布和标签。可以看到，第一个样本预测类别为 2（该行最大元素 0.6 在本行的索引为 2），与真实标签 0 不一致；第二个样本预测类别为 2（该行最大元素 0.5 在本行的索引为 2），与真实标签 2 一致。因此，这两个样本上的分类准确率为 0.5。

```
print(accuracy(y_hat, y))
```

输出：

0.5

类似地，我们可以评价模型 `net` 在数据集 `data_iter` 上的准确率。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用。该函数将被逐步改进：它的完整实现将在“图  
def evaluate_accuracy(data_iter, net):  
    acc_sum, n = 0.0, 0  
    for X, y in data_iter:  
        acc_sum += (net(X).argmax(dim=1) == y).float().sum().item()  
        n += y.shape[0]  
    return acc_sum / n
```

因为我们随机初始化了模型 `net`，所以这个随机模型的准确率应该接近于类别个数 10 的倒数即 0.1。

```
print(evaluate_accuracy(test_iter, net))
```

输出：

0.0681

3.6.7 训练模型

训练 softmax 回归的实现跟“[线性回归的从零开始实现](#)”一节介绍的线性回归中的实现非常相似。我们同样使用小批量随机梯度下降来优化模型的损失函数。在训练模型时，迭代周期数 `num_epochs` 和学习率 `lr` 都是可以调的超参数。改变它们的值可能会得到分类更准确的模型。

```

num_epochs, lr = 5, 0.1

# 本函数已保存在 d2lzh 包中方便以后使用
def train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size,
              params=None, lr=None, optimizer=None):
    for epoch in range(num_epochs):
        train_l_sum, train_acc_sum, n = 0.0, 0.0, 0
        for X, y in train_iter:
            y_hat = net(X)
            l = loss(y_hat, y).sum()

            # 梯度清零
            if optimizer is not None:
                optimizer.zero_grad()
            elif params is not None and params[0].grad is not None:
                for param in params:
                    param.grad.data.zero_()

            l.backward()
            if optimizer is None:
                d2l.sgd(params, lr, batch_size)
            else:
                optimizer.step()  # “softmax 回归的简洁实现”一节将用到

        train_l_sum += l.item()
        train_acc_sum += (y_hat.argmax(dim=1) == y).sum().item()
        n += y.shape[0]
        test_acc = evaluate_accuracy(test_iter, net)
        print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f'
              % (epoch + 1, train_l_sum / n, train_acc_sum / n, test_acc))

train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, batch_size, [W, b],
          output:
epoch 1, loss 0.7878, train acc 0.749, test acc 0.794

```

```
epoch 2, loss 0.5702, train acc 0.814, test acc 0.813
epoch 3, loss 0.5252, train acc 0.827, test acc 0.819
epoch 4, loss 0.5010, train acc 0.833, test acc 0.824
epoch 5, loss 0.4858, train acc 0.836, test acc 0.815
```

3.6.8 预测

训练完成后，现在就可以演示如何对图像进行分类了。给定一系列图像（第三行图像输出），我们比较一下它们的真实标签（第一行文本输出）和模型预测结果（第二行文本输出）。

```
X, y = iter(test_iter).next()

true_labels = d2l.get_fashion_mnist_labels(y.numpy())
pred_labels = d2l.get_fashion_mnist_labels(net(X).argmax(dim=1).numpy())
titles = [true + '\n' + pred for true, pred in zip(true_labels, pred_labels)]

d2l.show_fashion_mnist(X[0:9], titles[0:9])
```

小结

- 可以使用 softmax 回归做多类别分类。与训练线性回归相比，你会发现训练 softmax 回归的步骤和它非常相似：获取并读取数据、定义模型和损失函数并使用优化算法训练模型。事实上，绝大多数深度学习模型的训练都有着类似的步骤。

注：本节除了代码之外与原书基本相同，[原书传送门](#)

3.7 softmax 回归的简洁实现

我们在 3.3 节（线性回归的简洁实现）中已经了解了使用 Pytorch 实现模型的便利。下面，让我们再次使用 Pytorch 来实现一个 softmax 回归模型。首先导入所需的包或模块。

```
import torch
from torch import nn
from torch.nn import init
import numpy as np
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
```

3.7.1 获取和读取数据

我们仍然使用 Fashion-MNIST 数据集和上一节中设置的批量大小。

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

3.7.2 定义和初始化模型

在 3.4 节（softmax 回归）中提到，softmax 回归的输出层是一个全连接层，所以我们用一个线性模块就可以了。因为前面我们数据返回的每个 batch 样本 x 的形状为 $(batch_size, 1, 28, 28)$ ，所以我们要先用 `view()` 将 x 的形状转换成 $(batch_size, 784)$ 才送入全连接层。

```
num_inputs = 784
num_outputs = 10

class LinearNet(nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(LinearNet, self).__init__()
        self.linear = nn.Linear(num_inputs, num_outputs)
    def forward(self, x): # x shape: (batch, 1, 28, 28)
        y = self.linear(x.view(x.shape[0], -1))
        return y
```

```
net = LinearNet(num_inputs, num_outputs)
```

我们将对 x 的形状转换的这个功能自定义一个 `FlattenLayer` 并记录在 `d2lzh_pytorch` 中方便后面使用。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
class FlattenLayer(nn.Module):
    def __init__(self):
        super(FlattenLayer, self).__init__()
    def forward(self, x): # x shape: (batch, *, *, ...)
        return x.view(x.shape[0], -1)
```

这样我们就可以更方便地定义我们的模型：

```
from collections import OrderedDict
net = nn.Sequential(
    # FlattenLayer(),
    # nn.Linear(num_inputs, num_outputs)
    OrderedDict([
        ('flatten', FlattenLayer()),
        ('linear', nn.Linear(num_inputs, num_outputs))])
)
```

然后，我们使用均值为 0、标准差为 0.01 的正态分布随机初始化模型的权重参数。

```
init.normal_(net.linear.weight, mean=0, std=0.01)
init.constant_(net.linear.bias, val=0)
```

3.7.3 softmax 和交叉熵损失函数

如果做了上一节的练习，那么你可能意识到了分开定义 softmax 运算和交叉熵损失函数可能会造成数值不稳定。因此，PyTorch 提供了一个包括 softmax 运算和交叉熵损失计算的函数。它的数值稳定性更好。

```
loss = nn.CrossEntropyLoss()
```

3.7.4 定义优化算法

我们使用学习率为 0.1 的小批量随机梯度下降作为优化算法。

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
```

3.7.5 训练模型

接下来，我们使用上一节中定义的训练函数来训练模型。

```
num_epochs = 5
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None, None, o
```

输出：

```
epoch 1, loss 0.0031, train acc 0.745, test acc 0.790
epoch 2, loss 0.0022, train acc 0.812, test acc 0.807
epoch 3, loss 0.0021, train acc 0.825, test acc 0.806
epoch 4, loss 0.0020, train acc 0.832, test acc 0.810
epoch 5, loss 0.0019, train acc 0.838, test acc 0.823
```

小结

- PyTorch 提供的函数往往具有更好的数值稳定性。
- 可以使用 PyTorch 更简洁地实现 softmax 回归。

注：本节除了代码之外与原书基本相同，[原书传送门](#)

3.8 多层感知机

我们已经介绍了包括线性回归和 softmax 回归在内的单层神经网络。然而深度学习主要关注多层模型。在本节中，我们将以多层感知机（multilayer perceptron, MLP）为例，介绍多层神经网络的概念。

3.8.1 隐藏层

多层感知机在单层神经网络的基础上引入了一到多个隐藏层（hidden layer）。隐藏层位于输入层和输出层之间。图 3.3 展示了一个多层感知机的神经网络图，它含有一个隐藏层，该层中有 5 个隐藏单元。

图 3.3 带有隐藏层的多层感知机

在图 3.3 所示的多层感知机中，输入和输出个数分别为 4 和 3，中间的隐藏层中包含了 5 个隐藏单元（hidden unit）。由于输入层不涉及计算，图 3.3 中的多层感知机的层数为 2。由图 3.3 可见，隐藏层中的神经元和输入层中各个输入完全连接，输出层中的神经元和隐藏层中的各个神经元也完全连接。因此，多层感知机中的隐藏层和输出层都是全连接层。

具体来说，给定一个小批量样本 $\mathbf{X} \in \mathbb{R}^{n \times d}$ ，其批量大小为 n ，输入个数为 d 。假设多层感知机只有一个隐藏层，其中隐藏单元个数为 h 。记隐藏层的输出（也称为隐藏层变量或隐藏变量）为 \mathbf{H} ，有 $\mathbf{H} \in \mathbb{R}^{n \times h}$ 。因为隐藏层和输出层均是全连接层，可以设隐藏层的权重参数和偏差参数分别为 $\mathbf{W}_h \in \mathbb{R}^{d \times h}$ 和 $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ，输出层的权重和偏差参数分别为 $\mathbf{W}_o \in \mathbb{R}^{h \times q}$ 和 $\mathbf{b}_o \in \mathbb{R}^{1 \times q}$ 。

我们先来看一种含单隐藏层的多层感知机的设计。其输出 $\mathbf{O} \in \mathbb{R}^{n \times q}$ 的计算为

$$\begin{aligned}\mathbf{H} &= \mathbf{X}\mathbf{W}_h + \mathbf{b}_h, \\ \mathbf{O} &= \mathbf{H}\mathbf{W}_o + \mathbf{b}_o,\end{aligned}$$

也就是将隐藏层的输出直接作为输出层的输入。如果将以上两个式子联立起来，可以得到

$$\mathbf{O} = (\mathbf{X}\mathbf{W}_h + \mathbf{b}_h)\mathbf{W}_o + \mathbf{b}_o = \mathbf{X}\mathbf{W}_h\mathbf{W}_o + \mathbf{b}_h\mathbf{W}_o + \mathbf{b}_o.$$

从联立后的式子可以看出，虽然神经网络引入了隐藏层，却依然等价于一个单层神经网络：其中输出层权重参数为 $\mathbf{W}_h\mathbf{W}_o$ ，偏差参数为 $\mathbf{b}_h\mathbf{W}_o + \mathbf{b}_o$ 。不难发现，即便再添加更多的隐藏层，以上设计依然只能与仅含输出层的单层神经网络等价。

3.8.2 激活函数

上述问题的根源在于全连接层只是对数据做仿射变换（affine transformation），而多个仿射变换的叠加仍然是一个仿射变换。解决问题的一个方法是引入非线性变换，例如对隐藏变量使用按元素运算的非线性函数进行变换，然后再作为下一个全连接层的输入。这个非线性函数被称为激活函数（activation function）。下面我们介绍几个常用的激活函数。

3.8.2.1 ReLU 函数

ReLU (rectified linear unit) 函数提供了一个很简单的非线性变换。给定元素 x ，该函数定义为

$$\text{ReLU}(x) = \max(x, 0).$$

可以看出，ReLU 函数只保留正数元素，并将负数元素清零。为了直观地观察这一非线性变换，我们先定义一个绘图函数 `xyplot`。

```
%matplotlib inline
import torch
import numpy as np
import matplotlib.pyplot as plt
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l

def xyplot(x_vals, y_vals, name):
    d2l.set_figsize(figsize=(5, 2.5))
    d2l.plt.plot(x_vals.detach().numpy(), y_vals.detach().numpy())
    d2l.plt.xlabel('x')
    d2l.plt.ylabel(name + '(x)')
```

我们接下来通过 `NDArray` 提供的 `relu` 函数来绘制 ReLU 函数。可以看到，该激活函数是一个两段线性函数。

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = x.relu()
xyplot(x, y, 'relu')
```

显然，当输入为负数时，ReLU 函数的导数为 0；当输入为正数时，ReLU 函数的导数为 1。尽管输入为 0 时 ReLU 函数不可导，但是我们可以取此处的导数为 0。下面绘制 ReLU 函数的导数。

```
y.sum().backward()
xyplot(x, x.grad, 'grad of relu')
```

3.8.2.2 sigmoid 函数

sigmoid 函数可以将元素的值变换到 0 和 1 之间：

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

sigmoid 函数在早期的神经网络中较为普遍，但它目前逐渐被更简单的 ReLU 函数取代。在后面“循环神经网络”一章中我们会介绍如何利用它值域在 0 到 1 之间这一特性来控制信息在神经网络中的流动。下面绘制了 sigmoid 函数。当输入接近 0 时，sigmoid 函数接近线性变换。

```
y = x.sigmoid()
xyplot(x, y, 'sigmoid')
```

依据链式法则，sigmoid 函数的导数

$$\text{sigmoid}'(x) = \text{sigmoid}(x)(1 - \text{sigmoid}(x)).$$

下面绘制了 sigmoid 函数的导数。当输入为 0 时，sigmoid 函数的导数达到最大值 0.25；当输入越偏离 0 时，sigmoid 函数的导数越接近 0。

```
x.grad.zero_()
y.sum().backward()
xyplot(x, x.grad, 'grad of sigmoid')
```

3.8.2.3 tanh 函数

tanh（双曲正切）函数可以将元素的值变换到 -1 和 1 之间：

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

我们接着绘制 tanh 函数。当输入接近 0 时，tanh 函数接近线性变换。虽然该函数的形状和 sigmoid 函数的形状很像，但 tanh 函数在坐标系的原点上对称。

```
y = x.tanh()
xyplot(x, y, 'tanh')
```

依据链式法则， \tanh 函数的导数

$$\tanh'(x) = 1 - \tanh^2(x).$$

下面绘制了 \tanh 函数的导数。当输入为 0 时， \tanh 函数的导数达到最大值 1；当输入越偏离 0 时， \tanh 函数的导数越接近 0。

```
x.grad.zero_()
y.sum().backward()
xyplot(x, x.grad, 'grad of tanh')
```

3.8.3 多层感知机

多层感知机就是含有至少一个隐藏层的由全连接层组成的神经网络，且每个隐藏层的输出通过激活函数进行变换。多层感知机的层数和各隐藏层中隐藏单元个数都是超参数。以单隐藏层为例并沿用本节之前定义的符号，多层感知机按以下方式计算输出：

$$\begin{aligned}\mathbf{H} &= \phi(\mathbf{XW}_h + \mathbf{b}_h), \\ \mathbf{O} &= \mathbf{HW}_o + \mathbf{b}_o,\end{aligned}$$

其中 ϕ 表示激活函数。在分类问题中，我们可以对输出 \mathbf{O} 做 softmax 运算，并使用 softmax 回归中的交叉熵损失函数。在回归问题中，我们将输出层的输出个数设为 1，并将输出 \mathbf{O} 直接提供给线性回归中使用的平方损失函数。

小结

- 多层感知机在输出层与输入层之间加入了一个或多个全连接隐藏层，并通过激活函数对隐藏层输出进行变换。
- 常用的激活函数包括 ReLU 函数、sigmoid 函数和 \tanh 函数。

注：本节除了代码之外与原书基本相同，[原书传送门](#)

3.9 多层感知机的从零开始实现

我们已经从上一节里了解了多层感知机的原理。下面，我们一起来动手实现一个多层感知机。首先导入实现所需的包或模块。

```
import torch
import numpy as np
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
```

3.9.1 获得和读取数据

这里继续使用 Fashion-MNIST 数据集。我们将使用多层感知机对图像进行分类。

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

3.9.2 定义模型参数

我们在 3.6 节（softmax 回归的从零开始实现）里已经介绍了，Fashion-MNIST 数据集中图像形状为 28×28 ，类别数为 10。本节中我们依然使用长度为 $28 \times 28 = 784$ 的向量表示每一张图像。因此，输入个数为 784，输出个数为 10。实验中，我们设超参数隐藏单元个数为 256。

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = torch.tensor(np.random.normal(0, 0.01, (num_inputs, num_hiddens)), dtype=torch.float)
b1 = torch.zeros(num_hiddens, dtype=torch.float)
W2 = torch.tensor(np.random.normal(0, 0.01, (num_hiddens, num_outputs)), dtype=torch.float)
b2 = torch.zeros(num_outputs, dtype=torch.float)

params = [W1, b1, W2, b2]
for param in params:
    param.requires_grad_(requires_grad=True)
```

3.9.3 定义激活函数

这里我们使用基础的 `max` 函数来实现 ReLU，而非直接调用 `relu` 函数。

```
def relu(X):  
    return torch.max(input=X, other=torch.tensor(0.0))
```

3.9.4 定义模型

同 softmax 回归一样，我们通过 `view` 函数将每张原始图像改成长度为 `num_inputs` 的向量。然后我们实现上一节中多层感知机的计算表达式。

```
def net(X):  
    X = X.view((-1, num_inputs))  
    H = relu(torch.matmul(X, W1) + b1)  
    return torch.matmul(H, W2) + b2
```

3.9.5 定义损失函数

为了得到更好的数值稳定性，我们直接使用 PyTorch 提供的包括 softmax 运算和交叉熵损失计算的函数。

```
loss = torch.nn.CrossEntropyLoss()
```

3.9.6 训练模型

训练多层感知机的步骤和 3.6 节中训练 softmax 回归的步骤没什么区别。我们直接调用 `d2lzh_pytorch` 包中的 `train_ch3` 函数，它的实现已经在 3.6 节里介绍过。我们在这里设超参数迭代周期数为 5，学习率为 100.0。> 注：由于原书的 mxnet 中的 `SoftmaxCrossEntropyLoss` 在反向传播的时候相对于沿 batch 维求和了，而 PyTorch 默认的是求平均，所以用 PyTorch 计算得到的 loss 比 mxnet 小很多（大概是 maxnet 计算得到的 $1/batch_size$ 这个量级），所以反向传播得到的梯度也小很多，所以为了得到差不多的学习效果，我们把学习率调得成原书的约 `batch_size` 倍，原书的学习率为 0.5，这里设置成 100.0。（之所以这么大，应该是因为 `d2lzh_pytorch` 里面的 `sgd` 函数在更新的时候除以了 `batch_size`，其实 PyTorch 在计算 loss 的时候已经除过一次了，`sgd` 这里应该不用除了）

```
num_epochs, lr = 5, 100.0  
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, params, lr)
```

输出：

```
epoch 1, loss 0.0030, train acc 0.714, test acc 0.753
epoch 2, loss 0.0019, train acc 0.821, test acc 0.777
epoch 3, loss 0.0017, train acc 0.842, test acc 0.834
epoch 4, loss 0.0015, train acc 0.857, test acc 0.839
epoch 5, loss 0.0014, train acc 0.865, test acc 0.845
```

小结

- 可以通过手动定义模型及其参数来实现简单的多层感知机。
- 当多层感知机的层数较多时，本节的实现方法会显得较繁琐，例如在定义模型参数的时候。

注：本节除了代码之外与原书基本相同，[原书传送门](#)

3.10 多层感知机的简洁实现

下面我们使用 Gluon 来实现上一节中的多层感知机。首先导入所需的包或模块。

```
import torch
from torch import nn
from torch.nn import init
import numpy as np
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
```

3.10.1 定义模型

和 softmax 回归唯一的不同在于，我们多加了一个全连接层作为隐藏层。它的隐藏单元个数为 256，并使用 ReLU 函数作为激活函数。

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256
```

```
net = nn.Sequential(
    d2l.FlattenLayer(),
    nn.Linear(num_inputs, num_hiddens),
    nn.ReLU(),
    nn.Linear(num_hiddens, num_outputs),
)

for params in net.parameters():
    init.normal_(params, mean=0, std=0.01)
```

3.10.2 读取数据并训练模型

我们使用与 3.7 节中训练 softmax 回归几乎相同的步骤来读取数据并训练模型。>注：由于这里使用的是 PyTorch 的 SGD 而不是 d2lzh_pytorch 里面的 sgd，所以就不存在 3.9 节那样学习率看起来很大的问题了。

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
loss = torch.nn.CrossEntropyLoss()
```

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.5)
```

```
num_epochs = 5
```

```
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None, None, o
```

输出：

```
epoch 1, loss 0.0030, train acc 0.712, test acc 0.744
epoch 2, loss 0.0019, train acc 0.823, test acc 0.821
epoch 3, loss 0.0017, train acc 0.844, test acc 0.842
epoch 4, loss 0.0015, train acc 0.856, test acc 0.842
epoch 5, loss 0.0014, train acc 0.864, test acc 0.818
```

小结

- 通过 PyTorch 可以更简洁地实现多层感知机。

注：本节除了代码之外与原书基本相同，[原书传送门](#)

3.11 模型选择、欠拟合和过拟合

在前几节基于 Fashion-MNIST 数据集的实验中，我们评价了机器学习模型在训练数据集和测试数据集上的表现。如果你改变过实验中的模型结构或者超参数，你也许发现了：当模型在训练数据集上更准确时，它在测试数据集上却不一定更准确。这是为什么呢？

3.11.1 训练误差和泛化误差

在解释上述现象之前，我们需要区分训练误差（training error）和泛化误差（generalization error）。通俗来讲，前者指模型在训练数据集上表现出的误差，后者指模型在任意一个测试数据样本上表现出的误差的期望，并常常通过测试数据集上的误差来近似。计算训练误差和泛化误差可以使用之前介绍过的损失函数，例如线性回归用到的平方损失函数和 softmax 回归用到的交叉熵损失函数。

让我们以高考为例来直观地解释训练误差和泛化误差这两个概念。训练误差可以认为是做往年高考试题（训练题）时的错误率，泛化误差则可以通过真正参加高考（测试题）时的答题错误率来近似。假设训练题和测试题都随机采样于一个未知的依照相同考纲的巨大试题库。如果让一名未学习中学知识的小学生去答题，那么测试题和训练题的答题错误率可能很相近。但如果换成一名反复练习训练题的高三备考生答题，即使在训练题上做到了错误率为 0，也不代表真实的高考成绩会如此。

在机器学习里，我们通常假设训练数据集（训练题）和测试数据集（测试题）里的每一个样本都是从同一个概率分布中相互独立地生成的。基于该独立同分布假设，给定任意一个机器学习模型（含参数），它的训练误差的期望和泛化误差都是一样的。例如，如果我们将模型参数设成随机值（小学生），那么训练误差和泛化误差会非常相近。但我们在前面几节中已经了解到，模型的参数是通过在训练数据集上训练模型而学习出的，参数的选择依据了最小化训练误差（高三备考生）。所以，训练误差的期望小于或等于泛化误差。也就是说，一般情况下，由训练数据集学到的模型参数会使模型在训练数据集上的表现优于或等于在测试数据集上的表现。由于无法从训练误差估计泛化误差，一味地降低训练误差并不意味着泛化误差一定会降低。

机器学习模型应关注降低泛化误差。

3.11.2 模型选择

在机器学习中，通常需要评估若干候选模型的表现并从中选择模型。这一过程称为模型选择（model selection）。可供选择的候选模型可以是有着不同超参数的同类模型。以多层感知机为例，我们可以选择隐藏层的个数，以及每个隐藏层中隐藏单元个数和激

活函数。为了得到有效的模型，我们通常要在模型选择上下一番功夫。下面，我们来描述模型选择中经常使用的验证数据集（validation data set）。

3.11.2.1 验证数据集

从严格意义上讲，测试集只能在所有超参数和模型参数选定后使用一次。不可以使用测试数据选择模型，如调参。由于无法从训练误差估计泛化误差，因此也不应只依赖训练数据选择模型。鉴于此，我们可以预留一部分在训练数据集和测试数据集以外的数据来进行模型选择。这部分数据被称为验证数据集，简称验证集（validation set）。例如，我们可以从给定的训练集中随机选取一小部分作为验证集，而将剩余部分作为真正的训练集。

然而在实际应用中，由于数据不容易获取，测试数据极少只使用一次就丢弃。因此，实践中验证数据集和测试数据集的界限可能比较模糊。从严格意义上讲，除非明确说明，否则本书中实验所使用的测试集应为验证集，实验报告的测试结果（如测试准确率）应为验证结果（如验证准确率）。

3.11.2.3 K 折交叉验证

由于验证数据集不参与模型训练，当训练数据不够用时，预留大量的验证数据显得太奢侈。一种改善的方法是 K 折交叉验证（ K -fold cross-validation）。在 K 折交叉验证中，我们把原始训练数据集分割成 K 个不重合的子数据集，然后我们做 K 次模型训练和验证。每一次，我们使用一个子数据集验证模型，并使用其他 $K - 1$ 个子数据集来训练模型。在这 K 次训练和验证中，每次用来验证模型的子数据集都不同。最后，我们对这 K 次训练误差和验证误差分别求平均。

3.11.3 欠拟合和过拟合

接下来，我们将探究模型训练中经常出现的两类典型问题：一类是模型无法得到较低的训练误差，我们将这一现象称作欠拟合（underfitting）；另一类是模型的训练误差远小于它在测试数据集上的误差，我们称该现象为过拟合（overfitting）。在实践中，我们要尽可能同时应对欠拟合和过拟合。虽然有很多因素可能导致这两种拟合问题，在这里我们重点讨论两个因素：模型复杂度和训练数据集大小。

关于模型复杂度和训练集大小对学习的影响的详细理论分析可参见我写的[这篇博客](#)。

3.11.3.1 模型复杂度

为了解释模型复杂度，我们以多项式函数拟合为例。给定一个由标量数据特征 x 和对应的标量标签 y 组成的训练数据集，多项式函数拟合的目标是找一个 K 阶多项式函数

$$\hat{y} = b + \sum_{k=1}^K x^k w_k$$

来近似 y 。在上式中， w_k 是模型的权重参数， b 是偏差参数。与线性回归相同，多项式函数拟合也使用平方损失函数。特别地，一阶多项式函数拟合又叫线性函数拟合。

因为高阶多项式函数模型参数更多，模型函数的选择空间更大，所以高阶多项式函数比低阶多项式函数的复杂度更高。因此，高阶多项式函数比低阶多项式函数更容易在相同的训练数据集上得到更低的训练误差。给定训练数据集，模型复杂度和误差之间的关系通常如图 3.4 所示。给定训练数据集，如果模型的复杂度过低，很容易出现欠拟合；如果模型复杂度过高，很容易出现过拟合。应对欠拟合和过拟合的一个办法是针对数据集选择合适复杂度的模型。

图 3.4 模型复杂度对欠拟合和过拟合的影响

3.11.3.2 训练数据集大小

影响欠拟合和过拟合的另一个重要因素是训练数据集的大小。一般来说，如果训练数据集中样本数过少，特别是比模型参数数量（按元素计）更少时，过拟合更容易发生。此外，泛化误差不会随训练数据集里样本数量增加而增大。因此，在计算资源允许的范围之内，我们通常希望训练数据集大一些，特别是在模型复杂度较高时，例如层数较多的深度学习模型。

3.11.4 多项式函数拟合实验

为了理解模型复杂度和训练数据集大小对欠拟合和过拟合的影响，下面我们以多项式函数拟合为例来实验。首先导入实验需要的包或模块。

```
%matplotlib inline
import torch
import numpy as np
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
```

3.11.4.1 生成数据集

我们将生成一个人工数据集。在训练数据集和测试数据集中，给定样本特征 x ，我们使用如下的三阶多项式函数来生成该样本的标签：

$$y = 1.2x - 3.4x^2 + 5.6x^3 + 5 + \epsilon,$$

其中噪声项 ϵ 服从均值为 0、标准差为 0.01 的正态分布。训练数据集和测试数据集的样本数都设为 100。

```
n_train, n_test, true_w, true_b = 100, 100, [1.2, -3.4, 5.6], 5
features = torch.randn((n_train + n_test, 1))
poly_features = torch.cat((features, torch.pow(features, 2), torch.pow(features, 3)),
labels = (true_w[0] * poly_features[:, 0] + true_w[1] * poly_features[:, 1]
        + true_w[2] * poly_features[:, 2] + true_b)
labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size()), dtype=torch.fl
```

看一看生成的数据集的前两个样本。

```
features[:2], poly_features[:2], labels[:2]
```

输出：

```
(tensor([-1.0613, -0.8386]), tensor([-1.0613, 1.1264, -1.1954],
[-0.8386, 0.7032, -0.5897]), tensor([-6.8037, -1.7054]))
```

3.11.4.2 定义、训练和测试模型

我们先定义作图函数 `semilogy`，其中 y 轴使用了对数尺度。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def semilogy(x_vals, y_vals, x_label, y_label, x2_vals=None, y2_vals=None,
            legend=None, figsize=(3.5, 2.5)):
    d2l.set_figsize(figsize)
    d2l.plt.xlabel(x_label)
    d2l.plt.ylabel(y_label)
    d2l.plt.semilogy(x_vals, y_vals)
    if x2_vals and y2_vals:
        d2l.plt.semilogy(x2_vals, y2_vals, linestyle=':')
    d2l.plt.legend(legend)
```

和线性回归一样，多项式函数拟合也使用平方损失函数。因为我们将尝试使用不同复杂度的模型来拟合生成的数据集，所以我们把模型定义部分放在 `fit_and_plot` 函数中。多项式函数拟合的训练和测试步骤与 3.6 节（softmax 回归的从零开始实现）介绍的 softmax 回归中的相关步骤类似。

```
num_epochs, loss = 100, torch.nn.MSELoss()

def fit_and_plot(train_features, test_features, train_labels, test_labels):
    net = torch.nn.Linear(train_features.shape[-1], 1)
    # 通过 Linear 文档可知，pytorch 已经将参数初始化了，所以我们这里就不手动初始化了

    batch_size = min(10, train_labels.shape[0])
    dataset = torch.utils.data.TensorDataset(train_features, train_labels)
    train_iter = torch.utils.data.DataLoader(dataset, batch_size, shuffle=True)

    optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
    train_ls, test_ls = [], []
    for _ in range(num_epochs):
        for X, y in train_iter:
            l = loss(net(X), y.view(-1, 1))
            optimizer.zero_grad()
            l.backward()
            optimizer.step()
        train_labels = train_labels.view(-1, 1)
        test_labels = test_labels.view(-1, 1)
        train_ls.append(loss(net(train_features), train_labels).item())
        test_ls.append(loss(net(test_features), test_labels).item())
    print('final epoch: train loss', train_ls[-1], 'test loss', test_ls[-1])
    semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
              range(1, num_epochs + 1), test_ls, ['train', 'test'])
    print('weight:', net.weight.data,
          '\nbias:', net.bias.data)
```

3.11.4.3 三阶多项式函数拟合（正常）

我们先使用与数据生成函数同阶的三阶多项式函数拟合。实验表明，这个模型的训练误差和在测试数据集的误差都较低。训练出的模型参数也接近真实值： $w_1 = 1.2, w_2 =$

$-3.4, w_3 = 5.6, b = 5$ 。

```
fit_and_plot(poly_features[:n_train, :], poly_features[n_train:, :],
             labels[:n_train], labels[n_train:])
```

输出：

```
final epoch: train loss 0.00010175639908993617 test loss 9.790256444830447e-05
weight: tensor([[ 1.1982, -3.3992,  5.6002]])
bias: tensor([5.0014])
```

3.11.4.4 线性函数拟合（欠拟合）

我们再试试线性函数拟合。很明显，该模型的训练误差在迭代早期下降后便很难继续降低。在完成最后一次迭代周期后，训练误差依旧很高。线性模型在非线性模型（如三阶多项式函数）生成的数据集上容易欠拟合。

```
fit_and_plot(features[:n_train, :], features[n_train:, :], labels[:n_train],
             labels[n_train:])
```

输出：

```
final epoch: train loss 249.35157775878906 test loss 168.37705993652344
weight: tensor([[19.4123]])
bias: tensor([0.5805])
```

3.11.4.5 训练样本不足（过拟合）

事实上，即便使用与数据生成模型同阶的三阶多项式函数模型，如果训练样本不足，该模型依然容易过拟合。让我们只使用两个样本来训练模型。显然，训练样本过少了，甚至少于模型参数的数量。这使模型显得过于复杂，以至于容易被训练数据中的噪声影响。在迭代过程中，尽管训练误差较低，但是测试数据集上的误差却很高。这是典型的过拟合现象。

```
fit_and_plot(poly_features[0:2, :], poly_features[n_train:, :], labels[0:2],
             labels[n_train:])
```

输出：

```
final epoch: train loss 1.198514699935913 test loss 166.037109375
weight: tensor([[1.4741, 2.1198, 2.5674]])
bias: tensor([3.1207])
```

我们将在接下来的两个小节继续讨论过拟合问题以及应对过拟合的方法。

小结

- 由于无法从训练误差估计泛化误差，一味地降低训练误差并不意味着泛化误差一定会降低。机器学习模型应关注降低泛化误差。
 - 可以使用验证数据集来进行模型选择。
 - 欠拟合指模型无法得到较低的训练误差，过拟合指模型的训练误差远小于它在测试数据集上的误差。
 - 应选择复杂度合适的模型并避免使用过少的训练样本。
-

注：本节除了代码之外与原书基本相同，[原书传送门](#)

3.12 权重衰减

上一节中我们观察了过拟合现象，即模型的训练误差远小于它在测试集上的误差。虽然增大训练数据集可能会减轻过拟合，但是获取额外的训练数据往往代价高昂。本节介绍应对过拟合问题的常用方法：权重衰减（weight decay）。

3.12.1 方法

权重衰减等价于 L_2 范数正则化 (regularization)。正则化通过为模型损失函数添加惩罚项使学出的模型参数值较小，是应对过拟合的常用手段。我们先描述 L_2 范数正则化，再解释它为何又称权重衰减。

L_2 范数正则化在模型原损失函数基础上添加 L_2 范数惩罚项，从而得到训练所需要最小化的函数。 L_2 范数惩罚项指的是模型权重参数每个元素的平方和与一个正的常数的乘积。以 3.1 节（线性回归）中的线性回归损失函数

$$\ell(w_1, w_2, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} \left(x_1^{(i)} w_1 + x_2^{(i)} w_2 + b - y^{(i)} \right)^2$$

为例，其中 w_1, w_2 是权重参数， b 是偏差参数，样本 i 的输入为 $x_1^{(i)}, x_2^{(i)}$ ，标签为 $y^{(i)}$ ，样本数为 n 。将权重参数用向量 $\mathbf{w} = [w_1, w_2]$ 表示，带有 L_2 范数惩罚项的新损失函数为

$$\ell(w_1, w_2, b) + \frac{\lambda}{2n} \|\mathbf{w}\|^2,$$

其中超参数 $\lambda > 0$ 。当权重参数均为 0 时，惩罚项最小。当 λ 较大时，惩罚项在损失函数中的比重较大，这通常会使学到的权重参数的元素较接近 0。当 λ 设为 0 时，惩罚项完全不起作用。上式中 L_2 范数平方 $\|\mathbf{w}\|^2$ 展开后得到 $w_1^2 + w_2^2$ 。有了 L_2 范数惩罚项后，在小批量随机梯度下降中，我们将线性回归一节中权重 w_1 和 w_2 的迭代方式更改为

$$\begin{aligned} w_1 &\leftarrow \left(1 - \frac{\eta\lambda}{|\mathcal{B}|}\right) w_1 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} x_1^{(i)} \left(x_1^{(i)} w_1 + x_2^{(i)} w_2 + b - y^{(i)} \right), \\ w_2 &\leftarrow \left(1 - \frac{\eta\lambda}{|\mathcal{B}|}\right) w_2 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} x_2^{(i)} \left(x_1^{(i)} w_1 + x_2^{(i)} w_2 + b - y^{(i)} \right). \end{aligned}$$

可见， L_2 范数正则化令权重 w_1 和 w_2 先自乘小于 1 的数，再减去不含惩罚项的梯度。因此， L_2 范数正则化又叫权重衰减。权重衰减通过惩罚绝对值较大的模型参数为需要学习的模型增加了限制，这可能对过拟合有效。实际场景中，我们有时也在惩罚项中添加偏差元素的平方和。

3.12.2 高维线性回归实验

下面，我们以高维线性回归为例来引入一个过拟合问题，并使用权重衰减来应对过拟合。设数据样本特征的维度为 p 。对于训练数据集和测试数据集中特征为 x_1, x_2, \dots, x_p 的任一样本，我们使用如下的线性函数来生成该样本的标签：

$$y = 0.05 + \sum_{i=1}^p 0.01x_i + \epsilon$$

其中噪声项 ϵ 服从均值为 0、标准差为 0.01 的正态分布。为了较容易地观察过拟合，我们考虑高维线性回归问题，如设维度 $p = 200$ ；同时，我们特意把训练数据集的样本数设低，如 20。

```
%matplotlib inline
import torch
import torch.nn as nn
import numpy as np
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l

n_train, n_test, num_inputs = 20, 100, 200
true_w, true_b = torch.ones(num_inputs, 1) * 0.01, 0.05

features = torch.randn((n_train + n_test, num_inputs))
labels = torch.matmul(features, true_w) + true_b
labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size()), dtype=torch.float)
train_features, test_features = features[:n_train, :], features[n_train:, :]
train_labels, test_labels = labels[:n_train], labels[n_train:]
```

3.12.3 从零开始实现

下面先介绍从零开始实现权重衰减的方法。我们通过在目标函数后添加 L_2 范数惩罚项来实现权重衰减。

3.12.3.1 初始化模型参数

首先，定义随机初始化模型参数的函数。该函数为每个参数都附上梯度。

```
def init_params():
    w = torch.randn((num_inputs, 1), requires_grad=True)
    b = torch.zeros(1, requires_grad=True)
    return [w, b]
```

3.12.3.2 定义 L_2 范数惩罚项

下面定义 L_2 范数惩罚项。这里只惩罚模型的权重参数。

```
def l2_penalty(w):
    return (w**2).sum() / 2
```

3.12.3.3 定义训练和测试

下面定义如何在训练数据集和测试数据集上分别训练和测试模型。与前面几节中不同的是，这里在计算最终的损失函数时添加了 L_2 范数惩罚项。

```
batch_size, num_epochs, lr = 1, 100, 0.003
net, loss = d2l.linreg, d2l.squared_loss

dataset = torch.utils.data.TensorDataset(train_features, train_labels)
train_iter = torch.utils.data.DataLoader(dataset, batch_size, shuffle=True)

def fit_and_plot(lambd):
    w, b = init_params()
    train_ls, test_ls = [], []
    for _ in range(num_epochs):
        for X, y in train_iter:
            # 添加了 L2 范数惩罚项
            l = loss(net(X, w, b), y) + lambd * l2_penalty(w)
            l = l.sum()

            if w.grad is not None:
                w.grad.data.zero_()
                b.grad.data.zero_()
            l.backward()
            d2l.sgd([w, b], lr, batch_size)
```

```
train_ls.append(loss(net(train_features, w, b), train_labels).mean().item())
test_ls.append(loss(net(test_features, w, b), test_labels).mean().item())
d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
              range(1, num_epochs + 1), test_ls, ['train', 'test'])
print('L2 norm of w:', w.norm().item())
```

3.12.3.4 观察过拟合

接下来，让我们训练并测试高维线性回归模型。当 `lambda` 设为 0 时，我们没有使用权重衰减。结果训练误差远小于测试集上的误差。这是典型的过拟合现象。

```
fit_and_plot(lambda=0)
```

输出：

```
L2 norm of w: 15.114808082580566
```

3.12.3.5 使用权重衰减

下面我们使用权重衰减。可以看出，训练误差虽然有所提高，但测试集上的误差有所下降。过拟合现象得到一定程度的缓解。另外，权重参数的 L_2 范数比不使用权重衰减时的更小，此时的权重参数更接近 0。

```
fit_and_plot(lambda=3)
```

输出：

```
L2 norm of w: 0.035220853984355927
```

3.12.4 简洁实现

这里我们直接在构造优化器实例时通过 `weight_decay` 参数来指定权重衰减超参数。默认下，PyTorch 会对权重和偏差同时衰减。我们可以分别对权重和偏差构造优化器实例，从而只对权重衰减。

```
def fit_and_plot_pytorch(wd):
    # 对权重参数衰减。权重名称一般是以 weight 结尾
    net = nn.Linear(num_inputs, 1)
    nn.init.normal_(net.weight, mean=0, std=1)
    nn.init.normal_(net.bias, mean=0, std=1)
```

```
optimizer_w = torch.optim.SGD(params=[net.weight], lr=lr, weight_decay=wd) # 对权重衰减
optimizer_b = torch.optim.SGD(params=[net.bias], lr=lr) # 不对偏差参数衰减

train_ls, test_ls = [], []
for _ in range(num_epochs):
    for X, y in train_iter:
        l = loss(net(X), y).mean()
        optimizer_w.zero_grad()
        optimizer_b.zero_grad()

        l.backward()

    # 对两个 optimizer 实例分别调用 step 函数，从而分别更新权重和偏差
    optimizer_w.step()
    optimizer_b.step()
    train_ls.append(loss(net(train_features), train_labels).mean().item())
    test_ls.append(loss(net(test_features), test_labels).mean().item())
d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
              range(1, num_epochs + 1), test_ls, ['train', 'test'])
print('L2 norm of w:', net.weight.data.norm().item())
```

与从零开始实现权重衰减的实验现象类似，使用权重衰减可以在一定程度上缓解过拟合问题。

```
fit_and_plot_pytorch(0)
```

输出：

```
L2 norm of w: 12.86785888671875
```

```
fit_and_plot_pytorch(3)
```

输出：

```
L2 norm of w: 0.09631537646055222
```

小结

- 正则化通过为模型损失函数添加惩罚项使学出的模型参数值较小，是应对过拟合的常用手段。
 - 权重衰减等价于 L_2 范数正则化，通常会使学到的权重参数的元素较接近 0。
 - 权重衰减可以通过优化器中的 `weight_decay` 超参数来指定。
 - 可以定义多个优化器实例对不同的模型参数使用不同的迭代方法。
-

注：本节除了代码之外与原书基本相同，[原书传送门](#)

3.13 丢弃法

除了前一节介绍的权重衰减以外，深度学习模型常常使用丢弃法（dropout）[1]来应对过拟合问题。丢弃法有一些不同的变体。本节中提到的丢弃法特指倒置丢弃法（inverted dropout）。

3.13.1 方法

回忆一下，3.8 节（多层感知机）的图 3.3 描述了一个单隐藏层的多层感知机。其中输入个数为 4，隐藏单元个数为 5，且隐藏单元 h_i ($i = 1, \dots, 5$) 的计算表达式为

$$h_i = \phi(x_1 w_{1i} + x_2 w_{2i} + x_3 w_{3i} + x_4 w_{4i} + b_i)$$

这里 ϕ 是激活函数， x_1, \dots, x_4 是输入，隐藏单元 i 的权重参数为 w_{1i}, \dots, w_{4i} ，偏差参数为 b_i 。当对该隐藏层使用丢弃法时，该层的隐藏单元将有一定概率被丢弃掉。设丢弃概率为 p ，那么有 p 的概率 h_i 会被清零，有 $1 - p$ 的概率 h_i 会除以 $1 - p$ 做拉伸。丢弃概率是丢弃法的超参数。具体来说，设随机变量 ξ_i 为 0 和 1 的概率分别为 p 和 $1 - p$ 。使用丢弃法时我们计算新的隐藏单元 h'_i

$$h'_i = \frac{\xi_i}{1 - p} h_i$$

由于 $E(\xi_i) = 1 - p$ ，因此

$$E(h'_i) = \frac{E(\xi_i)}{1 - p} h_i = h_i$$

即丢弃法不改变其输入的期望值。让我们对图 3.3 中的隐藏层使用丢弃法，一种可能的结果如图 3.5 所示，其中 h_2 和 h_5 被清零。这时输出值的计算不再依赖 h_2 和 h_5 ，在反向传播时，与这两个隐藏单元相关的权重的梯度均为 0。由于在训练中隐藏层神经元的丢弃是随机的，即 h_1, \dots, h_5 都有可能被清零，输出层的计算无法过度依赖 h_1, \dots, h_5 中的任一个，从而在训练模型时起到正则化的作用，并可以用来应对过拟合。在测试模型时，我们为了拿到更加确定性的结果，一般不使用丢弃法。

图 3.5 隐藏层使用了丢弃法的多层感知机

3.13.2 从零开始实现

根据丢弃法的定义，我们可以很容易地实现它。下面的 `dropout` 函数将以 `drop_prob` 的概率丢弃 `x` 中的元素。

```
%matplotlib inline
import torch
import torch.nn as nn
import numpy as np
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l

def dropout(X, drop_prob):
    X = X.float()
    assert 0 <= drop_prob <= 1
    keep_prob = 1 - drop_prob
    # 这种情况下把全部元素都丢弃
    if keep_prob == 0:
        return torch.zeros_like(X)
    mask = (torch.randn(X.shape) < keep_prob).float()

    return mask * X / keep_prob
```

我们运行几个例子来测试一下 `dropout` 函数。其中丢弃概率分别为 0、0.5 和 1。

```
X = torch.arange(16).view(2, 8)
dropout(X, 0)

dropout(X, 0.5)

dropout(X, 1.0)
```

3.13.2.1 定义模型参数

实验中，我们依然使用 3.6 节（softmax 回归的从零开始实现）中介绍的 Fashion-MNIST 数据集。我们将定义一个包含两个隐藏层的多层感知机，其中两个隐藏层的输出个数都是 256。

```
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256
```

```
W1 = torch.tensor(np.random.normal(0, 0.01, size=(num_inputs, num_hiddens1)), dtype=t
b1 = torch.zeros(num_hiddens1, requires_grad=True)
```

```

W2 = torch.tensor(np.random.normal(0, 0.01, size=(num_hiddens1, num_hiddens2)), dtype=
b2 = torch.zeros(num_hiddens2, requires_grad=True)
W3 = torch.tensor(np.random.normal(0, 0.01, size=(num_hiddens2, num_outputs)), dtype=
b3 = torch.zeros(num_outputs, requires_grad=True)

params = [W1, b1, W2, b2, W3, b3]

```

3.13.2.2 定义模型

下面定义的模型将全连接层和激活函数 ReLU 串起来，并对每个激活函数的输出使用丢弃法。我们可以分别设置各个层的丢弃概率。通常的建议是把靠近输入层的丢弃概率设得小一点。在这个实验中，我们把第一个隐藏层的丢弃概率设为 0.2，把第二个隐藏层的丢弃概率设为 0.5。我们可以通过参数 `is_training` 函数来判断运行模式为训练还是测试，并只需在训练模式下使用丢弃法。

```
drop_prob1, drop_prob2 = 0.2, 0.5
```

```

def net(X, is_training=True):
    X = X.view(-1, num_inputs)
    H1 = (torch.matmul(X, W1) + b1).relu()
    if is_training: # 只在训练模型时使用丢弃法
        H1 = dropout(H1, drop_prob1) # 在第一层全连接后添加丢弃层
    H2 = (torch.matmul(H1, W2) + b2).relu()
    if is_training:
        H2 = dropout(H2, drop_prob2) # 在第二层全连接后添加丢弃层
    return torch.matmul(H2, W3) + b3

```

我们在对模型评估的时候不应该进行丢弃，所以我们修改一下 `d2lzh_pytorch` 中的 `evaluate_accuracy` 函数：

```

# 本函数已保存在 d2lzh_pytorch
def evaluate_accuracy(data_iter, net):
    acc_sum, n = 0.0, 0
    for X, y in data_iter:
        if isinstance(net, torch.nn.Module):
            net.eval() # 评估模式，这会关闭 dropout
        acc_sum += (net(X).argmax(dim=1) == y).float().sum().item()

```

```

    net.train() # 改回训练模式
else: # 自定义的模型
    if('is_training' in net.__code__.co_varnames): # 如果有 is_training 这个参数
        # 将 is_training 设置成 False
        acc_sum += (net(X, is_training=False).argmax(dim=1) == y).float().sum()
    else:
        acc_sum += (net(X).argmax(dim=1) == y).float().sum().item()
n += y.shape[0]
return acc_sum / n

```

注：将上诉 evaluate_accuracy 写回 d2lzh_pytorch 后要重启一下 jupyter kernel 才会生效。

3.13.2.3 训练和测试模型

这部分与之前多层感知机的训练和测试类似。

```

num_epochs, lr, batch_size = 5, 100.0, 256
loss = torch.nn.CrossEntropyLoss()
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, params, lr)

```

输出：

```

epoch 1, loss 0.0044, train acc 0.574, test acc 0.648
epoch 2, loss 0.0023, train acc 0.786, test acc 0.786
epoch 3, loss 0.0019, train acc 0.826, test acc 0.825
epoch 4, loss 0.0017, train acc 0.839, test acc 0.831
epoch 5, loss 0.0016, train acc 0.849, test acc 0.850

```

注：这里的学习率设置的很大，原因同 3.9.6 节。

3.13.3 简洁实现

在 PyTorch 中，我们只需要在全连接层后添加 `Dropout` 层并指定丢弃概率。在训练模型时，`Dropout` 层将以指定的丢弃概率随机丢弃上一层的输出元素；在测试模型时（即 `model.eval()` 后），`Dropout` 层并不发挥作用。

```
net = nn.Sequential(  
    d2l.FlattenLayer(),  
    nn.Linear(num_inputs, num_hiddens1),  
    nn.ReLU(),  
    nn.Dropout(drop_prob1),  
    nn.Linear(num_hiddens1, num_hiddens2),  
    nn.ReLU(),  
    nn.Dropout(drop_prob2),  
    nn.Linear(num_hiddens2, 10)  
)
```

```
for param in net.parameters():  
    nn.init.normal_(param, mean=0, std=0.01)
```

下面训练并测试模型。

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.5)
```

```
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None, None, o
```

输出：

```
epoch 1, loss 0.0045, train acc 0.553, test acc 0.715  
epoch 2, loss 0.0023, train acc 0.784, test acc 0.793  
epoch 3, loss 0.0019, train acc 0.822, test acc 0.817  
epoch 4, loss 0.0018, train acc 0.837, test acc 0.830  
epoch 5, loss 0.0016, train acc 0.848, test acc 0.839
```

注：由于这里使用的是 PyTorch 的 SGD 而不是 d2lzh_pytorch 里面的 sgd，所以就不存在 3.9.6 节那样学习率看起很大的问题了。

小结

- 我们可以通过使用丢弃法应对过拟合。
- 丢弃法只在训练模型时使用。

参考文献

- [1] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. JMLR

注：本节除了代码之外与原书基本相同，[原书传送门](#)

3.14 正向传播、反向传播和计算图

前面几节里我们使用了小批量随机梯度下降的优化算法来训练模型。在实现中，我们只提供了模型的正向传播 (forward propagation) 的计算，即对输入计算模型输出，然后通过 `autograd` 模块来调用系统自动生成的 `backward` 函数计算梯度。基于反向传播 (back-propagation) 算法的自动求梯度极大简化了深度学习模型训练算法的实现。本节我们将使用数学和计算图 (computational graph) 两个方式来描述正向传播和反向传播。具体来说，我们将以带 L_2 范数正则化的含单隐藏层的多层感知机为样例模型解释正向传播和反向传播。

3.14.1 正向传播

正向传播是指对神经网络沿着从输入层到输出层的顺序，依次计算并存储模型的中间变量（包括输出）。为简单起见，假设输入是一个特征为 $\mathbf{x} \in \mathbb{R}^d$ 的样本，且不考虑偏差项，那么中间变量

$$\mathbf{z} = \mathbf{W}^{(1)} \mathbf{x},$$

其中 $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ 是隐藏层的权重参数。把中间变量 $\mathbf{z} \in \mathbb{R}^h$ 输入按元素运算的激活函数 ϕ 后，将得到向量长度为 h 的隐藏层变量

$$\mathbf{h} = \phi(\mathbf{z}).$$

隐藏层变量 \mathbf{h} 也是一个中间变量。假设输出层参数只有权重 $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ ，可以得到向量长度为 q 的输出层变量

$$\mathbf{o} = \mathbf{W}^{(2)} \mathbf{h}.$$

假设损失函数为 ℓ ，且样本标签为 y ，可以计算出单个数据样本的损失项

$$L = \ell(\mathbf{o}, y).$$

根据 L_2 范数正则化的定义，给定超参数 λ ，正则化项即

$$s = \frac{\lambda}{2} \left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right),$$

其中矩阵的 Frobenius 范数等价于将矩阵变平为向量后计算 L_2 范数。最终，模型在给定的数据样本上带正则化的损失为

$$J = L + s.$$

我们将 J 称为有关给定数据样本的目标函数，并在以下的讨论中简称目标函数。

3.14.2 正向传播的计算图

我们通常绘制计算图来可视化运算符和变量在计算中的依赖关系。图 3.6 绘制了本节中样例模型正向传播的计算图，其中左下角是输入，右上角是输出。可以看到，图中箭头方向大多是向右和向上，其中方框代表变量，圆圈代表运算符，箭头表示从输入到输出之间的依赖关系。

图 3.6 正向传播的计算图

3.14.3 反向传播

反向传播指的是计算神经网络参数梯度的方法。总的来说，反向传播依据微积分中的链式法则，沿着从输出层到输入层的顺序，依次计算并存储目标函数有关神经网络各层的中间变量以及参数的梯度。对输入或输出 X, Y, Z 为任意形状张量的函数 $Y = f(X)$ 和 $Z = g(Y)$ ，通过链式法则，我们有

$$\frac{\partial Z}{\partial X} = \text{prod} \left(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X} \right),$$

其中 prod 运算符将根据两个输入的形状，在必要的操作（如转置和互换输入位置）后对两个输入做乘法。

回顾一下本节中样例模型，它的参数是 $\mathbf{W}^{(1)}$ 和 $\mathbf{W}^{(2)}$ ，因此反向传播的目标是计算 $\partial J / \partial \mathbf{W}^{(1)}$ 和 $\partial J / \partial \mathbf{W}^{(2)}$ 。我们将应用链式法则依次计算各中间变量和参数的梯度，其计算次序与前向传播中相应中间变量的计算次序恰恰相反。首先，分别计算目标函数 $J = L + s$ 有关损失项 L 和正则项 s 的梯度

$$\frac{\partial J}{\partial L} = 1, \quad \frac{\partial J}{\partial s} = 1.$$

其次，依据链式法则计算目标函数有关输出层变量的梯度 $\partial J / \partial \mathbf{o} \in \mathbb{R}^q$ ：

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod} \left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}} \right) = \frac{\partial L}{\partial \mathbf{o}}.$$

接下来，计算正则项有关两个参数的梯度：

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)}, \quad \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}.$$

现在，我们可以计算最靠近输出层的模型参数的梯度 $\partial J / \partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ 。依据链式法则，得到

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod} \left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}} \right) + \text{prod} \left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}} \right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}.$$

沿着输出层向隐藏层继续反向传播，隐藏层变量的梯度 $\partial J / \partial \mathbf{h} \in \mathbb{R}^h$ 可以这样计算：

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod} \left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}} \right) = \mathbf{W}^{(2)\top} \frac{\partial J}{\partial \mathbf{o}}.$$

由于激活函数 ϕ 是按元素运算的，中间变量 \mathbf{z} 的梯度 $\partial J / \partial \mathbf{z} \in \mathbb{R}^h$ 的计算需要使用按元素乘法符 \odot ：

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod} \left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z}).$$

最终，我们可以得到最靠近输入层的模型参数的梯度 $\partial J / \partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ 。依据链式法则，得到

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod} \left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} \right) + \text{prod} \left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}} \right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}.$$

3.14.4 训练深度学习模型

在训练深度学习模型时，正向传播和反向传播之间相互依赖。下面我们仍然以本节中的样例模型分别阐述它们之间的依赖关系。

一方面，正向传播的计算可能依赖于模型参数的当前值，而这些模型参数是在反向传播的梯度计算后通过优化算法迭代的。例如，计算正则化项 $s = (\lambda/2) (\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2)$ 依赖模型参数 $\mathbf{W}^{(1)}$ 和 $\mathbf{W}^{(2)}$ 的当前值，而这些当前值是优化算法最近一次根据反向传播算出梯度后迭代得到的。

另一方面，反向传播的梯度计算可能依赖于各变量的当前值，而这些变量的当前值是通过正向传播计算得到的。举例来说，参数梯度 $\partial J / \partial \mathbf{W}^{(2)} = (\partial J / \partial \mathbf{o}) \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$ 的计算需要依赖隐藏层变量的当前值 \mathbf{h} 。这个当前值是通过从输入层到输出层的正向传播计算并存储得到的。

因此，在模型参数初始化完成后，我们交替地进行正向传播和反向传播，并根据反向传播计算的梯度迭代模型参数。既然我们在反向传播中使用了正向传播中计算得到的中间变量来避免重复计算，那么这个复用也导致正向传播结束后不能立即释放中间变量内存。这也是训练要比预测占用更多内存的一个重要原因。另外需要指出的是，这些中间变量的个数大体上与网络层数线性相关，每个变量的大小跟批量大小和输入个数也是线性相关的，它们是导致较深的神经网络使用较大批量训练时更容易超内存的主要原因。

小结

- 正向传播沿着从输入层到输出层的顺序，依次计算并存储神经网络的中间变量。
 - 反向传播沿着从输出层到输入层的顺序，依次计算并存储神经网络中间变量和参数的梯度。
 - 在训练深度学习模型时，正向传播和反向传播相互依赖。
-

注：本节与原书基本相同，[原书传送门](#)

3.15 数值稳定性和模型初始化

理解了正向传播与反向传播以后，我们来讨论一下深度学习模型的数值稳定性问题以及模型参数的初始化方法。深度模型有关数值稳定性的典型问题是衰减（vanishing）和爆炸（explosion）。

3.15.1 衰减和爆炸

当神经网络的层数较多时，模型的数值稳定性容易变差。假设一个层数为 L 的多层感知机的第 l 层 $\mathbf{H}^{(l)}$ 的权重参数为 $\mathbf{W}^{(l)}$ ，输出层 $\mathbf{H}^{(L)}$ 的权重参数为 $\mathbf{W}^{(L)}$ 。为了便于讨论，不考虑偏差参数，且设所有隐藏层的激活函数为恒等映射（identity mapping） $\phi(x) = x$ 。给定输入 \mathbf{X} ，多层感知机的第 l 层的输出 $\mathbf{H}^{(l)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} \dots \mathbf{W}^{(l)}$ 。此时，如果层数 l 较大， $\mathbf{H}^{(l)}$ 的计算可能会出现衰减或爆炸。举个例子，假设输入和所有层的权重参数都是标量，如权重参数为 0.2 和 5，多层感知机的第 30 层输出为输入 \mathbf{X} 分别与 $0.2^{30} \approx 1 \times 10^{-21}$ （衰减）和 $5^{30} \approx 9 \times 10^{20}$ （爆炸）的乘积。类似地，当层数较多时，梯度的计算也更容易出现衰减或爆炸。

随着内容的不断深入，我们会在后面的章节进一步介绍深度学习的数值稳定性问题以及解决方法。

3.15.2 随机初始化模型参数

在神经网络中，通常需要随机初始化模型参数。下面我们来解释这样做的原因。

回顾 3.8 节（多层感知机）图 3.3 描述的多层感知机。为了方便解释，假设输出层只保留一个输出单元 o_1 （删去 o_2 和 o_3 以及指向它们的箭头），且隐藏层使用相同的激活函数。如果将每个隐藏单元的参数都初始化为相等的值，那么在正向传播时每个隐藏单元将根据相同的输入计算出相同的值，并传递至输出层。在反向传播中，每个隐藏单元的参数梯度值相等。因此，这些参数在使用基于梯度的优化算法迭代后值依然相等。之后的迭代也是如此。在这种情况下，无论隐藏单元有多少，隐藏层本质上只有 1 个隐藏单元在发挥作用。因此，正如在前面的实验中所做的那样，我们通常将神经网络的模型参数，特别是权重参数，进行随机初始化。

3.15.2.1 PyTorch 的默认随机初始化

随机初始化模型参数的方法有很多。在 3.3 节（线性回归的简洁实现）中，我们使用 `torch.nn.init.normal_()` 使模型 `net` 的权重参数采用正态分布的随机初始化方式。不过，PyTorch 中 `nn.Module` 的模块参数都采取了较为合理的初始化策略（不同类型的 layer 具体采样的哪一种初始化方法的可参考[源代码](#)），因此一般不用我们考虑。

3.15.2.2 Xavier 随机初始化

还有一种比较常用的随机初始化方法叫作 Xavier 随机初始化 [1]。假设某全连接层的输入个数为 a , 输出个数为 b , Xavier 随机初始化将使该层中权重参数的每个元素都随机采样于均匀分布

$$U\left(-\sqrt{\frac{6}{a+b}}, \sqrt{\frac{6}{a+b}}\right).$$

它的设计主要考虑到, 模型参数初始化后, 每层输出的方差不该受该层输入个数影响, 且每层梯度的方差也不该受该层输出个数影响。

小结

- 深度模型有关数值稳定性的典型问题是衰减和爆炸。当神经网络的层数较多时, 模型的数值稳定性容易变差。
- 我们通常需要随机初始化神经网络的模型参数, 如权重参数。

参考文献

- [1] Glorot, X., & Bengio, Y. (2010, March). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics (pp. 249-256).

3.16 实战 Kaggle 比赛：房价预测

作为深度学习基础篇章的总结，我们将对本章内容学以致用。下面，让我们动手实战一个 Kaggle 比赛：房价预测。本节将提供未经调优的数据的预处理、模型的设计和超参数的选择。我们希望读者通过动手操作、仔细观察实验现象、认真分析实验结果并不断调整方法，得到令自己满意的结果。

3.16.1 Kaggle 比赛

Kaggle 是一个著名的供机器学习爱好者交流的平台。图 3.7 展示了 Kaggle 网站的首页。为了便于提交结果，需要注册 Kaggle 账号。

图 3.7 Kaggle 网站首页

我们可以在房价预测比赛的网页上了解比赛信息和参赛者成绩，也可以下载数据集并提交自己的预测结果。该比赛的网页地址是 <https://www.kaggle.com/c/house-prices-advanced-regression-techniques>。

图 3.8 房价预测比赛的网页信息。比赛数据集可通过点击“Data”标签获取

图 3.8 展示了房价预测比赛的网页信息。

3.16.2 获取和读取数据集

比赛数据分为训练数据集和测试数据集。两个数据集都包括每栋房子的特征，如街道类型、建造年份、房顶类型、地下室状况等特征值。这些特征值有连续的数字、离散的标签甚至是缺失值“na”。只有训练数据集包括了每栋房子的价格，也就是标签。我们可以访问比赛网页，点击图 3.8 中的“Data”标签，并下载这些数据集。

我们将通过 `pandas` 库读入并处理数据。在导入本节需要的包前请确保已安装 `pandas` 库，否则请参考下面的代码注释。

```
# 如果没有安装 pandas，则反注释下面一行
# !pip install pandas

%matplotlib inline
import torch
import torch.nn as nn
import numpy as np
import pandas as pd
import sys
sys.path.append("..")
```

```
import d2lzh_pytorch as d2l

print(torch.__version__)
torch.set_default_tensor_type(torch.FloatTensor)
```

假设解压后的数据位于`../data/kaggle_house/`目录，它包括两个 csv 文件。下面使用 `pandas` 读取这两个文件。

```
train_data = pd.read_csv('..../data/kaggle_house/train.csv')
test_data = pd.read_csv('..../data/kaggle_house/test.csv')
```

训练数据集包括 1460 个样本、80 个特征和 1 个标签。

```
train_data.shape # 输出 (1460, 81)
```

测试数据集包括 1459 个样本和 80 个特征。我们需要将测试数据集中每个样本的标签预测出来。

```
test_data.shape # 输出 (1459, 80)
```

让我们来查看前 4 个样本的前 4 个特征、后 2 个特征和标签 (SalePrice)：

```
train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]]
```

可以看到第一个特征是 Id，它能帮助模型记住每个训练样本，但难以推广到测试样本，所以我们不使用它来训练。我们将所有的训练数据和测试数据的 79 个特征按样本连结。

```
all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

3.16.3 预处理数据

我们对连续数值的特征做标准化 (standardization)：设该特征在整个数据集上的均值为 μ ，标准差为 σ 。那么，我们可以将该特征的每个值先减去 μ 再除以 σ 得到标准化后的每个特征值。对于缺失的特征值，我们将其替换成该特征的均值。

```
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# 标准化后，每个特征的均值变为 0，所以可以直接用 0 来替换缺失值
all_features = all_features.fillna(0)
```

接下来将离散数值转成指示特征。举个例子，假设特征 MSZoning 里面有两个不同的离散值 RL 和 RM，那么这一步转换将去掉 MSZoning 特征，并新加两个特征 MSZoning_RL 和 MSZoning_RM，其值为 0 或 1。如果一个样本原来在 MSZoning 里的值为 RL，那么有 MSZoning_RL=1 且 MSZoning_RM=0。

```
# dummy_na=True 将缺失值也当作合法的特征值并为其创建指示特征
all_features = pd.get_dummies(all_features, dummy_na=True)
all_features.shape # (2919, 354)
```

可以看到这一步转换将特征数从 79 增加到了 354。

最后，通过 `values` 属性得到 NumPy 格式的数据，并转成 `NDArray` 方便后面的训练。

```
n_train = train_data.shape[0]
train_features = torch.tensor(all_features[:n_train].values, dtype=torch.float)
test_features = torch.tensor(all_features[n_train:].values, dtype=torch.float)
train_labels = torch.tensor(train_data.SalePrice.values, dtype=torch.float).view(-1,
```

3.16.4 训练模型

我们使用一个基本的线性回归模型和平方损失函数来训练模型。

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

def get_net(feature_num):
    net = nn.Linear(feature_num, 1)
    for param in net.parameters():
        nn.init.normal_(param, mean=0, std=0.01)
    return net
```

下面定义比赛用来评价模型的对数均方根误差。给定预测值 $\hat{y}_1, \dots, \hat{y}_n$ 和对应的真
实标签 y_1, \dots, y_n ，它的定义为

$$\sqrt{\frac{1}{n} \sum_{i=1}^n (\log(y_i) - \log(\hat{y}_i))^2}.$$

对数均方根误差的实现如下。

```
def log_rmse(net, features, labels):
    with torch.no_grad():
        # 将小于 1 的值设成 1，使得取对数时数值更稳定
        clipped_preds = torch.max(net(features), torch.tensor(1.0))
        rmse = torch.sqrt(2 * loss(clipped_preds.log(), labels.log()).mean())
    return rmse.item()
```

下面的训练函数跟本章中前几节的不同在于使用了 Adam 优化算法。相对之前使用的小批量随机梯度下降，它对学习率相对不那么敏感。我们将在之后的“优化算法”一章里详细介绍它。

```
def train(net, train_features, train_labels, test_features, test_labels,
          num_epochs, learning_rate, weight_decay, batch_size):
    train_ls, test_ls = [], []
    dataset = torch.utils.data.TensorDataset(train_features, train_labels)
    train_iter = torch.utils.data.DataLoader(dataset, batch_size, shuffle=True)
    # 这里使用了 Adam 优化算法
    optimizer = torch.optim.Adam(params=net.parameters(), lr=learning_rate, weight_de
    net = net.float()
    for epoch in range(num_epochs):
        for X, y in train_iter:
            l = loss(net(X.float()), y.float())
            optimizer.zero_grad()
            l.backward()
            optimizer.step()
        train_ls.append(log_rmse(net, train_features, train_labels))
        if test_labels is not None:
            test_ls.append(log_rmse(net, test_features, test_labels))
    return train_ls, test_ls
```

3.16.5 K 折交叉验证

我们在 3.11 节（模型选择、欠拟合和过拟合）中介绍了 K 折交叉验证。它将被用来选择模型设计并调节超参数。下面实现了一个函数，它返回第 i 折交叉验证时所需要的训练和验证数据。

```
def get_k_fold_data(k, i, X, y):
    # 返回第 i 折交叉验证时所需要的训练和验证数据
```

```

assert k > 1
fold_size = X.shape[0] // k
X_train, y_train = None, None
for j in range(k):
    idx = slice(j * fold_size, (j + 1) * fold_size)
    X_part, y_part = X[idx, :], y[idx]
    if j == i:
        X_valid, y_valid = X_part, y_part
    elif X_train is None:
        X_train, y_train = X_part, y_part
    else:
        X_train = torch.cat((X_train, X_part), dim=0)
        y_train = torch.cat((y_train, y_part), dim=0)
return X_train, y_train, X_valid, y_valid

```

在 K 折交叉验证中我们训练 K 次并返回训练和验证的平均误差。

```

def k_fold(k, X_train, y_train, num_epochs,
           learning_rate, weight_decay, batch_size):
    train_l_sum, valid_l_sum = 0, 0
    for i in range(k):
        data = get_k_fold_data(k, i, X_train, y_train)
        net = get_net(X_train.shape[1])
        train_ls, valid_ls = train(net, *data, num_epochs, learning_rate,
                                   weight_decay, batch_size)
        train_l_sum += train_ls[-1]
        valid_l_sum += valid_ls[-1]
        if i == 0:
            d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'rmse',
                          range(1, num_epochs + 1), valid_ls,
                          ['train', 'valid'])
        print('fold %d, train rmse %f, valid rmse %f' % (i, train_ls[-1], valid_ls[-1]))
    return train_l_sum / k, valid_l_sum / k

```

输出：

```
fold 0, train rmse 0.241054, valid rmse 0.221462
```

```
fold 1, train rmse 0.229857, valid rmse 0.268489
fold 2, train rmse 0.231413, valid rmse 0.238157
fold 3, train rmse 0.237733, valid rmse 0.218747
fold 4, train rmse 0.230720, valid rmse 0.258712
5-fold validation: avg train rmse 0.234155, avg valid rmse 0.241113
```

3.16.6 模型选择

我们使用一组未经调优的超参数并计算交叉验证误差。可以改动这些超参数来尽可能减小平均测试误差。

```
k, num_epochs, lr, weight_decay, batch_size = 5, 100, 5, 0, 64
train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr, weight_decay)
print('%d-fold validation: avg train rmse %f, avg valid rmse %f' % (k, train_l, valid_l))
```

有时候你会发现一组参数的训练误差可以达到很低，但是在 K 折交叉验证上的误差可能反而较高。这种现象很可能是由过拟合造成的。因此，当训练误差降低时，我们要观察 K 折交叉验证上的误差是否也相应降低。

3.16.7 预测并在 Kaggle 提交结果

下面定义预测函数。在预测之前，我们会使用完整的训练数据集来重新训练模型，并将预测结果存成提交所需要的格式。

```
def train_and_pred(train_features, test_features, train_labels, test_data,
                   num_epochs, lr, weight_decay, batch_size):
    net = get_net(train_features.shape[1])
    train_ls, _ = train(net, train_features, train_labels, None, None,
                        num_epochs, lr, weight_decay, batch_size)
    d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'rmse')
    print('train rmse %f' % train_ls[-1])
    preds = net(test_features).detach().numpy()
    test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
    submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
    submission.to_csv('./submission.csv', index=False)
```

设计好模型并调好超参数之后，下一步就是对测试数据集上的房屋样本做价格预测。如果我们得到与交叉验证时差不多的训练误差，那么这个结果很可能是理想的，可以在 Kaggle 上提交结果。

```
train_and_pred(train_features, test_features, train_labels, test_data, num_epochs, lr)
```

输出：

```
train rmse 0.229943
```

上述代码执行完之后会生成一个 submission.csv 文件。这个文件是符合 Kaggle 比赛要求的提交格式的。这时，我们可以在 Kaggle 上提交我们预测得出的结果，并且查看与测试数据集上真实房价（标签）的误差。具体来说有以下几个步骤：登录 Kaggle 网站，访问房价预测比赛网页，并点击右侧“Submit Predictions”或“Late Submission”按钮；然后，点击页面下方“Upload Submission File”图标所在的虚线框选择需要提交的预测结果文件；最后，点击页面最下方的“Make Submission”按钮就可以查看结果了，如图 3.9 所示。

图 3.9 Kaggle 预测房价比赛的预测结果提交页面

小结

- 通常需要对真实数据做预处理。
- 可以使用 K 折交叉验证来选择模型并调节超参数。

注：本节除了代码之外与原书基本相同，[原书传送门](#)

4.1 模型构造

让我们回顾一下在 3.10 节（多层感知机的简洁实现）中含单隐藏层的多层感知机的实现方法。我们首先构造 `Sequential` 实例，然后依次添加两个全连接层。其中第一层的输出大小为 256，即隐藏层单元个数是 256；第二层的输出大小为 10，即输出层单元个数是 10。我们在上一章的其他节中也使用了 `Sequential` 类构造模型。这里我们介绍另外一种基于 `Module` 类的模型构造方法：它让模型构造更加灵活。

注：其实前面我们陆陆续续已经使用了这些方法了，本节系统介绍一下。

4.1.1 继承 `Module` 类来构造模型

`Module` 类是 `nn` 模块里提供的一个模型构造类，是所有神经网络模块的基类，我们可以继承它来定义我们想要的模型。下面继承 `Module` 类构造本节开头提到的多层感知机。这里定义的 `MLP` 类重载了 `Module` 类的 `__init__` 函数和 `forward` 函数。它们分别用于创建模型参数和定义前向计算。前向计算也即正向传播。

```
import torch
from torch import nn

class MLP(nn.Module):
    # 声明带有模型参数的层，这里声明了两个全连接层
    def __init__(self, **kwargs):
        # 调用 MLP 父类 Block 的构造函数来进行必要的初始化。这样在构造实例时还可以指定其
        # 参数，如“模型参数的访问、初始化和共享”一节将介绍的模型参数 params
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Linear(784, 256) # 隐藏层
        self.act = nn.ReLU()
        self.output = nn.Linear(256, 10) # 输出层

    # 定义模型的前向计算，即如何根据输入 x 计算返回所需要的模型输出
    def forward(self, x):
        a = self.act(self.hidden(x))
        return self.output(a)
```

以上的 `MLP` 类中无须定义反向传播函数。系统将通过自动求梯度而自动生成反向传播所需的 `backward` 函数。

我们可以实例化 `MLP` 类得到模型变量 `net`。下面的代码初始化 `net` 并传入输入数据 `X` 做一次前向计算。其中，`net(X)` 会调用 `MLP` 继承自 `Module` 类的 `__call__` 函数，这个函数将调用 `MLP` 类定义的 `forward` 函数来完成前向计算。

```
X = torch.rand(2, 784)
net = MLP()
print(net)
net(X)
```

输出：

```
MLP(
  (hidden): Linear(in_features=784, out_features=256, bias=True)
  (act): ReLU()
  (output): Linear(in_features=256, out_features=10, bias=True)
)
tensor([[-0.1798, -0.2253,  0.0206, -0.1067, -0.0889,  0.1818, -0.1474,  0.1845,
        -0.1870,  0.1970],
       [-0.1843, -0.1562, -0.0090,  0.0351, -0.1538,  0.0992, -0.0883,  0.0911,
        -0.2293,  0.2360]], grad_fn=<ThAddmmBackward>)
```

注意，这里并没有将 `Module` 类命名为 `Layer`（层）或者 `Model`（模型）之类的名字，这是因为该类是一个可供自由组建的部件。它的子类既可以是一个层（如 PyTorch 提供的 `Linear` 类），又可以是一个模型（如这里定义的 `MLP` 类），或者是模型的一个部分。我们下面通过两个例子来展示它的灵活性。

4.1.2 Module 的子类

我们刚刚提到，`Module` 类是一个通用的部件。事实上，PyTorch 还实现了继承自 `Module` 的可以方便构建模型的类：如 `Sequential`、`ModuleList` 和 `ModuleDict` 等等。

4.1.2.1 Sequential 类

当模型的前向计算为简单串联各个层的计算时，`Sequential` 类可以通过更加简单的方式定义模型。这正是 `Sequential` 类的目的：它可以接收一个子模块的有序字典（`OrderedDict`）或者一系列子模块作为参数来逐一添加 `Module` 的实例，而模型的前向计算就是将这些实例按添加的顺序逐一计算。

下面我们实现一个与 `Sequential` 类有相同功能的 `MySequential` 类。这或许可以帮助读者更加清晰地理解 `Sequential` 类的工作机制。

```

class MySequential(nn.Module):
    from collections import OrderedDict
    def __init__(self, *args):
        super(MySequential, self).__init__()
        if len(args) == 1 and isinstance(args[0], OrderedDict): # 如果传入的是一个 Order
            for key, module in args[0].items():
                self.add_module(key, module) # add_module 方法会将 module 添加进 self._modules
        else: # 传入的是一些 Module
            for idx, module in enumerate(args):
                self.add_module(str(idx), module)
    def forward(self, input):
        # self._modules 返回一个 OrderedDict, 保证会按照成员添加时的顺序遍历成员
        for module in self._modules.values():
            input = module(input)
        return input

```

我们用 MySequential 类来实现前面描述的 MLP 类，并使用随机初始化的模型做一次前向计算。

```

net = MySequential(
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Linear(256, 10),
)
print(net)
net(X)

```

输出：

```

MySequential(
(0): Linear(in_features=784, out_features=256, bias=True)
(1): ReLU()
(2): Linear(in_features=256, out_features=10, bias=True)
)
tensor([[-0.0100, -0.2516,  0.0392, -0.1684, -0.0937,  0.2191, -0.1448,  0.0930,
        0.1228, -0.2540],
       [-0.1086, -0.1858,  0.0203, -0.2051, -0.1404,  0.2738, -0.0607,  0.0622,
        0.0817, -0.2574]], grad_fn=<ThAddmmBackward>)

```

可以观察到这里 `MySequential` 类的使用跟 3.10 节（多层感知机的简洁实现）中 `Sequential` 类的使用没什么区别。

4.1.2.2 `ModuleList` 类

`ModuleList` 接收一个子模块的列表作为输入，然后也可以类似 `List` 那样进行 `append` 和 `extend` 操作：

```
net = nn.ModuleList([nn.Linear(784, 256), nn.ReLU()])
net.append(nn.Linear(256, 10)) # # 类似 List 的 append 操作
print(net[-1]) # 类似 List 的索引访问
print(net)
```

输出：

```
Linear(in_features=256, out_features=10, bias=True)
ModuleList(
    (0): Linear(in_features=784, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=10, bias=True)
)
```

4.1.2.3 `ModuleDict` 类

`ModuleDict` 接收一个子模块的字典作为输入，然后也可以类似字典那样进行添加访问操作：

```
net = nn.ModuleDict({
    'linear': nn.Linear(784, 256),
    'act': nn.ReLU(),
})
net['output'] = nn.Linear(256, 10) # 添加
print(net['linear']) # 访问
print(net.output)
print(net)
```

输出：

```
Linear(in_features=784, out_features=256, bias=True)
Linear(in_features=256, out_features=10, bias=True)
ModuleDict(
    (act): ReLU()
    (linear): Linear(in_features=784, out_features=256, bias=True)
    (output): Linear(in_features=256, out_features=10, bias=True)
)
```

4.1.3 构造复杂的模型

虽然上面介绍的这些类可以使模型构造更加简单，且不需要定义 `forward` 函数，但直接继承 `Module` 类可以极大地拓展模型构造的灵活性。下面我们构造一个稍微复杂点的网络 `FancyMLP`。在这个网络中，我们通过 `get_constant` 函数创建训练中不被迭代的参数，即常数参数。在前向计算中，除了使用创建的常数参数外，我们还使用 `Tensor` 的函数和 Python 的控制流，并多次调用相同的层。

```
class FancyMLP(nn.Module):
    def __init__(self, **kwargs):
        super(FancyMLP, self).__init__(**kwargs)

        self.rand_weight = torch.rand((20, 20), requires_grad=False) # 不可训练参数 (常数参数)
        self.linear = nn.Linear(20, 20)

    def forward(self, x):
        x = self.linear(x)
        # 使用创建的常数参数，以及 nn.functional 中的 relu 函数和 mm 函数
        x = nn.functional.relu(torch.mm(x, self.rand_weight.data) + 1)

        # 复用全连接层。等价于两个全连接层共享参数
        x = self.linear(x)
        # 控制流，这里我们需要调用 item 函数来返回标量进行比较
        while x.norm().item() > 1:
            x /= 2
            if x.norm().item() < 0.8:
                x *= 10
        return x.sum()
```

在这个 FancyMLP 模型中，我们使用了常数权重 `rand_weight`（注意它不是可训练模型参数）、做了矩阵乘法操作（`torch.mm`）并重复使用了相同的 `Linear` 层。下面我们将来测试该模型的前向计算。

```
X = torch.rand(2, 20)
net = FancyMLP()
print(net)
net(X)
```

输出：

```
FancyMLP(
  (linear): Linear(in_features=20, out_features=20, bias=True)
)
tensor(0.8432, grad_fn=<SumBackward0>)
```

因为 `FancyMLP` 和 `Sequential` 类都是 `Module` 类的子类，所以我们可以嵌套调用它们。

```
class NestMLP(nn.Module):
    def __init__(self, **kwargs):
        super(NestMLP, self).__init__(**kwargs)
        self.net = nn.Sequential(nn.Linear(40, 30), nn.ReLU())

    def forward(self, x):
        return self.net(x)

net = nn.Sequential(NestMLP(), nn.Linear(30, 20), FancyMLP())

X = torch.rand(2, 40)
print(net)
net(X)
```

输出：

```
Sequential(
  (0): NestMLP(
    (net): Sequential(
```

```
(0): Linear(in_features=40, out_features=30, bias=True)
(1): ReLU()
)
)
(1): Linear(in_features=30, out_features=20, bias=True)
(2): FancyMLP(
    (linear): Linear(in_features=20, out_features=20, bias=True)
)
)
tensor(14.4908, grad_fn=<SumBackward0>)
```

小结

- 可以通过继承 `Module` 类来构造模型。
- `Sequential`、`ModuleList`、`ModuleDict` 类都继承自 `Module` 类。
- 虽然 `Sequential` 等类可以使模型构造更加简单，但直接继承 `Module` 类可以极大地拓展模型构造的灵活性。

注：本节与原书此节有一些不同，[原书传送门](#)

4.2 模型参数的访问、初始化和共享

在 3.3 节（线性回归的简洁实现）中，我们通过 `init` 模块来初始化模型的参数。我们也介绍了访问模型参数的简单方法。本节将深入讲解如何访问和初始化模型参数，以及如何在多个层之间共享同一份模型参数。

我们先定义一个与上一节中相同的含单隐藏层的多层感知机。我们依然使用默认方式初始化它的参数，并做一次前向计算。与之前不同的是，在这里我们从 `nn` 中导入了 `init` 模块，它包含了多种模型初始化方法。

```
import torch
from torch import nn
from torch.nn import init

net = nn.Sequential(nn.Linear(4, 3), nn.ReLU(), nn.Linear(3, 1)) # pytorch 已进行默认

print(net)
X = torch.rand(2, 4)
Y = net(X).sum()
```

输出：

```
Sequential(
  (0): Linear(in_features=4, out_features=3, bias=True)
  (1): ReLU()
  (2): Linear(in_features=3, out_features=1, bias=True)
)
```

4.2.1 访问模型参数

回忆一下上一节中提到的 `Sequential` 类与 `Module` 类的继承关系。对于 `Sequential` 实例中含模型参数的层，我们可以通过 `Module` 类的 `parameters()` 或者 `named_parameters` 方法来访问所有参数（以迭代器的形式返回），后者除了返回参数 `Tensor` 外还会返回其名字。下面，访问多层次感知机 `net` 的所有参数：

```
print(type(net.named_parameters()))
for name, param in net.named_parameters():
    print(name, param.size())
```

输出：

```
<class 'generator'>
0.weight torch.Size([3, 4])
0.bias torch.Size([3])
2.weight torch.Size([1, 3])
2.bias torch.Size([1])
```

可见返回的名字自动加上了层数的索引作为前缀。我们再来访问 `net` 中单层的参数。对于使用 `Sequential` 类构造的神经网络，我们可以通过方括号 `[]` 来访问网络的任一层。索引 0 表示隐藏层为 `Sequential` 实例最先添加的层。

```
for name, param in net[0].named_parameters():
    print(name, param.size(), type(param))
```

输出：

```
weight torch.Size([3, 4]) <class 'torch.nn.parameter.Parameter'>
bias torch.Size([3]) <class 'torch.nn.parameter.Parameter'>
```

因为这里是单层的所以没有了层数索引的前缀。另外返回的 `param` 的类型为 `torch.nn.parameter.Parameter`，其实这是 `Tensor` 的子类，和 `Tensor` 不同的是如果一个 `Tensor` 是 `Parameter`，那么它会自动被添加到模型的参数列表里，来看下面这个例子。

```
class MyModel(nn.Module):
    def __init__(self, **kwargs):
        super(MyModel, self).__init__(**kwargs)
        self.weight1 = nn.Parameter(torch.rand(20, 20))
        self.weight2 = torch.rand(20, 20)
    def forward(self, x):
        pass

n = MyModel()
for name, param in n.named_parameters():
    print(name)
```

输出：

```
weight1
```

上面的代码中 `weight1` 在参数列表中但是 `weight2` 却没在参数列表中。

因为 `Parameter` 是 `Tensor`, 即 `Tensor` 拥有的属性它都有, 比如可以根据 `data` 来访问参数数值, 用 `grad` 来访问参数梯度。

```
weight_0 = list(net[0].parameters())[0]
print(weight_0.data)
print(weight_0.grad) # 反向传播前梯度为 None
Y.backward()
print(weight_0.grad)
```

输出:

```
tensor([[ 0.2719, -0.0898, -0.2462,  0.0655],
        [-0.4669, -0.2703,  0.3230,  0.2067],
        [-0.2708,  0.1171, -0.0995,  0.3913]])
None
tensor([[[-0.2281, -0.0653, -0.1646, -0.2569],
        [-0.1916, -0.0549, -0.1382, -0.2158],
        [ 0.0000,  0.0000,  0.0000,  0.0000]]])
```

4.2.2 初始化模型参数

我们在 3.15 节（数值稳定性和模型初始化）中提到了 PyTorch 中 `nn.Module` 的模块参数都采取了较为合理的初始化策略（不同类型的 layer 具体采样的哪一种初始化方法的可参考[源代码](#)）。但我们经常需要使用其他方法来初始化权重。PyTorch 的 `init` 模块里提供了多种预设的初始化方法。在下面的例子中, 我们将权重参数初始化成均值为 0、标准差为 0.01 的正态分布随机数, 并依然将偏差参数清零。

```
for name, param in net.named_parameters():
    if 'weight' in name:
        init.normal_(param, mean=0, std=0.01)
        print(name, param.data)
```

输出:

```
0.weight tensor([[ 0.0030,  0.0094,  0.0070, -0.0010],  
                 [ 0.0001,  0.0039,  0.0105, -0.0126],  
                 [ 0.0105, -0.0135, -0.0047, -0.0006]])  
2.weight tensor([[-0.0074,  0.0051,  0.0066]])
```

下面使用常数来初始化权重参数。

```
for name, param in net.named_parameters():  
    if 'bias' in name:  
        init.constant_(param, val=0)  
        print(name, param.data)
```

输出：

```
0.bias tensor([0., 0., 0.])  
2.bias tensor([0.])
```

如果只想对某个特定参数进行初始化，我们可以调用 `Parameter` 类的 `initialize` 函数，它与 `Block` 类提供的 `initialize` 函数的使用方法一致。下例中我们对隐藏层的权重使用 Xavier 随机初始化方法。

4.2.3 自定义初始化方法

有时候我们需要的初始化方法并没有在 `init` 模块中提供。这时，可以实现一个初始化方法，从而能够像使用其他初始化方法那样使用它。在这之前我们先来看看 PyTorch 是怎么实现这些初始化方法的，例如 `torch.nn.init.normal_`：

```
def normal_(tensor, mean=0, std=1):  
    with torch.no_grad():  
        return tensor.normal_(mean, std)
```

可以看到这就是一个 `inplace` 改变 `Tensor` 值的函数，而且这个过程是不记录梯度的。类似的我们来实现一个自定义的初始化方法。在下面的例子里，我们令权重有一半概率初始化为 0，有另一半概率初始化为 $[-10, -5]$ 和 $[5, 10]$ 两个区间里均匀分布的随机数。

```
def init_weight_(tensor):  
    with torch.no_grad():  
        tensor.uniform_(-10, 10)
```

```

tensor *= (tensor.abs() >= 5).float()

for name, param in net.named_parameters():
    if 'weight' in name:
        init_weight_(param)
        print(name, param.data)

```

输出：

```

0.weight tensor([[ 7.0403,  0.0000, -9.4569,  7.0111],
                 [-0.0000, -0.0000,  0.0000,  0.0000],
                 [ 9.8063, -0.0000,  0.0000, -9.7993]])
2.weight tensor([[-5.8198,  7.7558, -5.0293]])

```

此外，参考 2.3.2 节，我们还可以通过改变这些参数的 `data` 来改写模型参数值同时不会影响梯度：

```

for name, param in net.named_parameters():
    if 'bias' in name:
        param.data += 1
        print(name, param.data)

```

输出：

```

0.bias tensor([1., 1., 1.])
2.bias tensor([1.])

```

4.2.4 共享模型参数

在有些情况下，我们希望在多个层之间共享模型参数。4.1.3 节提到了如何共享模型参数：`Module` 类的 `forward` 函数里多次调用同一个层。此外，如果我们传入 `Sequential` 的模块是同一个 `Module` 实例的话参数也是共享的，下面来看一个例子：

```

linear = nn.Linear(1, 1, bias=False)
net = nn.Sequential(linear, linear)
print(net)

for name, param in net.named_parameters():
    init.constant_(param, val=3)
    print(name, param.data)

```

输出:

```
Sequential(  
    (0): Linear(in_features=1, out_features=1, bias=False)  
    (1): Linear(in_features=1, out_features=1, bias=False)  
)  
0.weight tensor([[3.]])
```

在内存中，这两个线性层其实一个对象:

```
print(id(net[0]) == id(net[1]))  
print(id(net[0].weight) == id(net[1].weight))
```

输出:

```
True  
True
```

因为模型参数里包含了梯度，所以在反向传播计算时，这些共享的参数的梯度是累加的:

```
x = torch.ones(1, 1)  
y = net(x).sum()  
print(y)  
y.backward()  
print(net[0].weight.grad) # 单次梯度是 3, 两次所以就是 6
```

输出:

```
tensor(9., grad_fn=<SumBackward0>)  
tensor([[6.]])
```

小结

- 有多种方法来访问、初始化和共享模型参数。
- 可以自定义初始化方法。

注：本节与原书此节有一些不同，[原书传送门](#)

4.3 模型参数的延后初始化

由于使用 Gluon 创建的全连接层的时候不需要指定输入个数。所以当调用 `initialize` 函数时，由于隐藏层输入个数依然未知，系统也无法得知该层权重参数的形状。只有在当形状已知的输入 `x` 传进网络做前向计算 `net(x)` 时，系统才推断出该层的权重参数形状为多少，此时才进行真正的初始化操作。但是使用 PyTorch 在定义模型的时候就要指定输入的形状，所以也就不存在这个问题了，所以本节略。有兴趣的可以去看看原文，[传送门](#)。

4.4 自定义层

深度学习的一个魅力在于神经网络中各式各样的层，例如全连接层和后面章节中将要介绍的卷积层、池化层与循环层。虽然 PyTorch 提供了大量常用的层，但有时候我们依然希望自定义层。本节将介绍如何使用 `Module` 来自定义层，从而可以被重复调用。

4.4.1 不含模型参数的自定义层

我们先介绍如何定义一个不含模型参数的自定义层。事实上，这和 4.1 节（模型构造）中介绍的使用 `Module` 类构造模型类似。下面的 `CenteredLayer` 类通过继承 `Module` 类自定义了一个将输入减掉均值后输出的层，并将层的计算定义在了 `forward` 函数里。这个层里不含模型参数。

```
import torch
from torch import nn

class CenteredLayer(nn.Module):
    def __init__(self, **kwargs):
        super(CenteredLayer, self).__init__(**kwargs)
    def forward(self, x):
        return x - x.mean()
```

我们可以实例化这个层，然后做前向计算。

```
layer = CenteredLayer()
layer(torch.tensor([1, 2, 3, 4, 5], dtype=torch.float))
```

输出：

```
tensor([-2., -1.,  0.,  1.,  2.])
```

我们也可以用它来构造更复杂的模型。

```
net = nn.Sequential(nn.Linear(8, 128), CenteredLayer())
```

下面打印自定义层各个输出的均值。因为均值是浮点数，所以它的值是一个很接近 0 的数。

```
y = net(torch.rand(4, 8))
y.mean().item()
```

输出：

```
0.0
```

4.4.2 含模型参数的自定义层

我们还可以自定义含模型参数的自定义层。其中的模型参数可以通过训练学出。

在 4.2 节（模型参数的访问、初始化和共享）中介绍了 `Parameter` 类其实是 `Tensor` 的子类，如果一个 `Tensor` 是 `Parameter`，那么它会自动被添加到模型的参数列表里。所以在自定义含模型参数的层时，我们应该将参数定义成 `Parameter`，除了像 4.2.1 节那样直接定义成 `Parameter` 类外，还可以使用 `ParameterList` 和 `ParameterDict` 分别定义参数的列表和字典。

`ParameterList` 接收一个 `Parameter` 实例的列表作为输入然后得到一个参数列表，使用的时候可以用索引来访问某个参数，另外也可以使用 `append` 和 `extend` 在列表后面新增参数。

```
class MyDense(nn.Module):
    def __init__(self):
        super(MyDense, self).__init__()
        self.params = nn.ParameterList([nn.Parameter(torch.randn(4, 4)) for i in range(2)])
        self.params.append(nn.Parameter(torch.randn(4, 1)))

    def forward(self, x):
        for i in range(len(self.params)):
            x = torch.mm(x, self.params[i])
        return x

net = MyDense()
print(net)
```

输出：

```
MyDense(
  (params): ParameterList(
    (0): Parameter containing: [torch.FloatTensor of size 4x4]
    (1): Parameter containing: [torch.FloatTensor of size 4x4]
```

```
(2): Parameter containing: [torch.FloatTensor of size 4x4]
(3): Parameter containing: [torch.FloatTensor of size 4x1]
)
)
```

而 `ParameterDict` 接收一个 `Parameter` 实例的字典作为输入然后得到一个参数字典，然后可以按照字典的规则使用了。例如使用 `update()` 新增参数，使用 `keys()` 返回所有键值，使用 `items()` 返回所有键值对等等，可参考[官方文档](#)。

```
class MyDictDense(nn.Module):
    def __init__(self):
        super(MyDictDense, self).__init__()
        self.params = nn.ParameterDict({
            'linear1': nn.Parameter(torch.randn(4, 4)),
            'linear2': nn.Parameter(torch.randn(4, 1))
        })
        self.params.update({'linear3': nn.Parameter(torch.randn(4, 2))}) # 新增

    def forward(self, x, choice='linear1'):
        return torch.mm(x, self.params[choice])

net = MyDictDense()
print(net)
```

输出：

```
MyDictDense(
    (params): ParameterDict(
        (linear1): Parameter containing: [torch.FloatTensor of size 4x4]
        (linear2): Parameter containing: [torch.FloatTensor of size 4x1]
        (linear3): Parameter containing: [torch.FloatTensor of size 4x2]
    )
)
```

这样就可以根据传入的键值来进行不同的前向传播：

```
x = torch.ones(1, 4)
print(net(x, 'linear1'))
```

```
print(net(x, 'linear2'))
print(net(x, 'linear3'))
```

输出：

```
tensor([[1.5082, 1.5574, 2.1651, 1.2409]], grad_fn=<MmBackward>)
tensor([[-0.8783]], grad_fn=<MmBackward>)
tensor([[ 2.2193, -1.6539]], grad_fn=<MmBackward>)
```

我们也可以使用自定义层构造模型。它和 PyTorch 的其他层在使用上很类似。

```
net = nn.Sequential(
    MyDictDense(),
    MyListDense(),
)
print(net)
print(net(x))
```

输出：

```
Sequential(
(0): MyDictDense(
  (params): ParameterDict(
    (linear1): Parameter containing: [torch.FloatTensor of size 4x4]
    (linear2): Parameter containing: [torch.FloatTensor of size 4x1]
    (linear3): Parameter containing: [torch.FloatTensor of size 4x2]
  )
)
(1): MyListDense(
  (params): ParameterList(
    (0): Parameter containing: [torch.FloatTensor of size 4x4]
    (1): Parameter containing: [torch.FloatTensor of size 4x4]
    (2): Parameter containing: [torch.FloatTensor of size 4x4]
    (3): Parameter containing: [torch.FloatTensor of size 4x1]
  )
)
)
tensor([-101.2394], grad_fn=<MmBackward>)
```

小结

- 可以通过 `Module` 类自定义神经网络中的层，从而可以被重复调用。

注：本节与原书此节有一些不同，[原书传送门](#)

4.5 读取和存储

到目前为止，我们介绍了如何处理数据以及如何构建、训练和测试深度学习模型。然而在实际中，我们有时需要把训练好的模型部署到很多不同的设备。在这种情况下，我们可以把内存中训练好的模型参数存储在硬盘上供后续读取使用。

4.5.1 读写 Tensor

我们可以直接使用 `save` 函数和 `load` 函数分别存储和读取 `Tensor`。`save` 使用 Python 的 pickle 实用程序将对象进行序列化，然后将序列化的对象保存到 disk，使用 `save` 可以保存各种对象，包括模型、张量和字典等。而 `load` 使用 pickle unpickle 工具将 pickle 的对象文件反序列化为内存。

下面的例子创建了 `Tensor` 变量 `x`，并将其存在文件名同为 `x.pt` 的文件里。

```
import torch
from torch import nn

x = torch.ones(3)
torch.save(x, 'x.pt')
```

然后我们将数据从存储的文件读回内存。

```
x2 = torch.load('x.pt')
x2
```

输出：

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

我们还可以存储一个 `Tensor` 列表并读回内存。

```
y = torch.zeros(4)
torch.save([x, y], 'xy.pt')
xy_list = torch.load('xy.pt')
xy_list
```

输出：

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

存储并读取一个从字符串映射到 Tensor 的字典。

```
torch.save({'x': x, 'y': y}, 'xy_dict.pt')
xy = torch.load('xy_dict.pt')
xy
```

输出：

```
{'x': tensor([1., 1., 1.]), 'y': tensor([0., 0., 0., 0.])}
```

4.5.2 读写模型

4.5.2.1 state_dict

在 PyTorch 中，`Module` 的可学习参数 (即权重和偏差)，模块模型包含在参数中 (通过 `model.parameters()` 访问)。`state_dict` 是一个从参数名称映射到参数 Tensor 的字典对象。

```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.hidden = nn.Linear(3, 2)
        self.act = nn.ReLU()
        self.output = nn.Linear(2, 1)

    def forward(self, x):
        a = self.act(self.hidden(x))
        return self.output(a)

net = MLP()
net.state_dict()
```

输出：

```
OrderedDict([('hidden.weight', tensor([[ 0.2448,  0.1856, -0.5678],
                                         [ 0.2030, -0.2073, -0.0104]])),
              ('hidden.bias', tensor([-0.3117, -0.4232])),
              ('output.weight', tensor([[-0.4556,  0.4084]])),
              ('output.bias', tensor([-0.3573]))])
```

注意，只有具有可学习参数的层（卷积层、线性层等）才有 `state_dict` 中的条目。优化器 (`optim`) 也有一个 `state_dict`，其中包含关于优化器状态以及所使用的超参数的信息。

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
optimizer.state_dict()
```

输出：

```
{'param_groups': [ {'dampening': 0,
                     'lr': 0.001,
                     'momentum': 0.9,
                     'nesterov': False,
                     'params': [4736167728, 4736166648, 4736167368, 4736165352],
                     'weight_decay': 0} ],
  'state': {}}
```

4.5.2.2 保存和加载模型

PyTorch 中保存和加载训练模型有两种常见的方法：1. 仅保存和加载模型参数 (`state_dict`)；2. 保存和加载整个模型。#### 1. 保存和加载 `state_dict`(推荐方式) 保存：

```
torch.save(model.state_dict(), PATH) # 推荐的文件后缀名是 pt 或 pth
```

加载：

```
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
```

2. 保存和加载整个模型

保存：

```
torch.save(model, PATH)
```

加载：

```
model = torch.load(PATH)
```

我们采用推荐的方法一来实验一下：

```
X = torch.randn(2, 3)
Y = net(X)

PATH = "./net.pt"
torch.save(net.state_dict(), PATH)

net2 = MLP()
net2.load_state_dict(torch.load(PATH))
Y2 = net2(X)
Y2 == Y
```

输出：

```
tensor([[1],
       [1]], dtype=torch.uint8)
```

因为这 `net` 和 `net2` 都有同样的模型参数，那么对同一个输入 `X` 的计算结果将会是一样的。上面的输出也验证了这一点。

此外，还有一些其他使用场景，例如 GPU 与 CPU 之间的模型保存与读取、使用多块 GPU 的模型的存储等等，使用的时候可以参考[官方文档](#)。

小结

- 通过 `save` 函数和 `load` 函数可以很方便地读写 `Tensor`。
- 通过 `save` 函数和 `load_state_dict` 函数可以很方便地读写模型的参数。

注：本节与原书此节有一些不同，[原书传送门](#)

4.6 GPU 计算

到目前为止，我们一直在使用 CPU 计算。对复杂的神经网络和大规模的数据来说，使用 CPU 来计算可能不够高效。在本节中，我们将介绍如何使用单块 NVIDIA GPU 来计算。所以需要确保已经安装好了 PyTorch GPU 版本。准备工作都完成后，下面就可以通过 `nvidia-smi` 命令来查看显卡信息了。

```
!nvidia-smi # 对 Linux/macOS 用户有效
```

输出：

```
Sun Mar 17 14:59:57 2019
```

```
+-----+  
| NVIDIA-SMI 390.48          Driver Version: 390.48 |  
|-----+-----+  
| GPU  Name      Persistence-M| Bus-Id      Disp.A  | Volatile Uncorr. ECC |  
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |  
|=====+=====+=====+=====+=====+=====+=====+  
| 0  GeForce GTX 1050     Off  | 00000000:01:00.0 Off |                  N/A |  
| 20%   36C     P5    N/A / 75W |  1223MiB / 2000MiB |      0%  Default |  
+-----+-----+-----+
```

```
+-----+  
| Processes:                               GPU Memory |  
| GPU     PID  Type  Process name           Usage  |  
|=====+=====+=====+=====+=====+=====+  
| 0       1235  G    /usr/lib/xorg/Xorg        434MiB |  
| 0       2095  G    compiz                   163MiB |  
| 0       2660  G    /opt/teamviewer/tv_bin/TeamViewer  5MiB |  
| 0       4166  G    /proc/self/exe            416MiB |  
| 0       13274  C    /home/tss/anaconda3/bin/python  191MiB |  
+-----+
```

可以看到我这里只有一块 GTX 1050，显存一共只有 2000M（太惨了）。

4.6.1 计算设备

PyTorch 可以指定用来存储和计算的设备，如使用内存的 CPU 或者使用显存的 GPU。默认情况下，PyTorch 会将数据创建在内存，然后利用 CPU 来计算。

用 `torch.cuda.is_available()` 查看 GPU 是否可用:

```
import torch
from torch import nn

torch.cuda.is_available() # 输出 True
```

查看 GPU 数量:

```
torch.cuda.device_count() # 输出 1
```

查看当前 GPU 索引号, 索引号从 0 开始:

```
torch.cuda.current_device() # 输出 0
```

根据索引号查看 GPU 名字:

```
torch.cuda.get_device_name(0) # 输出 'GeForce GTX 1050'
```

4.6.2 Tensor 的 GPU 计算

默认情况下, `Tensor` 会被存在内存上。因此, 之前我们每次打印 `Tensor` 的时候看不到 GPU 相关标识。

```
x = torch.tensor([1, 2, 3])
x
```

输出:

```
tensor([1, 2, 3])
```

使用 `.cuda()` 可以将 CPU 上的 `Tensor` 转换(复制)到 GPU 上。如果有多块 GPU, 我们用 `.cuda(i)` 来表示第 i 块 GPU 及相应的显存(i 从 0 开始)且 `cuda(0)` 和 `cuda()` 等价。

```
x = x.cuda(0)
x
```

输出:

```
tensor([1, 2, 3], device='cuda:0')
```

我们可以通过 Tensor 的 device 属性来查看该 Tensor 所在的设备。

```
x.device
```

输出:

```
device(type='cuda', index=0)
```

我们可以直接在创建的时候就指定设备。

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

```
x = torch.tensor([1, 2, 3], device=device)
# or
x = torch.tensor([1, 2, 3]).to(device)
x
```

输出:

```
tensor([1, 2, 3], device='cuda:0')
```

如果对在 GPU 上的数据进行运算，那么结果还是存放在 GPU 上。

```
y = x**2
y
```

输出:

```
tensor([1, 4, 9], device='cuda:0')
```

需要注意的是，存储在不同位置中的数据是不可以直接进行计算的。即存放在 CPU 上的数据不可以直接与存放在 GPU 上的数据进行运算，位于不同 GPU 上的数据也是不能直接进行计算的。

```
z = y + x.cpu()
```

会报错:

```
RuntimeError: Expected object of type torch.cuda.LongTensor but found type torch.Long
```

4.6.3 模型的 GPU 计算

同 `Tensor` 类似，PyTorch 模型也可以通过`.cuda` 转换到 GPU 上。我们可以通过检查模型的参数的 `device` 属性来查看存放模型的设备。

```
net = nn.Linear(3, 1)
list(net.parameters())[0].device
```

输出：

```
device(type='cpu')
```

可见模型在 CPU 上，将其转换到 GPU 上：

```
net.cuda()
list(net.parameters())[0].device
```

输出：

```
device(type='cuda', index=0)
```

同样的，我么需要保证模型输入的 `Tensor` 和模型都在同一设备上，否则会报错。

```
x = torch.rand(2,3).cuda()
net(x)
```

输出：

```
tensor([[-0.5800],
       [-0.2995]], device='cuda:0', grad_fn=<ThAddmmBackward>)
```

小结

- PyTorch 可以指定用来存储和计算的设备，如使用内存的 CPU 或者使用显存的 GPU。在默认情况下，PyTorch 会将数据创建在内存，然后利用 CPU 来计算。
- PyTorch 要求计算的所有输入数据都在内存或同一块显卡的显存上。

注：本节与原书此节有一些不同，[原书传送门](#)

5.1 二维卷积层

卷积神经网络 (convolutional neural network) 是含有卷积层 (convolutional layer) 的神经网络。本章中介绍的卷积神经网络均使用最常见的二维卷积层。它有高和宽两个空间维度，常用来处理图像数据。本节中，我们将介绍简单形式的二维卷积层的工作原理。

5.1.1 二维互相关运算

虽然卷积层得名于卷积 (convolution) 运算，但我们通常在卷积层中使用更加直观的互相关 (cross-correlation) 运算。在二维卷积层中，一个二维输入数组和一个二维核 (kernel) 数组通过互相关运算输出一个二维数组。我们用一个具体例子来解释二维互相关运算的含义。如图 5.1 所示，输入是一个高和宽均为 3 的二维数组。我们将该数组的形状记为 3×3 或 $(3, 3)$ 。核数组的高和宽分别为 2。该数组在卷积计算中又称卷积核或过滤器 (filter)。卷积核窗口 (又称卷积窗口) 的形状取决于卷积核的高和宽，即 2×2 。图 5.1 中的阴影部分为第一个输出元素及其计算所使用的输入和核数组元素： $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$ 。

图 5.1 二维互相关运算

在二维互相关运算中，卷积窗口从输入数组的最左上方开始，按从左往右、从上往下的顺序，依次在输入数组上滑动。当卷积窗口滑动到某一位置时，窗口中的输入子数组与核数组按元素相乘并求和，得到输出数组中相应位置的元素。图 5.1 中的输出数组高和宽分别为 2，其中的 4 个元素由二维互相关运算得出：

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19, 1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25, 3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37, 4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 49$$

下面我们将上述过程实现在 `corr2d` 函数里。它接受输入数组 `X` 与核数组 `K`，并输出数组 `Y`。

```
import torch
from torch import nn

def corr2d(X, K):  # 本函数已保存在 d2lzh_pytorch 包中方便以后使用
    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
```

```

Y[i, j] = (X[i: i + h, j: j + w] * K).sum()
return Y

```

我们可以构造图 5.1 中的输入数组 X 、核数组 K 来验证二维互相关运算的输出。

```

X = torch.tensor([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
K = torch.tensor([[0, 1], [2, 3]])
corr2d(X, K)

```

输出：

```

tensor([[19., 25.],
       [37., 43.]])

```

5.1.2 二维卷积层

二维卷积层将输入和卷积核做互相关运算，并加上一个标量偏差来得到输出。卷积层的模型参数包括了卷积核和标量偏差。在训练模型的时候，通常我们先对卷积核随机初始化，然后不断迭代卷积核和偏差。

下面基于 `corr2d` 函数来实现一个自定义的二维卷积层。在构造函数 `__init__` 里我们声明 `weight` 和 `bias` 这两个模型参数。前向计算函数 `forward` 则是直接调用 `corr2d` 函数再加上偏差。

```

class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super(Conv2D, self).__init__()
        self.weight = nn.Parameter(torch.randn(kernel_size))
        self.bias = nn.Parameter(torch.randn(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias

```

卷积窗口形状为 $p \times q$ 的卷积层称为 $p \times q$ 卷积层。同样， $p \times q$ 卷积或 $p \times q$ 卷积核说明卷积核的高和宽分别为 p 和 q 。

5.1.3 图像中物体边缘检测

下面我们来看一个卷积层的简单应用：检测图像中物体的边缘，即找到像素变化的位置。首先我们构造一张 6×8 的图像（即高和宽分别为 6 像素和 8 像素的图像）。它中间 4 列为黑（0），其余为白（1）。

```
X = torch.ones(6, 8)
X[:, 2:6] = 0
X
```

输出：

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

然后我们构造一个高和宽分别为 1 和 2 的卷积核 K 。当它与输入做互相关运算时，如果横向相邻元素相同，输出为 0；否则输出为非 0。

```
K = torch.tensor([[1, -1]])
```

下面将输入 X 和我们设计的卷积核 K 做互相关运算。可以看出，我们将从白到黑的边缘和从黑到白的边缘分别检测成了 1 和 -1。其余部分的输出全是 0。

```
Y = corr2d(X, K)
Y
```

输出：

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

由此，我们可以看出，卷积层可通过重复使用卷积核有效地表征局部空间。

5.1.4 通过数据学习核数组

最后我们来看一个例子，它使用物体边缘检测中的输入数据 X 和输出数据 Y 来学习我们构造的核数组 K 。我们首先构造一个卷积层，其卷积核将被初始化成随机数组。接下来在每一次迭代中，我们使用平方误差来比较 Y 和卷积层的输出，然后计算梯度来更新权重。

```
# 构造一个核数组形状是 (1, 2) 的二维卷积层
conv2d = Conv2D(kernel_size=(1, 2))

step = 20
lr = 0.01
for i in range(step):
    Y_hat = conv2d(X)
    l = ((Y_hat - Y) ** 2).sum()
    l.backward()

    # 梯度下降
    conv2d.weight.data -= lr * conv2d.weight.grad
    conv2d.bias.data -= lr * conv2d.bias.grad

    # 梯度清 0
    conv2d.weight.grad.fill_(0)
    conv2d.bias.grad.fill_(0)
    if (i + 1) % 5 == 0:
        print('Step %d, loss %.3f' % (i + 1, l.item()))
```

输出：

```
Step 5, loss 1.844
Step 10, loss 0.206
Step 15, loss 0.023
Step 20, loss 0.003
```

可以看到，20 次迭代后误差已经降到了一个比较小的值。现在来看一下学习到的卷积核的参数。

```
print("weight: ", conv2d.weight.data)
print("bias: ", conv2d.bias.data)
```

输出：

```
weight: tensor([[ 0.9948, -1.0092]])  
bias:  tensor([0.0080])
```

可以看到，学到的卷积核的权重参数与我们之前定义的核数组 K 较接近，而偏置参数接近 0。

5.1.5 互相关运算和卷积运算

实际上，卷积运算与互相关运算类似。为了得到卷积运算的输出，我们只需将核数组左右翻转并上下翻转，再与输入数组做互相关运算。可见，卷积运算和互相关运算虽然类似，但如果它们使用相同的核数组，对于同一个输入，输出往往并不相同。

那么，你也许会好奇卷积层为何能使用互相关运算替代卷积运算。其实，在深度学习中核数组都是学出来的：卷积层无论使用互相关运算或卷积运算都不影响模型预测时的输出。为了解释这一点，假设卷积层使用互相关运算学出图 5.1 中的核数组。设其他条件不变，使用卷积运算学出的核数组即图 5.1 中的核数组按上下、左右翻转。也就是说，图 5.1 中的输入与学出的已翻转的核数组再做卷积运算时，依然得到图 5.1 中的输出。为了与大多数深度学习文献一致，如无特别说明，本书中提到的卷积运算均指互相关运算。

注：感觉深度学习中的卷积运算实际上是互相关运算是个面试题考点。

5.1.6 特征图和感受野

二维卷积层输出的二维数组可以看作是输入在空间维度（宽和高）上某一级的表征，也叫特征图（feature map）。影响元素 x 的前向计算的所有可能输入区域（可能大于输入的实际尺寸）叫做 x 的感受野（receptive field）。以图 5.1 为例，输入中阴影部分的四个元素是输出中阴影部分元素的感受野。我们将图 5.1 中形状为 2×2 的输出记为 Y ，并考虑一个更深的卷积神经网络：将 Y 与另一个形状为 2×2 的核数组做互相关运算，输出单个元素 z 。那么， z 在 Y 上的感受野包括 Y 的全部四个元素，在输入上的感受野包括其中全部 9 个元素。可见，我们可以通过更深的卷积神经网络使特征图中单个元素的感受野变得更加广阔，从而捕捉输入上更大尺寸的特征。

我们常使用“元素”一词来描述数组或矩阵中的成员。在神经网络的术语中，这些元素也可称为“单元”。当含义明确时，本书不对这两个术语做严格区分。

小结

- 二维卷积层的核心计算是二维互相关运算。在最简单的形式下，它对二维输入数据和卷积核做互相关运算然后加上偏差。
 - 我们可以设计卷积核来检测图像中的边缘。
 - 我们可以通过数据来学习卷积核。
-

注：除代码外本节与原书此节基本相同，[原书传送门](#)

5.2 填充和步幅

在上一节的例子里，我们使用高和宽为 3 的输入与高和宽为 2 的卷积核得到高和宽为 2 的输出。一般来说，假设输入形状是 $n_h \times n_w$ ，卷积核窗口形状是 $k_h \times k_w$ ，那么输出形状将会是

$$(n_h - k_h + 1) \times (n_w - k_w + 1).$$

所以卷积层的输出形状由输入形状和卷积核窗口形状决定。本节我们将介绍卷积层的两个超参数，即填充和步幅。它们可以对给定形状的输入和卷积核改变输出形状。

5.2.1 填充

填充 (padding) 是指在输入高和宽的两侧填充元素（通常是 0 元素）。图 5.2 里我们在原输入高和宽的两侧分别添加了值为 0 的元素，使得输入高和宽从 3 变成了 5，并导致输出高和宽由 2 增加到 4。图 5.2 中的阴影部分为第一个输出元素及其计算所使用的输入和核数组元素： $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$ 。

图 5.2 在输入的高和宽两侧分别填充了 0 元素的二维互相关计算

一般来说，如果在高的两侧一共填充 p_h 行，在宽的两侧一共填充 p_w 列，那么输出形状将会是

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1),$$

也就是说，输出的高和宽会分别增加 p_h 和 p_w 。

在很多情况下，我们会设置 $p_h = k_h - 1$ 和 $p_w = k_w - 1$ 来使输入和输出具有相同的高和宽。这样会方便在构造网络时推测每个层的输出形状。假设这里 k_h 是奇数，我们会在高的两侧分别填充 $p_h/2$ 行。如果 k_h 是偶数，一种可能是在输入的顶端一侧填充 $\lceil p_h/2 \rceil$ 行，而在底端一侧填充 $\lfloor p_h/2 \rfloor$ 行。在宽的两侧填充同理。

卷积神经网络经常使用奇数高宽的卷积核，如 1、3、5 和 7，所以两端上的填充个数相等。对任意的二维数组 X ，设它的第 i 行第 j 列的元素为 $X[i, j]$ 。当两端上的填充个数相等，并使输入和输出具有相同的高和宽时，我们就知道输出 $Y[i, j]$ 是由输入以 $X[i, j]$ 为中心的窗口同卷积核进行互相关计算得到的。

下面的例子里我们创建一个高和宽为 3 的二维卷积层，然后设输入高和宽两侧的填充数分别为 1。给定一个高和宽为 8 的输入，我们发现输出的高和宽也是 8。

```
import torch
from torch import nn
```

```
# 定义一个函数来计算卷积层。它对输入和输出做相应的升维和降维
def comp_conv2d(conv2d, X):
    # (1, 1) 代表批量大小和通道数（“多输入通道和多输出通道”一节将介绍）均为 1
    X = X.view((1, 1) + X.shape)
    Y = conv2d(X)
    return Y.view(Y.shape[2:])
# 注意这里是两侧分别填充 1 行或列，所以在两侧一共填充 2 行或列
conv2d = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3, padding=1)

X = torch.rand(8, 8)
comp_conv2d(conv2d, X).shape
```

输出：

```
torch.Size([8, 8])
```

当卷积核的高和宽不同时，我们也可以通过设置高和宽上不同的填充数使输出和输入具有相同的高和宽。

```
# 使用高为 5、宽为 3 的卷积核。在高和宽两侧的填充数分别为 2 和 1
conv2d = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

输出：

```
torch.Size([8, 8])
```

5.2.2 步幅

在上一节里我们介绍了二维互相关运算。卷积窗口从输入数组的最左上方开始，按从左往右、从上往下的顺序，依次在输入数组上滑动。我们将每次滑动的行数和列数称为步幅（stride）。

目前我们看到的例子里，在高和宽两个方向上步幅均为 1。我们也可以使用更大步幅。图 5.3 展示了在高上步幅为 3、在宽上步幅为 2 的二维互相关运算。可以看到，输出第一列第二个元素时，卷积窗口向下滑动了 3 行，而在输出第一行第二个元素时卷积窗口向右滑动了 2 列。当卷积窗口在输入上再向右滑动 2 列时，由于输入元素无法填

满窗口，无结果输出。图 5.3 中的阴影部分为输出元素及其计算所使用的输入和核数组元素： $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$ 、 $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$ 。

图 5.3 高和宽上步幅分别为 3 和 2 的二维互相关运算

一般来说，当高上步幅为 s_h ，宽上步幅为 s_w 时，输出形状为

$$\lfloor (n_h - k_h + p_h + s_h) / s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w) / s_w \rfloor.$$

如果设置 $p_h = k_h - 1$ 和 $p_w = k_w - 1$ ，那么输出形状将简化为 $\lfloor (n_h + s_h - 1) / s_h \rfloor \times \lfloor (n_w + s_w - 1) / s_w \rfloor$ 。更进一步，如果输入的高和宽能分别被高和宽上的步幅整除，那么输出形状将是 $(n_h / s_h) \times (n_w / s_w)$ 。

下面我们令高和宽上的步幅均为 2，从而使输入的高和宽减半。

```
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

输出：

```
torch.Size([4, 4])
```

接下来是一个稍微复杂点儿的例子。

```
conv2d = nn.Conv2d(1, 1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
```

输出：

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

为了表述简洁，当输入的高和宽两侧的填充数分别为 p_h 和 p_w 时，我们称填充为 (p_h, p_w) 。特别地，当 $p_h = p_w = p$ 时，填充为 p 。当在高和宽上的步幅分别为 s_h 和 s_w 时，我们称步幅为 (s_h, s_w) 。特别地，当 $s_h = s_w = s$ 时，步幅为 s 。在默认情况下，填充为 0，步幅为 1。

小结

- 填充可以增加输出的高和宽。这常用来使输出与输入具有相同的高和宽。
- 步幅可以减小输出的高和宽，例如输出的高和宽仅为输入的高和宽的 $1/n$ (n 为大于 1 的整数)。

5.3 多输入通道和多输出通道

前面两节里我们用到的输入和输出都是二维数组，但真实数据的维度经常更高。例如，彩色图像在高和宽 2 个维度外还有 RGB（红、绿、蓝）3 个颜色通道。假设彩色图像的高和宽分别是 h 和 w （像素），那么它可以表示为一个 $3 \times h \times w$ 的多维数组。我们将大小为 3 的这一维称为通道（channel）维。本节我们将介绍含多个输入通道或多个输出通道的卷积核。

5.3.1 多输入通道

当输入数据含多个通道时，我们需要构造一个输入通道数与输入数据的通道数相同的卷积核，从而能够与含多通道的输入数据做互相关运算。假设输入数据的通道数为 c_i ，那么卷积核的输入通道数同样为 c_i 。设卷积核窗口形状为 $k_h \times k_w$ 。当 $c_i = 1$ 时，我们知道卷积核只包含一个形状为 $k_h \times k_w$ 的二维数组。当 $c_i > 1$ 时，我们将会为每个输入通道各分配一个形状为 $k_h \times k_w$ 的核数组。把这 c_i 个数组在输入通道维上连结，即得到一个形状为 $c_i \times k_h \times k_w$ 的卷积核。由于输入和卷积核各有 c_i 个通道，我们可以在各个通道上对输入的二维数组和卷积核的二维核数组做互相关运算，再将这 c_i 个互相关运算的二维输出按通道相加，得到一个二维数组。这就是含多个通道的输入数据与多输入通道的卷积核做二维互相关运算的输出。

图 5.4 展示了含 2 个输入通道的二维互相关计算的例子。在每个通道上，二维输入数组与二维核数组做互相关运算，再按通道相加即得到输出。图 5.4 中阴影部分为第一个输出元素及其计算所使用的输入和核数组元素： $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$ 。

图 5.4 含 2 个输入通道的互相关计算

接下来我们实现含多个输入通道的互相关运算。我们只需要对每个通道做互相关运算，然后通过 `add_n` 函数来进行累加。

```
import torch
from torch import nn
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l

def corr2d_multi_in(X, K):
    # 沿着 X 和 K 的第 0 维（通道维）分别计算再相加
    res = d2l.corr2d(X[0, :, :], K[0, :, :])
    for i in range(1, X.shape[0]):
        res = d2l.add_n(res, d2l.corr2d(X[i, :, :], K[i, :, :]))
    return res
```

```
for i in range(1, X.shape[0]):
    res += d2l.corr2d(X[i, :, :], K[i, :, :])
return res
```

我们可以构造图 5.4 中的输入数组 X 、核数组 K 来验证互相关运算的输出。

```
X = torch.tensor([[0, 1, 2], [3, 4, 5], [6, 7, 8]],
                 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]])
K = torch.tensor([[0, 1], [2, 3]], [[1, 2], [3, 4]]])

corr2d_multi_in(X, K)
```

输出：

```
tensor([[ 56.,  72.],
        [104., 120.]])
```

5.3.2 多输出通道

当输入通道有多个时，因为我们将各个通道的结果做了累加，所以不论输入通道数是多少，输出通道数总是为 1。设卷积核输入通道数和输出通道数分别为 c_i 和 c_o ，高和宽分别为 k_h 和 k_w 。如果希望得到含多个通道的输出，我们可以为每个输出通道分别创建形状为 $c_i \times k_h \times k_w$ 的核数组。将它们在输出通道维上连结，卷积核的形状即 $c_o \times c_i \times k_h \times k_w$ 。在做互相关运算时，每个输出通道上的结果由卷积核在该输出通道上的核数组与整个输入数组计算而来。

下面我们实现一个互相关运算函数来计算多个通道的输出。

```
def corr2d_multi_in_out(X, K):
    # 对 K 的第 0 维遍历，每次同输入 X 做互相关计算。所有结果使用 stack 函数合并在一起
    return torch.stack([corr2d_multi_in(X, k) for k in K])
```

我们将核数组 K 同 $K+1$ (K 中每个元素加一) 和 $K+2$ 连结在一起构造一个输出通道数为 3 的卷积核。

```
K = torch.stack([K, K + 1, K + 2])
K.shape # torch.Size([3, 2, 2, 2])
```

下面我们对输入数组 X 与核数组 K 做互相关运算。此时的输出含有 3 个通道。其中第一个通道的结果与之前输入数组 X 与多输入通道、单输出通道核的计算结果一致。

```
corr2d_multi_in_out(X, K)
```

输出：

```
tensor([[[ 56.,  72.],
        [104., 120.]],

       [[ 76., 100.],
        [148., 172.]],

       [[ 96., 128.],
        [192., 224.]]])
```

5.3.3 1×1 卷积层

最后我们讨论卷积窗口形状为 1×1 ($k_h = k_w = 1$) 的多通道卷积层。我们通常称之为 1×1 卷积层，并将其中的卷积运算称为 1×1 卷积。因为使用了最小窗口， 1×1 卷积失去了卷积层可以识别高和宽维度上相邻元素构成的模式的功能。实际上， 1×1 卷积的主要计算发生在通道维上。图 5.5 展示了使用输入通道数为 3、输出通道数为 2 的 1×1 卷积核的互相关计算。值得注意的是，输入和输出具有相同的高和宽。输出中的每个元素来自输入中在高和宽上相同位置的元素在不同通道之间的按权重累加。假设我们将通道维当作特征维，将高和宽维度上的元素当成数据样本，那么 1×1 卷积层的作用与全连接层等价。

图 5.5 1×1 卷积核的互相关计算。输入和输出具有相同的高和宽

下面我们使用全连接层中的矩阵乘法来实现 1×1 卷积。这里需要在矩阵乘法运算前后对数据形状做一些调整。

```
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = X.view(c_i, h * w)
    K = K.view(c_o, c_i)
    Y = torch.mm(K, X)  # 全连接层的矩阵乘法
    return Y.view(c_o, h, w)
```

经验证，做 1×1 卷积时，以上函数与之前实现的互相关运算函数 `corr2d_multi_in_out` 等价。

```
X = torch.rand(3, 3, 3)
K = torch.rand(2, 3, 1, 1)

Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)

(Y1 - Y2).norm().item() < 1e-6
```

输出：

True

在之后的模型里我们将会看到 1×1 卷积层被当作保持高和宽维度形状不变的全连接层使用。于是，我们可以通过调整网络层之间的通道数来控制模型复杂度。

小结

- 使用多通道可以拓展卷积层的模型参数。
- 假设将通道维当作特征维，将高和宽维度上的元素当成数据样本，那么 1×1 卷积层的作用与全连接层等价。
- 1×1 卷积层通常用来调整网络层之间的通道数，并控制模型复杂度。

注：除代码外本节与原书此节基本相同，[原书传送门](#)

5.4 池化层

回忆一下，在 5.1 节（二维卷积层）里介绍的图像物体边缘检测应用中，我们构造卷积核从而精确地找到了像素变化的位置。设任意二维数组 X 的 i 行 j 列的元素为 $X[i, j]$ 。如果我们构造的卷积核输出 $Y[i, j]=1$ ，那么说明输入中 $X[i, j]$ 和 $X[i, j+1]$ 数值不一样。这可能意味着物体边缘通过这两个元素之间。但实际图像里，我们感兴趣的物体不会总出现在固定位置：即使我们连续拍摄同一个物体也极有可能出现像素位置上的偏移。这会导致同一个边缘对应的输出可能出现在卷积输出 Y 中的不同位置，进而对后面的模式识别造成不便。

在本节中我们介绍池化（pooling）层，它的提出是为了缓解卷积层对位置的过度敏感性。

5.4.1 二维最大池化层和平均池化层

同卷积层一样，池化层每次对输入数据的一个固定形状窗口（又称池化窗口）中的元素计算输出。不同于卷积层里计算输入和核的互相关性，池化层直接计算池化窗口内元素的最大值或者平均值。该运算也分别叫做最大池化或平均池化。在二维最大池化中，池化窗口从输入数组的最左上方开始，按从左往右、从上往下的顺序，依次在输入数组上滑动。当池化窗口滑动到某一位置时，窗口中的输入子数组的最大值即输出数组中相应位置的元素。

图 5.6 池化窗口形状为 2×2 的最大池化

图 5.6 展示了池化窗口形状为 2×2 的最大池化，阴影部分为第一个输出元素及其计算所使用的输入元素。输出数组的高和宽分别为 2，其中的 4 个元素由取最大值运算 \max 得出：

$$\max(0, 1, 3, 4) = 4, \max(1, 2, 4, 5) = 5, \max(3, 4, 6, 7) = 7, \max(4, 5, 7, 8) = 8.$$

二维平均池化的工作原理与二维最大池化类似，但将最大运算符替换成平均运算符。池化窗口形状为 $p \times q$ 的池化层称为 $p \times q$ 池化层，其中的池化运算叫作 $p \times q$ 池化。

让我们再次回到本节开始提到的物体边缘检测的例子。现在我们将卷积层的输出作为 2×2 最大池化的输入。设该卷积层输入是 X 、池化层输出为 Y 。无论是 $X[i, j]$ 和 $X[i, j+1]$ 值不同，还是 $X[i, j+1]$ 和 $X[i, j+2]$ 不同，池化层输出均有 $Y[i, j]=1$ 。也就是说，使用 2×2 最大池化层时，只要卷积层识别的模式在高和宽上移动不超过一个元素，我们依然可以将它检测出来。

下面把池化层的前向计算实现在 `pool2d` 函数里。它跟 5.1 节（二维卷积层）里 `corr2d` 函数非常类似，唯一的区别在计算输出 `Y` 上。

```
import torch
from torch import nn

def pool2d(X, pool_size, mode='max'):
    X = X.float()
    p_h, p_w = pool_size
    Y = torch.zeros(X.shape[0] - p_h + 1, X.shape[1] - p_w + 1)
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i: i + p_h, j: j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y
```

我们可以构造图 5.6 中的输入数组 `X` 来验证二维最大池化层的输出。

```
X = torch.tensor([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
pool2d(X, (2, 2))
```

输出:

```
tensor([[4., 5.],
       [7., 8.]])
```

同时我们实验一下平均池化层。

```
pool2d(X, (2, 2), 'avg')
```

输出:

```
tensor([[2., 3.],
       [5., 6.]])
```

5.4.2 填充和步幅

同卷积层一样，池化层也可以在输入的高和宽两侧的填充并调整窗口的移动步幅来改变输出形状。池化层填充和步幅与卷积层填充和步幅的工作机制一样。我们将通过 `nn` 模块里的二维最大池化层 `MaxPool2d` 来演示池化层填充和步幅的工作机制。我们先构造一个形状为 $(1, 1, 4, 4)$ 的输入数据，前两个维度分别是批量和通道。

```
X = torch.arange(16, dtype=torch.float).view((1, 1, 4, 4))  
X
```

输出：

```
tensor([[[[ 0.,  1.,  2.,  3.],  
         [ 4.,  5.,  6.,  7.],  
         [ 8.,  9., 10., 11.],  
         [12., 13., 14., 15.]]]])
```

默认情况下，`MaxPool2d` 实例里步幅和池化窗口形状相同。下面使用形状为 $(3, 3)$ 的池化窗口，默认获得形状为 $(3, 3)$ 的步幅。

```
pool2d = nn.MaxPool2d(3)  
pool2d(X)
```

输出：

```
tensor([[[[10.]]]])
```

我们可以手动指定步幅和填充。

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)  
pool2d(X)
```

输出：

```
tensor([[[[ 5.,  7.],  
         [13., 15.]]]])
```

当然，我们也可以指定非正方形的池化窗口，并分别指定高和宽上的填充和步幅。

```
pool2d = nn.MaxPool2d((2, 4), padding=(1, 2), stride=(2, 3))  
pool2d(X)
```

输出：

```
tensor([[[[ 1.,  3.],  
         [ 9., 11.],  
         [13., 15.]]]])
```

5.4.3 多通道

在处理多通道输入数据时，池化层对每个输入通道分别池化，而不是像卷积层那样将各通道的输入按通道相加。这意味着池化层的输出通道数与输入通道数相等。下面将数组 `X` 和 `X+1` 在通道维上连结来构造通道数为 2 的输入。

```
X = torch.cat((X, X + 1), dim=1)  
X
```

输出：

```
tensor([[[[ 0.,  1.,  2.,  3.],  
         [ 4.,  5.,  6.,  7.],  
         [ 8.,  9., 10., 11.],  
         [12., 13., 14., 15.]],  
  
        [[[ 1.,  2.,  3.,  4.],  
          [ 5.,  6.,  7.,  8.],  
          [ 9., 10., 11., 12.],  
          [13., 14., 15., 16.]]]])
```

池化后，我们发现输出通道数仍然是 2。

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)  
pool2d(X)
```

输出：

```
tensor([[[[ 5.,  7.],  
         [13., 15.]],  
  
        [[[ 6.,  8.],  
          [14., 16.]]]])
```

小结

- 最大池化和平均池化分别取池化窗口中输入元素的最大值和平均值作为输出。
- 池化层的一个主要作用是缓解卷积层对位置的过度敏感性。
- 可以指定池化层的填充和步幅。

- 池化层的输出通道数跟输入通道数相同。

注：除代码外本节与原书此节基本相同，[原书传送门](#)

5.5 卷积神经网络 (LeNet)

在 3.9 节 (多层感知机的从零开始实现) 里我们构造了一个含单隐藏层的多层感知机模型来对 Fashion-MNIST 数据集中的图像进行分类。每张图像高和宽均是 28 像素。我们将图像中的像素逐行展开，得到长度为 784 的向量，并输入进全连接层中。然而，这种分类方法有一定的局限性。

1. 图像在同一列邻近的像素在这个向量中可能相距较远。它们构成的模式可能难以被模型识别。
2. 对于大尺寸的输入图像，使用全连接层容易造成模型过大。假设输入是高和宽均为 1000 像素的彩色照片 (含 3 个通道)。即使全连接层输出个数仍是 256，该层权重参数的形状是 $3,000,000 \times 256$ ：它占用了大约 3 GB 的内存或显存。这带来过复杂的模型和过高的存储开销。

卷积层尝试解决这两个问题。一方面，卷积层保留输入形状，使图像的像素在高和宽两个方向上的相关性均可能被有效识别；另一方面，卷积层通过滑动窗口将同一卷积核与不同位置的输入重复计算，从而避免参数尺寸过大。

卷积神经网络就是含卷积层的网络。本节里我们将介绍一个早期用来识别手写数字图像的卷积神经网络：LeNet [1]。这个名字来源于 LeNet 论文的第一作者 Yann LeCun。LeNet 展示了通过梯度下降训练卷积神经网络可以达到手写数字识别在当时最先进的结果。这个奠基性的工作第一次将卷积神经网络推上舞台，为世人所知。LeNet 的网络结构如下图所示。

LeNet 网络结构

5.5.1 LeNet 模型

LeNet 分为卷积层块和全连接层块两个部分。下面我们分别介绍这两个模块。

卷积层块里的基本单位是卷积层后接最大池化层：卷积层用来识别图像里的空间模式，如线条和物体局部，之后的最大池化层则用来降低卷积层对位置的敏感性。卷积层块由两个这样的基本单位重复堆叠构成。在卷积层块中，每个卷积层都使用 5×5 的窗口，并在输出上使用 sigmoid 激活函数。第一个卷积层输出通道数为 6，第二个卷积层输出通道数则增加到 16。这是因为第二个卷积层比第一个卷积层的输入的高和宽要小，所以增加输出通道使两个卷积层的参数尺寸类似。卷积层块的两个最大池化层的窗口形状均为 2×2 ，且步幅为 2。由于池化窗口与步幅形状相同，池化窗口在输入上每次滑动所覆盖的区域互不重叠。

卷积层块的输出形状为 (批量大小, 通道, 高, 宽)。当卷积层块的输出传入全连接层块时，全连接层块会将小批量中每个样本变平 (flatten)。也就是说，全连接层的输入形

状将变成二维，其中第一维是小批量中的样本，第二维是每个样本变平后的向量表示，且向量长度为通道、高和宽的乘积。全连接层块含 3 个全连接层。它们的输出个数分别是 120、84 和 10，其中 10 为输出的类别个数。

下面我们通过 `Sequential` 类来实现 LeNet 模型。

```
import time
import torch
from torch import nn, optim

import sys
sys.path.append("../")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, 6, 5), # in_channels, out_channels, kernel_size
            nn.Sigmoid(),
            nn.MaxPool2d(2, 2), # kernel_size, stride
            nn.Conv2d(6, 16, 5),
            nn.Sigmoid(),
            nn.MaxPool2d(2, 2)
        )
        self.fc = nn.Sequential(
            nn.Linear(16*4*4, 120),
            nn.Sigmoid(),
            nn.Linear(120, 84),
            nn.Sigmoid(),
            nn.Linear(84, 10)
        )

    def forward(self, img):
        feature = self.conv(img)
        output = self.fc(feature.view(img.shape[0], -1))
```

```
    return output
```

接下来查看每个层的形状。

```
net = LeNet()  
print(net)
```

输出：

```
LeNet(  
  (conv): Sequential(  
    (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))  
    (1): Sigmoid()  
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    (3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))  
    (4): Sigmoid()  
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
  )  
  (fc): Sequential(  
    (0): Linear(in_features=256, out_features=120, bias=True)  
    (1): Sigmoid()  
    (2): Linear(in_features=120, out_features=84, bias=True)  
    (3): Sigmoid()  
    (4): Linear(in_features=84, out_features=10, bias=True)  
  )  
)
```

可以看到，在卷积层块中输入的高和宽在逐层减小。卷积层由于使用高和宽均为 5 的卷积核，从而将高和宽分别减小 4，而池化层则将高和宽减半，但通道数则从 1 增加到 16。全连接层则逐层减少输出个数，直到变成图像的类别数 10。

5.5.2 获得数据和训练模型

下面我们来实验 LeNet 模型。实验中，我们仍然使用 Fashion-MNIST 作为训练数据集。

```
batch_size = 256  
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)
```

因为卷积神经网络计算比多层感知机要复杂，建议使用 GPU 来加速计算。因此，我们对 3.6 节 (softmax 回归的从零开始实现) 中描述的 `evaluate_accuracy` 函数略作修改，使其支持 GPU 计算。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用。该函数将被逐步改进。
def evaluate_accuracy(data_iter, net,
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')):
    acc_sum, n = 0.0, 0
    with torch.no_grad():
        for X, y in data_iter:
            if isinstance(net, torch.nn.Module):
                net.eval() # 评估模式，这会关闭 dropout
                acc_sum += (net(X.to(device)).argmax(dim=1) == y.to(device)).float()
                net.train() # 改回训练模式
            else: # 自定义的模型，3.13 节之后不会用到，不考虑 GPU
                if('is_training' in net.__code__.co_varnames): # 如果有 is_training 这
                    # 将 is_training 设置成 False
                acc_sum += (net(X, is_training=False).argmax(dim=1) == y).float()
            else:
                acc_sum += (net(X).argmax(dim=1) == y).float().sum().item()
            n += y.shape[0]
    return acc_sum / n
```

我们同样对 3.6 节中定义的 `train_ch3` 函数略作修改，确保计算使用的数据和模型同在内存或显存上。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs):
    net = net.to(device)
    print("training on ", device)
    loss = torch.nn.CrossEntropyLoss()
    batch_count = 0
    for epoch in range(num_epochs):
        train_l_sum, train_acc_sum, n, start = 0.0, 0.0, 0, time.time()
        for X, y in train_iter:
            X = X.to(device)
            y = y.to(device)
```

```

y_hat = net(X)
l = loss(y_hat, y)
optimizer.zero_grad()
l.backward()
optimizer.step()
train_l_sum += l.cpu().item()
train_acc_sum += (y_hat.argmax(dim=1) == y).sum().cpu().item()
n += y.shape[0]
batch_count += 1
test_acc = evaluate_accuracy(test_iter, net)
print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f, time %.1f sec'
      % (epoch + 1, train_l_sum / batch_count, train_acc_sum / n, test_acc, t)
    )

```

学习率采用 0.001，训练算法使用 Adam 算法，损失函数使用交叉熵损失函数。

```

lr, num_epochs = 0.001, 5
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)

```

输出：

```

training on  cuda
epoch 1, loss 0.0072, train acc 0.322, test acc 0.584, time 3.7 sec
epoch 2, loss 0.0037, train acc 0.649, test acc 0.699, time 1.8 sec
epoch 3, loss 0.0030, train acc 0.718, test acc 0.724, time 1.7 sec
epoch 4, loss 0.0027, train acc 0.741, test acc 0.746, time 1.6 sec
epoch 5, loss 0.0024, train acc 0.759, test acc 0.759, time 1.7 sec

```

注：本节代码在 GPU 和 CPU 上都已测试过。

小结

- 卷积神经网络就是含卷积层的网络。
- LeNet 交替使用卷积层和最大池化层后接全连接层来进行图像分类。

参考文献

- [1] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

注：除代码外本节与原书此节基本相同，[原书传送门](#)

5.6 深度卷积神经网络 (AlexNet)

在 LeNet 提出后的将近 20 年里，神经网络一度被其他机器学习方法超越，如支持向量机。虽然 LeNet 可以在早期的小数据集上取得好的成绩，但是在更大的真实数据集上的表现并不尽如人意。一方面，神经网络计算复杂。虽然 20 世纪 90 年代也有过一些针对神经网络的加速硬件，但并没有像之后 GPU 那样大量普及。因此，训练一个多通道、多层和有大量参数的卷积神经网络在当年很难完成。另一方面，当年研究者还没有大量深入研究参数初始化和非凸优化算法等诸多领域，导致复杂的神经网络的训练通常较困难。

我们在上一节看到，神经网络可以直接基于图像的原始像素进行分类。这种称为端到端 (end-to-end) 的方法节省了很多中间步骤。然而，在很长一段时间里更流行的是研究者通过勤劳与智慧所设计并生成的手工特征。这类图像分类研究的主要流程是：

1. 获取图像数据集；
2. 使用已有的特征提取函数生成图像的特征；
3. 使用机器学习模型对图像的特征分类。

当时认为的机器学习部分仅限最后一步。如果那时候跟机器学习研究者交谈，他们会认为机器学习既重要又优美。优雅的定理证明了许多分类器的性质。机器学习领域生机勃勃、严谨而且极其有用。然而，如果跟计算机视觉研究者交谈，则是另外一幅景象。他们会告诉你图像识别里“不可告人”的现实是：计算机视觉流程中真正重要的是数据和特征。也就是说，使用较干净的数据集和较有效的特征甚至比机器学习模型的选择对图像分类结果的影响更大。

5.6.1 学习特征表示

既然特征如此重要，它该如何表示呢？

我们已经提到，在相当长的时间里，特征都是基于各式各样手工设计的函数从数据中提取的。事实上，不少研究者通过提出新的特征提取函数不断改进图像分类结果。这一度为计算机视觉的发展做出了重要贡献。

然而，另一些研究者则持异议。他们认为特征本身也应该由学习得来。他们还相信，为了表征足够复杂的输入，特征本身应该分级表示。持这一想法的研究者相信，多层神经网络可能可以学得数据的多级表征，并逐级表示越来越抽象的概念或模式。以图像分类为例，并回忆 5.1 节（二维卷积层）中物体边缘检测的例子。在多层神经网络中，图像的第一级的表示可以是在特定的位置和角度是否出现边缘；而第二级的表示说不定能够将这些边缘组合出有趣的模式，如花纹；在第三级的表示中，也许上一级的花纹能进一步汇合成对应物体特定部位的模式。这样逐级表示下去，最终，模型能够较容易根

据最后一级的表示完成分类任务。需要强调的是，输入的逐级表示由多层模型中的参数决定，而这些参数都是学出来的。

尽管一直有一群执着的研究者不断钻研，试图学习视觉数据的逐级表征，然而很长一段时间里这些野心都未能实现。这其中有着诸多因素值得我们一一分析。

5.6.1.1 缺失要素一：数据

包含许多特征的深度模型需要大量的有标签的数据才能表现得比其他经典方法更好。限于早期计算机有限的存储和 90 年代有限的研究预算，大部分研究只基于小的公开数据集。例如，不少研究论文基于加州大学欧文分校 (UCI) 提供的若干个公开数据集，其中许多数据集只有几百至几千张图像。这一状况在 2010 年前后兴起的大数据浪潮中得到改善。特别是，2009 年诞生的 ImageNet 数据集包含了 1,000 大类物体，每类有多达数千张不同的图像。这一规模是当时其他公开数据集无法与之相提并论的。ImageNet 数据集同时推动计算机视觉和机器学习研究进入新的阶段，使此前的传统方法不再有优势。

5.6.1.2 缺失要素二：硬件

深度学习对计算资源要求很高。早期的硬件计算能力有限，这使训练较复杂的神经网络变得很困难。然而，通用 GPU 的到来改变了这一格局。很久以来，GPU 都是为图像处理和计算机游戏设计的，尤其是针对大吞吐量的矩阵和向量乘法从而服务于基本的图形变换。值得庆幸的是，这其中的数学表达与深度网络中的卷积层的表达类似。通用 GPU 这个概念在 2001 年开始兴起，涌现出诸如 OpenCL 和 CUDA 之类的编程框架。这使得 GPU 也在 2010 年前后开始被机器学习社区使用。

5.6.2 AlexNet

2012 年，AlexNet 横空出世。这个模型的名字来源于论文第一作者的姓名 Alex Krizhevsky [1]。AlexNet 使用了 8 层卷积神经网络，并以很大的优势赢得了 ImageNet 2012 图像识别挑战赛。它首次证明了学习到的特征可以超越手工设计的特征，从而一举打破计算机视觉研究的前状。

AlexNet 网络结构

AlexNet 与 LeNet 的设计理念非常相似，但也有显著的区别。

第一，与相对较小的 LeNet 相比，AlexNet 包含 8 层变换，其中有 5 层卷积和 2 层全连接隐藏层，以及 1 个全连接输出层。下面我们来详细描述这些层的设计。

AlexNet 第一层中的卷积窗口形状是 11×11 。因为 ImageNet 中绝大多数图像的高和宽均比 MNIST 图像的高和宽大 10 倍以上，ImageNet 图像的物体占用更多的像素，

所以需要更大的卷积窗口来捕获物体。第二层中的卷积窗口形状减小到 5×5 ，之后全采用 3×3 。此外，第一、第二和第五个卷积层之后都使用了窗口形状为 3×3 、步幅为 2 的最大池化层。而且，AlexNet 使用的卷积通道数也大于 LeNet 中的卷积通道数数十倍。

紧接着最后一个卷积层的是两个输出个数为 4096 的全连接层。这两个巨大的全连接层带来将近 1 GB 的模型参数。由于早期显存的限制，最早的 AlexNet 使用双数据流的设计使一个 GPU 只需要处理一半模型。幸运的是，显存在过去几年得到了长足的发展，因此通常我们不再需要这样的特别设计了。

第二，AlexNet 将 sigmoid 激活函数改成了更加简单的 ReLU 激活函数。一方面，ReLU 激活函数的计算更简单，例如它并没有 sigmoid 激活函数中的求幂运算。另一方面，ReLU 激活函数在不同的参数初始化方法下使模型更容易训练。这是由于当 sigmoid 激活函数输出极接近 0 或 1 时，这些区域的梯度几乎为 0，从而造成反向传播无法继续更新部分模型参数；而 ReLU 激活函数在正区间的梯度恒为 1。因此，若模型参数初始化不当，sigmoid 函数可能在正区间得到几乎为 0 的梯度，从而令模型无法得到有效训练。

第三，AlexNet 通过丢弃法（参见 3.13 节）来控制全连接层的模型复杂度。而 LeNet 并没有使用丢弃法。

第四，AlexNet 引入了大量的图像增广，如翻转、裁剪和颜色变化，从而进一步扩大数据集来缓解过拟合。我们将在后面的 9.1 节（图像增广）详细介绍这种方法。

下面我们实现稍微简化过的 AlexNet。

```
import time
import torch
from torch import nn, optim
import torchvision

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, 96, 11, 4), # in_channels, out_channels, kernel_size, stride
```

```

        nn.ReLU(),
        nn.MaxPool2d(3, 2), # kernel_size, stride
        # 减小卷积窗口，使用填充为 2 来使得输入与输出的高和宽一致，且增大输出通道数
        nn.Conv2d(96, 256, 5, 1, 2),
        nn.ReLU(),
        nn.MaxPool2d(3, 2),
        # 连续 3 个卷积层，且使用更小的卷积窗口。除了最后的卷积层外，进一步增大了输出
        # 前两个卷积层后不使用池化层来减小输入的高和宽
        nn.Conv2d(256, 384, 3, 1, 1),
        nn.ReLU(),
        nn.Conv2d(384, 384, 3, 1, 1),
        nn.ReLU(),
        nn.Conv2d(384, 256, 3, 1, 1),
        nn.ReLU(),
        nn.MaxPool2d(3, 2)
    )
    # 这里全连接层的输出个数比 LeNet 中的大数倍。使用丢弃层来缓解过拟合
    self.fc = nn.Sequential(
        nn.Linear(256*5*5, 4096),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(4096, 4096),
        nn.ReLU(),
        nn.Dropout(0.5),
        # 输出层。由于这里使用 Fashion-MNIST，所以用类别数为 10，而非论文中的 1000
        nn.Linear(4096, 10),
    )

    def forward(self, img):
        feature = self.conv(img)
        output = self.fc(feature.view(img.shape[0], -1))
        return output

```

打印看看网络结构。

```

net = AlexNet()
print(net)

```

输出：

```
AlexNet(  
    (conv): Sequential(  
        (0): Conv2d(1, 96, kernel_size=(11, 11), stride=(4, 4))  
        (1): ReLU()  
        (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)  
        (3): Conv2d(96, 256, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))  
        (4): ReLU()  
        (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)  
        (6): Conv2d(256, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (7): ReLU()  
        (8): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (9): ReLU()  
        (10): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (11): ReLU()  
        (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)  
    )  
    (fc): Sequential(  
        (0): Linear(in_features=6400, out_features=4096, bias=True)  
        (1): ReLU()  
        (2): Dropout(p=0.5)  
        (3): Linear(in_features=4096, out_features=4096, bias=True)  
        (4): ReLU()  
        (5): Dropout(p=0.5)  
        (6): Linear(in_features=4096, out_features=10, bias=True)  
    )  
)
```

5.6.3 读取数据

虽然论文中 AlexNet 使用 ImageNet 数据集，但因为 ImageNet 数据集训练时间较长，我们仍用前面的 Fashion-MNIST 数据集来演示 AlexNet。读取数据的时候我们额外做了一步将图像高和宽扩大到 AlexNet 使用的图像高和宽 224。这个可以通过 `torchvision.transforms.Resize` 实例来实现。也就是说，我们在 `ToTensor` 实例前使用 `Resize` 实例，然后使用 `Compose` 实例来将这两个变换串联以方便调用。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def load_data_fashion_mnist(batch_size, resize=None, root='~/Datasets/FashionMNIST'):
    """Download the fashion mnist dataset and then load into memory."""
    trans = []
    if resize:
        trans.append(torchvision.transforms.Resize(size=resize))
    trans.append(torchvision.transforms.ToTensor())

    transform = torchvision.transforms.Compose(trans)
    mnist_train = torchvision.datasets.FashionMNIST(root=root, train=True, download=True)
    mnist_test = torchvision.datasets.FashionMNIST(root=root, train=False, download=True)

    train_iter = torch.utils.data.DataLoader(mnist_train, batch_size=batch_size, shuffle=True)
    test_iter = torch.utils.data.DataLoader(mnist_test, batch_size=batch_size, shuffle=True)

    return train_iter, test_iter

batch_size = 128
# 如出现 "out of memory" 的报错信息，可减小 batch_size 或 resize
train_iter, test_iter = load_data_fashion_mnist(batch_size, resize=224)
```

5.6.4 训练

这时候我们可以开始训练 AlexNet 了。相对于 LeNet，由于图片尺寸变大了而且模型变大了，所以需要更大的显存，也需要更长的训练时间了。

```
lr, num_epochs = 0.001, 5
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)
```

输出：

```
training on  cuda
epoch 1, loss 0.0047, train acc 0.770, test acc 0.865, time 128.3 sec
epoch 2, loss 0.0025, train acc 0.879, test acc 0.889, time 128.8 sec
epoch 3, loss 0.0022, train acc 0.898, test acc 0.901, time 130.4 sec
epoch 4, loss 0.0019, train acc 0.908, test acc 0.900, time 131.4 sec
epoch 5, loss 0.0018, train acc 0.913, test acc 0.902, time 129.9 sec
```

小结

- AlexNet 跟 LeNet 结构类似，但使用了更多的卷积层和更大的参数空间来拟合大规模数据集 ImageNet。它是浅层神经网络和深度神经网络的分界线。
- 虽然看上去 AlexNet 的实现比 LeNet 的实现也就多了几行代码而已，但这个观念上的转变和真正优秀实验结果的产生令学术界付出了很多年。

参考文献

- [1] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).
-

注：除代码外本节与原书此节基本相同，[原书传送门](#)

5.7 使用重复元素的网络（VGG）

AlexNet 在 LeNet 的基础上增加了 3 个卷积层。但 AlexNet 作者对它们的卷积窗口、输出通道数和构造顺序均做了大量的调整。虽然 AlexNet 指明了深度卷积神经网络可以取得出色的结果，但并没有提供简单的规则以指导后来的研究者如何设计新的网络。我们将在本章的后续几节里介绍几种不同的深度网络设计思路。

本节介绍 VGG，它的名字来源于论文作者所在的实验室 Visual Geometry Group [1]。VGG 提出了可以通过重复使用简单的基础块来构建深度模型的思路。

5.7.1 VGG 块

VGG 块的组成规律是：连续使用数个相同的填充为 1、窗口形状为 3×3 的卷积层后接上一个步幅为 2、窗口形状为 2×2 的最大池化层。卷积层保持输入的高和宽不变，而池化层则对其减半。我们使用 `vgg_block` 函数来实现这个基础的 VGG 块，它可以指定卷积层的数量和输入输出通道数。

对于给定的感受野（与输出有关的输入图片的局部大小），采用堆积的小卷积核优于采用大的卷积核，因为可以增加网络深度来保证学习更复杂的模式，而且代价还比较小（参数更少）。例如，在 VGG 中，使用了 3 个 3×3 卷积核来代替 7×7 卷积核，使用了 2 个 3×3 卷积核来代替 5×5 卷积核，这样做的主要目的是在保证具有相同感知野的条件下，提升了网络的深度，在一定程度上提升了神经网络的效果。

```
import time
import torch
from torch import nn, optim

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

def vgg_block(num_convs, in_channels, out_channels):
    blk = []
    for i in range(num_convs):
        if i == 0:
            blk.append(nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1))
        else:
            blk.append(nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1))
    if num_convs > 1:
        blk.append(nn.MaxPool2d(2))
    return nn.Sequential(*blk)
```

```
    blk.append(nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1))
    blk.append(nn.ReLU())
    blk.append(nn.MaxPool2d(kernel_size=2, stride=2)) # 这里会使宽高减半
return nn.Sequential(*blk)
```

5.7.2 VGG 网络

与 AlexNet 和 LeNet 一样，VGG 网络由卷积层模块后接全连接层模块构成。卷积层模块串联数个 `vgg_block`，其超参数由变量 `conv_arch` 定义。该变量指定了每个 VGG 块里卷积层个数和输入输出通道数。全连接模块则跟 AlexNet 中的一样。

现在我们构造一个 VGG 网络。它有 5 个卷积块，前 2 块使用单卷积层，而后 3 块使用双卷积层。第一块的输入输出通道分别是 1（因为下面要使用的 Fashion-MNIST 数据的通道数为 1）和 64，之后每次对输出通道数翻倍，直到变为 512。因为这个网络使用了 8 个卷积层和 3 个全连接层，所以经常被称为 VGG-11。

```

conv_arch = ((1, 1, 64), (1, 64, 128), (2, 128, 256), (2, 256, 512), (2, 512, 512))
# 经过 5 个 vgg_block, 宽高会减半 5 次, 变成  $224/32 = 7$ 
fc_features = 512 * 7 * 7 # c * w * h
fc_hidden_units = 4096 # 任意

```

下面我们实现 VGG-11。

```

        nn.Linear(fc_hidden_units, 10)
    ))
return net

```

下面构造一个高和宽均为 224 的单通道数据样本来观察每一层的输出形状。

```

net = vgg(conv_arch, fc_features, fc_hidden_units)
X = torch.rand(1, 1, 224, 224)

```

```

# named_children 获取一级子模块及其名字 (named_modules 会返回所有子模块, 包括子模块的子
for name, blk in net.named_children():
    X = blk(X)
    print(name, 'output shape: ', X.shape)

```

输出：

```

vgg_block_1 output shape: torch.Size([1, 64, 112, 112])
vgg_block_2 output shape: torch.Size([1, 128, 56, 56])
vgg_block_3 output shape: torch.Size([1, 256, 28, 28])
vgg_block_4 output shape: torch.Size([1, 512, 14, 14])
vgg_block_5 output shape: torch.Size([1, 512, 7, 7])
fc output shape: torch.Size([1, 10])

```

可以看到，每次我们将输入的高和宽减半，直到最终高和宽变成 7 后传入全连接层。与此同时，输出通道数每次翻倍，直到变成 512。因为每个卷积层的窗口大小一样，所以每层的模型参数尺寸和计算复杂度与输入高、输入宽、输入通道数和输出通道数的乘积成正比。VGG 这种高和宽减半以及通道翻倍的设计使得多数卷积层都有相同的模型参数尺寸和计算复杂度。

5.7.3 获得数据和训练模型

因为 VGG-11 计算上比 AlexNet 更加复杂，出于测试的目的我们构造一个通道数更小，或者说更窄的网络在 Fashion-MNIST 数据集上进行训练。

```

ratio = 8
small_conv_arch = [(1, 1, 64//ratio), (1, 64//ratio, 128//ratio), (2, 128//ratio, 256//ratio),
                   (2, 256//ratio, 512//ratio), (2, 512//ratio, 512//ratio)]
net = vgg(small_conv_arch, fc_features // ratio, fc_hidden_units // ratio)
print(net)

```

输出：

```
Sequential(  
    (vgg_block_1): Sequential(  
        (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): ReLU()  
        (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    )  
    (vgg_block_2): Sequential(  
        (0): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): ReLU()  
        (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    )  
    (vgg_block_3): Sequential(  
        (0): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): ReLU()  
        (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (3): ReLU()  
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    )  
    (vgg_block_4): Sequential(  
        (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): ReLU()  
        (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (3): ReLU()  
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    )  
    (vgg_block_5): Sequential(  
        (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): ReLU()  
        (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (3): ReLU()  
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    )  
    (fc): Sequential(  
        (0): FlattenLayer()
```

```

(1): Linear(in_features=3136, out_features=512, bias=True)
(2): ReLU()
(3): Dropout(p=0.5)
(4): Linear(in_features=512, out_features=512, bias=True)
(5): ReLU()
(6): Dropout(p=0.5)
(7): Linear(in_features=512, out_features=10, bias=True)
)
)

```

模型训练过程与上一节的 AlexNet 中的类似。

```

batch_size = 64
# 如出现 “out of memory” 的报错信息，可减小 batch_size 或 resize
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)

lr, num_epochs = 0.001, 5
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)

```

输出：

```

training on  cuda
epoch 1, loss 0.0101, train acc 0.755, test acc 0.859, time 255.9 sec
epoch 2, loss 0.0051, train acc 0.882, test acc 0.902, time 238.1 sec
epoch 3, loss 0.0043, train acc 0.900, test acc 0.908, time 225.5 sec
epoch 4, loss 0.0038, train acc 0.913, test acc 0.914, time 230.3 sec
epoch 5, loss 0.0035, train acc 0.919, test acc 0.918, time 153.9 sec

```

小结

- VGG-11 通过 5 个可以重复使用的卷积块来构造网络。根据每块里卷积层个数和输出通道数的不同可以定义出不同的 VGG 模型。

参考文献

- [1] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

注：除代码外本节与原书此节基本相同，[原书传送门](#)

5.8 网络中的网络 (NiN)

前几节介绍的 LeNet、AlexNet 和 VGG 在设计上的共同之处是：先以由卷积层构成的模块充分抽取空间特征，再以由全连接层构成的模块来输出分类结果。其中，AlexNet 和 VGG 对 LeNet 的改进主要在于如何对这两个模块加宽（增加通道数）和加深。本节我们介绍网络中的网络 (NiN) [1]。它提出了另外一个思路，即串联多个由卷积层和“全连接”层构成的小网络来构建一个深层网络。

5.8.1 NiN 块

我们知道，卷积层的输入和输出通常是四维数组（样本，通道，高，宽），而全连接层的输入和输出则通常是二维数组（样本，特征）。如果想在全连接层后再接上卷积层，则需要将全连接层的输出变换为四维。回忆在 5.3 节（多输入通道和多输出通道）里介绍的 1×1 卷积层。它可以看成全连接层，其中空间维度（高和宽）上的每个元素相当于样本，通道相当于特征。因此，NiN 使用 1×1 卷积层来替代全连接层，从而使空间信息能够自然传递到后面的层中去。图 5.7 对比了 NiN 同 AlexNet 和 VGG 等网络在结构上的主要区别。

图 5.7 左图是 AlexNet 和 VGG 的网络结构局部，右图是 NiN 的网络结构局部

NiN 块是 NiN 中的基础块。它由一个卷积层加两个充当全连接层的 1×1 卷积层串联而成。其中第一个卷积层的超参数可以自行设置，而第二和第三个卷积层的超参数一般是固定的。

```
import time
import torch
from torch import nn, optim

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

def nin_block(in_channels, out_channels, kernel_size, stride, padding):
    blk = nn.Sequential(nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding),
                        nn.ReLU(),
                        nn.Conv2d(out_channels, out_channels, kernel_size=1),
                        nn.ReLU(),
```

```

        nn.Conv2d(out_channels, out_channels, kernel_size=1),
        nn.ReLU())
return blk

```

5.8.2 NiN 模型

NiN 是在 AlexNet 问世不久后提出的。它们的卷积层设定有类似之处。NiN 使用卷积窗口形状分别为 11×11 、 5×5 和 3×3 的卷积层，相应的输出通道数也与 AlexNet 中的一致。每个 NiN 块后接一个步幅为 2、窗口形状为 3×3 的最大池化层。

除使用 NiN 块以外，NiN 还有一个设计与 AlexNet 显著不同：NiN 去掉了 AlexNet 最后的 3 个全连接层，取而代之地，NiN 使用了输出通道数等于标签类别数的 NiN 块，然后使用全局平均池化层对每个通道中所有元素求平均并直接用于分类。这里的全局平均池化层即窗口形状等于输入空间维形状的平均池化层。NiN 的这个设计的好处是可以显著减小模型参数尺寸，从而缓解过拟合。然而，该设计有时会造成获得有效模型的训练时间的增加。

```

# 已保存在 d2lzh_pytorch
class GlobalAvgPool2d(nn.Module):
    # 全局平均池化层可通过将池化窗口形状设置成输入的高和宽实现
    def __init__(self):
        super(GlobalAvgPool2d, self).__init__()
    def forward(self, x):
        return F.avg_pool2d(x, kernel_size=x.size()[2:])

net = nn.Sequential(
    nin_block(96, kernel_size=11, stride=4, padding=0),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nin_block(256, kernel_size=5, stride=1, padding=2),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nin_block(384, kernel_size=3, stride=1, padding=1),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Dropout(0.5),
    # 标签类别数是 10
    nin_block(384, 10, kernel_size=3, stride=1, padding=1),
    GlobalAvgPool2d(),
    # 将四维的输出转成二维的输出，其形状为 (批量大小, 10)
)

```

```
d2l.FlattenLayer())
```

我们构建一个数据样本来查看每一层的输出形状。

```
X = torch.rand(1, 1, 224, 224)
for name, blk in net.named_children():
    X = blk(X)
    print(name, 'output shape: ', X.shape)
```

输出：

```
0 output shape:  torch.Size([1, 96, 54, 54])
1 output shape:  torch.Size([1, 96, 26, 26])
2 output shape:  torch.Size([1, 256, 26, 26])
3 output shape:  torch.Size([1, 256, 12, 12])
4 output shape:  torch.Size([1, 384, 12, 12])
5 output shape:  torch.Size([1, 384, 5, 5])
6 output shape:  torch.Size([1, 384, 5, 5])
7 output shape:  torch.Size([1, 10, 5, 5])
8 output shape:  torch.Size([1, 10, 1, 1])
9 output shape:  torch.Size([1, 10])
```

5.8.3 获取数据和训练模型

我们依然使用 Fashion-MNIST 数据集来训练模型。NiN 的训练与 AlexNet 和 VGG 的类似，但这里使用的学习率更大。

```
batch_size = 128
# 如出现 “out of memory”的报错信息，可减小 batch_size 或 resize
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)

lr, num_epochs = 0.002, 5
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)
```

输出：

```
training on  cuda
epoch 1, loss 0.0101, train acc 0.513, test acc 0.734, time 260.9 sec
epoch 2, loss 0.0050, train acc 0.763, test acc 0.754, time 175.1 sec
epoch 3, loss 0.0041, train acc 0.808, test acc 0.826, time 151.0 sec
epoch 4, loss 0.0037, train acc 0.828, test acc 0.827, time 151.0 sec
epoch 5, loss 0.0034, train acc 0.839, test acc 0.831, time 151.0 sec
```

小结

- NiN 重复使用由卷积层和代替全连接层的 1×1 卷积层构成的 NiN 块来构建深层网络。
- NiN 去除了容易造成过拟合的全连接输出层，而是将其替换成输出通道数等于标签类别数的 NiN 块和全局平均池化层。
- NiN 的以上设计思想影响了后面一系列卷积神经网络的设计。

参考文献

- [1] Lin, M., Chen, Q., & Yan, S. (2013). Network in network. arXiv preprint arXiv:1312.4400.
-

注：除代码外本节与原书此节基本相同，[原书传送门](#)

5.9 合并行连结的网络 (GoogLeNet)

在 2014 年的 ImageNet 图像识别挑战赛中，一个名叫 GoogLeNet 的网络结构大放异彩 [1]。它虽然在名字上向 LeNet 致敬，但在网络结构上已经很难看到 LeNet 的影子。GoogLeNet 吸收了 NiN 中网络串联网络的思想，并在此基础上做了很大改进。在随后的几年里，研究员对 GoogLeNet 进行了数次改进，本节将介绍这个模型系列的第一个版本。

5.9.1 Inception 块

GoogLeNet 中的基础卷积块叫作 Inception 块，得名于同名电影《盗梦空间》(Inception)。与上一节介绍的 NiN 块相比，这个基础块在结构上更加复杂，如图 5.8 所示。

图 5.8 Inception 块的结构

由图 5.8 可以看出，Inception 块里有 4 条并行的线路。前 3 条线路使用窗口大小分别是 1×1 、 3×3 和 5×5 的卷积层来抽取不同空间尺寸下的信息，其中中间 2 个线路会对输入先做 1×1 卷积来减少输入通道数，以降低模型复杂度。第四条线路则使用 3×3 最大池化层，后接 1×1 卷积层来改变通道数。4 条线路都使用了合适的填充来使输入与输出的高和宽一致。最后我们将每条线路的输出在通道维上连结，并输入接下来的层中去。

Inception 块中可以自定义的超参数是每个层的输出通道数，我们以此来控制模型复杂度。

```
import time
import torch
from torch import nn, optim
import torch.nn.functional as F

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

class Inception(nn.Module):
    # c1 - c4 为每条线路里的层的输出通道数
    def __init__(self, in_c, c1, c2, c3, c4):
```

```

super(Inception, self).__init__()
# 线路 1, 单 1 x 1 卷积层
self.p1_1 = nn.Conv2d(in_c, c1, kernel_size=1)
# 线路 2, 1 x 1 卷积层后接 3 x 3 卷积层
self.p2_1 = nn.Conv2d(in_c, c2[0], kernel_size=1)
self.p2_2 = nn.Conv2d(c2[0], c2[1], kernel_size=3, padding=1)
# 线路 3, 1 x 1 卷积层后接 5 x 5 卷积层
self.p3_1 = nn.Conv2d(in_c, c3[0], kernel_size=1)
self.p3_2 = nn.Conv2d(c3[0], c3[1], kernel_size=5, padding=2)
# 线路 4, 3 x 3 最大池化层后接 1 x 1 卷积层
self.p4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
self.p4_2 = nn.Conv2d(in_c, c4, kernel_size=1)

def forward(self, x):
    p1 = F.relu(self.p1_1(x))
    p2 = F.relu(self.p2_2(F.relu(self.p2_1(x))))
    p3 = F.relu(self.p3_2(F.relu(self.p3_1(x))))
    p4 = F.relu(self.p4_2(self.p4_1(x)))
    return torch.cat((p1, p2, p3, p4), dim=1) # 在通道维上连结输出

```

5.9.2 GoogLeNet 模型

GoogLeNet 跟 VGG 一样，在主体卷积部分中使用 5 个模块（block），每个模块之间使用步幅为 2 的 3×3 最大池化层来减小输出高宽。第一模块使用一个 64 通道的 7×7 卷积层。

```
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
                   nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

第二模块使用 2 个卷积层：首先是 64 通道的 1×1 卷积层，然后是将通道增大 3 倍的 3×3 卷积层。它对应 Inception 块中的第二条线路。

```
b2 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=1),
                   nn.Conv2d(64, 192, kernel_size=3, padding=1),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

第三模块串联 2 个完整的 Inception 块。第一个 Inception 块的输出通道数为 $64 + 128 + 32 + 32 = 256$ ，其中 4 条线路的输出通道数比例为 $64 : 128 : 32 : 32 = 2 : 4 : 1 : 1$ 。其中第二、第三条线路先分别将输入通道数减小至 $96/192 = 1/2$ 和 $16/192 = 1/12$ 后，再接上第二层卷积层。第二个 Inception 块输出通道数增至 $128 + 192 + 96 + 64 = 480$ ，每条线路的输出通道数之比为 $128 : 192 : 96 : 64 = 4 : 6 : 3 : 2$ 。其中第二、第三条线路先分别将输入通道数减小至 $128/256 = 1/2$ 和 $32/256 = 1/8$ 。

```
b3 = nn.Sequential(Inception(192, 64, (96, 128), (16, 32), 32),
                   Inception(256, 128, (128, 192), (32, 96), 64),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

第四模块更加复杂。它串联了 5 个 Inception 块，其输出通道数分别是 $192 + 208 + 48 + 64 = 512$ 、 $160 + 224 + 64 + 64 = 512$ 、 $128 + 256 + 64 + 64 = 512$ 、 $112 + 288 + 64 + 64 = 528$ 和 $256 + 320 + 128 + 128 = 832$ 。这些线路的通道数分配和第三模块中的类似，首先含 3×3 卷积层的第二条线路输出最多通道，其次是仅含 1×1 卷积层的第一条线路，之后是含 5×5 卷积层的第三条线路和含 3×3 最大池化层的第四条线路。其中第二、第三条线路都会先按比例减小通道数。这些比例在各个 Inception 块中都略有不同。

```
b4 = nn.Sequential(Inception(480, 192, (96, 208), (16, 48), 64),
                   Inception(512, 160, (112, 224), (24, 64), 64),
                   Inception(512, 128, (128, 256), (24, 64), 64),
                   Inception(512, 112, (144, 288), (32, 64), 64),
                   Inception(528, 256, (160, 320), (32, 128), 128),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

第五模块有输出通道数为 $256 + 320 + 128 + 128 = 832$ 和 $384 + 384 + 128 + 128 = 1024$ 的两个 Inception 块。其中每条线路的通道数的分配思路和第三、第四模块中的一致，只是在具体数值上有所不同。需要注意的是，第五模块的后面紧跟输出层，该模块同 NiN 一样使用全局平均池化层来将每个通道的高和宽变成 1。最后我们将输出变成二维数组后接上一个输出个数为标签类别数的全连接层。

```
b5 = nn.Sequential(Inception(832, 256, (160, 320), (32, 128), 128),
                   Inception(832, 384, (192, 384), (48, 128), 128),
                   d21.GlobalAvgPool2d())
net = nn.Sequential(b1, b2, b3, b4, b5,
                    d21.FlattenLayer(), nn.Linear(1024, 10))
```

GoogLeNet 模型的计算复杂，而且不如 VGG 那样便于修改通道数。本节里我们将输入的高和宽从 224 降到 96 来简化计算。下面演示各个模块之间的输出的形状变化。

```
net = nn.Sequential(b1, b2, b3, b4, b5, d21.FlattenLayer(), nn.Linear(1024, 10))
X = torch.rand(1, 1, 96, 96)
for blk in net.children():
    X = blk(X)
    print('output shape: ', X.shape)
```

输出：

```
output shape: torch.Size([1, 64, 24, 24])
output shape: torch.Size([1, 192, 12, 12])
output shape: torch.Size([1, 480, 6, 6])
output shape: torch.Size([1, 832, 3, 3])
output shape: torch.Size([1, 1024, 1, 1])
output shape: torch.Size([1, 1024])
output shape: torch.Size([1, 10])
```

5.9.3 获取数据和训练模型

我们使用高和宽均为 96 像素的图像来训练 GoogLeNet 模型。训练使用的图像依然来自 Fashion-MNIST 数据集。

```
batch_size = 128
# 如出现 “out of memory”的报错信息，可减小 batch_size 或 resize
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)

lr, num_epochs = 0.001, 5
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)
```

输出：

```
training on  cuda
epoch 1, loss 0.0087, train acc 0.570, test acc 0.831, time 45.5 sec
epoch 2, loss 0.0032, train acc 0.851, test acc 0.853, time 48.5 sec
epoch 3, loss 0.0026, train acc 0.880, test acc 0.883, time 45.4 sec
epoch 4, loss 0.0022, train acc 0.895, test acc 0.887, time 46.6 sec
epoch 5, loss 0.0020, train acc 0.906, test acc 0.896, time 43.5 sec
```

小结

- Inception 块相当于一个有 4 条线路的子网络。它通过不同窗口形状的卷积层和最大池化层来并行抽取信息，并使用 1×1 卷积层减少通道数从而降低模型复杂度。
- GoogLeNet 将多个设计精细的 Inception 块和其他层串联起来。其中 Inception 块的通道数分配之比是在 ImageNet 数据集上通过大量的实验得来的。
- GoogLeNet 和它的后继者们一度是 ImageNet 上最高效的模型之一：在类似的测试精度下，它们的计算复杂度往往更低。

参考文献

- [1] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., & Anguelov, D. & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).
- [2] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.
- [3] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 2818-2826).
- [4] Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017, February). Inception-v4, inception-resnet and the impact of residual connections on learning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 4, p. 12).

注：除代码外本节与原书此节基本相同，[原书传送门](#)

5.10 批量归一化

本节我们介绍批量归一化 (batch normalization) 层，它能让较深的神经网络的训练变得更加容易 [1]。在 3.16 节（实战 Kaggle 比赛：预测房价）里，我们对输入数据做了标准化处理：处理后的任意一个特征在数据集中所有样本上的均值为 0、标准差为 1。标准化处理输入数据使各个特征的分布相近：这往往更容易训练出有效的模型。

通常来说，数据标准化预处理对于浅层模型就足够有效了。随着模型训练的进行，当每层中参数更新时，靠近输出层的输出较难出现剧烈变化。但对深层神经网络来说，即使输入数据已做标准化，训练中模型参数的更新依然很容易造成靠近输出层输出的剧烈变化。这种计算数值的不稳定性通常令我们难以训练出有效的深度模型。

批量归一化的提出正是为了应对深度模型训练的挑战。在模型训练时，批量归一化利用小批量上的均值和标准差，不断调整神经网络中间输出，从而使整个神经网络在各层的中间输出的数值更稳定。**批量归一化和下一节将要介绍的残差网络为训练和设计深度模型提供了两类重要思路。**

5.10.1 批量归一化层

对全连接层和卷积层做批量归一化的方法稍有不同。下面我们将分别介绍这两种情况下的批量归一化。

5.10.1.1 对全连接层做批量归一化

我们先考虑如何对全连接层做批量归一化。通常，我们将批量归一化层置于全连接层中的仿射变换和激活函数之间。设全连接层的输入为 \mathbf{u} ，权重参数和偏差参数分别为 \mathbf{W} 和 \mathbf{b} ，激活函数为 ϕ 。设批量归一化的运算符为 BN。那么，使用批量归一化的全连接层的输出为

$$\phi(\text{BN}(\mathbf{x})),$$

其中批量归一化输入 \mathbf{x} 由仿射变换

$$\mathbf{x} = \mathbf{W}\mathbf{u} + \mathbf{b}$$

得到。考虑一个由 m 个样本组成的小批量，仿射变换的输出为一个新的小批量 $\mathcal{B} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ 。它们正是批量归一化层的输入。对于小批量 \mathcal{B} 中任意样本 $\mathbf{x}^{(i)} \in \mathbb{R}^d, 1 \leq i \leq m$ ，批量归一化层的输出同样是 d 维向量

$$\mathbf{y}^{(i)} = \text{BN}(\mathbf{x}^{(i)}),$$

并由以下几步求得。首先，对小批量 \mathcal{B} 求均值和方差：

$$\boldsymbol{\mu}_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m \mathbf{x}^{(i)},$$

$$\boldsymbol{\sigma}_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (\mathbf{x}^{(i)} - \boldsymbol{\mu}_{\mathcal{B}})^2,$$

其中的平方计算是按元素求平方。接下来，使用按元素开方和按元素除法对 $\mathbf{x}^{(i)}$ 标准化：

$$\hat{\mathbf{x}}^{(i)} \leftarrow \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_{\mathcal{B}}}{\sqrt{\boldsymbol{\sigma}_{\mathcal{B}}^2 + \epsilon}},$$

这里 $\epsilon > 0$ 是一个很小的常数，保证分母大于 0。在上面标准化的基础上，批量归一化层引入了两个可以学习的模型参数，拉伸（scale）参数 γ 和偏移（shift）参数 β 。这两个参数和 $\mathbf{x}^{(i)}$ 形状相同，皆为 d 维向量。它们与 $\mathbf{x}^{(i)}$ 分别做按元素乘法（符号 \odot ）和加法计算：

$$\mathbf{y}^{(i)} \leftarrow \gamma \odot \hat{\mathbf{x}}^{(i)} + \beta.$$

至此，我们得到了 $\mathbf{x}^{(i)}$ 的批量归一化的输出 $\mathbf{y}^{(i)}$ 。值得注意的是，可学习的拉伸和偏移参数保留了不对 $\hat{\mathbf{x}}^{(i)}$ 做批量归一化的可能：此时只需学出 $\gamma = \sqrt{\boldsymbol{\sigma}_{\mathcal{B}}^2 + \epsilon}$ 和 $\beta = \boldsymbol{\mu}_{\mathcal{B}}$ 。我们可以对此这样理解：如果批量归一化无益，理论上，学出的模型可以不使用批量归一化。

5.10.1.2 对卷积层做批量归一化

对卷积层来说，批量归一化发生在卷积计算之后、应用激活函数之前。如果卷积计算输出多个通道，我们需要对这些通道的输出分别做批量归一化，且**每个通道都拥有独立的拉伸和偏移参数，并均为标量**。设小批量中有 m 个样本。在单个通道上，假设卷积计算输出的高和宽分别为 p 和 q 。我们需要对该通道中 $m \times p \times q$ 个元素同时做批量归一化。对这些元素做标准化计算时，我们使用相同的均值和方差，即该通道中 $m \times p \times q$ 个元素的均值和方差。

5.10.1.3 预测时的批量归一化

使用批量归一化训练时，我们可以将批量大小设得大一点，从而使批量内样本的均值和方差的计算都较为准确。将训练好的模型用于预测时，我们希望模型对于任意输入都有确定的输出。因此，单个样本的输出不应取决于批量归一化所需要的随机小批量中的均值和方差。一种常用的方法是通过移动平均估算整个训练数据集的样本均值和方

差，并在预测时使用它们得到确定的输出。可见，和丢弃层一样，批量归一化层在训练模式和预测模式下的计算结果也是不一样的。

5.10.2 从零开始实现

下面我们自己实现批量归一化层。

```
import time
import torch
from torch import nn, optim
import torch.nn.functional as F

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

def batch_norm(is_training, X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # 判断当前模式是训练模式还是预测模式
    if not is_training:
        # 如果是在预测模式下，直接使用传入的移动平均所得的均值和方差
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # 使用全连接层的情况，计算特征维上的均值和方差
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # 使用二维卷积层的情况，计算通道维上 (axis=1) 的均值和方差。这里我们需要保持
            # X 的形状以便后面可以做广播运算
            mean = X.mean(dim=0, keepdim=True).mean(dim=2, keepdim=True).mean(dim=3,
            var = ((X - mean) ** 2).mean(dim=0, keepdim=True).mean(dim=2, keepdim=True)
        # 训练模式下用当前的均值和方差做标准化
        X_hat = (X - mean) / torch.sqrt(var + eps)
    # 更新移动平均的均值和方差
    moving_mean = moving_mean * momentum + X_hat * (1 - momentum)
    moving_var = moving_var * momentum + var * (1 - momentum)
```

```

    moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
    moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta # 拉伸和偏移
    return Y, moving_mean, moving_var

```

接下来，我们自定义一个 `BatchNorm` 层。它保存参与求梯度和迭代的拉伸参数 `gamma` 和偏移参数 `beta`，同时也维护移动平均得到的均值和方差，以便能够在模型预测时被使用。`BatchNorm` 实例所需指定的 `num_features` 参数对于全连接层来说应为输出个数，对于卷积层来说则为输出通道数。该实例所需指定的 `num_dims` 参数对于全连接层和卷积层来说分别为 2 和 4。

```

class BatchNorm(nn.Module):
    def __init__(self, num_features, num_dims):
        super(BatchNorm, self).__init__()
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # 参与求梯度和迭代的拉伸和偏移参数，分别初始化成 0 和 1
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # 不参与求梯度和迭代的变量，全在内存上初始化成 0
        self.moving_mean = torch.zeros(shape)
        self.moving_var = torch.zeros(shape)

    def forward(self, X):
        # 如果 X 不在内存上，将 moving_mean 和 moving_var 复制到 X 所在显存上
        if self.moving_mean.device != X.device:
            self.moving_mean = self.moving_mean.to(X.device)
            self.moving_var = self.moving_var.to(X.device)
        # 保存更新过的 moving_mean 和 moving_var, Module 实例的 training 属性默认为 true
        Y, self.moving_mean, self.moving_var = batch_norm(self.training,
                                                          X, self.gamma, self.beta, self.moving_mean,
                                                          self.moving_var, eps=1e-5, momentum=0.9)
        return Y

```

5.10.2.1 使用批量归一化层的 LeNet

下面我们修改 5.5 节（卷积神经网络（LeNet））介绍的 LeNet 模型，从而应用批量归一化层。我们在所有的卷积层或全连接层之后、激活层之前加入批量归一化层。

```
net = nn.Sequential(
    nn.Conv2d(1, 6, 5), # in_channels, out_channels, kernel_size
    BatchNorm(6, num_dims=4),
    nn.Sigmoid(),
    nn.MaxPool2d(2, 2), # kernel_size, stride
    nn.Conv2d(6, 16, 5),
    BatchNorm(16, num_dims=4),
    nn.Sigmoid(),
    nn.MaxPool2d(2, 2),
    d2l.FlattenLayer(),
    nn.Linear(16*4*4, 120),
    BatchNorm(120, num_dims=2),
    nn.Sigmoid(),
    nn.Linear(120, 84),
    BatchNorm(84, num_dims=2),
    nn.Sigmoid(),
    nn.Linear(84, 10)
)
```

下面我们训练修改后的模型。

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

lr, num_epochs = 0.001, 5
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)
```

输出：

```
training on  cuda
epoch 1, loss 0.0039, train acc 0.790, test acc 0.835, time 2.9 sec
epoch 2, loss 0.0018, train acc 0.866, test acc 0.821, time 3.2 sec
```

```
epoch 3, loss 0.0014, train acc 0.879, test acc 0.857, time 2.6 sec
epoch 4, loss 0.0013, train acc 0.886, test acc 0.820, time 2.7 sec
epoch 5, loss 0.0012, train acc 0.891, test acc 0.859, time 2.8 sec
```

最后我们查看第一个批量归一化层学习到的拉伸参数 `gamma` 和偏移参数 `beta`。

```
net[1].gamma.view((-1,)), net[1].beta.view((-1,))
```

输出：

```
(tensor([ 1.2537,  1.2284,  1.0100,  1.0171,  0.9809,  1.1870], device='cuda:0'),
 tensor([ 0.0962,  0.3299, -0.5506,  0.1522, -0.1556,  0.2240], device='cuda:0'))
```

5.10.3 简洁实现

与我们刚刚自己定义的 `BatchNorm` 类相比，Pytorch 中 `nn` 模块定义的 `BatchNorm1d` 和 `BatchNorm2d` 类使用起来更加简单，二者分别用于全连接层和卷积层，都需要指定输入的 `num_features` 参数值。下面我们用 PyTorch 实现使用批量归一化的 LeNet。

```
net = nn.Sequential(
    nn.Conv2d(1, 6, 5), # in_channels, out_channels, kernel_size
    nn.BatchNorm2d(6),
    nn.Sigmoid(),
    nn.MaxPool2d(2, 2), # kernel_size, stride
    nn.Conv2d(6, 16, 5),
    nn.BatchNorm2d(16),
    nn.Sigmoid(),
    nn.MaxPool2d(2, 2),
    d2l.FlattenLayer(),
    nn.Linear(16*4*4, 120),
    nn.BatchNorm1d(120),
    nn.Sigmoid(),
    nn.Linear(120, 84),
    nn.BatchNorm1d(84),
    nn.Sigmoid(),
    nn.Linear(84, 10)
)
```

使用同样的超参数进行训练。

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

lr, num_epochs = 0.001, 5
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)
```

输出：

```
training on  cuda
epoch 1, loss 0.0054, train acc 0.767, test acc 0.795, time 2.0 sec
epoch 2, loss 0.0024, train acc 0.851, test acc 0.748, time 2.0 sec
epoch 3, loss 0.0017, train acc 0.872, test acc 0.814, time 2.2 sec
epoch 4, loss 0.0014, train acc 0.883, test acc 0.818, time 2.1 sec
epoch 5, loss 0.0013, train acc 0.889, test acc 0.734, time 1.8 sec
```

小结

- 在模型训练时，批量归一化利用小批量上的均值和标准差，不断调整神经网络的中间输出，从而使整个神经网络在各层的中间输出的数值更稳定。
- 对全连接层和卷积层做批量归一化的方法稍有不同。
- 批量归一化层和丢弃层一样，在训练模式和预测模式的计算结果是不一样的。
- PyTorch 提供了 BatchNorm 类方便使用。

参考文献

- [1] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

注：除代码外本节与原书此节基本相同，[原书传送门](#)

5.11 残差网络 (ResNet)

让我们先思考一个问题：对神经网络模型添加新的层，充分训练后的模型是否只可能更有效地降低训练误差？理论上，原模型解的空间只是新模型解的空间的子空间。也就是说，如果我们能将新添加的层训练成恒等映射 $f(x) = x$ ，新模型和原模型将同样有效。由于新模型可能得出更优的解来拟合训练数据集，因此添加层似乎更容易降低训练误差。然而在实践中，添加过多的层后训练误差往往不降反升。即使利用批量归一化带来的数值稳定性使训练深层模型更加容易，该问题仍然存在。针对这一问题，何恺明等人提出了残差网络（ResNet）[1]。它在 2015 年的 ImageNet 图像识别挑战赛夺魁，并深刻影响了后来的深度神经网络的设计。

5.11.2 残差块

让我们聚焦于神经网络局部。如图 5.9 所示，设输入为 \mathbf{x} 。假设我们希望学出的理想映射为 $f(\mathbf{x})$ ，从而作为图 5.9 上方激活函数的输入。左图虚线框中的部分需要直接拟合出该映射 $f(\mathbf{x})$ ，而右图虚线框中的部分则需要拟合出有关恒等映射的残差映射 $f(\mathbf{x}) - \mathbf{x}$ 。残差映射在实际中往往更容易优化。以本节开头提到的恒等映射作为我们希望学出的理想映射 $f(\mathbf{x})$ 。我们只需将图 5.9 中右图虚线框内上方的加权运算（如仿射）的权重和偏差参数学成 0，那么 $f(\mathbf{x})$ 即为恒等映射。实际上，当理想映射 $f(\mathbf{x})$ 极接近于恒等映射时，残差映射也易于捕捉恒等映射的细微波动。图 5.9 右图也是 ResNet 的基础块，即残差块（residual block）。在残差块中，输入可通过跨层的数据线路更快地向前传播。

图 5.9 普通的网络结构（左）与加入残差连接的网络结构（右）

ResNet 沿用了 VGG 全 3×3 卷积层的设计。残差块里首先有 2 个有相同输出通道数的 3×3 卷积层。每个卷积层后接一个批量归一化层和 ReLU 激活函数。然后我们将输入跳过这两个卷积运算后直接加在最后的 ReLU 激活函数前。这样的设计要求两个卷积层的输出与输入形状一样，从而可以相加。如果想改变通道数，就需要引入一个额外的 1×1 卷积层来将输入变换成需要的形状后再做相加运算。

残差块的实现如下。它可以设定输出通道数、是否使用额外的 1×1 卷积层来修改通道数以及卷积层的步幅。

```
import time
import torch
from torch import nn, optim
import torch.nn.functional as F
```

```

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

class Residual(nn.Module): # 本类已保存在 d2lzh_pytorch 包中方便以后使用
    def __init__(self, in_channels, out_channels, use_1x1conv=False, stride=1):
        super(Residual, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1, s)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=s)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return F.relu(Y + X)

```

下面我们来查看输入和输出形状一致的情况。

```

blk = Residual(3, 3)
X = torch.rand((4, 3, 6, 6))
blk(X).shape # torch.Size([4, 3, 6, 6])

```

我们也可以在增加输出通道数的同时减半输出的高和宽。

```

blk = Residual(3, 6, use_1x1conv=True, stride=2)
blk(X).shape # torch.Size([4, 6, 3, 3])

```

5.11.2 ResNet 模型

ResNet 的前两层跟之前介绍的 GoogLeNet 中的一样：在输出通道数为 64、步幅为 2 的 7×7 卷积层后接步幅为 2 的 3×3 的最大池化层。不同之处在于 ResNet 每个卷

积层后增加的批量归一化层。

```
net = nn.Sequential(
    nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

GoogLeNet 在后面接了 4 个由 Inception 块组成的模块。ResNet 则使用 4 个由残差块组成的模块，每个模块使用若干个同样输出通道数的残差块。第一个模块的通道数同输入通道数一致。由于之前已经使用了步幅为 2 的最大池化层，所以无须减小高和宽。之后的每个模块在第一个残差块里将上一个模块的通道数翻倍，并将高和宽减半。

下面我们来实现这个模块。注意，这里对第一个模块做了特别处理。

```
def resnet_block(in_channels, out_channels, num_residuals, first_block=False):
    if first_block:
        assert in_channels == out_channels # 第一个模块的通道数同输入通道数一致
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(in_channels, out_channels, use_1x1conv=True, stride=2))
        else:
            blk.append(Residual(out_channels, out_channels))
    return nn.Sequential(*blk)
```

接着我们为 ResNet 加入所有残差块。这里每个模块使用两个残差块。

```
net.add_module("resnet_block1", resnet_block(64, 64, 2, first_block=True))
net.add_module("resnet_block2", resnet_block(64, 128, 2))
net.add_module("resnet_block3", resnet_block(128, 256, 2))
net.add_module("resnet_block4", resnet_block(256, 512, 2))
```

最后，与 GoogLeNet 一样，加入全局平均池化层后接上全连接层输出。

```
net.add_module("global_avg_pool", d2l.GlobalAvgPool2d()) # GlobalAvgPool2d 的输出：(B, 512)
net.add_module("fc", nn.Sequential(d2l.FlattenLayer(), nn.Linear(512, 10)))
```

这里每个模块里有 4 个卷积层（不计算 1×1 卷积层），加上最开始的卷积层和最后的全连接层，共计 18 层。这个模型通常也被称为 ResNet-18。通过配置不同的通道数和模块里的残差块数可以得到不同的 ResNet 模型，例如更深的含 152 层的 ResNet-152。虽然 ResNet 的主体架构跟 GoogLeNet 的类似，但 ResNet 结构更简单，修改也更方便。这些因素都导致了 ResNet 迅速被广泛使用。

在训练 ResNet 之前，我们来观察一下输入形状在 ResNet 不同模块之间的变化。

```
X = torch.rand((1, 1, 224, 224))
for name, layer in net.named_children():
    X = layer(X)
    print(name, ' output shape:\t', X.shape)
```

输出：

```
0  output shape:      torch.Size([1, 64, 112, 112])
1  output shape:      torch.Size([1, 64, 112, 112])
2  output shape:      torch.Size([1, 64, 112, 112])
3  output shape:      torch.Size([1, 64, 56, 56])
resnet_block1  output shape:      torch.Size([1, 64, 56, 56])
resnet_block2  output shape:      torch.Size([1, 128, 28, 28])
resnet_block3  output shape:      torch.Size([1, 256, 14, 14])
resnet_block4  output shape:      torch.Size([1, 512, 7, 7])
global_avg_pool  output shape:   torch.Size([1, 512, 1, 1])
fc  output shape:      torch.Size([1, 10])
```

5.11.3 获得数据和训练模型

下面我们在 Fashion-MNIST 数据集上训练 ResNet。

```
batch_size = 256
# 如出现 “out of memory”的报错信息，可减小 batch_size 或 resize
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)

lr, num_epochs = 0.001, 5
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)
```

输出：

```
training on  cuda
epoch 1, loss 0.0015, train acc 0.853, test acc 0.885, time 31.0 sec
epoch 2, loss 0.0010, train acc 0.910, test acc 0.899, time 31.8 sec
epoch 3, loss 0.0008, train acc 0.926, test acc 0.911, time 31.6 sec
epoch 4, loss 0.0007, train acc 0.936, test acc 0.916, time 31.8 sec
epoch 5, loss 0.0006, train acc 0.944, test acc 0.926, time 31.5 sec
```

小结

- 残差块通过跨层的数据通道从而能够训练出有效的深度神经网络。
- ResNet 深刻影响了后来的深度神经网络的设计。

参考文献

[1] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[2] He, K., Zhang, X., Ren, S., & Sun, J. (2016, October). Identity mappings in deep residual networks. In European Conference on Computer Vision (pp. 630-645). Springer, Cham.

注：除代码外本节与原书此节基本相同，[原书传送门](#)

5.12 稠密连接网络 (DenseNet)

ResNet 中的跨层连接设计引申出了数个后续工作。本节我们介绍其中的一个：稠密连接网络 (DenseNet) [1]。它与 ResNet 的主要区别如图 5.10 所示。

图 5.10 ResNet (左) 与 DenseNet (右) 在跨层连接上的主要区别：使用相加和使用连结

图 5.10 中将部分前后相邻的运算抽象为模块 A 和模块 B。与 ResNet 的主要区别在于，DenseNet 里模块 B 的输出不是像 ResNet 那样和模块 A 的输出相加，而是在通道维上连结。这样模块 A 的输出可以直接传入模块 B 后面的层。在这个设计里，模块 A 直接跟模块 B 后面的所有层连接在了一起。这也是它被称为“稠密连接”的原因。

DenseNet 的主要构建模块是稠密块 (dense block) 和过渡层 (transition layer)。前者定义了输入和输出是如何连结的，后者则用来控制通道数，使之不过大。

5.12.1 稠密块

DenseNet 使用了 ResNet 改良版的“批量归一化、激活和卷积”结构，我们首先在 `conv_block` 函数里实现这个结构。

```
import time
import torch
from torch import nn, optim
import torch.nn.functional as F

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

def conv_block(in_channels, out_channels):
    blk = nn.Sequential(nn.BatchNorm2d(in_channels),
                        nn.ReLU(),
                        nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1))
    return blk
```

稠密块由多个 `conv_block` 组成，每块使用相同的输出通道数。但在前向计算时，我们将每块的输入和输出在通道维上连结。

```

class DenseBlock(nn.Module):
    def __init__(self, num_convs, in_channels, out_channels):
        super(DenseBlock, self).__init__()
        net = []
        for i in range(num_convs):
            in_c = in_channels + i * out_channels
            net.append(conv_block(in_c, out_channels))
        self.net = nn.ModuleList(net)
        self.out_channels = in_channels + num_convs * out_channels # 计算输出通道数

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            X = torch.cat((X, Y), dim=1) # 在通道维上将输入和输出连结
        return X

```

在下面的例子中，我们定义一个有 2 个输出通道数为 10 的卷积块。使用通道数为 3 的输入时，我们会得到通道数为 $3 + 2 \times 10 = 23$ 的输出。卷积块的通道数控制了输出通道数相对于输入通道数的增长，因此也被称为增长率（growth rate）。

```

blk = DenseBlock(2, 3, 10)
X = torch.rand(4, 3, 8, 8)
Y = blk(X)
Y.shape # torch.Size([4, 23, 8, 8])

```

5.12.2 过渡层

由于每个稠密块都会带来通道数的增加，使用过多则会带来过于复杂的模型。过渡层用来控制模型复杂度。它通过 1×1 卷积层来减小通道数，并使用步幅为 2 的平均池化层减半高和宽，从而进一步降低模型复杂度。

```

def transition_block(in_channels, out_channels):
    blk = nn.Sequential(
        nn.BatchNorm2d(in_channels),
        nn.ReLU(),
        nn.Conv2d(in_channels, out_channels, kernel_size=1),
        nn.AvgPool2d(kernel_size=2, stride=2))
    return blk

```

对上一个例子中稠密块的输出使用通道数为 10 的过渡层。此时输出的通道数减为 10，高和宽均减半。

```
blk = transition_block(23, 10)
blk(Y).shape # torch.Size([4, 10, 4, 4])
```

5.12.3 DenseNet 模型

我们来构造 DenseNet 模型。DenseNet 首先使用同 ResNet 一样的单卷积层和最大池化层。

```
net = nn.Sequential(
    nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

类似于 ResNet 接下来使用的 4 个残差块，DenseNet 使用的是 4 个稠密块。同 ResNet 一样，我们可以设置每个稠密块使用多少个卷积层。这里我们设成 4，从而与上一节的 ResNet-18 保持一致。稠密块里的卷积层通道数（即增长率）设为 32，所以每个稠密块将增加 128 个通道。

ResNet 里通过步幅为 2 的残差块在每个模块之间减小高和宽。这里我们则使用过渡层来减半高和宽，并减半通道数。

```
num_channels, growth_rate = 64, 32 # num_channels 为当前的通道数
num_convs_in_dense_blocks = [4, 4, 4, 4]

for i, num_convs in enumerate(num_convs_in_dense_blocks):
    DB = DenseBlock(num_convs, num_channels, growth_rate)
    net.add_module("DenseBlock_%d" % i, DB)
    # 上一个稠密块的输出通道数
    num_channels = DB.out_channels
    # 在稠密块之间加入通道数减半的过渡层
    if i != len(num_convs_in_dense_blocks) - 1:
        net.add_module("transition_block_%d" % i, transition_block(num_channels, num_
            num_channels = num_channels // 2
```

同 ResNet 一样，最后接上全局池化层和全连接层来输出。

```
net.add_module("BN", nn.BatchNorm2d(num_channels))
net.add_module("relu", nn.ReLU())
net.add_module("global_avg_pool", d2l.GlobalAvgPool2d() # GlobalAvgPool2d 的输出: (B, C)
net.add_module("fc", nn.Sequential(d2l.FlattenLayer(), nn.Linear(num_channels, 10)))
```

我们尝试打印每个子模块的输出维度确保网络无误：

```
X = torch.rand(1, 1, 96, 96)
for name, layer in net.named_children():
    X = layer(X)
    print(name, ' output shape:\t', X.shape)
```

输出：

```
0  output shape:      torch.Size([1, 64, 48, 48])
1  output shape:      torch.Size([1, 64, 48, 48])
2  output shape:      torch.Size([1, 64, 48, 48])
3  output shape:      torch.Size([1, 64, 24, 24])
DenseBlosk_0  output shape:  torch.Size([1, 192, 24, 24])
transition_block_0  output shape:  torch.Size([1, 96, 12, 12])
DenseBlosk_1  output shape:  torch.Size([1, 224, 12, 12])
transition_block_1  output shape:  torch.Size([1, 112, 6, 6])
DenseBlosk_2  output shape:  torch.Size([1, 240, 6, 6])
transition_block_2  output shape:  torch.Size([1, 120, 3, 3])
DenseBlosk_3  output shape:  torch.Size([1, 248, 3, 3])
BN  output shape:      torch.Size([1, 248, 3, 3])
relu  output shape:  torch.Size([1, 248, 3, 3])
global_avg_pool  output shape:  torch.Size([1, 248, 1, 1])
fc  output shape:      torch.Size([1, 10])
```

5.12.4 获取数据并训练模型

由于这里使用了比较深的网络，本节里我们将输入高和宽从 224 降到 96 来简化计算。

```
batch_size = 256
# 如出现“out of memory”的报错信息，可减小 batch_size 或 resize
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
```

```
lr, num_epochs = 0.001, 5
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)
```

输出：

```
training on  cuda
epoch 1, loss 0.0020, train acc 0.834, test acc 0.749, time 27.7 sec
epoch 2, loss 0.0011, train acc 0.900, test acc 0.824, time 25.5 sec
epoch 3, loss 0.0009, train acc 0.913, test acc 0.839, time 23.8 sec
epoch 4, loss 0.0008, train acc 0.921, test acc 0.889, time 24.9 sec
epoch 5, loss 0.0008, train acc 0.929, test acc 0.884, time 24.3 sec
```

小结

- 在跨层连接上，不同于 ResNet 中将输入与输出相加，DenseNet 在通道维上连结输入与输出。
- DenseNet 的主要构建模块是稠密块和过渡层。

参考文献

- [1] Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (Vol. 1, No. 2).
-

注：除代码外本节与原书此节基本相同，[原书传送门](#)

6.1 语言模型

语言模型 (language model) 是自然语言处理的重要技术。自然语言处理中最常见的数据是文本数据。我们可以把一段自然语言文本看作一段离散的时间序列。假设一段长度为 T 的文本中的词依次为 w_1, w_2, \dots, w_T , 那么在离散的时间序列中, $w_t (1 \leq t \leq T)$ 可看作在时间步 (time step) t 的输出或标签。给定一个长度为 T 的词的序列 w_1, w_2, \dots, w_T , 语言模型将计算该序列的概率:

$$P(w_1, w_2, \dots, w_T).$$

语言模型可用于提升语音识别和机器翻译的性能。例如, 在语音识别中, 给定一段“厨房里食油用完了”的语音, 有可能会输出“厨房里食油用完了”和“厨房里石油用完了”这两个读音完全一样的文本序列。如果语言模型判断出前者的概率大于后者的概率, 我们就可以根据相同读音的语音输出“厨房里食油用完了”的文本序列。在机器翻译中, 如果对英文“you go first”逐词翻译成中文的话, 可能得到“你走先”“你先走”等排列方式的文本序列。如果语言模型判断出“你先走”的概率大于其他排列方式的文本序列的概率, 我们就可以把“you go first”翻译成“你先走”。

6.1.1 语言模型的计算

既然语言模型很有用, 那该如何计算它呢? 假设序列 w_1, w_2, \dots, w_T 中的每个词是依次生成的, 我们有

$$P(w_1, w_2, \dots, w_T) = \prod_{t=1}^T P(w_t | w_1, \dots, w_{t-1}).$$

例如, 一段含有 4 个词的文本序列的概率

$$P(w_1, w_2, w_3, w_4) = P(w_1)P(w_2 | w_1)P(w_3 | w_1, w_2)P(w_4 | w_1, w_2, w_3).$$

为了计算语言模型, 我们需要计算词的概率, 以及一个词在给定前几个词的情况下条件概率, 即语言模型参数。设训练数据集为一个大型文本语料库, 如维基百科的所有条目。词的概率可以通过该词在训练数据集中的相对词频来计算。例如, $P(w_1)$ 可以计算为 w_1 在训练数据集中的词频 (词出现的次数) 与训练数据集的总词数之比。因此, 根据条件概率定义, 一个词在给定前几个词的情况下条件概率也可以通过训练数据集中的相对词频计算。例如, $P(w_2 | w_1)$ 可以计算为 w_1, w_2 两词相邻的频率与 w_1 词频的比值, 因为该比值即 $P(w_1, w_2)$ 与 $P(w_1)$ 之比; 而 $P(w_3 | w_1, w_2)$ 同理可以计算为 w_1, w_2 和 w_3 三词相邻的频率与 w_1 和 w_2 两词相邻的频率的比值。以此类推。

6.1.2 n 元语法

当序列长度增加时，计算和存储多个词共同出现的概率的复杂度会呈指数级增加。 n 元语法通过马尔可夫假设（虽然并不一定成立）简化了语言模型的计算。这里的马尔可夫假设是指一个词的出现只与前面 n 个词相关，即 n 阶马尔可夫链（Markov chain of order n ）。如果 $n = 1$ ，那么有 $P(w_3 | w_1, w_2) = P(w_3 | w_2)$ 。如果基于 $n - 1$ 阶马尔可夫链，我们可以将语言模型改写为

$$P(w_1, w_2, \dots, w_T) \approx \prod_{t=1}^T P(w_t | w_{t-(n-1)}, \dots, w_{t-1}).$$

以上也叫 n 元语法 (n -grams)。它是基于 $n - 1$ 阶马尔可夫链的概率语言模型。当 n 分别为 1、2 和 3 时，我们将其分别称作一元语法 (unigram)、二元语法 (bigram) 和三元语法 (trigram)。例如，长度为 4 的序列 w_1, w_2, w_3, w_4 在一元语法、二元语法和三元语法中的概率分别为

$$\begin{aligned} P(w_1, w_2, w_3, w_4) &= P(w_1)P(w_2)P(w_3)P(w_4), \\ P(w_1, w_2, w_3, w_4) &= P(w_1)P(w_2 | w_1)P(w_3 | w_2)P(w_4 | w_3), \\ P(w_1, w_2, w_3, w_4) &= P(w_1)P(w_2 | w_1)P(w_3 | w_1, w_2)P(w_4 | w_2, w_3). \end{aligned}$$

当 n 较小时， n 元语法往往并不准确。例如，在一元语法中，由三个词组成的句子“你走先”和“你先走”的概率是一样的。然而，当 n 较大时， n 元语法需要计算并存储大量的词频和多词相邻频率。

那么，有没有方法在语言模型中更好地平衡以上这两点呢？我们将在本章探究这样的方法。

小结

- 语言模型是自然语言处理的重要技术。
- N 元语法是基于 $n - 1$ 阶马尔可夫链的概率语言模型，其中 n 权衡了计算复杂度和模型准确性。

注：本节与原书此节完全相同，[原书传送门](#)

6.2 循环神经网络

上一节介绍的 n 元语法中，时间步 t 的词 w_t 基于前面所有词的条件概率只考虑了最近时间步的 $n - 1$ 个词。如果要考虑比 $t - (n - 1)$ 更早时间步的词对 w_t 的可能影响，我们需要增大 n 。但这样模型参数的数量将随之呈指数级增长。

本节将介绍循环神经网络。它并非刚性地记忆所有固定长度的序列，而是通过隐藏状态来存储之前时间步的信息。首先我们回忆一下前面介绍过的多层次感知机，然后描述如何添加隐藏状态来将它变成循环神经网络。

6.2.1 不含隐藏状态的神经网络

让我们考虑一个含单隐藏层的多层次感知机。给定样本数为 n 、输入个数（特征数或特征向量维度）为 d 的小批量数据样本 $\mathbf{X} \in \mathbb{R}^{n \times d}$ 。设隐藏层的激活函数为 ϕ ，那么隐藏层的输出 $\mathbf{H} \in \mathbb{R}^{n \times h}$ 计算为

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h),$$

其中隐藏层权重参数 $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ ，隐藏层偏差参数 $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ， h 为隐藏单元个数。上式相加的两项形状不同，因此将按照广播机制相加。把隐藏变量 \mathbf{H} 作为输出层的输入，且设输出个数为 q （如分类问题中的类别数），输出层的输出为

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q,$$

其中输出变量 $\mathbf{O} \in \mathbb{R}^{n \times q}$ ，输出层权重参数 $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ ，输出层偏差参数 $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ 。如果是分类问题，我们可以使用 $\text{softmax}(\mathbf{O})$ 来计算输出类别的概率分布。

6.2.2 含隐藏状态的循环神经网络

现在我们考虑输入数据存在时间相关性的情况。假设 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ 是序列中时间步 t 的小批量输入， $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ 是该时间步的隐藏变量。与多层次感知机不同的是，这里我们保存上一时间步的隐藏变量 \mathbf{H}_{t-1} ，并引入一个新的权重参数 $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ ，该参数用来描述在当前时间步如何使用上一时间步的隐藏变量。具体来说，时间步 t 的隐藏变量的计算由当前时间步的输入和上一时间步的隐藏变量共同决定：

$$\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h).$$

与多层次感知机相比，我们在这里添加了 $\mathbf{H}_{t-1}\mathbf{W}_{hh}$ 一项。由上式中相邻时间步的隐藏变量 \mathbf{H}_t 和 \mathbf{H}_{t-1} 之间的关系可知，这里的隐藏变量能够捕捉截至当前时间步的序列

的历史信息，就像是神经网络当前时间步的状态或记忆一样。因此，该隐藏变量也称为隐藏状态。由于隐藏状态在当前时间步的定义使用了上一时间步的隐藏状态，上式的计算是循环的。使用循环计算的网络即循环神经网络 (recurrent neural network)。

循环神经网络有很多种不同的构造方法。含上式所定义的隐藏状态的循环神经网络是极为常见的一种。若无特别说明，本章中的循环神经网络均基于上式中隐藏状态的循环计算。在时间步 t ，输出层的输出和多层感知机中的计算类似：

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q.$$

循环神经网络的参数包括隐藏层的权重 $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ 、 $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ 和偏差 $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ，以及输出层的权重 $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ 和偏差 $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ 。值得一提的是，即便在不同时间步，循环神经网络也始终使用这些模型参数。因此，循环神经网络模型参数的数量不随时间步的增加而增长。

图 6.1 展示了循环神经网络在 3 个相邻时间步的计算逻辑。在时间步 t ，隐藏状态的计算可以看成是将输入 \mathbf{X}_t 和前一时间步隐藏状态 \mathbf{H}_{t-1} 连结后输入一个激活函数为 ϕ 的全连接层。该全连接层的输出就是当前时间步的隐藏状态 \mathbf{H}_t ，且模型参数为 \mathbf{W}_{xh} 与 \mathbf{W}_{hh} 的连结，偏差为 \mathbf{b}_h 。当前时间步 t 的隐藏状态 \mathbf{H}_t 将参与下一个时间步 $t+1$ 的隐藏状态 \mathbf{H}_{t+1} 的计算，并输入到当前时间步的全连接输出层。

图 6.1 含隐藏状态的循环神经网络

我们刚刚提到，隐藏状态中 $\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh}$ 的计算等价于 \mathbf{X}_t 与 \mathbf{H}_{t-1} 连结后的矩阵乘以 \mathbf{W}_{xh} 与 \mathbf{W}_{hh} 连结后的矩阵。接下来，我们用一个具体的例子来验证这一点。首先，我们构造矩阵 \mathbf{X} 、 \mathbf{W}_{xh} 、 \mathbf{H} 和 \mathbf{W}_{hh} ，它们的形状分别为 $(3, 1)$ 、 $(1, 4)$ 、 $(3, 4)$ 和 $(4, 4)$ 。将 \mathbf{X} 与 \mathbf{W}_{xh} 、 \mathbf{H} 与 \mathbf{W}_{hh} 分别相乘，再把两个乘法运算的结果相加，得到形状为 $(3, 4)$ 的矩阵。

```
import torch
```

```
X, W_xh = torch.randn(3, 1), torch.randn(1, 4)
H, W_hh = torch.randn(3, 4), torch.randn(4, 4)
torch.matmul(X, W_xh) + torch.matmul(H, W_hh)
```

输出：

```
tensor([[ 5.2633, -3.2288,  0.6037, -1.3321],
       [ 9.4012, -6.7830,  1.0630, -0.1809],
       [ 7.0355, -2.2361,  0.7469, -3.4667]])
```

将矩阵 X 和 H 按列（维度 1）连结，连结后的矩阵形状为 $(3, 5)$ 。可见，连结后矩阵在维度 1 的长度为矩阵 X 和 H 在维度 1 的长度之和 $(1 + 4)$ 。然后，将矩阵 W_{xh} 和 W_{hh} 按行（维度 0）连结，连结后的矩阵形状为 $(5, 4)$ 。最后将两个连结后的矩阵相乘，得到与上面代码输出相同的形状为 $(3, 4)$ 的矩阵。

```
torch.matmul(torch.cat((X, H), dim=1), torch.cat((W_xh, W_hh), dim=0))
```

输出：

```
tensor([[ 5.2633, -3.2288,  0.6037, -1.3321],
        [ 9.4012, -6.7830,  1.0630, -0.1809],
        [ 7.0355, -2.2361,  0.7469, -3.4667]])
```

6.2.3 应用：基于字符级循环神经网络的语言模型

最后我们介绍如何应用循环神经网络来构建一个语言模型。设小批量中样本数为 1，文本序列为“想”“要”“有”“直”“升”“机”。图 6.2 演示了如何使用循环神经网络基于当前和过去的字符来预测下一个字符。在训练时，我们对每个时间步的输出层输出使用 softmax 运算，然后使用交叉熵损失函数来计算它与标签的误差。在图 6.2 中，由于隐藏层中隐藏状态的循环计算，时间步 3 的输出 O_3 取决于文本序列“想”“要”“有”。由于训练数据中该序列的下一个词为“直”，时间步 3 的损失将取决于该时间步基于序列“想”“要”“有”生成下一个词的概率分布与该时间步的标签“直”。

图 6.2 基于字符级循环神经网络的语言模型。

因为每个输入词是一个字符，因此这个模型被称为字符级循环神经网络 (character-level recurrent neural network)。因为不同字符的个数远小于不同词的个数（对于英文尤其如此），所以字符级循环神经网络的计算通常更加简单。在接下来的几节里，我们将介绍它的具体实现。

小结

- 使用循环计算的网络即循环神经网络。
- 循环神经网络的隐藏状态可以捕捉截至当前时间步的序列的历史信息。
- 循环神经网络模型参数的数量不随时间步的增加而增长。
- 可以基于字符级循环神经网络来创建语言模型。

注：除代码外本节与原书此节基本相同，[原书传送门](#)

6.3 语言模型数据集（周杰伦专辑歌词）

本节将介绍如何预处理一个语言模型数据集，并将其转换成字符级循环神经网络所需要的输入格式。为此，我们收集了周杰伦从第一张专辑《Jay》到第十张专辑《跨时代》中的歌词，并在后面几节里应用循环神经网络来训练一个语言模型。当模型训练好后，我们就可以用这个模型来创作歌词。

6.3.1 读取数据集

首先读取这个数据集，看看前 40 个字符是什么样的。

```
import torch
import random
import zipfile

with zipfile.ZipFile('.../data/jaychou_lyrics.txt.zip') as zin:
    with zin.open('jaychou_lyrics.txt') as f:
        corpus_chars = f.read().decode('utf-8')
corpus_chars[:40]
```

输出：

```
'想要有直升机\n想要和你飞到宇宙去\n想要和你融化在一起\n融化在宇宙里\n我每天每天每'
```

这个数据集有 6 万多个字符。为了打印方便，我们把换行符替换成空格，然后仅使用前 1 万个字符来训练模型。

```
corpus_chars = corpus_chars.replace('\n', ' ').replace('\r', ' ')
corpus_chars = corpus_chars[0:10000]
```

6.3.2 建立字符索引

我们将每个字符映射成一个从 0 开始的连续整数，又称索引，来方便之后的数据处理。为了得到索引，我们将数据集里所有不同字符取出来，然后将其逐一映射到索引来构造词典。接着，打印 `vocab_size`，即词典中不同字符的个数，又称词典大小。

```
idx_to_char = list(set(corpus_chars))
char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
vocab_size = len(char_to_idx)
vocab_size # 1027
```

之后，将训练数据集中每个字符转化为索引，并打印前 20 个字符及其对应的索引。

```
corpus_indices = [char_to_idx[char] for char in corpus_chars]
sample = corpus_indices[:20]
print('chars:', ''.join([idx_to_char[idx] for idx in sample]))
print('indices:', sample)
```

输出：

```
chars: 想要有直升机 想要和你飞到宇宙去 想要和
```

```
indices: [250, 164, 576, 421, 674, 653, 357, 250, 164, 850, 217, 910, 1012, 261, 275,
```

我们将以上代码封装在 `d2lzh_pytorch` 包里的 `load_data_jay_lyrics` 函数中，以方便后面章节调用。调用该函数后会依次得到 `corpus_indices`、`char_to_idx`、`idx_to_char` 和 `vocab_size` 这 4 个变量。

6.3.3 时序数据的采样

在训练中我们需要每次随机读取小批量样本和标签。与之前章节的实验数据不同的是，时序数据的一个样本通常包含连续的字符。假设时间步数为 5，样本序列为 5 个字符，即“想”“要”“有”“直”“升”。该样本的标签序列为这些字符分别在训练集中的下一个字符，即“要”“有”“直”“升”“机”。我们有两种方式对时序数据进行采样，分别是随机采样和相邻采样。

6.3.3.1 随机采样

下面的代码每次从数据里随机采样一个小批量。其中批量大小 `batch_size` 指每个小批量的样本数，`num_steps` 为每个样本所包含的时间步数。在随机采样中，每个样本是原始序列上任意截取的一段序列。相邻的两个随机小批量在原始序列上的位置不一定相毗邻。因此，我们无法用一个小批量最终时间步的隐藏状态来初始化下一个批量的隐藏状态。在训练模型时，每次随机采样前都需要重新初始化隐藏状态。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def data_iter_random(corpus_indices, batch_size, num_steps, device=None):
    # 减 1 是因为输出的索引 x 是相应输入的索引 y 加 1
    num_examples = (len(corpus_indices) - 1) // num_steps
    epoch_size = num_examples // batch_size
    example_indices = list(range(num_examples))
```

```
random.shuffle(example_indices)

# 返回从 pos 开始的长为 num_steps 的序列
def _data(pos):
    return corpus_indices[pos: pos + num_steps]
if device is None:
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

for i in range(epoch_size):
    # 每次读取 batch_size 个随机样本
    i = i * batch_size
    batch_indices = example_indices[i: i + batch_size]
    X = [_data(j * num_steps) for j in batch_indices]
    Y = [_data(j * num_steps + 1) for j in batch_indices]
    yield torch.tensor(X, dtype=torch.float32, device=device), torch.tensor(Y, dt
```

让我们输入一个从 0 到 29 的连续整数的人工序列。设批量大小和时间步数分别为 2 和 6。打印随机采样每次读取的小批量样本的输入 X 和标签 Y。可见，相邻的两个随机小批量在原始序列上的位置不一定相毗邻。

```
my_seq = list(range(30))
for X, Y in data_iter_random(my_seq, batch_size=2, num_steps=6):
    print('X: ', X, '\nY: ', Y, '\n')
```

输出：

```
X: tensor([[18., 19., 20., 21., 22., 23.],
           [12., 13., 14., 15., 16., 17.]])
Y: tensor([[19., 20., 21., 22., 23., 24.],
           [13., 14., 15., 16., 17., 18.]])
```



```
X: tensor([[ 0.,  1.,  2.,  3.,  4.,  5.],
           [ 6.,  7.,  8.,  9., 10., 11.]])
Y: tensor([[ 1.,  2.,  3.,  4.,  5.,  6.],
           [ 7.,  8.,  9., 10., 11., 12.]])
```

6.3.3.2 相邻采样

除对原始序列做随机采样之外，我们还可以令相邻的两个随机小批量在原始序列上的位置相毗邻。这时候，我们就可以用一个小批量最终时间步的隐藏状态来初始化下一个小批量的隐藏状态，从而使下一个批量的输出也取决于当前小批量的输入，并如此循环下去。这对实现循环神经网络造成了两方面影响：一方面，在训练模型时，我们只需在每一个迭代周期开始时初始化隐藏状态；另一方面，当多个相邻小批量通过传递隐藏状态串联起来时，模型参数的梯度计算将依赖所有串联起来的小批量序列。同一迭代周期中，随着迭代次数的增加，梯度的计算开销会越来越大。为了使模型参数的梯度计算只依赖一次迭代读取的小批量序列，我们可以在每次读取小批量前将隐藏状态从计算图中分离出来。我们将在下一节（循环神经网络的从零开始实现）的实现中了解这种处理方式。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def data_iter_consecutive(corpus_indices, batch_size, num_steps, device=None):
    if device is None:
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    corpus_indices = torch.tensor(corpus_indices, dtype=torch.float32, device=device)
    data_len = len(corpus_indices)
    batch_len = data_len // batch_size
    indices = corpus_indices[0: batch_size*batch_len].view(batch_size, batch_len)
    epoch_size = (batch_len - 1) // num_steps
    for i in range(epoch_size):
        i = i * num_steps
        X = indices[:, i: i + num_steps]
        Y = indices[:, i + 1: i + num_steps + 1]
        yield X, Y
```

同样的设置下，打印相邻采样每次读取的小批量样本的输入 X 和标签 Y 。相邻的两个随机小批量在原始序列上的位置相毗邻。

```
for X, Y in data_iter_consecutive(my_seq, batch_size=2, num_steps=6):
    print('X: ', X, '\nY: ', Y, '\n')
```

输出：

```
X:  tensor([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [15., 16., 17., 18., 19., 20.]])
```

```
Y: tensor([[ 1.,  2.,  3.,  4.,  5.,  6.],
           [16., 17., 18., 19., 20., 21.]])  
  
X: tensor([[ 6.,  7.,  8.,  9., 10., 11.],
           [21., 22., 23., 24., 25., 26.]])  
Y: tensor([[ 7.,  8.,  9., 10., 11., 12.],
           [22., 23., 24., 25., 26., 27.]])
```

小结

- 时序数据采样方式包括随机采样和相邻采样。使用这两种方式的循环神经网络训练在实现上略有不同。

注：除代码外本节与原书此节基本相同，[原书传送门](#)

6.4 循环神经网络的从零开始实现

在本节中，我们将从零开始实现一个基于字符级循环神经网络的语言模型，并在周杰伦专辑歌词数据集上训练一个模型来进行歌词创作。首先，我们读取周杰伦专辑歌词数据集：

```
import time
import math
import numpy as np
import torch
from torch import nn, optim
import torch.nn.functional as F

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

(corpus_indices, char_to_idx, idx_to_char, vocab_size) = d2l.load_data_jay_lyrics()
```

6.4.1 one-hot 向量

为了将词表示成向量输入到神经网络，一个简单的办法是使用 one-hot 向量。假设词典中不同字符的数量为 N （即词典大小 `vocab_size`），每个字符已经同一个从 0 到 $N - 1$ 的连续整数值索引一一对应。如果一个字符的索引是整数 i ，那么我们创建一个全 0 的长为 N 的向量，并将其位置为 i 的元素设成 1。该向量就是对原字符的 one-hot 向量。下面分别展示了索引为 0 和 2 的 one-hot 向量，向量长度等于词典大小。> pytorch 没有自带 one-hot 函数（新版好像有了），下面自己实现一个

```
def one_hot(x, n_class, dtype=torch.float32):
    # X shape: (batch), output shape: (batch, n_class)
    x = x.long()
    res = torch.zeros(x.shape[0], n_class, dtype=dtype, device=x.device)
    res.scatter_(1, x.view(-1, 1), 1)
    return res
```

```
x = torch.tensor([0, 2])
one_hot(x, vocab_size)
```

我们每次采样的小批量的形状是 (批量大小, 时间步数)。下面的函数将这样的小批量变换为数个可以输入进网络的形状为 (批量大小, 词典大小) 的矩阵, 矩阵个数等于时间步数。也就是说, 时间步 t 的输入为 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$, 其中 n 为批量大小, d 为输入个数, 即 one-hot 向量长度 (词典大小)。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def to_onehot(X, n_class):
    # X shape: (batch, seq_len), output: seq_len elements of (batch, n_class)
    return [one_hot(X[:, i], n_class) for i in range(X.shape[1])]

X = torch.arange(10).view(2, 5)
inputs = to_onehot(X, vocab_size)
print(len(inputs), inputs[0].shape)
```

输出:

```
5 torch.Size([2, 1027])
```

6.4.2 初始化模型参数

接下来, 我们初始化模型参数。隐藏单元个数 `num_hiddens` 是一个超参数。

```
num_inputs, num_hiddens, num_outputs = vocab_size, 256, vocab_size
print('will use', device)

def get_params():
    def _one(shape):
        ts = torch.tensor(np.random.normal(0, 0.01, size=shape), device=device, dtype=torch.float32)
        return torch.nn.Parameter(ts, requires_grad=True)

    # 隐藏层参数
    W_xh = _one((num_inputs, num_hiddens))
    W_hh = _one((num_hiddens, num_hiddens))
    b_h = torch.nn.Parameter(torch.zeros(num_hiddens, device=device, requires_grad=True))

    # 输出层参数
    W_hq = _one((num_hiddens, num_outputs))
    b_q = torch.nn.Parameter(torch.zeros(num_outputs, device=device, requires_grad=True))

    return W_xh, W_hh, b_h, W_hq, b_q
```

```

W_hq = _one((num_hiddens, num_outputs))
b_q = torch.nn.Parameter(torch.zeros(num_outputs, device=device, requires_grad=True))
return nn.ParameterList([W_xh, W_hh, b_h, W_hq, b_q])

```

6.4.3 定义模型

我们根据循环神经网络的计算表达式实现该模型。首先定义 `init_rnn_state` 函数来返回初始化的隐藏状态。它返回由一个形状为 (批量大小, 隐藏单元个数) 的值为 0 的 `NDArray` 组成的元组。使用元组是为了更便于处理隐藏状态含有多个 `NDArray` 的情况。

```

def init_rnn_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device), )

```

下面的 `rnn` 函数定义了在一个时间步里如何计算隐藏状态和输出。这里的激活函数使用了 `tanh` 函数。3.8 节（多层感知机）中介绍过，当元素在实数域上均匀分布时，`tanh` 函数值的均值为 0。

```

def rnn(inputs, state, params):
    # inputs 和 outputs 皆为 num_steps 个形状为 (batch_size, vocab_size) 的矩阵
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        H = torch.tanh(torch.matmul(X, W_xh) + torch.matmul(H, W_hh) + b_h)
        Y = torch.matmul(H, W_hq) + b_q
        outputs.append(Y)
    return outputs, (H,)

```

做个简单的测试来观察输出结果的个数（时间步数），以及第一个时间步的输出层输出的形状和隐藏状态的形状。

```

state = init_rnn_state(X.shape[0], num_hiddens, device)
inputs = to_onehot(X.to(device), vocab_size)
params = get_params()
outputs, state_new = rnn(inputs, state, params)
print(len(outputs), outputs[0].shape, state_new[0].shape)

```

输出：

```
5 torch.Size([2, 1027]) torch.Size([2, 256])
```

6.4.4 定义预测函数

以下函数基于前缀 `prefix`（含有数个字符的字符串）来预测接下来的 `num_chars` 个字符。这个函数稍显复杂，其中我们将循环神经单元 `rnn` 设置成了函数参数，这样在后面小节介绍其他循环神经网络时能重复使用这个函数。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def predict_rnn(prefix, num_chars, rnn, params, init_rnn_state,
                num_hiddens, vocab_size, device, idx_to_char, char_to_idx):
    state = init_rnn_state(1, num_hiddens, device)
    output = [char_to_idx[prefix[0]]]
    for t in range(num_chars + len(prefix) - 1):
        # 将上一时间步的输出作为当前时间步的输入
        X = to_onehot(torch.tensor([[output[-1]]], device=device), vocab_size)
        # 计算输出和更新隐藏状态
        (Y, state) = rnn(X, state, params)
        # 下一个时间步的输入是 prefix 里的字符或者当前的最佳预测字符
        if t < len(prefix) - 1:
            output.append(char_to_idx[prefix[t + 1]])
        else:
            output.append(int(Y[0].argmax(dim=1).item()))
    return ''.join([idx_to_char[i] for i in output])
```

我们先测试一下 `predict_rnn` 函数。我们将根据前缀“分开”创作长度为 10 个字符（不考虑前缀长度）的一段歌词。因为模型参数为随机值，所以预测结果也是随机的。

```
predict_rnn(' 分开', 10, rnn, params, init_rnn_state, num_hiddens, vocab_size,
            device, idx_to_char, char_to_idx)
```

输出：

'分开西圈绪升王凝瓜必客映'

6.4.5 裁剪梯度

循环神经网络中较容易出现梯度衰减或梯度爆炸。我们会在 6.6 节（通过时间反向传播）中解释原因。为了应对梯度爆炸，我们可以裁剪梯度（clip gradient）。假设我们把所有模型参数梯度的元素拼接成一个向量 \mathbf{g} ，并设裁剪的阈值是 θ 。裁剪后的梯度

$$\min \left(\frac{\theta}{\|g\|}, 1 \right) g$$

的 L_2 范数不超过 θ 。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def grad_clipping(params, theta, device):
    norm = torch.tensor([0.0], device=device)
    for param in params:
        norm += (param.grad.data ** 2).sum()
    norm = norm.sqrt().item()
    if norm > theta:
        for param in params:
            param.grad.data *= (theta / norm)
```

6.4.6 困惑度

我们通常使用困惑度 (perplexity) 来评价语言模型的好坏。回忆一下 3.4 节 (softmax 回归) 中交叉熵损失函数的定义。困惑度是对交叉熵损失函数做指数运算后得到的值。特别地，

- 最佳情况下，模型总是把标签类别的概率预测为 1，此时困惑度为 1；
- 最坏情况下，模型总是把标签类别的概率预测为 0，此时困惑度为正无穷；
- 基线情况下，模型总是预测所有类别的概率都相同，此时困惑度为类别个数。

显然，任何一个有效模型的困惑度必须小于类别个数。在本例中，困惑度必须小于词典大小 `vocab_size`。

6.4.7 定义模型训练函数

跟之前章节的模型训练函数相比，这里的模型训练函数有以下几点不同：

1. 使用困惑度评价模型。
2. 在迭代模型参数前裁剪梯度。
3. 对时序数据采用不同采样方法将导致隐藏状态初始化的不同。相关讨论可参考 6.3 节（语言模型数据集（周杰伦专辑歌词））。

另外，考虑到后面将介绍的其他循环神经网络，为了更通用，这里的函数实现更长一些。

```

# 本函数已保存在 d2lzh_pytorch 包中方便以后使用

def train_and_predict_rnn(rnn, get_params, init_rnn_state, num_hiddens,
                          vocab_size, device, corpus_indices, idx_to_char,
                          char_to_idx, is_random_iter, num_epochs, num_steps,
                          lr, clipping_theta, batch_size, pred_period,
                          pred_len, prefixes):

    if is_random_iter:
        data_iter_fn = d2l.data_iter_random
    else:
        data_iter_fn = d2l.data_iter_consecutive
    params = get_params()
    loss = nn.CrossEntropyLoss()

    for epoch in range(num_epochs):
        if not is_random_iter: # 如使用相邻采样，在 epoch 开始时初始化隐藏状态
            state = init_rnn_state(batch_size, num_hiddens, device)
        l_sum, n, start = 0.0, 0, time.time()
        data_iter = data_iter_fn(corpus_indices, batch_size, num_steps, device)
        for X, Y in data_iter:
            if is_random_iter: # 如使用随机采样，在每个小批量更新前初始化隐藏状态
                state = init_rnn_state(batch_size, num_hiddens, device)
            else:
                # 否则需要使用 detach 函数从计算图分离隐藏状态，这是为了
                # 使模型参数的梯度计算只依赖一次迭代读取的小批量序列（防止梯度计算开销太大）
                for s in state:
                    s.detach_()

            inputs = to_onehot(X, vocab_size)
            # outputs 有 num_steps 个形状为 (batch_size, vocab_size) 的矩阵
            (outputs, state) = rnn(inputs, state, params)
            # 拼接之后形状为 (num_steps * batch_size, vocab_size)
            outputs = torch.cat(outputs, dim=0)
            # Y 的形状是 (batch_size, num_steps)，转置后再变成长度为
            # batch * num_steps 的向量，这样跟输出的行一一对应
            y = torch.transpose(Y, 0, 1).contiguous().view(-1)


```

```

# 使用交叉熵损失计算平均分类误差
l = loss(outputs, y.long())

# 梯度清 0
if params[0].grad is not None:
    for param in params:
        param.grad.data.zero_()
l.backward()
grad_clipping(params, clipping_theta, device) # 裁剪梯度
d2l.sgd(params, lr, 1) # 因为误差已经取过均值，梯度不用再做平均
l_sum += l.item() * y.shape[0]
n += y.shape[0]

if (epoch + 1) % pred_period == 0:
    print('epoch %d, perplexity %f, time %.2f sec' %
          (epoch + 1, math.exp(l_sum / n), time.time() - start))
    for prefix in prefixes:
        print(' - ', predict_rnn(prefix, pred_len, rnn, params, init_rnn_state,
                               num_hiddens, vocab_size, device, idx_to_char, char_to_idx))

```

6.4.8 训练模型并创作歌词

现在我们可以训练模型了。首先，设置模型超参数。我们将根据前缀“分开”和“不分开”分别创作长度为 50 个字符（不考虑前缀长度）的一段歌词。我们每过 50 个迭代周期便根据当前训练的模型创作一段歌词。

```
num_epochs, num_steps, batch_size, lr, clipping_theta = 250, 35, 32, 1e-2
pred_period, pred_len, prefixes = 50, 50, ['分开', '不分开']
```

下面采用随机采样训练模型并创作歌词。

```
train_and_predict_rnn(rnn, get_params, init_rnn_state, num_hiddens,
                      vocab_size, device, corpus_indices, idx_to_char,
                      char_to_idx, True, num_epochs, num_steps, lr,
                      clipping_theta, batch_size, pred_period, pred_len,
                      prefixes)
```

输出：

epoch 50, perplexity 70.039647, time 0.11 sec

- 分开 我不要再想 我不能 想你的让我 我的可 你怎么 一颗四 一颗四 我不要 一颗两 一颗四

- 不分开 我不要再 你你的外 在人 别你的让我 狂的可 语人两 我不要 一颗两 一颗四 一颗四

epoch 100, perplexity 9.726828, time 0.12 sec

- 分开 一直的美栈人 一起看 我不要好生活 你知不觉 我已好好生活 我知道好生活 后知不觉

- 不分开堡 我不要再想 我不 我不 我不要再想你 不知不觉 你已经离开我 不知不觉 我跟了好

epoch 150, perplexity 2.864874, time 0.11 sec

- 分开 一只会停留 有不它元羞 这蜴什么奇怪的事都有 包括像猫的狗 印地安老斑鳩 平常话不

- 不分开扫 我不你再想 我不能再想 我不 我不 我不要再想你 不知不觉 你已经离开我 不知不

epoch 200, perplexity 1.597790, time 0.11 sec

- 分开 有杰伦 干 载颗拳满的让空美空主 相爱还有个人 再狠狠忘记 你爱过我的证 有晶莹的

- 不分开扫 我叫你爸 你打我妈 这样对吗干嘛这样 何必让它牵鼻子走 瞎 说底牵打我妈要 难道

epoch 250, perplexity 1.303903, time 0.12 sec

- 分开 有杰人开留 仙唱它怕羞 蜥蜴横著走 这里什么奇怪的事都有 包括像猫的狗 印地安老斑

- 不分开简 我不能再想 我不 我不 我不能 爱情走的太快就像龙卷风 不能承受我已无处可躲 我

接下来采用相邻采样训练模型并创作歌词。

```
train_and_predict_rnn(rnn, get_params, init_rnn_state, num_hiddens,
                      vocab_size, device, corpus_indices, idx_to_char,
                      char_to_idx, False, num_epochs, num_steps, lr,
                      clipping_theta, batch_size, pred_period, pred_len,
                      prefixes)
```

输出：

epoch 50, perplexity 59.514416, time 0.11 sec

- 分开 我想要这 我想了空 我想了空 我想了空 我想了空 我想了空 我想了空 我想了空 我想了

- 不分开 我不要这 全使了双 我想了这 我想了空 我想了空 我想了空 我想了空 我想了空 我想了

epoch 100, perplexity 6.801417, time 0.11 sec

- 分开 我说的这样笑 想你都 不着我 我想就这样牵 你你的回不笑多难的 它在云实 有一条事

- 不分开觉 你已经离开我 不知不觉 我跟好这节活 我该好好生活 不知不觉 你跟了离开我 不知

epoch 150, perplexity 2.063730, time 0.16 sec

- 分开 我有到这样牵着你的手不放开 爱可不可以简简单单没有伤 古有你烦 我有多烦恼向 你

- 不分开觉 你已经很个我 不知不觉 我跟了这节奏 后知后觉 又过了一个秋 后哼哈兮 快使用双

epoch 200, perplexity 1.300031, time 0.11 sec

- 分开 我想要这样牵着你的手不放开 爱能不能够永远单甜没有伤害 你 靠着我的肩膀 你 在我

- 不分开觉 你已经离开我 不知不觉 我跟了这节奏 后知后觉 又过了一个秋 后知后觉 我该好好epoch 250, perplexity 1.164455, time 0.11 sec
- 分开 我有一这样布 对你依依不舍 连隔壁邻居都猜到我现在的感受 河边的风 在吹着头发飘动
- 不分开觉 你已经离开我 不知不觉 我跟了这节奏 后知后觉 又过了一个秋 后知后觉 我该好好

小结

- 可以用基于字符级循环神经网络的语言模型来生成文本序列，例如创作歌词。
 - 当训练循环神经网络时，为了应对梯度爆炸，可以裁剪梯度。
 - 困惑度是对交叉熵损失函数做指数运算后得到的值。
-

注：除代码外本节与原书此节基本相同，[原书传送门](#)

6.5 循环神经网络的简洁实现

本节将使用 PyTorch 来更简洁地实现基于循环神经网络的语言模型。首先，我们读取周杰伦专辑歌词数据集。

```
import time
import math
import numpy as np
import torch
from torch import nn, optim
import torch.nn.functional as F

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

(corpus_indices, char_to_idx, idx_to_char, vocab_size) = d2l.load_data_jay_lyrics()
```

6.5.1 定义模型

PyTorch 中的 `nn` 模块提供了循环神经网络的实现。下面构造一个含单隐藏层、隐藏单元个数为 256 的循环神经网络层 `rnn_layer`。

```
num_hiddens = 256
# rnn_layer = nn.LSTM(input_size=vocab_size, hidden_size=num_hiddens) # 已测试
rnn_layer = nn.RNN(input_size=vocab_size, hidden_size=num_hiddens)
```

与上一节中实现的循环神经网络不同，这里 `rnn_layer` 的输入形状为 (时间步数, 批量大小, 输入个数)。其中输入个数即 one-hot 向量长度 (词典大小)。此外，`rnn_layer` 作为 `nn.RNN` 实例，在前向计算后会分别返回输出和隐藏状态 `h`，其中输出指的是隐藏层在**各个时间步**上计算并输出的隐藏状态，它们通常作为后续输出层的输入。需要强调的是，该“输出”本身并不涉及输出层计算，形状为 (时间步数, 批量大小, 隐藏单元个数)。而 `nn.RNN` 实例在前向计算返回的隐藏状态指的是隐藏层在**最后时间步**的隐藏状态：当隐藏层有多层时，每一层的隐藏状态都会记录在该变量中；对于像长短期记忆 (LSTM)，隐藏状态是一个元组 (`h, c`)，即 hidden state 和 cell state。我们会在本章的后面介绍长短期记忆和深度循环神经网络。关于循环神经网络（以 LSTM 为例）的输出，可以参考下图 ([图片来源](#))。

循环神经网络（以 LSTM 为例）的输出

来看看我们的例子，输出形状为（时间步数，批量大小，输入个数），隐藏状态 h 的形状为（层数，批量大小，隐藏单元个数）。

```
num_steps = 35
batch_size = 2
state = None
X = torch.rand(num_steps, batch_size, vocab_size)
Y, state_new = rnn_layer(X, state)
print(Y.shape, len(state_new), state_new[0].shape)
```

输出：

```
torch.Size([35, 2, 256]) 1 torch.Size([2, 256])
```

如果 `rnn_layer` 是 `nn.LSTM` 实例，那么上面的输出是什么？

接下来我们继承 `Module` 类来定义一个完整的循环神经网络。它首先将输入数据使用 one-hot 向量表示后输入到 `rnn_layer` 中，然后使用全连接输出层得到输出。输出个数等于词典大小 `vocab_size`。

```
# 本类已保存在 d2lzh_pytorch 包中方便以后使用
class RNNModel(nn.Module):
    def __init__(self, rnn_layer, vocab_size):
        super(RNNModel, self).__init__()
        self.rnn = rnn_layer
        self.hidden_size = rnn_layer.hidden_size * (2 if rnn_layer.bidirectional else 1)
        self.vocab_size = vocab_size
        self.dense = nn.Linear(self.hidden_size, vocab_size)
        self.state = None

    def forward(self, inputs, state): # inputs: (batch, seq_len)
        # 获取 one-hot 向量表示
        X = d2l.to_onehot(inputs, self.vocab_size) # X 是个 list
        Y, self.state = self.rnn(torch.stack(X), state)
        # 全连接层会首先将 Y 的形状变成 (num_steps * batch_size, num_hiddens)，它的输出
        # 形状为 (num_steps * batch_size, vocab_size)
        output = self.dense(Y.view(-1, Y.shape[-1]))
        return output, self.state
```

6.5.2 训练模型

同上一节一样，下面定义一个预测函数。这里的实现区别在于前向计算和初始化隐藏状态的函数接口。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def predict_rnn_pytorch(prefix, num_chars, model, vocab_size, device, idx_to_char,
                        char_to_idx):
    state = None
    output = [char_to_idx[prefix[0]]] # output 会记录 prefix 加上输出
    for t in range(num_chars + len(prefix) - 1):
        X = torch.tensor([output[-1]], device=device).view(1, 1)
        if state is not None:
            if isinstance(state, tuple): # LSTM, state:(h, c)
                state = (state[0].to(device), state[1].to(device))
            else:
                state = state.to(device)

        (Y, state) = model(X, state)
        if t < len(prefix) - 1:
            output.append(char_to_idx[prefix[t + 1]])
        else:
            output.append(int(Y.argmax(dim=1).item()))
    return ''.join([idx_to_char[i] for i in output])
```

让我们使用权重为随机值的模型来预测一次。

```
model = RNNModel(rnn_layer, vocab_size).to(device)
predict_rnn_pytorch('分开', 10, model, vocab_size, device, idx_to_char, char_to_idx)
```

输出：

```
'分开戏想暖迎凉想征凉征征'
```

接下来实现训练函数。算法同上一节的一样，但这里只使用了相邻采样来读取数据。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size, device,
```

```
corpus_indices, idx_to_char, char_to_idx,
num_epochs, num_steps, lr, clipping_theta,
batch_size, pred_period, pred_len, prefixes):

loss = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
model.to(device)
state = None

for epoch in range(num_epochs):
    l_sum, n, start = 0.0, 0, time.time()
    data_iter = d2l.data_iter_consecutive(corpus_indices, batch_size, num_steps,
    for X, Y in data_iter:
        if state is not None:
            # 使用 detach 函数从计算图分离隐藏状态，这是为了
            # 使模型参数的梯度计算只依赖一次迭代读取的小批量序列（防止梯度计算开销过大）
            if isinstance(state, tuple): # LSTM, state:(h, c)
                state = (state[0].detach(), state[1].detach())
            else:
                state = state.detach()

        (output, state) = model(X, state) # output: 形状为 (num_steps * batch_size, vocab_size)

        # Y 的形状是 (batch_size, num_steps)，转置后再变成长度为
        # batch * num_steps 的向量，这样跟输出的行一一对应
        y = torch.transpose(Y, 0, 1).contiguous().view(-1)
        l = loss(output, y.long())

        optimizer.zero_grad()
        l.backward()
        # 梯度裁剪
        d2l.grad_clipping(model.parameters(), clipping_theta, device)
        optimizer.step()
        l_sum += l.item() * y.shape[0]
        n += y.shape[0]

    try:
```

```

perplexity = math.exp(l_sum / n)
except OverflowError:
    perplexity = float('inf')
if (epoch + 1) % pred_period == 0:
    print('epoch %d, perplexity %f, time %.2f sec' %
          (epoch + 1, perplexity, time.time() - start))
for prefix in prefixes:
    print(' -', predict_rnn_pytorch(
        prefix, pred_len, model, vocab_size, device, idx_to_char,
        char_to_idx))

```

使用和上一节实验中一样的超参数（除了学习率）来训练模型。

```

num_epochs, batch_size, lr, clipping_theta = 250, 32, 1e-3, 1e-2 # 注意这里的学习率设置
pred_period, pred_len, prefixes = 50, 50, ['分开', '不分开']
train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size, device,
                               corpus_indices, idx_to_char, char_to_idx,
                               num_epochs, num_steps, lr, clipping_theta,
                               batch_size, pred_period, pred_len, prefixes)

```

输出：

```

epoch 50, perplexity 10.658418, time 0.05 sec
- 分开始我妈 想要你 我不多 让我心到的 我妈妈 我不能再想 我不多再想 我不要再想 我不多
- 不分开 我想要你不你 我 你不要 让我心到的 我妈人 可爱女人 坏坏的让我疯狂的可爱女人
epoch 100, perplexity 1.308539, time 0.05 sec
- 分开不会痛 不要 你在黑色幽默 开始了美丽全脸的梦滴 闪烁成回忆 伤人的美丽 你的完美主
- 不分开不是我不要再想你 我不能这样牵着你的手不放开 爱可不可以简简单单没有伤害 你靠
epoch 150, perplexity 1.070370, time 0.05 sec
- 分开不能去河南嵩山 学少林跟武当 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈兮 习武之
- 不分开 在我会想通 是谁开没有全有开始 他心今天 一切人看 我 一口令秋软语的姑娘缓缓走
epoch 200, perplexity 1.034663, time 0.05 sec
- 分开不能去吗周杰伦 才离 没要你在一场悲剧 我的完美主义 太彻底 分手的话像语言暴力 我
- 不分开 让我面对你 爱情来的太快就像龙卷风 离不开暴风圈来不及逃 我不能再想 我不能再想
epoch 250, perplexity 1.021437, time 0.05 sec
- 分开 我我外的家边 你知道这 我爱不看的太 我想一个又重来不以 迷已文一只剩下回忆 让我
- 不分开 我我想想和 是你听没不 我不能不想 不知不觉 你已经离开我 不知不觉 我跟了这节

```

小结

- PyTorch 的 `nn` 模块提供了循环神经网络层的实现。
 - PyTorch 的 `nn.RNN` 实例在前向计算后会分别返回输出和隐藏状态。该前向计算并不涉及输出层计算。
-

注：除代码外本节与原书此节基本相同，[原书传送门](#)

6.6 通过时间反向传播

在前面两节中，如果不裁剪梯度，模型将无法正常训练。为了深刻理解这一现象，本节将介绍循环神经网络中梯度的计算和存储方法，即通过时间反向传播（back-propagation through time）。

我们在 3.14 节（正向传播、反向传播和计算图）中介绍了神经网络中梯度计算与存储的一般思路，并强调正向传播和反向传播相互依赖。正向传播在循环神经网络中比较直观，而通过时间反向传播其实是反向传播在循环神经网络中的具体应用。我们需要将循环神经网络按时间步展开，从而得到模型变量和参数之间的依赖关系，并依据链式法则应用反向传播计算并存储梯度。

6.6.1 定义模型

简单起见，我们考虑一个无偏差项的循环神经网络，且激活函数为恒等映射 ($\phi(x) = x$)。设时间步 t 的输入为单样本 $\mathbf{x}_t \in \mathbb{R}^d$ ，标签为 y_t ，那么隐藏状态 $\mathbf{h}_t \in \mathbb{R}^h$ 的计算表达式为

$$\mathbf{h}_t = \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1},$$

其中 $\mathbf{W}_{hx} \in \mathbb{R}^{h \times d}$ 和 $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ 是隐藏层权重参数。设输出层权重参数 $\mathbf{W}_{qh} \in \mathbb{R}^{q \times h}$ ，时间步 t 的输出层变量 $\mathbf{o}_t \in \mathbb{R}^q$ 计算为

$$\mathbf{o}_t = \mathbf{W}_{qh}\mathbf{h}_t.$$

设时间步 t 的损失为 $\ell(\mathbf{o}_t, y_t)$ 。时间步数为 T 的损失函数 L 定义为

$$L = \frac{1}{T} \sum_{t=1}^T \ell(\mathbf{o}_t, y_t).$$

我们将 L 称为有关给定时间步的数据样本的目标函数，并在本节后续讨论中简称为目标函数。

6.6.2 模型计算图

为了可视化循环神经网络中模型变量和参数在计算中的依赖关系，我们可以绘制模型计算图，如图 6.3 所示。例如，时间步 3 的隐藏状态 \mathbf{h}_3 的计算依赖模型参数 \mathbf{W}_{hx} 、 \mathbf{W}_{hh} 、上一时间步隐藏状态 \mathbf{h}_2 以及当前时间步输入 \mathbf{x}_3 。

图 6.3 时间步数为 3 的循环神经网络模型计算中的依赖关系。方框代表变量（无阴影）或参数（有阴影），圆圈代表运算符

6.6.3 方法

刚刚提到，图 6.3 中的模型的参数是 \mathbf{W}_{hx} , \mathbf{W}_{hh} 和 \mathbf{W}_{qh} 。与 3.14 节（正向传播、反向传播和计算图）中的类似，训练模型通常需要模型参数的梯度 $\partial L / \partial \mathbf{W}_{hx}$ 、 $\partial L / \partial \mathbf{W}_{hh}$ 和 $\partial L / \partial \mathbf{W}_{qh}$ 。根据图 6.3 中的依赖关系，我们可以按照其中箭头所指的反方向依次计算并存储梯度。为了表述方便，我们依然采用 3.14 节中表达链式法则的运算符 prod。

首先，目标函数有关各时间步输出层变量的梯度 $\partial L / \partial \mathbf{o}_t \in \mathbb{R}^q$ 很容易计算：

$$\frac{\partial L}{\partial \mathbf{o}_t} = \frac{\partial \ell(\mathbf{o}_t, y_t)}{T \cdot \partial \mathbf{o}_t}.$$

下面，我们可以计算目标函数有关模型参数 \mathbf{W}_{qh} 的梯度 $\partial L / \partial \mathbf{W}_{qh} \in \mathbb{R}^{q \times h}$ 。根据图 6.3, L 通过 $\mathbf{o}_1, \dots, \mathbf{o}_T$ 依赖 \mathbf{W}_{qh} 。依据链式法则，

$$\frac{\partial L}{\partial \mathbf{W}_{qh}} = \sum_{t=1}^T \text{prod} \left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{qh}} \right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{o}_t} \mathbf{h}_t^\top.$$

其次，我们注意到隐藏状态之间也存在依赖关系。在图 6.3 中， L 只通过 \mathbf{o}_T 依赖最终时间步 T 的隐藏状态 \mathbf{h}_T 。因此，我们先计算目标函数有关最终时间步隐藏状态的梯度 $\partial L / \partial \mathbf{h}_T \in \mathbb{R}^h$ 。依据链式法则，我们得到

$$\frac{\partial L}{\partial \mathbf{h}_T} = \text{prod} \left(\frac{\partial L}{\partial \mathbf{o}_T}, \frac{\partial \mathbf{o}_T}{\partial \mathbf{h}_T} \right) = \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_T}.$$

接下来对于时间步 $t < T$, 在图 6.3 中， L 通过 \mathbf{h}_{t+1} 和 \mathbf{o}_t 依赖 \mathbf{h}_t 。依据链式法则，目标函数有关时间步 $t < T$ 的隐藏状态的梯度 $\partial L / \partial \mathbf{h}_t \in \mathbb{R}^h$ 需要按照时间步从大到小依次计算：

$$\frac{\partial L}{\partial \mathbf{h}_t} = \text{prod} \left(\frac{\partial L}{\partial \mathbf{h}_{t+1}}, \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \right) + \text{prod} \left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t} \right) = \mathbf{W}_{hh}^\top \frac{\partial L}{\partial \mathbf{h}_{t+1}} + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_t}$$

将上面的递归公式展开，对任意时间步 $1 \leq t \leq T$, 我们可以得到目标函数有关隐藏状态梯度的通项公式

$$\frac{\partial L}{\partial \mathbf{h}_t} = \sum_{i=t}^T \left(\mathbf{W}_{hh}^\top \right)^{T-i} \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{T+t-i}}.$$

由上式中的指数项可见，当时间步数 T 较大或者时间步 t 较小时，目标函数有关隐藏状态的梯度较容易出现衰减和爆炸。这也会影响其他包含 $\partial L / \partial \mathbf{h}_t$ 项的梯度，例如隐藏层中模型参数的梯度 $\partial L / \partial \mathbf{W}_{hx} \in \mathbb{R}^{h \times d}$ 和 $\partial L / \partial \mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ 。在图 6.3 中， L 通过 $\mathbf{h}_1, \dots, \mathbf{h}_T$ 依赖这些模型参数。依据链式法则，我们有

$$\frac{\partial L}{\partial \mathbf{W}_{hx}} = \sum_{t=1}^T \text{prod} \left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hx}} \right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{x}_t^\top,$$

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^T \text{prod} \left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}} \right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{h}_{t-1}^\top.$$

我们已在 3.14 节里解释过，每次迭代中，我们在依次计算完以上各个梯度后，会将它们存储起来，从而避免重复计算。例如，由于隐藏状态梯度 $\partial L / \partial \mathbf{h}_t$ 被计算和存储，之后的模型参数梯度 $\partial L / \partial \mathbf{W}_{hx}$ 和 $\partial L / \partial \mathbf{W}_{hh}$ 的计算可以直接读取 $\partial L / \partial \mathbf{h}_t$ 的值，而无须重复计算它们。此外，反向传播中的梯度计算可能会依赖变量的当前值。它们正是通过正向传播计算出来的。举例来说，参数梯度 $\partial L / \partial \mathbf{W}_{hh}$ 的计算需要依赖隐藏状态在时间步 $t = 0, \dots, T - 1$ 的当前值 \mathbf{h}_t (\mathbf{h}_0 是初始化得到的)。这些值是通过从输入层到输出层的正向传播计算并存储得到的。

小结

- 通过时间反向传播是反向传播在循环神经网络中的具体应用。
- 当总的时间步数较大或者当前时间步较小时，循环神经网络的梯度较容易出现衰减或爆炸。

注：本节与原书基本相同，[原书传送门](#)

6.7 门控循环单元 (GRU)

上一节介绍了循环神经网络中的梯度计算方法。我们发现，当时间步数较大或者时间步较小时，循环神经网络的梯度较容易出现衰减或爆炸。虽然裁剪梯度可以应对梯度爆炸，但无法解决梯度衰减的问题。通常由于这个原因，循环神经网络在实际中较难捕捉时间序列中时间步距离较大的依赖关系。

门控循环神经网络 (gated recurrent neural network) 的提出，正是为了更好地捕捉时间序列中时间步距离较大的依赖关系。它通过可以学习的门来控制信息的流动。其中，门控循环单元 (gated recurrent unit, GRU) 是一种常用的门控循环神经网络 [1, 2]。另一种常用的门控循环神经网络则将在下一节中介绍。

6.7.1 门控循环单元

下面将介绍门控循环单元的设计。它引入了重置门 (reset gate) 和更新门 (update gate) 的概念，从而修改了循环神经网络中隐藏状态的计算方式。

6.7.1.1 重置门和更新门

如图 6.4 所示，门控循环单元中的重置门和更新门的输入均为当前时间步输入 \mathbf{X}_t 与上一时间步隐藏状态 \mathbf{H}_{t-1} ，输出由激活函数为 sigmoid 函数的全连接层计算得到。

图 6.4 门控循环单元中重置门和更新门的计算

具体来说，假设隐藏单元个数为 h ，给定时间步 t 的小批量输入 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (样本数为 n ，输入个数为 d) 和上一时间步隐藏状态 $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ 。重置门 $\mathbf{R}_t \in \mathbb{R}^{n \times h}$ 和更新门 $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$ 的计算如下：

$$\begin{aligned}\mathbf{R}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r), \\ \mathbf{Z}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z),\end{aligned}$$

其中 $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$ 和 $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$ 是权重参数， $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$ 是偏差参数。3.8 节 (多层感知机) 节中介绍过，sigmoid 函数可以将元素的值变换到 0 和 1 之间。因此，重置门 \mathbf{R}_t 和更新门 \mathbf{Z}_t 中每个元素的值域都是 $[0, 1]$ 。

6.7.1.2 候选隐藏状态

接下来，门控循环单元将计算候选隐藏状态来辅助稍后的隐藏状态计算。如图 6.5 所示，我们将当前时间步重置门的输出与上一时间步隐藏状态做按元素乘法 (符号为 \odot)。如果重置门中元素值接近 0，那么意味着重置对应隐藏状态元素为 0，即丢弃上一时间步的隐藏状态。如果元素值接近 1，那么表示保留上一时间步的隐藏状态。然后，

将按元素乘法的结果与当前时间步的输入连结，再通过含激活函数 \tanh 的全连接层计算出候选隐藏状态，其所有元素的值域为 $[-1, 1]$ 。

图 6.5 门控循环单元中候选隐藏状态的计算

具体来说，时间步 t 的候选隐藏状态 $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ 的计算为

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h),$$

其中 $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ 和 $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ 是权重参数， $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ 是偏差参数。从上面这个公式可以看出，重置门控制了上一时间步的隐藏状态如何流入当前时间步的候选隐藏状态。而上一时间步的隐藏状态可能包含了时间序列截至上一时间步的全部历史信息。因此，重置门可以用来丢弃与预测无关的历史信息。

6.7.1.3 隐藏状态

最后，时间步 t 的隐藏状态 $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ 的计算使用当前时间步的更新门 \mathbf{Z}_t 来对上一时间步的隐藏状态 \mathbf{H}_{t-1} 和当前时间步的候选隐藏状态 $\tilde{\mathbf{H}}_t$ 做组合：

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.$$

图 6.6 门控循环单元中隐藏状态的计算

值得注意的是，更新门可以控制隐藏状态应该如何被包含当前时间步信息的候选隐藏状态所更新，如图 6.6 所示。假设更新门在时间步 t' 到 t ($t' < t$) 之间一直近似 1。那么，在时间步 t' 到 t 之间的输入信息几乎没有流入时间步 t 的隐藏状态 \mathbf{H}_t 。实际上，这可以看作是较早时刻的隐藏状态 $\mathbf{H}_{t'-1}$ 一直通过时间保存并传递至当前时间步 t 。这个设计可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时间序列中时间步距离较大的依赖关系。

我们对门控循环单元的设计稍作总结：

- 重置门有助于捕捉时间序列里短期的依赖关系；
- 更新门有助于捕捉时间序列里长期的依赖关系。

6.7.2 读取数据集

为了实现并展示门控循环单元，下面依然使用周杰伦歌词数据集来训练模型作词。这里除门控循环单元以外的实现已在 6.2 节（循环神经网络）中介绍过。以下为读取数据集部分。

```
import numpy as np
import torch
```

```
from torch import nn, optim
import torch.nn.functional as F

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

(corpus_indices, char_to_idx, idx_to_char, vocab_size) = d2l.load_data_jay_lyrics()
```

6.7.3 从零开始实现

我们先介绍如何从零开始实现门控循环单元。

6.7.3.1 初始化模型参数

下面的代码对模型参数进行初始化。超参数 `num_hiddens` 定义了隐藏单元的个数。

```
num_inputs, num_hiddens, num_outputs = vocab_size, 256, vocab_size
print('will use', device)

def get_params():
    def _one(shape):
        ts = torch.tensor(np.random.normal(0, 0.01, size=shape), device=device, dtype=torch.float32)
        return torch.nn.Parameter(ts, requires_grad=True)
    def _three():
        return (_one((num_inputs, num_hiddens)),
                _one((num_hiddens, num_hiddens)),
                torch.nn.Parameter(torch.zeros(num_hiddens, device=device, dtype=torch.float32)))

    W_xz, W_hz, b_z = _three() # 更新门参数
    W_xr, W_hr, b_r = _three() # 重置门参数
    W_xh, W_hh, b_h = _three() # 候选隐藏状态参数

    # 输出层参数
    W_hq = _one((num_hiddens, num_outputs))
```

```
b_q = torch.nn.Parameter(torch.zeros(num_outputs, device=device, dtype=torch.float))
return nn.ParameterList([W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq,
```

6.7.3.2 定义模型

下面的代码定义隐藏状态初始化函数 `init_gru_state`。同 6.4 节（循环神经网络的从零开始实现）中定义的 `init_rnn_state` 函数一样，它返回由一个形状为（批量大小, 隐藏单元个数）的值为 0 的 `Tensor` 组成的元组。

```
def init_gru_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device), )
```

下面根据门控循环单元的计算表达式定义模型。

```
def gru(inputs, state, params):
    W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        Z = torch.sigmoid(torch.matmul(X, W_xz) + torch.matmul(H, W_hz) + b_z)
        R = torch.sigmoid(torch.matmul(X, W_xr) + torch.matmul(H, W_hr) + b_r)
        H_tilda = torch.tanh(torch.matmul(X, W_xh) + R * torch.matmul(H, W_hh) + b_h)
        H = Z * H + (1 - Z) * H_tilda
        Y = torch.matmul(H, W_hq) + b_q
        outputs.append(Y)
    return outputs, (H,)
```

6.7.3.3 训练模型并创作歌词

我们在训练模型时只使用相邻采样。设置好超参数后，我们将训练模型并根据前缀“分开”和“不分开”分别创作长度为 50 个字符的一段歌词。

```
num_epochs, num_steps, batch_size, lr, clipping_theta = 160, 35, 32, 1e2, 1e-2
pred_period, pred_len, prefixes = 40, 50, ['分开', '不分开']
```

我们每过 40 个迭代周期便根据当前训练的模型创作一段歌词。

```
d2l.train_and_predict_rnn(gru, get_params, init_gru_state, num_hiddens,
                           vocab_size, device, corpus_indices, idx_to_char,
```

```
char_to_idx, False, num_epochs, num_steps, lr,
clipping_theta, batch_size, pred_period, pred_len,
prefixes)
```

输出：

```
epoch 40, perplexity 149.477598, time 1.08 sec
- 分开 我不不你 我想你你的爱我 你不你的让我 你不你的让我 你不你的让我 你不你的让我 你不你的让我 你不你的让我
- 不分开 我想你你的让我 你不你的让我 你不你的让我 你不你的让我 你不你的让我 你不你的让我 你不你的让我 epoch 80, perplexity 31.689210, time 1.10 sec
- 分开 我想要你 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想 我
- 不分开 我想要你 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想 epoch 120, perplexity 4.866115, time 1.08 sec
- 分开 我想要这样牵着你的手不放开 爱过 让我来的肩膀 一起好酒 你来了这节秋 后知后觉 我
- 不分开 你已经不了我不要 我不要再想你 我不要再想你 我不要再想你 不知不觉 我跟了这节 epoch 160, perplexity 1.442282, time 1.51 sec
- 分开 我一定好生忧 唱着歌 一直走 我想就这样牵着你的手不放开 爱可不可以简简单单没有你
- 不分开 你已经离开我 不知不觉 我跟了这节奏 后知后觉 又过了一个秋 后知后觉 我该好好生
```

6.7.4 简洁实现

在 PyTorch 中我们直接调用 nn 模块中的 GRU 类即可。

```
lr = 1e-2 # 注意调整学习率
gru_layer = nn.GRU(input_size=vocab_size, hidden_size=num_hiddens)
model = d2l.RNNModel(gru_layer, vocab_size).to(device)
d2l.train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size, device,
                                   corpus_indices, idx_to_char, char_to_idx,
                                   num_epochs, num_steps, lr, clipping_theta,
                                   batch_size, pred_period, pred_len, prefixes)
```

输出：

```
epoch 40, perplexity 1.022157, time 1.02 sec
- 分开手牵手 一步两步三步四步望著天 看星星 一颗两颗三颗四颗 连成线背著背默默许下心愿
- 不分开暴风圈来不及逃 我不能再想 我不能再想 我不 我不 我不能 爱情走的太快就像龙卷风 epoch 80, perplexity 1.014535, time 1.04 sec
- 分开始想像 爸和妈当年的模样 说著一口吴侬软语的姑娘缓缓走过外滩 消失的 旧时光 一九四
```

- 不分开始爱像 不知不觉 你已经离开我 不知不觉 我跟了这节奏 后知后觉 又过了一个秋 后 epoch 120, perplexity 1.147843, time 1.04 sec
- 分开都靠我 你拿着球不投 又不会掩护我 选你这种队友 瞎透了我 说你说 分数怎么停留 所有
- 不分开球我有多烦恼多 牧草有没有危险 一场梦 我面对我 甩开球我满腔的怒火 我想揍你已经 epoch 160, perplexity 1.018370, time 1.05 sec
- 分开爱上你 那场悲剧 是你完美演出的一场戏 宁愿心碎哭泣 再狠狠忘记 你爱过我的证据 让
- 不分开始 担心今天的你过得好不好 整个画面是你 想你想的睡不著 嘴嘟嘟那可爱的模样 还有

小结

- 门控循环神经网络可以更好地捕捉时间序列中时间步距离较大的依赖关系。
- 门控循环单元引入了门的概念，从而修改了循环神经网络中隐藏状态的计算方式。它包括重置门、更新门、候选隐藏状态和隐藏状态。
- 重置门有助于捕捉时间序列里短期的依赖关系。
- 更新门有助于捕捉时间序列里长期的依赖关系。

参考文献

- [1] Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259.
 - [2] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555.
-

注：除代码外本节与原书此节基本相同，[原书传送门](#)

6.8 长短期记忆 (LSTM)

本节将介绍另一种常用的门控循环神经网络:长短期记忆(long short-term memory, LSTM) [1]。它比门控循环单元的结构稍微复杂一点。

6.8.1 长短期记忆

LSTM 中引入了 3 个门, 即输入门 (input gate)、遗忘门 (forget gate) 和输出门 (output gate), 以及与隐藏状态形状相同的记忆细胞 (某些文献把记忆细胞当成一种特殊的隐藏状态), 从而记录额外的信息。

6.8.1.1 输入门、遗忘门和输出门

与门控循环单元中的重置门和更新门一样, 如图 6.7 所示, 长短期记忆的门的输入均为当前时间步输入 \mathbf{X}_t 与上一时间步隐藏状态 \mathbf{H}_{t-1} , 输出由激活函数为 sigmoid 函数的全连接层计算得到。如此一来, 这 3 个门元素的值域均为 $[0, 1]$ 。

图 6.7 长短期记忆中输入门、遗忘门和输出门的计算

具体来说, 假设隐藏单元个数为 h , 给定时间步 t 的小批量输入 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (样本数为 n , 输入个数为 d) 和上一时间步隐藏状态 $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ 。时间步 t 的输入门 $\mathbf{I}_t \in \mathbb{R}^{n \times h}$ 、遗忘门 $\mathbf{F}_t \in \mathbb{R}^{n \times h}$ 和输出门 $\mathbf{O}_t \in \mathbb{R}^{n \times h}$ 分别计算如下:

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i), \\ \mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f), \\ \mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),\end{aligned}$$

其中的 $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$ 和 $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$ 是权重参数, $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$ 是偏差参数。

6.8.1.2 候选记忆细胞

接下来, 长短期记忆需要计算候选记忆细胞 $\tilde{\mathbf{C}}_t$ 。它的计算与上面介绍的 3 个门类似, 但使用了值域在 $[-1, 1]$ 的 \tanh 函数作为激活函数, 如图 6.8 所示。

图 6.8 长短期记忆中候选记忆细胞的计算

具体来说, 时间步 t 的候选记忆细胞 $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$ 的计算为

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c),$$

其中 $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$ 和 $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$ 是权重参数, $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$ 是偏差参数。

6.8.1.3 记忆细胞

我们可以通过元素值域在 $[0, 1]$ 的输入门、遗忘门和输出门来控制隐藏状态中信息的流动，这一般也是通过使用按元素乘法（符号为 \odot ）来实现的。当前时间步记忆细胞 $\mathbf{C}_t \in \mathbb{R}^{n \times h}$ 的计算组合了上一时间步记忆细胞和当前时间步候选记忆细胞的信息，并通过遗忘门和输入门来控制信息的流动：

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t.$$

如图 6.9 所示，遗忘门控制上一时间步的记忆细胞 \mathbf{C}_{t-1} 中的信息是否传递到当前时间步，而输入门则控制当前时间步的输入 \mathbf{X}_t 通过候选记忆细胞 $\tilde{\mathbf{C}}_t$ 如何流入当前时间步的记忆细胞。如果遗忘门一直近似 1 且输入门一直近似 0，过去的信息将一直通过时间保存并传递至当前时间步。这个设计可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时间序列中时间步距离较大的依赖关系。

图 6.9 长短期记忆中记忆细胞的计算

6.8.1.4 隐藏状态

有了记忆细胞以后，接下来我们还可以通过输出门来控制从记忆细胞到隐藏状态 $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ 的信息的流动：

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t).$$

这里的 \tanh 函数确保隐藏状态元素值在 -1 到 1 之间。需要注意的是，当输出门近似 1 时，记忆细胞信息将传递到隐藏状态供输出层使用；当输出门近似 0 时，记忆细胞信息只自己保留。图 6.10 展示了长短期记忆中隐藏状态的计算。

图 6.10 长短期记忆中隐藏状态的计算

6.8.2 读取数据集

下面我们开始实现并展示长短期记忆。和前几节中的实验一样，这里依然使用周杰伦歌词数据集来训练模型作词。

```
import numpy as np
import torch
from torch import nn, optim
import torch.nn.functional as F

import sys
```

```

sys.path.append("..")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

(corpus_indices, char_to_idx, idx_to_char, vocab_size) = d2l.load_data_jay_lyrics()

```

6.8.3 从零开始实现

我们先介绍如何从零开始实现长短期记忆。

6.8.3.1 初始化模型参数

下面的代码对模型参数进行初始化。超参数 `num_hiddens` 定义了隐藏单元的个数。

```

num_inputs, num_hiddens, num_outputs = vocab_size, 256, vocab_size
print('will use', device)

def get_params():
    def _one(shape):
        ts = torch.tensor(np.random.normal(0, 0.01, size=shape), device=device, dtype=torch.float32)
        return torch.nn.Parameter(ts, requires_grad=True)
    def _three():
        return (_one((num_inputs, num_hiddens)),
                _one((num_hiddens, num_hiddens)),
                torch.nn.Parameter(torch.zeros(num_hiddens, device=device, dtype=torch.float32)))
    W_xi, W_hi, b_i = _three() # 输入门参数
    W_xf, W_hf, b_f = _three() # 遗忘门参数
    W_xo, W_ho, b_o = _three() # 输出门参数
    W_xc, W_hc, b_c = _three() # 候选记忆细胞参数

    # 输出层参数
    W_hq = _one((num_hiddens, num_outputs))
    b_q = torch.nn.Parameter(torch.zeros(num_outputs, device=device, dtype=torch.float32))
    return nn.ParameterList([W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc,
                           W_hq, b_q])

```

6.8.4 定义模型

在初始化函数中，长短期记忆的隐藏状态需要返回额外的形状为（批量大小，隐藏单元个数）的值为 0 的记忆细胞。

```
def init_lstm_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device),
            torch.zeros((batch_size, num_hiddens), device=device))
```

下面根据长短期记忆的计算表达式定义模型。需要注意的是，只有隐藏状态会传递到输出层，而记忆细胞不参与输出层的计算。

```
def lstm(inputs, state, params):
    [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c, W_hq, b_q] =
        (H, C) = state
    outputs = []
    for X in inputs:
        I = torch.sigmoid(torch.matmul(X, W_xi) + torch.matmul(H, W_hi) + b_i)
        F = torch.sigmoid(torch.matmul(X, W_xf) + torch.matmul(H, W_hf) + b_f)
        O = torch.sigmoid(torch.matmul(X, W_xo) + torch.matmul(H, W_ho) + b_o)
        C_tilda = torch.tanh(torch.matmul(X, W_xc) + torch.matmul(H, W_hc) + b_c)
        C = F * C + I * C_tilda
        H = O * C.tanh()
        Y = torch.matmul(H, W_hq) + b_q
        outputs.append(Y)
    return outputs, (H, C)
```

6.8.4.1 训练模型并创作歌词

同上一节一样，我们在训练模型时只使用相邻采样。设置好超参数后，我们将训练模型并根据前缀“分开”和“不分开”分别创作长度为 50 个字符的一段歌词。

```
num_epochs, num_steps, batch_size, lr, clipping_theta = 160, 35, 32, 1e2, 1e-2
pred_period, pred_len, prefixes = 40, 50, ['分开', '不分开']
```

我们每过 40 个迭代周期便根据当前训练的模型创作一段歌词。

```
d2l.train_and_predict_rnn(lstm, get_params, init_lstm_state, num_hiddens,
                           vocab_size, device, corpus_indices, idx_to_char,
```

```
char_to_idx, False, num_epochs, num_steps, lr,
clipping_theta, batch_size, pred_period, pred_len,
prefixes)
```

输出：

```
epoch 40, perplexity 211.416571, time 1.37 sec
- 分开 我不的我 我不的我 我不的我 我不的我 我不的我 我不的我 我不的我 我不的我 我不的我
- 不分开 我不的我 我不的我 我不的我 我不的我 我不的我 我不的我 我不的我 我不的我 我不的我
epoch 80, perplexity 67.048346, time 1.35 sec
- 分开 我想你你 我不要再想 我不要这我 我不要这我 我不要这我 我不要这我 我不要这我 我不要
- 不分开 我想你你想你 我不要这不样 我不要这我 我不要这我 我不要这我 我不要这我 我不要
epoch 120, perplexity 15.552743, time 1.36 sec
- 分开 我想带你的微笑 像这在 你想我 我想你 说你我 说你了 说给怎么 有你在空 你在空
- 不分开 我想要你已经堡 一样样 说你了 我想就这样着你 不知不觉 你已了离开活 后知后觉
epoch 160, perplexity 4.274031, time 1.35 sec
- 分开 我想带你 你不一外在半空 我只能够远远著她 这些我 你想我难头 一话看人对落我一
- 不分开 我想你这生堡 我知好烦 你不的节我 后知后觉 我该了这节奏 后知后觉 又过了一个秋
```

6.8.5 简洁实现

在 Gluon 中我们可以直接调用 `rnn` 模块中的 `LSTM` 类。

```
lr = 1e-2 # 注意调整学习率
lstm_layer = nn.LSTM(input_size=vocab_size, hidden_size=num_hiddens)
model = d2l.RNNModel(lstm_layer, vocab_size)
d2l.train_and_predict_rnn_pytorch(model, num_hiddens, vocab_size, device,
                                  corpus_indices, idx_to_char, char_to_idx,
                                  num_epochs, num_steps, lr, clipping_theta,
                                  batch_size, pred_period, pred_len, prefixes)
```

输出：

```
epoch 40, perplexity 1.020401, time 1.54 sec
- 分开始想担 妈跟我 一定是我妈在 因为分手前那句抱歉 在感动 穿梭时间的画面的钟 从反方
- 不分开始想像 妈跟我 我将我的寂寞封闭 然后在这里 不限日期 然后将过去 慢慢温习 让我爱
epoch 80, perplexity 1.011164, time 1.34 sec
- 分开始想担 你的 从前的可爱女人 温柔的让我心疼的可爱女人 透明的让我感动的可爱女人 块
```

- 不分开 我满了 让我疯狂的可爱女人 漂亮的让我面红的可爱女人 温柔的让我心疼的可爱女人
epoch 120, perplexity 1.025348, time 1.39 sec
- 分开始共渡每一天 手牵手 一步两步三步四步望著天 看星星 一颗两颗三颗四颗 连成线背著背
- 不分开 我不懂 说了没用 他的笑容 有何不同 在你心中 我不再受宠 我的天空 是雨是风 还是
epoch 160, perplexity 1.017492, time 1.42 sec
- 分开始乡相信命运 感谢地心引力 让我碰到你 漂亮的让我面红的可爱女人 温柔的让我心疼的
- 不分开 我不能再想 我不 我不 我不能 爱情走的太快就像龙卷风 不能承受我已无处可躲 我不

小结

- 长短期记忆的隐藏层输出包括隐藏状态和记忆细胞。只有隐藏状态会传递到输出层。
- 长短期记忆的输入门、遗忘门和输出门可以控制信息的流动。
- 长短期记忆可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时间序列中时间步距离较大的依赖关系。

参考文献

- [1] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735-1780.
-

注：除代码外本节与原书此节基本相同，[原书传送门](#)

6.9 深度循环神经网络

本章到目前为止介绍的循环神经网络只有一个单向的隐藏层，在深度学习应用里，我们通常会用到含有多个隐藏层的循环神经网络，也称作深度循环神经网络。图 6.11 演示了一个有 L 个隐藏层的深度循环神经网络，每个隐藏状态不断传递至当前层的下一时间步和当前时间步的下一层。

图 6.11 深度循环神经网络的架构

具体来说，在时间步 t 里，设小批量输入 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ （样本数为 n ，输入个数为 d ），第 ℓ 隐藏层 ($\ell = 1, \dots, L$) 的隐藏状态为 $\mathbf{H}_t^{(\ell)} \in \mathbb{R}^{n \times h}$ （隐藏单元个数为 h ），输出层变量为 $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ （输出个数为 q ），且隐藏层的激活函数为 ϕ 。第 1 隐藏层的隐藏状态和之前的计算一样：

$$\mathbf{H}_t^{(1)} = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(1)} + \mathbf{H}_{t-1}^{(1)} \mathbf{W}_{hh}^{(1)} + \mathbf{b}_h^{(1)}),$$

其中权重 $\mathbf{W}_{xh}^{(1)} \in \mathbb{R}^{d \times h}$ 、 $\mathbf{W}_{hh}^{(1)} \in \mathbb{R}^{h \times h}$ 和偏差 $\mathbf{b}_h^{(1)} \in \mathbb{R}^{1 \times h}$ 分别为第 1 隐藏层的模型参数。

当 $1 < \ell \leq L$ 时，第 ℓ 隐藏层的隐藏状态的表达式为

$$\mathbf{H}_t^{(\ell)} = \phi(\mathbf{H}_t^{(\ell-1)} \mathbf{W}_{xh}^{(\ell)} + \mathbf{H}_{t-1}^{(\ell)} \mathbf{W}_{hh}^{(\ell)} + \mathbf{b}_h^{(\ell)}),$$

其中权重 $\mathbf{W}_{xh}^{(\ell)} \in \mathbb{R}^{h \times h}$ 、 $\mathbf{W}_{hh}^{(\ell)} \in \mathbb{R}^{h \times h}$ 和偏差 $\mathbf{b}_h^{(\ell)} \in \mathbb{R}^{1 \times h}$ 分别为第 ℓ 隐藏层的模型参数。

最终，输出层的输出只需基于第 L 隐藏层的隐藏状态：

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{hq} + \mathbf{b}_q,$$

其中权重 $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ 和偏差 $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ 为输出层的模型参数。

同多层感知机一样，隐藏层个数 L 和隐藏单元个数 h 都是超参数。此外，如果将隐藏状态的计算换成门控循环单元或者长短期记忆的计算，我们可以得到深度门控循环神经网络。

小结

- 在深度循环神经网络中，隐藏状态的信息不断传递至当前层的下一时间步和当前时间步的下一层。

注：本节与原书基本相同，[原书传送门](#)

6.10 双向循环神经网络

之前介绍的循环神经网络模型都是假设当前时间步是由前面的较早时间步的序列决定的，因此它们都将信息通过隐藏状态从前往后传递。有时候，当前时间步也可能由后面时间步决定。例如，当我们写下一个句子时，可能会根据句子后面的词来修改句子前面的用词。双向循环神经网络通过增加从后往前传递信息的隐藏层来更灵活地处理这类信息。图 6.12 演示了一个含单隐藏层的双向循环神经网络的架构。

图 6.12 双向循环神经网络的架构

下面我们来介绍具体的定义。给定时间步 t 的小批量输入 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (样本数为 n , 输入个数为 d) 和隐藏层激活函数为 ϕ 。在双向循环神经网络的架构中，设该时间步正向隐藏状态为 $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ (正向隐藏单元个数为 h)，反向隐藏状态为 $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ (反向隐藏单元个数为 h)。我们可以分别计算正向隐藏状态和反向隐藏状态：

$$\begin{aligned}\overrightarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}), \\ \overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}),\end{aligned}$$

其中权重 $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}$ 、 $\mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}$ 、 $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$ 、 $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$ 和偏差 $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$ 、 $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$ 均为模型参数。

然后我们连结两个方向的隐藏状态 $\overrightarrow{\mathbf{H}}_t$ 和 $\overleftarrow{\mathbf{H}}_t$ 来得到隐藏状态 $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$ ，并将其实输入到输出层。输出层计算输出 $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (输出个数为 q)：

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q,$$

其中权重 $\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$ 和偏差 $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ 为输出层的模型参数。不同方向上的隐藏单元个数也可以不同。

小结

- 双向循环神经网络在每个时间步的隐藏状态同时取决于该时间步之前和之后的子序列（包括当前时间步的输入）。

注：本节与原书基本相同，[原书传送门](#)

7.1 优化与深度学习

本节将讨论优化与深度学习的关系，以及优化在深度学习中的挑战。在一个深度学习问题中，我们通常会预先定义一个损失函数。有了损失函数以后，我们就可以使用优化算法试图将其最小化。在优化中，这样的损失函数通常被称作优化问题的目标函数（objective function）。依据惯例，优化算法通常只考虑最小化目标函数。其实，任何最大化问题都可以很容易地转化为最小化问题，只需令目标函数的相反数为新的目标函数即可。

7.1.1 优化与深度学习的关系

虽然优化为深度学习提供了最小化损失函数的方法，但本质上，优化与深度学习的目标是有区别的。在 3.11 节（模型选择、欠拟合和过拟合）中，我们区分了训练误差和泛化误差。由于优化算法的目标函数通常是一个基于训练数据集的损失函数，优化的目标在于降低训练误差。而深度学习的目标在于降低泛化误差。为了降低泛化误差，除了使用优化算法降低训练误差以外，还需要注意应对过拟合。

本章中，我们只关注优化算法在最小化目标函数上的表现，而不关注模型的泛化误差。

7.1.2 优化在深度学习中的挑战

我们在 3.1 节（线性回归）中对优化问题的解析解和数值解做了区分。深度学习中绝大多数目标函数都很复杂。因此，很多优化问题并不存在解析解，而需要使用基于数值方法的优化算法找到近似解，即数值解。本书中讨论的优化算法都是这类基于数值方法的算法。为了求得最小化目标函数的数值解，我们将通过优化算法有限次迭代模型参数来尽可能降低损失函数的值。

优化在深度学习中有很多挑战。下面描述了其中的两个挑战，即局部最小值和鞍点。为了更好地描述问题，我们先导入本节中实验需要的包或模块。

```
%matplotlib inline
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
from mpl_toolkits import mplot3d # 三维画图
import numpy as np
```

7.1.2.1 局部最小值

对于目标函数 $f(x)$, 如果 $f(x)$ 在 x 上的值比在 x 邻近的其他点的值更小, 那么 $f(x)$ 可能是一个局部最小值 (local minimum)。如果 $f(x)$ 在 x 上的值是目标函数在整个定义域上的最小值, 那么 $f(x)$ 是全局最小值 (global minimum)。

举个例子, 给定函数

$$f(x) = x \cdot \cos(\pi x), \quad -1.0 \leq x \leq 2.0,$$

我们可以大致找出该函数的局部最小值和全局最小值的位置。需要注意的是, 图中箭头所指示的只是大致位置。

```
def f(x):
    return x * np.cos(np.pi * x)

d2l.set_figsize((4.5, 2.5))
x = np.arange(-1.0, 2.0, 0.1)
fig, = d2l.plt.plot(x, f(x))
fig.axes.annotate('local minimum', xy=(-0.3, -0.25), xytext=(-0.77, -1.0),
                   arrowprops=dict(arrowstyle='->'))
fig.axes.annotate('global minimum', xy=(1.1, -0.95), xytext=(0.6, 0.8),
                   arrowprops=dict(arrowstyle='->'))
d2l.plt.xlabel('x')
d2l.plt.ylabel('f(x)');
```

深度学习模型的目标函数可能有若干局部最优值。当一个优化问题的数值解在局部最优解附近时, 由于目标函数有关解的梯度接近或变成零, 最终迭代求得的数值解可能只令目标函数局部最小化而非全局最小化。

7.1.2.2 鞍点

刚刚我们提到, 梯度接近或变成零可能是由于当前解在局部最优解附近造成的。事实上, 另一种可能性是当前解在鞍点 (saddle point) 附近。

举个例子, 给定函数

$$f(x) = x^3,$$

我们可以找出该函数的鞍点位置。

```
x = np.arange(-2.0, 2.0, 0.1)
fig, = d2l.plt.plot(x, x**3)
fig.axes.annotate('saddle point', xy=(0, -0.2), xytext=(-0.52, -5.0),
                  arrowprops=dict(arrowstyle='->'))
d2l.plt.xlabel('x')
d2l.plt.ylabel('f(x)');
```

再举个定义在二维空间的函数的例子，例如：

$$f(x, y) = x^2 - y^2.$$

我们可以找出该函数的鞍点位置。也许你已经发现了，该函数看起来像一个马鞍，而鞍点恰好是马鞍上可坐区域的中心。

```
x, y = np.mgrid[-1: 1: 31j, -1: 1: 31j]
z = x**2 - y**2

ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 2, 'cstride': 2})
ax.plot([0], [0], [0], 'rx')
ticks = [-1, 0, 1]
d2l.plt.xticks(ticks)
d2l.plt.yticks(ticks)
ax.set_zticks(ticks)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y');
```

在图的鞍点位置，目标函数在 x 轴方向上是局部最小值，但在 y 轴方向上是局部最大值。

假设一个函数的输入为 k 维向量，输出为标量，那么它的海森矩阵 (Hessian matrix) 有 k 个特征值。该函数在梯度为 0 的位置上可能是局部最小值、局部最大值或者鞍点。

- 当函数的海森矩阵在梯度为零的位置上的特征值全为正时，该函数得到局部最小值。
- 当函数的海森矩阵在梯度为零的位置上的特征值全为负时，该函数得到局部最大值。
- 当函数的海森矩阵在梯度为零的位置上的特征值有正有负时，该函数得到鞍点。

随机矩阵理论告诉我们，对于一个大的高斯随机矩阵来说，任一特征值是正或者是负的概率都是 0.5 [1]。那么，以上第一种情况的概率为 0.5^k 。由于深度学习模型参数通常都是高维的 (k 很大)，目标函数的鞍点通常比局部最小值更常见。

在深度学习中，虽然找到目标函数的全局最优解很难，但这并非必要。我们将在本章接下来的几节中逐一介绍深度学习中常用的优化算法，它们在很多实际问题中都能够训练出十分有效的深度学习模型。

小结

- 由于优化算法的目标函数通常是一个基于训练数据集的损失函数，优化的目标在于降低训练误差。
- 由于深度学习模型参数通常都是高维的，目标函数的鞍点通常比局部最小值更常见。

参考文献

-
- [1] Wigner, E. P. (1958). On the distribution of the roots of certain symmetric matrices. *Annals of Mathematics*, 325-327.

注：本节与原书基本相同，[原书传送门](#)

7.2 梯度下降和随机梯度下降

在本节中，我们将介绍梯度下降（gradient descent）的工作原理。虽然梯度下降在深度学习中很少被直接使用，但理解梯度的意义以及沿着梯度反方向更新自变量可能降低目标函数值的原因是学习后续优化算法的基础。随后，我们将引出随机梯度下降（stochastic gradient descent）。

7.2.1 一维梯度下降

我们先以简单的一维梯度下降为例，解释梯度下降算法可能降低目标函数值的原因。假设连续可导的函数 $f : \mathbb{R} \rightarrow \mathbb{R}$ 的输入和输出都是标量。给定绝对值足够小的数 ϵ ，根据泰勒展开公式，我们得到以下的近似：

$$f(x + \epsilon) \approx f(x) + \epsilon f'(x).$$

这里 $f'(x)$ 是函数 f 在 x 处的梯度。一维函数的梯度是一个标量，也称导数。

接下来，找到一个常数 $\eta > 0$ ，使得 $|\eta f'(x)|$ 足够小，那么可以将 ϵ 替换为 $-\eta f'(x)$ 并得到

$$f(x - \eta f'(x)) \approx f(x) - \eta f'(x)^2.$$

如果导数 $f'(x) \neq 0$ ，那么 $\eta f'(x)^2 > 0$ ，所以

$$f(x - \eta f'(x)) \lesssim f(x).$$

这意味着，如果通过

$$x \leftarrow x - \eta f'(x)$$

来迭代 x ，函数 $f(x)$ 的值可能会降低。因此在梯度下降中，我们先选取一个初始值 x 和常数 $\eta > 0$ ，然后不断通过上式来迭代 x ，直到达到停止条件，例如 $f'(x)^2$ 的值已足够小或迭代次数已达到某个值。

下面我们以目标函数 $f(x) = x^2$ 为例来看一看梯度下降是如何工作的。虽然我们知道最小化 $f(x)$ 的解为 $x = 0$ ，这里依然使用这个简单函数来观察 x 是如何被迭代的。首先，导入本节实验所需的包或模块。

```
%matplotlib inline
import numpy as np
import torch
```

```
import math
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
```

接下来使用 $x = 10$ 作为初始值，并设 $\eta = 0.2$ 。使用梯度下降对 x 迭代 10 次，可见最终 x 的值较接近最优解。

```
def gd(eta):
    x = 10
    results = [x]
    for i in range(10):
        x -= eta * 2 * x #  $f(x) = x * x$  的导数为  $f'(x) = 2 * x$ 
        results.append(x)
    print('epoch 10, x:', x)
    return results

res = gd(0.2)
```

输出：

```
epoch 10, x: 0.06046617599999997
```

下面将绘制出自变量 x 的迭代轨迹。

```
def show_trace(res):
    n = max(abs(min(res)), abs(max(res)), 10)
    f_line = np.arange(-n, n, 0.1)
    d2l.set_figsize()
    d2l.plt.plot(f_line, [x * x for x in f_line])
    d2l.plt.plot(res, [x * x for x in res], '-o')
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('f(x)')

show_trace(res)
```

7.2.2 学习率

上述梯度下降算法中的正数 η 通常叫作学习率。这是一个超参数，需要人工设定。如果使用过小的学习率，会导致 x 更新缓慢从而需要更多的迭代才能得到较好的解。

下面展示使用学习率 $\eta = 0.05$ 时自变量 x 的迭代轨迹。可见，同样迭代 10 次后，当学习率过小时，最终 x 的值依然与最优解存在较大偏差。

```
show_trace(gd(0.05))
```

输出：

```
epoch 10, x: 3.4867844009999995
```

如果使用过大的学习率， $|\eta f'(x)|$ 可能会过大从而使前面提到的一阶泰勒展开公式不再成立：这时我们无法保证迭代 x 会降低 $f(x)$ 的值。

举个例子，当设学习率 $\eta = 1.1$ 时，可以看到 x 不断越过 (overshoot) 最优解 $x = 0$ 并逐渐发散。

```
show_trace(gd(1.1))
```

输出：

```
epoch 10, x: 61.917364224000096
```

7.2.3 多维梯度下降

在了解了一维梯度下降之后，我们再考虑一种更广义的情况：目标函数的输入为向量，输出为标量。假设目标函数 $f : \mathbb{R}^d \rightarrow \mathbb{R}$ 的输入是一个 d 维向量 $\mathbf{x} = [x_1, x_2, \dots, x_d]^\top$ 。目标函数 $f(\mathbf{x})$ 有关 \mathbf{x} 的梯度是一个由 d 个偏导数组成的向量：

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_d} \right]^\top.$$

为表示简洁，我们用 $\nabla f(\mathbf{x})$ 代替 $\nabla_{\mathbf{x}} f(\mathbf{x})$ 。梯度中每个偏导数元素 $\partial f(\mathbf{x})/\partial x_i$ 代表着 f 在 \mathbf{x} 有关输入 x_i 的变化率。为了测量 f 沿着单位向量 \mathbf{u} (即 $\|\mathbf{u}\| = 1$) 方向上的变化率，在多元微积分中，我们定义 f 在 \mathbf{x} 上沿着 \mathbf{u} 方向的方向导数为

$$D_{\mathbf{u}} f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{u}) - f(\mathbf{x})}{h}.$$

依据方向导数性质 [1, 14.6 节定理三]，以上方向导数可以改写为

$$D_u f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{u}.$$

方向导数 $D_u f(\mathbf{x})$ 给出了 f 在 \mathbf{x} 上沿着所有可能方向的变化率。为了最小化 f , 我们希望找到 f 能被降低最快的方向。因此, 我们可以通过单位向量 \mathbf{u} 来最小化方向导数 $D_u f(\mathbf{x})$ 。

由于 $D_u f(\mathbf{x}) = \|\nabla f(\mathbf{x})\| \cdot \|\mathbf{u}\| \cdot \cos(\theta) = \|\nabla f(\mathbf{x})\| \cdot \cos(\theta)$, 其中 θ 为梯度 $\nabla f(\mathbf{x})$ 和单位向量 \mathbf{u} 之间的夹角, 当 $\theta = \pi$ 时, $\cos(\theta)$ 取得最小值 -1 。因此, 当 \mathbf{u} 在梯度方向 $\nabla f(\mathbf{x})$ 的相反方向时, 方向导数 $D_u f(\mathbf{x})$ 被最小化。因此, 我们可能通过梯度下降算法来不断降低目标函数 f 的值:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x}).$$

同样, 其中 η (取正数) 称作学习率。

下面我们构造一个输入为二维向量 $\mathbf{x} = [x_1, x_2]^\top$ 和输出为标量的目标函数 $f(\mathbf{x}) = x_1^2 + 2x_2^2$ 。那么, 梯度 $\nabla f(\mathbf{x}) = [2x_1, 4x_2]^\top$ 。我们将观察梯度下降从初始位置 $[-5, -2]$ 开始对自变量 \mathbf{x} 的迭代轨迹。我们先定义两个辅助函数, 第一个函数使用给定的自变量更新函数, 从初始位置 $[-5, -2]$ 开始迭代自变量 \mathbf{x} 共 20 次, 第二个函数对自变量 \mathbf{x} 的迭代轨迹进行可视化。

```
def train_2d(trainer):  # 本函数将保存在 d2lzh_pytorch 包中方便以后使用
    x1, x2, s1, s2 = -5, -2, 0, 0  # s1 和 s2 是自变量状态, 本章后续几节会使用
    results = [(x1, x2)]
    for i in range(20):
        x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
        results.append((x1, x2))
    print('epoch %d, x1 %f, x2 %f' % (i + 1, x1, x2))
    return results

def show_trace_2d(f, results):  # 本函数将保存在 d2lzh_pytorch 包中方便以后使用
    d2l.plt.plot(*zip(*results), '-o', color='#ff7f0e')
    x1, x2 = np.meshgrid(np.arange(-5.5, 1.0, 0.1), np.arange(-3.0, 1.0, 0.1))
    d2l.plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')
    d2l.plt.xlabel('x1')
    d2l.plt.ylabel('x2')
```

然后, 观察学习率为 0.1 时自变量的迭代轨迹。使用梯度下降对自变量 \mathbf{x} 迭代 20 次后, 可见最终 \mathbf{x} 的值较接近最优解 $[0, 0]$ 。

```

eta = 0.1

def f_2d(x1, x2):  # 目标函数
    return x1 ** 2 + 2 * x2 ** 2

def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 2 * x1, x2 - eta * 4 * x2, 0, 0)

show_trace_2d(f_2d, train_2d(gd_2d))

```

输出：

```
epoch 20, x1 -0.057646, x2 -0.000073
```

7.2.4 随机梯度下降

在深度学习里，目标函数通常是训练数据集中有关各个样本的损失函数的平均。设 $f_i(\mathbf{x})$ 是有关索引为 i 的训练数据样本的损失函数， n 是训练数据样本数， \mathbf{x} 是模型的参数向量，那么目标函数定义为

$$f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n f_i(\mathbf{x}).$$

目标函数在 \mathbf{x} 处的梯度计算为

$$\nabla f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{x}).$$

如果使用梯度下降，每次自变量迭代的计算开销为 $\mathcal{O}(n)$ ，它随着 n 线性增长。因此，当训练数据样本数很大时，梯度下降每次迭代的计算开销很高。

随机梯度下降 (stochastic gradient descent, SGD) 减少了每次迭代的计算开销。在随机梯度下降的每次迭代中，我们随机均匀采样的一个样本索引 $i \in \{1, \dots, n\}$ ，并计算梯度 $\nabla f_i(\mathbf{x})$ 来迭代 \mathbf{x} ：

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}).$$

这里 η 同样是学习率。可以看到每次迭代的计算开销从梯度下降的 $\mathcal{O}(n)$ 降到了常数 $\mathcal{O}(1)$ 。值得强调的是，随机梯度 $\nabla f_i(\mathbf{x})$ 是对梯度 $\nabla f(\mathbf{x})$ 的无偏估计：

$$E_i \nabla f_i(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}).$$

这意味着，平均来说，随机梯度是对梯度的一个良好的估计。

下面我们通过在梯度中添加均值为 0 的随机噪声来模拟随机梯度下降，以此来比较它与梯度下降的区别。

```
def sgd_2d(x1, x2, s1, s2):
    return (x1 - eta * (2 * x1 + np.random.normal(0.1)),
            x2 - eta * (4 * x2 + np.random.normal(0.1)), 0, 0)

show_trace_2d(f_2d, train_2d(sgd_2d))
```

输出：

```
epoch 20, x1 -0.047150, x2 -0.075628
```

可以看到，随机梯度下降中自变量的迭代轨迹相对于梯度下降中的来说更为曲折。这是由于实验所添加的噪声使模拟的随机梯度的准确度下降。在实际中，这些噪声通常指训练数据集中的无意义的干扰。

小结

- 使用适当的学习率，沿着梯度反方向更新自变量可能降低目标函数值。梯度下降重复这一更新过程直到得到满足要求的解。
- 学习率过大或过小都有问题。一个合适的学习率通常是需要通过多次实验找到的。
- 当训练数据集的样本较多时，梯度下降每次迭代的计算开销较大，因而随机梯度下降通常更受青睐。

参考文献

[1] Stewart, J. (2010). Calculus: early transcendentals. 7th ed. Cengage Learning.

注：本节与原书基本相同，[原书传送门](#)

7.3 小批量随机梯度下降

在每一次迭代中，梯度下降使用整个训练数据集来计算梯度，因此它有时也被称为批量梯度下降（batch gradient descent）。而随机梯度下降在每次迭代中只随机采样一个样本来计算梯度。正如我们在前几章中所看到的，我们还可以在每轮迭代中随机均匀采样多个样本来组成一个小批量，然后使用这个小批量来计算梯度。下面就来描述小批量随机梯度下降。

设目标函数 $f(\mathbf{x}) : \mathbb{R}^d \rightarrow \mathbb{R}$ 。在迭代开始前的时间步设为 0。该时间步的自变量记为 $\mathbf{x}_0 \in \mathbb{R}^d$ ，通常由随机初始化得到。在接下来的每一个时间步 $t > 0$ 中，小批量随机梯度下降随机均匀采样一个由训练数据样本索引组成的小批量 \mathcal{B}_t 。我们可以通过重复采样（sampling with replacement）或者不重复采样（sampling without replacement）得到一个小批量中的各个样本。前者允许同一个小批量中出现重复的样本，后者则不允许如此，且更常见。对于这两者间的任一种方式，都可以使用

$$\mathbf{g}_t \leftarrow \nabla f_{\mathcal{B}_t}(\mathbf{x}_{t-1}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \nabla f_i(\mathbf{x}_{t-1})$$

来计算时间步 t 的小批量 \mathcal{B}_t 上目标函数位于 \mathbf{x}_{t-1} 处的梯度 \mathbf{g}_t 。这里 $|\mathcal{B}|$ 代表批量大小，即小批量中样本的个数，是一个超参数。同随机梯度一样，重复采样所得的小批量随机梯度 \mathbf{g}_t 也是对梯度 $\nabla f(\mathbf{x}_{t-1})$ 的无偏估计。给定学习率 η_t （取正数），小批量随机梯度下降对自变量的迭代如下：

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{g}_t.$$

基于随机采样得到的梯度的方差在迭代过程中无法减小，因此在实际中，（小批量）随机梯度下降的学习率可以在迭代过程中自我衰减，例如 $\eta_t = \eta t^\alpha$ （通常 $\alpha = -1$ 或者 -0.5 ）、 $\eta_t = \eta \alpha^t$ （如 $\alpha = 0.95$ ）或者每迭代若干次后将学习率衰减一次。如此一来，学习率和（小批量）随机梯度乘积的方差会减小。而梯度下降在迭代过程中一直使用目标函数的真实梯度，无须自我衰减学习率。

小批量随机梯度下降中每次迭代的计算开销为 $\mathcal{O}(|\mathcal{B}|)$ 。当批量大小为 1 时，该算法即为随机梯度下降；当批量大小等于训练数据样本数时，该算法即为梯度下降。当批量较小时，每次迭代中使用的样本少，这会导致并行处理和内存使用效率变低。这使得在计算同样数目样本的情况下比使用更大批量时所花时间更多。当批量较大时，每个小批量梯度里可能含有更多的冗余信息。为了得到较好的解，批量较大时比批量较小时需要计算的样本数目可能更多，例如增大迭代周期数。

7.3.1 读取数据

本章里我们将使用一个来自 NASA 的测试不同飞机机翼噪音的数据集来比较各个优化算法 [1]。我们使用该数据集的前 1,500 个样本和 5 个特征，并使用标准化对数据进行预处理。

```
%matplotlib inline
import numpy as np
import time
import torch
from torch import nn, optim
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l

def get_data_ch7(): # 本函数已保存在 d2lzh_pytorch 包中方便以后使用
    data = np.genfromtxt('..../data/airfoil_self_noise.dat', delimiter='\t')
    data = (data - data.mean(axis=0)) / data.std(axis=0)
    return torch.tensor(data[:1500, :-1], dtype=torch.float32), \
        torch.tensor(data[:1500, -1], dtype=torch.float32) # 前 1500 个样本 (每个样本 5 个特征)

features, labels = get_data_ch7()
features.shape # torch.Size([1500, 5])
```

7.3.2 从零开始实现

3.2 节（线性回归的从零开始实现）中已经实现过小批量随机梯度下降算法。我们在这里将它的输入参数变得更加通用，主要是为了方便本章后面介绍的其他优化算法也可以使用同样的输入。具体来说，我们添加了一个状态输入 `states` 并将超参数放在字典 `hyperparams` 里。此外，我们将在训练函数里对各个小批量样本的损失求平均，因此优化算法里的梯度不需要除以批量大小。

```
def sgd(params, states, hyperparams):
    for p in params:
        p.data -= hyperparams['lr'] * p.grad.data
```

下面实现一个通用的训练函数，以方便本章后面介绍的其他优化算法使用。它初始化一个线性回归模型，然后可以使用小批量随机梯度下降以及后续小节介绍的其他算

法来训练模型。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def train_ch7(optimizer_fn, states, hyperparams, features, labels,
    batch_size=10, num_epochs=2):
    # 初始化模型
    net, loss = d2l.linreg, d2l.squared_loss

    w = torch.nn.Parameter(torch.tensor(np.random.normal(0, 0.01, size=(features.shape[1], 1)),
                                         requires_grad=True))
    b = torch.nn.Parameter(torch.zeros(1, dtype=torch.float32), requires_grad=True)

    def eval_loss():
        return loss(net(features, w, b), labels).mean().item()

    ls = [eval_loss()]
    data_iter = torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(features, labels), batch_size, shuffle=True)

    for _ in range(num_epochs):
        start = time.time()
        for batch_i, (X, y) in enumerate(data_iter):
            l = loss(net(X, w, b), y).mean()  # 使用平均损失

            # 梯度清零
            if w.grad is not None:
                w.grad.data.zero_()
                b.grad.data.zero_()

            l.backward()
            optimizer_fn([w, b], states, hyperparams)  # 迭代模型参数
            if (batch_i + 1) * batch_size % 100 == 0:
                ls.append(eval_loss())  # 每 100 个样本记录下当前训练误差

        # 打印结果和作图
        print('loss: %f, %f sec per epoch' % (ls[-1], time.time() - start))
        d2l.set_figsize()
```

```
d2l=plt.plot(np.linspace(0, num_epochs, len(ls)), ls)
d2l=plt.xlabel('epoch')
d2l=plt.ylabel('loss')
```

当批量大小为样本总数 1,500 时，优化使用的是梯度下降。梯度下降的 1 个迭代周期对模型参数只迭代 1 次。可以看到 6 次迭代后目标函数值（训练损失）的下降趋向了平稳。

```
def train_sgd(lr, batch_size, num_epochs=2):
    train_ch7(sgd, None, {'lr': lr}, features, labels, batch_size, num_epochs)

train_sgd(1, 1500, 6)
```

输出：

```
loss: 0.243605, 0.014335 sec per epoch
```

当批量大小为 1 时，优化使用的是随机梯度下降。为了简化实现，有关（小批量）随机梯度下降的实验中，我们未对学习率进行自我衰减，而是直接采用较小的常数学习率。随机梯度下降中，每处理一个样本会更新一次自变量（模型参数），一个迭代周期里会对自变量进行 1,500 次更新。可以看到，目标函数值的下降在 1 个迭代周期后就变得较为平缓。

```
train_sgd(0.005, 1)
```

输出：

```
loss: 0.243433, 0.270011 sec per epoch
```

虽然随机梯度下降和梯度下降在一个迭代周期里都处理了 1,500 个样本，但实验中随机梯度下降的一个迭代周期耗时更多。这是因为随机梯度下降在一个迭代周期里做了更多次的自变量迭代，而且单样本的梯度计算难以有效利用矢量计算。

当批量大小为 10 时，优化使用的是小批量随机梯度下降。它在每个迭代周期的耗时介于梯度下降和随机梯度下降的耗时之间。

```
train_sgd(0.05, 10)
```

输出：

```
loss: 0.242805, 0.078792 sec per epoch
```

7.3.3 简洁实现

在 PyTorch 里可以通过创建 `optimizer` 实例来调用优化算法。这能让实现更简洁。下面实现一个通用的训练函数，它通过优化算法的函数 `optimizer_fn` 和超参数 `optimizer_hyperparams` 来创建 `optimizer` 实例。

```
# 本函数与原书不同的是这里第一个参数优化器函数而不是优化器的名字
# 例如: optimizer_fn=torch.optim.SGD, optimizer_hyperparams={"lr": 0.05}
def train_pytorch_ch7(optimizer_fn, optimizer_hyperparams, features, labels,
                      batch_size=10, num_epochs=2):
    # 初始化模型
    net = nn.Sequential(
        nn.Linear(features.shape[-1], 1)
    )
    loss = nn.MSELoss()
    optimizer = optimizer_fn(net.parameters(), **optimizer_hyperparams)

    def eval_loss():
        return loss(net(features).view(-1), labels).item() / 2

    ls = [eval_loss()]
    data_iter = torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(features, labels), batch_size, shuffle=True)

    for _ in range(num_epochs):
        start = time.time()
        for batch_i, (X, y) in enumerate(data_iter):
            # 除以 2 是为了和 train_ch7 保持一致, 因为 squared_loss 中除了 2
            l = loss(net(X).view(-1), y) / 2

            optimizer.zero_grad()
            l.backward()
            optimizer.step()
            if (batch_i + 1) * batch_size % 100 == 0:
                ls.append(eval_loss())
    # 打印结果和作图
```

```
print('loss: %f, %f sec per epoch' % (ls[-1], time.time() - start))
d2l.set_figsize()
d2l=plt.plot(np.linspace(0, num_epochs, len(ls)), ls)
d2l.xlabel('epoch')
d2l.ylabel('loss')
```

使用 PyTorch 重复上一个实验。

```
train_pytorch_ch7(optim.SGD, {"lr": 0.05}, features, labels, 10)
```

输出：

```
loss: 0.245491, 0.044150 sec per epoch
```

小结

- 小批量随机梯度每次随机均匀采样一个小批量的训练样本来计算梯度。
- 在实际中，（小批量）随机梯度下降的学习率可以在迭代过程中自我衰减。
- 通常，小批量随机梯度在每个迭代周期的耗时介于梯度下降和随机梯度下降的耗时之间。

参考文献

[1] 飞机机翼噪音数据集。<https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise>

注：除代码外本节与原书此节基本相同，[原书传送门](#)

7.4 动量法

在 7.2 节（梯度下降和随机梯度下降）中我们提到，目标函数有关自变量的梯度代表了目标函数在自变量当前位置下降最快的方向。因此，梯度下降也叫作最陡下降 (steepest descent)。在每次迭代中，梯度下降根据自变量当前位置，沿着当前位置的梯度更新自变量。然而，如果自变量的迭代方向仅仅取决于自变量当前位置，这可能会带来一些问题。

7.4.1 梯度下降的问题

让我们考虑一个输入和输出分别为二维向量 $\mathbf{x} = [x_1, x_2]^\top$ 和标量的目标函数 $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ 。与 7.2 节中不同，这里将 x_1^2 系数从 1 减小到了 0.1。下面实现基于这个目标函数的梯度下降，并演示使用学习率为 0.4 时自变量的迭代轨迹。

```
%matplotlib inline
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
import torch

eta = 0.4 # 学习率

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)

d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

输出：

```
epoch 20, x1 -0.943467, x2 -0.000073
```

可以看到，同一位置上，目标函数在竖直方向 (x_2 轴方向) 比在水平方向 (x_1 轴方向) 的斜率的绝对值更大。因此，给定学习率，梯度下降迭代自变量时会使自变量在竖直方向比在水平方向移动幅度更大。那么，我们需要一个较小的学习率从而避免自变

量在竖直方向上越过目标函数最优解。然而，这会造成自变量在水平方向上朝最优解移动变慢。

下面我们试着将学习率调得稍大一点，此时自变量在竖直方向不断越过最优解并逐渐发散。

```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

输出：

```
epoch 20, x1 -0.387814, x2 -1673.365109
```

7.4.2 动量法

动量法的提出是为了解决梯度下降的上述问题。由于小批量随机梯度下降比梯度下降更为广义，本章后续讨论将沿用 7.3 节（小批量随机梯度下降）中时间步 t 的小批量随机梯度 \mathbf{g}_t 的定义。设时间步 t 的自变量为 \mathbf{x}_t ，学习率为 η_t 。在时间步 0，动量法创建速度变量 \mathbf{v}_0 ，并将其元素初始化成 0。在时间步 $t > 0$ ，动量法对每次迭代的步骤做如下修改：

$$\begin{aligned}\mathbf{v}_t &\leftarrow \gamma \mathbf{v}_{t-1} + \eta_t \mathbf{g}_t, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \mathbf{v}_t,\end{aligned}$$

其中，动量超参数 γ 满足 $0 \leq \gamma < 1$ 。当 $\gamma = 0$ 时，动量法等价于小批量随机梯度下降。

在解释动量法的数学原理前，让我们先从实验中观察梯度下降在使用动量法后的迭代轨迹。

```
def momentum_2d(x1, x2, v1, v2):
    v1 = gamma * v1 + eta * 0.2 * x1
    v2 = gamma * v2 + eta * 4 * x2
    return x1 - v1, x2 - v2, v1, v2

eta, gamma = 0.4, 0.5
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

输出：

```
epoch 20, x1 -0.062843, x2 0.001202
```

可以看到使用较小的学习率 $\eta = 0.4$ 和动量超参数 $\gamma = 0.5$ 时，动量法在竖直方向上的移动更加平滑，且在水平方向上更快逼近最优解。下面使用较大的学习率 $\eta = 0.6$ ，此时自变量也不再发散。

```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

输出：

```
epoch 20, x1 0.007188, x2 0.002553
```

7.4.2.1 指数加权移动平均

为了从数学上理解动量法，让我们先解释一下指数加权移动平均（exponentially weighted moving average）。给定超参数 $0 \leq \gamma < 1$ ，当前时间步 t 的变量 y_t 是上一时间步 $t - 1$ 的变量 y_{t-1} 和当前时间步另一变量 x_t 的线性组合：

$$y_t = \gamma y_{t-1} + (1 - \gamma)x_t.$$

我们可以对 y_t 展开：

$$\begin{aligned} y_t &= (1 - \gamma)x_t + \gamma y_{t-1} \\ &= (1 - \gamma)x_t + (1 - \gamma) \cdot \gamma x_{t-1} + \gamma^2 y_{t-2} \\ &= (1 - \gamma)x_t + (1 - \gamma) \cdot \gamma x_{t-1} + (1 - \gamma) \cdot \gamma^2 x_{t-2} + \gamma^3 y_{t-3} \\ &\dots \end{aligned}$$

令 $n = 1/(1 - \gamma)$ ，那么 $(1 - 1/n)^n = \gamma^{1/(1-\gamma)}$ 。因为

$$\lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right)^n = \exp(-1) \approx 0.3679,$$

所以当 $\gamma \rightarrow 1$ 时， $\gamma^{1/(1-\gamma)} = \exp(-1)$ ，如 $0.95^{20} \approx \exp(-1)$ 。如果把 $\exp(-1)$ 当作一个比较小的数，我们可以在近似中忽略所有含 $\gamma^{1/(1-\gamma)}$ 和比 $\gamma^{1/(1-\gamma)}$ 更高阶的系数的项。例如，当 $\gamma = 0.95$ 时，

$$y_t \approx 0.05 \sum_{i=0}^{19} 0.95^i x_{t-i}.$$

因此，在实际中，我们常常将 y_t 看作是对最近 $1/(1 - \gamma)$ 个时间步的 x_t 值的加权平均。例如，当 $\gamma = 0.95$ 时， y_t 可以被看作对最近 20 个时间步的 x_t 值的加权平均；当 $\gamma = 0.9$ 时， y_t 可以看作是对最近 10 个时间步的 x_t 值的加权平均。而且，离当前时间步 t 越近的 x_t 值获得的权重越大（越接近 1）。

7.4.2.2 由指数加权移动平均理解动量法

现在，我们对动量法的速度变量做变形：

$$\mathbf{v}_t \leftarrow \gamma \mathbf{v}_{t-1} + (1 - \gamma) \left(\frac{\eta_t}{1 - \gamma} \mathbf{g}_t \right).$$

由指数加权移动平均的形式可得，速度变量 \mathbf{v}_t 实际上对序列 $\{\eta_{t-i} \mathbf{g}_{t-i} / (1 - \gamma) : i = 0, \dots, 1/(1 - \gamma) - 1\}$ 做了指数加权移动平均。换句话说，相比于小批量随机梯度下降，动量法在每个时间步的自变量更新量近似于将最近 $1/(1 - \gamma)$ 个时间步的普通更新量（即学习率乘以梯度）做了指数加权移动平均后再除以 $1 - \gamma$ 。所以，在动量法中，自变量在各个方向上的移动幅度不仅取决于当前梯度，还取决于过去的各个梯度在各个方向上是否一致。在本节之前示例的优化问题中，所有梯度在水平方向上为正（向右），而在竖直方向上时正（向上）时负（向下）。这样，我们就可以使用较大的学习率，从而使自变量向最优解更快移动。

7.4.3 从零开始实现

相对于小批量随机梯度下降，动量法需要对每一个自变量维护一个同它一样形状的速度变量，且超参数里多了动量超参数。实现中，我们将速度变量用更广义的状态变量 `states` 表示。

```
features, labels = d2l.get_data_ch7()

def init_momentum_states():
    v_w = torch.zeros((features.shape[1], 1), dtype=torch.float32)
    v_b = torch.zeros(1, dtype=torch.float32)
    return (v_w, v_b)

def sgd_momentum(params, states, hyperparams):
    for p, v in zip(params, states):
        v.data = hyperparams['momentum'] * v.data + hyperparams['lr'] * p.grad.data
        p.data -= v.data
```

我们先将动量超参数 `momentum` 设 0.5，这时可以看成是特殊的小批量随机梯度下降：其小批量随机梯度为最近 2 个时间步的 2 倍小批量梯度的加权平均。> 注：个人认为这里不应该是“加权平均”而应该是“加权和”，因为根据 7.4.2.2 节分析，加权平均最后除以了 $1 - \gamma$ ，所以就相当于没有进行平均。

```
d2l.train_ch7(sgd_momentum, init_momentum_states(),
    {'lr': 0.02, 'momentum': 0.5}, features, labels)
```

输出：

```
loss: 0.245518, 0.042304 sec per epoch
```

将动量超参数 `momentum` 增大到 0.9，这时依然可以看成是特殊的小批量随机梯度下降：其小批量随机梯度为最近 10 个时间步的 10 倍小批量梯度的加权平均。我们先保持学习率 0.02 不变。> 同理，这里不应该是“加权平均”而应该是“加权和”。

```
d2l.train_ch7(sgd_momentum, init_momentum_states(),
    {'lr': 0.02, 'momentum': 0.9}, features, labels)
```

输出：

```
loss: 0.252046, 0.095708 sec per epoch
```

可见目标函数值在后期迭代过程中的变化不够平滑。直觉上，10 倍小批量梯度比 2 倍小批量梯度大了 5 倍，我们可以试着将学习率减小到原来的 1/5。此时目标函数值在下降了一段时间后变化更加平滑。> 这也印证了刚刚的观点。

```
d2l.train_ch7(sgd_momentum, init_momentum_states(),
    {'lr': 0.004, 'momentum': 0.9}, features, labels)
```

输出：

```
loss: 0.242905, 0.073496 sec per epoch
```

7.4.4 简洁实现

在 PyTorch 中，只需要通过参数 `momentum` 来指定动量超参数即可使用动量法。

```
d2l.train_pytorch_ch7(torch.optim.SGD, {'lr': 0.004, 'momentum': 0.9},
    features, labels)
```

输出：

```
loss: 0.253280, 0.060247 sec per epoch
```

小结

- 动量法使用了指数加权移动平均的思想。它将过去时间步的梯度做了加权平均，且权重按时间步指数衰减。
 - 动量法使得相邻时间步的自变量更新在方向上更加一致。
-

注：除代码外本节与原书此节基本相同，[原书传送门](#)

7.5 AdaGrad 算法

在之前介绍过的优化算法中，目标函数自变量的每一个元素在相同时间步都使用同一个学习率来自我迭代。举个例子，假设目标函数为 f ，自变量为一个二维向量 $[x_1, x_2]^\top$ ，该向量中每一个元素在迭代时都使用相同的学习率。例如，在学习率为 η 的梯度下降中，元素 x_1 和 x_2 都使用相同的学习率 η 来自我迭代：

$$x_1 \leftarrow x_1 - \eta \frac{\partial f}{\partial x_1}, \quad x_2 \leftarrow x_2 - \eta \frac{\partial f}{\partial x_2}.$$

在 7.4 节（动量法）里我们看到当 x_1 和 x_2 的梯度值有较大差别时，需要选择足够小的学习率使得自变量在梯度值较大的维度上不发散。但这样会导致自变量在梯度值较小的维度上迭代过慢。动量法依赖指数加权移动平均使得自变量的更新方向更加一致，从而降低发散的可能。本节我们介绍 AdaGrad 算法，它根据自变量在每个维度的梯度值的大小来调整各个维度上的学习率，从而避免统一的学习率难以适应所有维度的问题 [1]。

7.5.1 算法

AdaGrad 算法会使用一个小批量随机梯度 \mathbf{g}_t 按元素平方的累加变量 \mathbf{s}_t 。在时间步 0，AdaGrad 将 \mathbf{s}_0 中每个元素初始化为 0。在时间步 t ，首先将小批量随机梯度 \mathbf{g}_t 按元素平方后累加到变量 \mathbf{s}_t ：

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + \mathbf{g}_t \odot \mathbf{g}_t,$$

其中 \odot 是按元素相乘。接着，我们将目标函数自变量中每个元素的学习率通过按元素运算重新调整一下：

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t,$$

其中 η 是学习率， ϵ 是为了维持数值稳定性而添加的常数，如 10^{-6} 。这里开方、除法和乘法的运算都是按元素运算的。这些按元素运算使得目标函数自变量中每个元素都分别拥有自己的学习率。

7.5.2 特点

需要强调的是，小批量随机梯度按元素平方的累加变量 \mathbf{s}_t 出现在学习率的分母项中。因此，如果目标函数有关自变量中某个元素的偏导数一直都较大，那么该元素的学习率将下降较快；反之，如果目标函数有关自变量中某个元素的偏导数一直都较小，那

么该元素的学习率将下降较慢。然而，由于 s_t 一直在累加按元素平方的梯度，自变量中每个元素的学习率在迭代过程中一直在降低（或不变）。所以，当学习率在迭代早期降得较快且当前解依然不佳时，AdaGrad 算法在迭代后期由于学习率过小，可能较难找到一个有用的解。

下面我们仍然以目标函数 $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ 为例观察 AdaGrad 算法对自变量的迭代轨迹。我们实现 AdaGrad 算法并使用和上一节实验中相同的学习率 0.4。可以看到，自变量的迭代轨迹较平滑。但由于 s_t 的累加效果使学习率不断衰减，自变量在迭代后期的移动幅度较小。

```
%matplotlib inline
import math
import torch
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l

def adagrad_2d(x1, x2, s1, s2):
    g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6 # 前两项为自变量梯度
    s1 += g1 ** 2
    s2 += g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta = 0.4
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

输出：

```
epoch 20, x1 -2.382563, x2 -0.158591
```

下面将学习率增大到 2。可以看到自变量更为迅速地逼近了最优解。

```
eta = 2
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

输出:

```
epoch 20, x1 -0.002295, x2 -0.000000
```

7.5.3 从零开始实现

同动量法一样，AdaGrad 算法需要对每个自变量维护同它一样形状的状态变量。我们根据 AdaGrad 算法中的公式实现该算法。

```
features, labels = d2l.get_data_ch7()

def init_adagrad_states():
    s_w = torch.zeros((features.shape[1], 1), dtype=torch.float32)
    s_b = torch.zeros(1, dtype=torch.float32)
    return (s_w, s_b)

def adagrad(params, states, hyperparams):
    eps = 1e-6
    for p, s in zip(params, states):
        s.data += (p.grad.data**2)
        p.data -= hyperparams['lr'] * p.grad.data / torch.sqrt(s + eps)
```

与 7.3 节（小批量随机梯度下降）中的实验相比，这里使用更大的学习率来训练模型。

```
d2l.train_ch7(adagrad, init_adagrad_states(), {'lr': 0.1}, features, labels)
```

输出:

```
loss: 0.243675, 0.049749 sec per epoch
```

7.5.4 简洁实现

通过名称为 `Adagrad` 的优化器方法，我们便可使用 PyTorch 提供的 AdaGrad 算法来训练模型。

```
d2l.train_pytorch_ch7(torch.optim.Adagrad, {'lr': 0.1}, features, labels)
```

输出:

```
loss: 0.243147, 0.040675 sec per epoch
```

小结

- AdaGrad 算法在迭代过程中不断调整学习率，并让目标函数自变量中每个元素都分别拥有自己的学习率。
- 使用 AdaGrad 算法时，自变量中每个元素的学习率在迭代过程中一直在降低（或不变）。

参考文献

- [1] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul), 2121-2159.
-

注：除代码外本节与原书此节基本相同，[原书传送门](#)

7.6 RMSProp 算法

我们在 7.5 节（AdaGrad 算法）中提到，因为调整学习率时分母上的变量 s_t 一直在累加按元素平方的小批量随机梯度，所以目标函数自变量每个元素的学习率在迭代过程中一直在降低（或不变）。因此，当学习率在迭代早期降得较快且当前解依然不佳时，AdaGrad 算法在迭代后期由于学习率过小，可能较难找到一个有用的解。为了解决这一问题，RMSProp 算法对 AdaGrad 算法做了一点小小的修改。该算法源自 Coursera 上的一门课程，即“机器学习的神经网络”[1]。

7.6.1 算法

我们在 7.4 节（动量法）里介绍过指数加权移动平均。不同于 AdaGrad 算法里状态变量 s_t 是截至时间步 t 所有小批量随机梯度 \mathbf{g}_t 按元素平方和，RMSProp 算法将这些梯度按元素平方做指数加权移动平均。具体来说，给定超参数 $0 \leq \gamma < 1$ ，RMSProp 算法在时间步 $t > 0$ 计算

$$\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t \odot \mathbf{g}_t.$$

和 AdaGrad 算法一样，RMSProp 算法将目标函数自变量中每个元素的学习率通过按元素运算重新调整，然后更新自变量

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t,$$

其中 η 是学习率， ϵ 是为了维持数值稳定性而添加的常数，如 10^{-6} 。因为 RMSProp 算法的状态变量 s_t 是对平方项 $\mathbf{g}_t \odot \mathbf{g}_t$ 的指数加权移动平均，所以可以看作是最近 $1/(1 - \gamma)$ 个时间步的小批量随机梯度平方项的加权平均。如此一来，自变量每个元素的学习率在迭代过程中就不再一直降低（或不变）。

照例，让我们先观察 RMSProp 算法对目标函数 $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ 中自变量的迭代轨迹。回忆在 7.5 节（AdaGrad 算法）使用的学习率为 0.4 的 AdaGrad 算法，自变量在迭代后期的移动幅度较小。但在同样的学习率下，RMSProp 算法可以更快逼近最优解。

```
%matplotlib inline
import math
import torch
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
```

```

def rmsprop_2d(x1, x2, s1, s2):
    g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
    s1 = gamma * s1 + (1 - gamma) * g1 ** 2
    s2 = gamma * s2 + (1 - gamma) * g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta, gamma = 0.4, 0.9
d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))

```

输出：

```
epoch 20, x1 -0.010599, x2 0.000000
```

7.6.2 从零开始实现

接下来按照 RMSProp 算法中的公式实现该算法。

```

features, labels = d2l.get_data_ch7()

def init_rmsprop_states():
    s_w = torch.zeros((features.shape[1], 1), dtype=torch.float32)
    s_b = torch.zeros(1, dtype=torch.float32)
    return (s_w, s_b)

def rmsprop(params, states, hyperparams):
    gamma, eps = hyperparams['gamma'], 1e-6
    for p, s in zip(params, states):
        s.data = gamma * s.data + (1 - gamma) * (p.grad.data)**2
        p.data -= hyperparams['lr'] * p.grad.data / torch.sqrt(s + eps)

```

我们将初始学习率设为 0.01，并将超参数 γ 设为 0.9。此时，变量 s_t 可看作是最近 $1/(1 - 0.9) = 10$ 个时间步的平方项 $\mathbf{g}_t \odot \mathbf{g}_t$ 的加权平均。

```
d2l.train_ch7(rmsprop, init_rmsprop_states(), {'lr': 0.01, 'gamma': 0.9},  
              features, labels)
```

输出：

```
loss: 0.243452, 0.049984 sec per epoch
```

7.6.3 简洁实现

通过名称为 `RMSprop` 的优化器方法，我们便可使用 PyTorch 提供的 `RMSProp` 算法来训练模型。注意，超参数 γ 通过 `alpha` 指定。

```
d2l.train_pytorch_ch7(torch.optim.RMSprop, {'lr': 0.01, 'alpha': 0.9},  
                      features, labels)
```

输出：

```
loss: 0.243676, 0.043637 sec per epoch
```

小结

- RMSProp 算法和 AdaGrad 算法的不同在于，RMSProp 算法使用了小批量随机梯度按元素平方的指数加权移动平均来调整学习率。

参考文献

[1] Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning, 4(2), 26-31.

注：除代码外本节与原书此节基本相同，[原书传送门](#)

7.7 AdaDelta 算法

除了 RMSProp 算法以外，另一个常用优化算法 AdaDelta 算法也针对 AdaGrad 算法在迭代后期可能较难找到有用解的问题做了改进 [1]。有意思的是，AdaDelta 算法没有学习率这一超参数。

7.7.1 算法

AdaDelta 算法也像 RMSProp 算法一样，使用了小批量随机梯度 \mathbf{g}_t 按元素平方的指数加权移动平均变量 \mathbf{s}_t 。在时间步 0，它的所有元素被初始化为 0。给定超参数 $0 \leq \rho < 1$ （对应 RMSProp 算法中的 γ ），在时间步 $t > 0$ ，同 RMSProp 算法一样计算

$$\mathbf{s}_t \leftarrow \rho \mathbf{s}_{t-1} + (1 - \rho) \mathbf{g}_t \odot \mathbf{g}_t.$$

与 RMSProp 算法不同的是，AdaDelta 算法还维护一个额外的状态变量 $\Delta\mathbf{x}_t$ ，其元素同样在时间步 0 时被初始化为 0。我们使用 $\Delta\mathbf{x}_{t-1}$ 来计算自变量的变化量：

$$\mathbf{g}'_t \leftarrow \sqrt{\frac{\Delta\mathbf{x}_{t-1} + \epsilon}{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t,$$

其中 ϵ 是为了维持数值稳定性而添加的常数，如 10^{-5} 。接着更新自变量：

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}'_t.$$

最后，我们使用 $\Delta\mathbf{x}_t$ 来记录自变量变化量 \mathbf{g}'_t 按元素平方的指数加权移动平均：

$$\Delta\mathbf{x}_t \leftarrow \rho \Delta\mathbf{x}_{t-1} + (1 - \rho) \mathbf{g}'_t \odot \mathbf{g}'_t.$$

可以看到，如不考虑 ϵ 的影响，AdaDelta 算法跟 RMSProp 算法的不同之处在于使用 $\sqrt{\Delta\mathbf{x}_{t-1}}$ 来替代学习率 η 。

7.7.2 从零开始实现

AdaDelta 算法需要对每个自变量维护两个状态变量，即 \mathbf{s}_t 和 $\Delta\mathbf{x}_t$ 。我们按 AdaDelta 算法中的公式实现该算法。

```
%matplotlib inline
import torch
import sys
sys.path.append("../")
```

```
import d2lzh_pytorch as d2l

features, labels = d2l.get_data_ch7()

def init_adadelta_states():
    s_w, s_b = torch.zeros((features.shape[1], 1), dtype=torch.float32), torch.zeros(
        delta_w, delta_b = torch.zeros((features.shape[1], 1), dtype=torch.float32), torch.zeros(
    return ((s_w, delta_w), (s_b, delta_b))

def adadelta(params, states, hyperparams):
    rho, eps = hyperparams['rho'], 1e-5
    for p, (s, delta) in zip(params, states):
        s[:] = rho * s + (1 - rho) * (p.grad.data**2)
        g = p.grad.data * torch.sqrt((delta + eps) / (s + eps))
        p.data -= g
        delta[:] = rho * delta + (1 - rho) * g * g
```

使用超参数 $\rho = 0.9$ 来训练模型。

```
d2l.train_ch7(adadelta, init_adadelta_states(), {'rho': 0.9}, features, labels)
```

输出：

```
loss: 0.243728, 0.062991 sec per epoch
```

7.7.3 简洁实现

通过名称为 Adadelta 的优化器方法，我们便可使用 PyTorch 提供的 AdaDelta 算法。它的超参数可以通过 `rho` 来指定。

```
d2l.train_pytorch_ch7(torch.optim.Adadelta, {'rho': 0.9}, features, labels)
```

输出：

```
loss: 0.242104, 0.047702 sec per epoch
```

小结

- AdaDelta 算法没有学习率超参数，它通过使用有关自变量更新量平方的指数加权移动平均的项来替代 RMSProp 算法中的学习率。

参考文献

- [1] Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. arXiv preprint arXiv:1212.5701.
-

注：除代码外本节与原书此节基本相同，[原书传送门](#)

7.8 Adam 算法

Adam 算法在 RMSProp 算法基础上对小批量随机梯度也做了指数加权移动平均 [1]。下面我们来介绍这个算法。> 所以 Adam 算法可以看做是 RMSProp 算法与动量法的结合。

7.8.1 算法

Adam 算法使用了动量变量 \mathbf{v}_t 和 RMSProp 算法中小批量随机梯度按元素平方的指数加权移动平均变量 \mathbf{s}_t ，并在时间步 0 将它们中每个元素初始化为 0。给定超参数 $0 \leq \beta_1 < 1$ （算法作者建议设为 0.9），时间步 t 的动量变量 \mathbf{v}_t 即小批量随机梯度 \mathbf{g}_t 的指数加权移动平均：

$$\mathbf{v}_t \leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1) \mathbf{g}_t.$$

和 RMSProp 算法中一样，给定超参数 $0 \leq \beta_2 < 1$ （算法作者建议设为 0.999），将小批量随机梯度按元素平方后的项 $\mathbf{g}_t \odot \mathbf{g}_t$ 做指数加权移动平均得到 \mathbf{s}_t ：

$$\mathbf{s}_t \leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t \odot \mathbf{g}_t.$$

由于我们将 \mathbf{v}_0 和 \mathbf{s}_0 中的元素都初始化为 0，在时间步 t 我们得到 $\mathbf{v}_t = (1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} \mathbf{g}_i$ 。将过去各时间步小批量随机梯度的权值相加，得到 $(1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} = 1 - \beta_1^t$ 。需要注意的是，当 t 较小时，过去各时间步小批量随机梯度权值之和会较小。例如，当 $\beta_1 = 0.9$ 时， $\mathbf{v}_1 = 0.1 \mathbf{g}_1$ 。为了消除这样的影响，对于任意时间步 t ，我们可以将 \mathbf{v}_t 再除以 $1 - \beta_1^t$ ，从而使过去各时间步小批量随机梯度权值之和为 1。这也叫作偏差修正。在 Adam 算法中，我们对变量 \mathbf{v}_t 和 \mathbf{s}_t 均作偏差修正：

$$\hat{\mathbf{v}}_t \leftarrow \frac{\mathbf{v}_t}{1 - \beta_1^t},$$

$$\hat{\mathbf{s}}_t \leftarrow \frac{\mathbf{s}_t}{1 - \beta_2^t}.$$

接下来，Adam 算法使用以上偏差修正后的变量 $\hat{\mathbf{v}}_t$ 和 $\hat{\mathbf{s}}_t$ ，将模型参数中每个元素的学习率通过按元素运算重新调整：

$$\mathbf{g}'_t \leftarrow \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon},$$

其中 η 是学习率， ϵ 是为了维持数值稳定性而添加的常数，如 10^{-8} 。和 AdaGrad 算法、RMSProp 算法以及 AdaDelta 算法一样，目标函数自变量中每个元素都分别拥有自己的学习率。最后，使用 \mathbf{g}'_t 迭代自变量：

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}'_t.$$

7.8.2 从零开始实现

我们按照 Adam 算法中的公式实现该算法。其中时间步 t 通过 `hyperparams` 参数传入 `adam` 函数。

```
%matplotlib inline
import torch
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l

features, labels = d2l.get_data_ch7()

def init_adam_states():
    v_w, v_b = torch.zeros((features.shape[1], 1), dtype=torch.float32), torch.zeros(
        features.shape[1], 1, dtype=torch.float32)
    s_w, s_b = torch.zeros((features.shape[1], 1), dtype=torch.float32), torch.zeros(
        features.shape[1], 1, dtype=torch.float32)
    return ((v_w, s_w), (v_b, s_b))

def adam(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s) in zip(params, states):
        v[:] = beta1 * v + (1 - beta1) * p.grad.data
        s[:] = beta2 * s + (1 - beta2) * p.grad.data**2
        v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
        s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
        p.data -= hyperparams['lr'] * v_bias_corr / (torch.sqrt(s_bias_corr) + eps)
    hyperparams['t'] += 1
```

使用学习率为 0.01 的 Adam 算法来训练模型。

```
d2l.train_ch7(adam, init_adam_states(), {'lr': 0.01, 't': 1}, features, labels)
```

输出：

```
loss: 0.245370, 0.065155 sec per epoch
```

7.8.3 简洁实现

通过名称为“adam”的 Trainer 实例，我们便可使用 Gluon 提供的 Adam 算法。

```
d2l.train_pytorch_ch7(torch.optim.Adam, {'lr': 0.01}, features, labels)
```

输出：

```
loss: 0.242066, 0.056867 sec per epoch
```

小结

- Adam 算法在 RMSProp 算法的基础上对小批量随机梯度也做了指数加权移动平均。
- Adam 算法使用了偏差修正。

参考文献

- [1] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.
-

注：除代码外本节与原书此节基本相同，[原书传送门](#)

8.1 命令式和符号式混合编程

本书到目前为止一直都在使用命令式编程，它使用编程语句改变程序状态。考虑下面这段简单的命令式程序。

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

fancy_func(1, 2, 3, 4) # 10
```

和我们预期的一样，在运行语句 `e = add(a, b)` 时，Python 会做加法运算并将结果存储在变量 `e` 中，从而令程序的状态发生改变。类似地，后面的两条语句 `f = add(c, d)` 和 `g = add(e, f)` 会依次做加法运算并存储变量。

虽然使用命令式编程很方便，但它的运行可能很慢。一方面，即使 `fancy_func` 函数中的 `add` 是被重复调用的函数，Python 也会逐一执行这 3 条函数调用语句。另一方面，我们需要保存变量 `e` 和 `f` 的值直到 `fancy_func` 中所有语句执行结束。这是因为在执行 `e = add(a, b)` 和 `f = add(c, d)` 这 2 条语句之后我们并不知道变量 `e` 和 `f` 是否会被程序的其他部分使用。

与命令式编程不同，符号式编程通常在计算流程完全定义好后才被执行。多个深度学习框架，如 `Theano` 和 `TensorFlow`，都使用了符号式编程。通常，符号式编程的程序需要下面 3 个步骤：

1. 定义计算流程；
2. 把计算流程编译成可执行的程序；
3. 给定输入，调用编译好的程序执行。

下面我们用符号式编程重新实现本节开头给出的命令式编程代码。

```
def add_str():
    return ''
def add(a, b):
```

```
    return a + b
    ...

def fancy_func_str():
    return ''

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
    ...

def evoke_str():
    return add_str() + fancy_func_str() + ''
print(fancy_func(1, 2, 3, 4))
...
...  
  
prog = evoke_str()
print(prog)
y = compile(prog, '', 'exec')
exec(y)
```

输出：

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))
```

以上定义的 3 个函数都仅以字符串的形式返回计算流程。最后，我们通过 `compile` 函数编译完整的计算流程并运行。由于在编译时系统能够完整地获取整个程序，因此有更多空间优化计算。例如，编译的时候可以将程序改写成 `print((1 + 2) + (3 + 4))`，甚至直接改写成 `print(10)`。这样不仅减少了函数调用，还节省了内存。

对比这两种编程方式，我们可以看到以下两点。

- 命令式编程更方便。当我们在 Python 里使用命令式编程时，大部分代码编写起来都很直观。同时，命令式编程更容易调试。这是因为我们可以很方便地获取并打印所有的中间变量值，或者使用 Python 的调试工具。
- 符号式编程更高效并更容易移植。一方面，在编译的时候系统容易做更多优化；另一方面，符号式编程可以将程序变成一个与 Python 无关的格式，从而可以使程序在非 Python 环境下运行，以避开 Python 解释器的性能问题。

8.1.1 混合式编程取两者之长

大部分深度学习框架在命令式编程和符号式编程之间二选一。例如，**Theano** 和受其启发的后来者 **TensorFlow** 使用了符号式编程，**Chainer** 和它的追随者 **PyTorch** 使用了命令式编程，而 **Gluon** 则采用了混合式编程的方式。

.....

由于 PyTorch 仅仅采用了命令式编程，所以跳过本节剩余部分，感兴趣的可以去看[原文](#)

8.2 异步计算

此节内容对应 PyTorch 的版本本人没怎么用过，网上参考资料也比较少，所以略：），有兴趣的可以去看看[原文](#)。

关于 PyTorch 的异步执行我只在[官方文档](#)找到了一段：> By default, GPU operations are asynchronous. When you call a function that uses the GPU, the operations are enqueued to the particular device, but not necessarily executed until later. This allows us to execute more computations in parallel, including operations on CPU or other GPUs. In general, the effect of asynchronous computation is invisible to the caller, because (1) each device executes operations in the order they are queued, and (2) PyTorch automatically performs necessary synchronization when copying data between CPU and GPU or between two GPUs. Hence, computation will proceed as if every operation was executed synchronously. You can force synchronous computation by setting environment variable CUDA_LAUNCH_BLOCKING=1. This can be handy when an error occurs on the GPU. (With asynchronous execution, such an error isn't reported until after the operation is actually executed, so the stack trace does not show where it was requested.)

大致翻译一下就是：默认情况下，PyTorch 中的 GPU 操作是异步的。当调用一个使用 GPU 的函数时，这些操作会在特定的设备上排队但不一定会在稍后立即执行。这就使我们可以并行更多的计算，包括 CPU 或其他 GPU 上的操作。一般情况下，异步计算的效果对调用者是不可见的，因为（1）每个设备按照它们排队的顺序执行操作，（2）在 CPU 和 GPU 之间或两个 GPU 之间复制数据时，PyTorch 会自动执行必要的同步操作。因此，计算将按每个操作同步执行的方式进行。可以通过设置环境变量 CUDA_LAUNCH_BLOCKING = 1 来强制进行同步计算。当 GPU 产生 error 时，这可能非常有用。（异步执行时，只有在实际执行操作之后才会报告此类错误，因此堆栈跟踪不会显示请求的位置。）

8.3 自动并行计算

上一节提到，默认情况下，GPU 操作是异步的。当调用一个使用 GPU 的函数时，这些操作会在特定的设备上排队，但不一定会在稍后执行。这允许我们并行更多的计算，包括 CPU 或其他 GPU 上的操作。下面看一个简单的例子。

首先导入本节中实验所需的包或模块。注意，需要至少 2 块 GPU 才能运行本节实验。

```
import torch
import time

assert torch.cuda.device_count() >= 2
```

我们先实现一个简单的计时类。

```
class Benchmark(): # 本类已保存在 d2lzh_pytorch 包中方便以后使用
    def __init__(self, prefix=None):
        self.prefix = prefix + ' ' if prefix else ''

    def __enter__(self):
        self.start = time.time()

    def __exit__(self, *args):
        print('%stime: %.4f sec' % (self.prefix, time.time() - self.start))
```

再定义 `run` 函数，令它做 20000 次矩阵乘法。

```
def run(x):
    for _ in range(20000):
        y = torch.mm(x, x)
```

接下来，分别在两块 GPU 上创建 Tensor。

```
x_gpu1 = torch.rand(size=(100, 100), device='cuda:0')
x_gpu2 = torch.rand(size=(100, 100), device='cuda:1')
```

然后，分别使用它们运行 `run` 函数并打印运行所需时间。

```
with Benchmark('Run on GPU1.'):
    run(x_gpu1)
    torch.cuda.synchronize()

with Benchmark('Then run on GPU2.'):
    run(x_gpu2)
    torch.cuda.synchronize()
```

输出：

```
Run on GPU1. time: 0.2989 sec
Then run on GPU2. time: 0.3518 sec
```

尝试系统能自动并行这两个任务：

```
with Benchmark('Run on both GPU1 and GPU2 in parallel.'):
    run(x_gpu1)
    run(x_gpu2)
    torch.cuda.synchronize()
```

输出：

```
Run on both GPU1 and GPU2 in parallel. time: 0.5076 sec
```

可以看到，当两个计算任务一起执行时，执行总时间小于它们分开执行的总和。这表明，PyTorch 能有效地实现在不同设备上自动并行计算。

注：本节与原书有很多不同，[原书传送门](#)

8.4 多 GPU 计算

注：相对于本章的前面几节，我们实际中更可能遇到本节所讨论的情况：多 GPU 计算。原书将 MXNet 的多 GPU 计算分成了 8.4 和 8.5 两节，但我们将关于 PyTorch 的多 GPU 计算统一放在本节讨论。需要注意的是，这里我们谈论的是单主机多 GPU 计算而不是分布式计算。如果对分布式计算感兴趣可以参考[PyTorch 官方文档](#)。

本节中我们将展示如何使用多块 GPU 计算，例如，使用多块 GPU 训练同一个模型。正如所期望的那样，运行本节中的程序需要至少 2 块 GPU。事实上，一台机器上安装多块 GPU 很常见，这是因为主板上通常会有多个 PCIe 插槽。如果正确安装了 NVIDIA 驱动，我们可以通过在命令行输入 `nvidia-smi` 命令来查看当前计算机上的全部 GPU（或者在 jupyter notebook 中运行`!nvidia-smi`）。

```
nvidia-smi
```

输出：

```
Wed May 15 23:12:38 2019
```

NVIDIA-SMI 390.48										Driver Version: 390.48	
GPU	Name	Persistence-M Bus-Id			Disp.A Volatile Uncorr.		ECC			Memory-Usage GPU-Util	Compute M.
		Fan	Temp	Perf	Pwr:Usage/Cap	Memory	Usage				
0	TITAN X (Pascal)	Off	00000000:02:00.0	Off						N/A	
46%	76C	P2	87W / 250W		10995MiB / 12196MiB		0%			Default	
1	TITAN X (Pascal)	Off	00000000:04:00.0	Off						N/A	
53%	84C	P2	143W / 250W		11671MiB / 12196MiB		4%			Default	
2	TITAN X (Pascal)	Off	00000000:83:00.0	Off						N/A	
62%	87C	P2	190W / 250W		12096MiB / 12196MiB		100%			Default	
3	TITAN X (Pascal)	Off	00000000:84:00.0	Off						N/A	
51%	83C	P2	255W / 250W		8144MiB / 12196MiB		58%			Default	

Processes:				GPU Memory
GPU	PID	Type	Process name	Usage
<hr/>				
0	44683	C	python	3289MiB
0	155760	C	python	4345MiB
0	158310	C	python	2297MiB
0	172338	C	/home/yzs/anaconda3/bin/python	1031MiB
1	139985	C	python	11653MiB
2	38630	C	python	5547MiB
2	43127	C	python	5791MiB
2	156710	C	python3	725MiB
3	14444	C	python3	1891MiB
3	43407	C	python	5841MiB
3	88478	C	/home/tangss/.conda/envs/py36/bin/python	379MiB

从上面的输出可以看到一共有四块 TITAN X GPU，每一块总共有约 12 个 G 的显存，此时每块的显存都占得差不多了……此外还可以看到 GPU 利用率、运行的所有程序等信息。

Pytorch 在 0.4.0 及以后的版本中已经提供了多 GPU 训练的方式，本文用一个简单的例子讲解下使用 Pytorch 多 GPU 训练的方式以及一些注意的地方。

8.4.1 多 GPU 计算

先定义一个模型：

```
import torch
net = torch.nn.Linear(10, 1).cuda()
net
```

输出：

```
Linear(in_features=10, out_features=1, bias=True)
```

要想使用 PyTorch 进行多 GPU 计算，最简单的方法是直接用 `torch.nn.DataParallel` 将模型 wrap 一下即可：

```
net = torch.nn.DataParallel(net)
net
```

输出:

```
DataParallel(
  (module): Linear(in_features=10, out_features=1, bias=True)
)
```

这时， 默认所有存在的 GPU 都会被使用。

如果我们机子中有很多 GPU(例如上面显示我们有 4 张显卡，但是只有第 0、3 块还剩下一点点显存)，但我们只想使用 0、3 号显卡，那么我们可以用参数 `device_ids` 指定即可:`torch.nn.DataParallel(net, device_ids=[0, 3])`。

8.4.2 多 GPU 模型的保存与加载

我们现在来尝试一下按照 4.5 节（读取和存储）推荐的方式进行一下模型的保存与加载。保存模型:

```
torch.save(net.state_dict(), "./8.4_model.pt")
```

加载模型前我们一般要先进行一下模型定义，此时的 `new_net` 并没有使用多 GPU:

```
new_net = torch.nn.Linear(10, 1)
new_net.load_state_dict(torch.load("./8.4_model.pt"))
```

然后我们发现报错了:

```
RuntimeError: Error(s) in loading state_dict for Linear:
  Missing key(s) in state_dict: "weight", "bias".
  Unexpected key(s) in state_dict: "module.weight", "module.bias".
```

事实上 `DataParallel` 也是一个 `nn.Module`，只是这个类其中有一个 `module` 就是传入的实际模型。因此当我们调用 `DataParallel` 后，模型结构变了（在外面加了一层而已，从 8.4.1 节两个输出可以对比看出来）。所以直接加载肯定会报错的，因为模型结构对不上。

所以正确的方法是保存的时候只保存 `net.module`:

```
torch.save(net.module.state_dict(), "./8.4_model.pt")
new_net.load_state_dict(torch.load("./8.4_model.pt")) # 加载成功
```

或者先将 `new_net` 用 `DataParallel` 包括以下再用上面报错的方法进行模型加载:

```
torch.save(net.state_dict(), "./8.4_model.pt")
new_net = torch.nn.Linear(10, 1)
new_net = torch.nn.DataParallel(new_net)
new_net.load_state_dict(torch.load("./8.4_model.pt")) # 加载成功
```

注意这两种方法的区别，推荐用第一种方法，因为可以按照普通的加载方法进行正确加载。

注：本节与原书基本不同，[原书传送门](#)

9.1 图像增广

在 5.6 节（深度卷积神经网络）里我们提到过，大规模数据集是成功应用深度神经网络的前提。图像增广（image augmentation）技术通过对训练图像做一系列随机改变，来产生相似但又不同的训练样本，从而扩大训练数据集的规模。图像增广的另一种解释是，随机改变训练样本可以降低模型对某些属性的依赖，从而提高模型的泛化能力。例如，我们可以对图像进行不同方式的裁剪，使感兴趣的物体出现在不同位置，从而减轻模型对物体出现位置的依赖性。我们也可以调整亮度、色彩等因素来降低模型对色彩的敏感度。可以说，在当年 AlexNet 的成功中，图像增广技术功不可没。本节我们将讨论这个在计算机视觉里被广泛使用的技术。

首先，导入实验所需的包或模块。

```
%matplotlib inline
import time
import torch
from torch import nn, optim
from torch.utils.data import Dataset, DataLoader
import torchvision
from PIL import Image

import sys
sys.path.append("../")
import d2lzh_pytorch as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

9.1.1 常用的图像增广方法

我们来读取一张形状为 400×500 （高和宽分别为 400 像素和 500 像素）的图像作为实验的样例。

```
d2l.set_figsize()
img = Image.open('../img/cat1.jpg')
d2l.plt.imshow(img)
```

下面定义绘图函数 `show_images`。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def show_images(imgs, num_rows, num_cols, scale=2):
```

```
figsize = (num_cols * scale, num_rows * scale)
_, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
for i in range(num_rows):
    for j in range(num_cols):
        axes[i][j].imshow(imgs[i * num_cols + j])
        axes[i][j].axes.get_xaxis().set_visible(False)
        axes[i][j].axes.get_yaxis().set_visible(False)
return axes
```

大部分图像增广方法都有一定的随机性。为了方便观察图像增广的效果，接下来我们定义一个辅助函数 `apply`。这个函数对输入图像 `img` 多次运行图像增广方法 `aug` 并展示所有的结果。

```
def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
    Y = [aug(img) for _ in range(num_rows * num_cols)]
    show_images(Y, num_rows, num_cols, scale)
```

9.1.1.1 翻转和裁剪

左右翻转图像通常不改变物体的类别。它是最早也是最广泛使用的一种图像增广方法。下面我们通过 `torchvision.transforms` 模块创建 `RandomHorizontalFlip` 实例来实现一半概率的图像水平（左右）翻转。

```
apply(img, torchvision.transforms.RandomHorizontalFlip())
```

上下翻转不如左右翻转通用。但是至少对于样例图像，上下翻转不会造成识别障碍。下面我们创建 `RandomVerticalFlip` 实例来实现一半概率的图像垂直（上下）翻转。

```
apply(img, torchvision.transforms.RandomVerticalFlip())
```

在我们使用的样例图像里，猫在图像正中间，但一般情况下可能不是这样。在 5.4 节（池化层）里我们解释了池化层能降低卷积层对目标位置的敏感度。除此之外，我们还可以通过对图像随机裁剪来让物体以不同的比例出现在图像的不同位置，这同样能够降低模型对目标位置的敏感性。

在下面的代码里，我们每次随机裁剪出一块面积为原面积 $10\% \sim 100\%$ 的区域，且该区域的宽和高之比随机取自 $0.5 \sim 2$ ，然后再将该区域的宽和高分别缩放到 200 像素。若无特殊说明，本节中 a 和 b 之间的随机数指的是从区间 $[a, b]$ 中随机均匀采样所得到的连续值。

```
shape_aug = torchvision.transforms.RandomResizedCrop(200, scale=(0.1, 1), ratio=(0.5,
```

9.1.1.2 变化颜色

另一类增广方法是变化颜色。我们可以从 4 个方面改变图像的颜色：亮度（`brightness`）、对比度（`contrast`）、饱和度（`saturation`）和色调（`hue`）。在下面的例子里，我们将图像的亮度随机变化为原图亮度的 50% ($1 - 0.5$) ~ 150% ($1 + 0.5$)。

```
apply(img, torchvision.transforms.ColorJitter(brightness=0.5))
```

我们也可以随机变化图像的色调。

```
apply(img, torchvision.transforms.ColorJitter(hue=0.5))
```

类似地，我们也可以随机变化图像的对比度。

```
apply(img, torchvision.transforms.ColorJitter(contrast=0.5))
```

我们也可以同时设置如何随机变化图像的亮度（`brightness`）、对比度（`contrast`）、饱和度（`saturation`）和色调（`hue`）。

```
color_aug = torchvision.transforms.ColorJitter(  
    brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)  
apply(img, color_aug)
```

9.1.1.3 叠加多个图像增广方法

实际应用中我们会将多个图像增广方法叠加使用。我们可以通过 `Compose` 实例将上面定义的多个图像增广方法叠加起来，再应用到每张图像之上。

```
augs = torchvision.transforms.Compose([  
    torchvision.transforms.RandomHorizontalFlip(), color_aug, shape_aug])  
apply(img, augs)
```

9.1.2 使用图像增广训练模型

下面我们来看一个将图像增广应用在实际训练中的例子。这里我们使用 CIFAR-10 数据集，而不是之前我们一直使用的 Fashion-MNIST 数据集。这是因为 Fashion-MNIST 数据集中物体的位置和尺寸都已经经过归一化处理，而 CIFAR-10 数据集中物体的颜色和大小区别更加显著。下面展示了 CIFAR-10 数据集中前 32 张训练图像。

```
all_imgs = torchvision.datasets.CIFAR10(train=True, root="/Datasets/CIFAR", download=True)
# all_imgs 的每一个元素都是 (image, label)
show_images([all_imgs[i][0] for i in range(32)], 4, 8, scale=0.8);
```

为了在预测时得到确定的结果，我们通常只将图像增广应用在训练样本上，而不在预测时使用含随机操作的图像增广。在这里我们只使用最简单的随机左右翻转。此外，我们使用 `ToTensor` 将小批量图像转成 PyTorch 需要的格式，即形状为 (批量大小, 通道数, 高, 宽)、值域在 0 到 1 之间且类型为 32 位浮点数。

```
flip_aug = torchvision.transforms.Compose([
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor()])

no_aug = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor()])
```

接下来我们定义一个辅助函数来方便读取图像并应用图像增广。有关 `DataLoader` 的详细介绍，可参考更早的 3.5 节图像分类数据集 (Fashion-MNIST)。

```
num_workers = 0 if sys.platform.startswith('win32') else 4
def load_cifar10(is_train, augs, batch_size, root="/Datasets/CIFAR"):
    dataset = torchvision.datasets.CIFAR10(root=root, train=is_train, transform=augs,
                                           download=True)
    return DataLoader(dataset, batch_size=batch_size, shuffle=is_train, num_workers=num_workers)
```

9.1.2.1 使用图像增广训练模型

原书本节使用的多 GPU，由于我这里卡比较紧张就不使用多 GPU 了...关于 PyTorch 多 GPU 的使用可参考 8.4 节。

我们在 CIFAR-10 数据集上训练 5.11 节（残差网络）中介绍的 ResNet-18 模型。我们先定义 `train` 函数使用 GPU 训练并评价模型。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def train(train_iter, test_iter, net, loss, optimizer, device, num_epochs):
    net = net.to(device)
    print("training on ", device)
    batch_count = 0
    for epoch in range(num_epochs):
```

```

train_l_sum, train_acc_sum, n, start = 0.0, 0.0, 0, time.time()
for X, y in train_iter:
    X = X.to(device)
    y = y.to(device)
    y_hat = net(X)
    l = loss(y_hat, y)
    optimizer.zero_grad()
    l.backward()
    optimizer.step()
    train_l_sum += l.cpu().item()
    train_acc_sum += (y_hat.argmax(dim=1) == y).sum().cpu().item()
    n += y.shape[0]
    batch_count += 1
test_acc = d2l.evaluate_accuracy(test_iter, net)
print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f, time %.1f sec'
      % (epoch + 1, train_l_sum / batch_count, train_acc_sum / n, test_acc, t)

```

然后就可以定义 `train_with_data_aug` 函数使用图像增广来训练模型了。该函数使用 Adam 算法作为训练使用的优化算法，然后将图像增广应用于训练数据集之上，最后调用刚才定义的 `train` 函数训练并评价模型。

```

def train_with_data_aug(train_augs, test_augs, lr=0.001):
    batch_size, net = 256, d2l.resnet18(10)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    loss = torch.nn.CrossEntropyLoss()
    train_iter = load_cifar10(True, train_augs, batch_size)
    test_iter = load_cifar10(False, test_augs, batch_size)
    train(train_iter, test_iter, net, loss, optimizer, device, num_epochs=10)

```

下面使用随机左右翻转的图像增广来训练模型。

```
train_with_data_aug(flip_aug, no_aug)
```

输出：

```

training on  cuda
epoch 1, loss 1.3615, train acc 0.505, test acc 0.493, time 123.2 sec
epoch 2, loss 0.5003, train acc 0.645, test acc 0.620, time 123.0 sec

```

```
epoch 3, loss 0.2811, train acc 0.703, test acc 0.616, time 123.1 sec
epoch 4, loss 0.1890, train acc 0.735, test acc 0.686, time 123.0 sec
epoch 5, loss 0.1346, train acc 0.765, test acc 0.671, time 123.1 sec
epoch 6, loss 0.1029, train acc 0.787, test acc 0.674, time 123.1 sec
epoch 7, loss 0.0803, train acc 0.804, test acc 0.749, time 123.1 sec
epoch 8, loss 0.0644, train acc 0.822, test acc 0.717, time 123.1 sec
epoch 9, loss 0.0526, train acc 0.836, test acc 0.750, time 123.0 sec
epoch 10, loss 0.0433, train acc 0.851, test acc 0.754, time 123.1 sec
```

小结

- 图像增广基于现有训练数据生成随机图像从而应对过拟合。
 - 为了在预测时得到确定的结果，通常只将图像增广应用在训练样本上，而不是在预测时使用含随机操作的图像增广。
 - 可以从 torchvision 的 `transforms` 模块中获取有关图片增广的类。
-

注：本节与原书有一些不同，[原书传送门](#)

9.2 微调

在前面的一些章节中，我们介绍了如何在只有 6 万张图像的 Fashion-MNIST 训练数据集上训练模型。我们还描述了学术界当下使用最广泛的大规模图像数据集 ImageNet，它有超过 1,000 万的图像和 1,000 类的物体。然而，我们平常接触到数据集的规模通常在这两者之间。

假设我们想从图像中识别出不同种类的椅子，然后将购买链接推荐给用户。一种可能的方法是先找出 100 种常见的椅子，为每种椅子拍摄 1,000 张不同角度的图像，然后在收集到的图像数据集上训练一个分类模型。这个椅子数据集虽然可能比 Fashion-MNIST 数据集要庞大，但样本数仍然不及 ImageNet 数据集中样本数的十分之一。这可能会导致适用于 ImageNet 数据集的复杂模型在这个椅子数据集上过拟合。同时，因为数据量有限，最终训练得到的模型的精度也可能达不到实用的要求。

为了应对上述问题，一个显而易见的解决办法是收集更多的数据。然而，收集和标注数据会花费大量的时间和资金。例如，为了收集 ImageNet 数据集，研究人员花费了数百万美元的研究经费。虽然目前的数据采集成本已降低了不少，但其成本仍然不可忽略。

另外一种解决办法是应用迁移学习（transfer learning），将从源数据集学到的知识迁移到目标数据集上。例如，虽然 ImageNet 数据集的图像大多跟椅子无关，但在该数据集上训练的模型可以抽取较通用的图像特征，从而能够帮助识别边缘、纹理、形状和物体组成等。这些类似的特征对于识别椅子也可能同样有效。

本节我们介绍迁移学习中的一种常用技术：微调（fine tuning）。如图 9.1 所示，微调由以下 4 步构成。

1. 在源数据集（如 ImageNet 数据集）上预训练一个神经网络模型，即源模型。
2. 创建一个新的神经网络模型，即目标模型。它复制了源模型上除了输出层外的所有模型设计及其参数。我们假设这些模型参数包含了源数据集上学习到的知识，且这些知识同样适用于目标数据集。我们还假设源模型的输出层跟源数据集的标签紧密相关，因此在目标模型中不予采用。
3. 为目标模型添加一个输出大小为目标数据集类别个数的输出层，并随机初始化该层的模型参数。
4. 在目标数据集（如椅子数据集）上训练目标模型。我们将从头训练输出层，而其余层的参数都是基于源模型的参数微调得到的。

图 9.1 微调

当目标数据集远小于源数据集时，微调有助于提升模型的泛化能力。

9.2.1 热狗识别

接下来我们来实践一个具体的例子：热狗识别。我们将基于一个小数据集对在 ImageNet 数据集上训练好的 ResNet 模型进行微调。该小数据集含有数千张包含热狗和不包含热狗的图像。我们将使用微调得到的模型来识别一张图像中是否包含热狗。

首先，导入实验所需的包或模块。torchvision 的**models**包提供了常用的预训练模型。如果希望获取更多的预训练模型，可以使用使用**pretrained-models.pytorch**仓库。

```
%matplotlib inline
import torch
from torch import nn, optim
from torch.utils.data import Dataset, DataLoader
import torchvision
from torchvision.datasets import ImageFolder
from torchvision import transforms
from torchvision import models
import os

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l

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

9.2.1.1 获取数据集

我们使用的热狗数据集（[点击下载](#)）是从网上抓取的，它含有 1400 张包含热狗的正类图像，和同样多包含其他食品的负类图像。各类的 1000 张图像被用于训练，其余则用于测试。

我们首先将压缩后的数据集下载到路径 `data_dir` 之下，然后在该路径将下载好的数据集解压，得到两个文件夹 `hotdog/train` 和 `hotdog/test`。这两个文件夹下面均有 `hotdog` 和 `not-hotdog` 两个类别文件夹，每个类别文件夹里面是图像文件。

```
data_dir = '/S1/CSCL/tangss/Datasets'
os.listdir(os.path.join(data_dir, "hotdog")) # ['train', 'test']
```

我们创建两个 `ImageFolder` 实例来分别读取训练数据集和测试数据集中的所有图像文件。

```
train_imgs = ImageFolder(os.path.join(data_dir, 'hotdog/train'))
test_imgs = ImageFolder(os.path.join(data_dir, 'hotdog/test'))
```

下面画出前 8 张正类图像和最后 8 张负类图像。可以看到，它们的大小和高宽比各不相同。

```
hotdogs = [train_imgs[i][0] for i in range(8)]
not_hotdogs = [train_imgs[-i - 1][0] for i in range(8)]
d2l.show_images(hotdogs + not_hotdogs, 2, 8, scale=1.4);
```

在训练时，我们先从图像中裁剪出随机大小和随机高宽比的一块随机区域，然后将该区域缩放为高和宽均为 224 像素的输入。测试时，我们将图像的高和宽均缩放为 256 像素，然后从中裁剪出高和宽均为 224 像素的中心区域作为输入。此外，我们对 RGB（红、绿、蓝）三个颜色通道的数值做标准化：每个数值减去该通道所有数值的平均值，再除以该通道所有数值的标准差作为输出。> 注：在使用预训练模型时，一定要和预训练时作同样的预处理。如果你使用的是 `torchvision` 的 `models`，那就要求：All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225]. 如果你使用的是 [pretrained-models.pytorch](#) 仓库，请务必阅读其 README，其中说明了如何预处理。

```
# 指定 RGB 三个通道的均值和方差来将图像通道归一化
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
train_augs = transforms.Compose([
    transforms.RandomResizedCrop(size=224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    normalize
])

test_augs = transforms.Compose([
    transforms.Resize(size=256),
    transforms.CenterCrop(size=224),
    transforms.ToTensor(),
    normalize
])
```

9.2.1.2 定义和初始化模型

我们使用在 ImageNet 数据集上预训练的 ResNet-18 作为源模型。这里指定 `pretrained=True` 来自动下载并加载预训练的模型参数。在第一次使用时需要联网下载模型参数。

```
pretrained_net = models.resnet18(pretrained=True)
```

不管你是使用的 torchvision 的 `models` 还是 `pretrained-models.pytorch` 仓库， 默认都会将预训练好的模型参数下载到你的 home 目录下 `.torch` 文件夹。你可以通过修改环境变量 `$TORCH_MODEL_ZOO` 来更改下载目录：

```
export TORCH_MODEL_ZOO="/local/pretrainedmodels
```

另外我比较常使用的方法是，在其源码中找到下载地址直接浏览器输入地址下载， 下载好后将其放到环境变量 `$TORCH_MODEL_ZOO` 所指文件夹即可， 这样比较快。

下面打印源模型的成员变量 `fc`。作为一个全连接层，它将 ResNet 最终的全局平均池化层输出变换成 ImageNet 数据集上 1000 类的输出。

```
print(pretrained_net.fc)
```

输出：

```
Linear(in_features=512, out_features=1000, bias=True)
```

注：如果你使用的是其他模型，那可能没有成员变量 `fc`（比如 `models` 中的 VGG 预训练模型），所以正确做法是查看对应模型源码中其定义部分，这样既不会出错也能加深我们对模型的理解。`pretrained-models.pytorch` 仓库貌似统一了接口，但是我还是建议使用时查看一下对应模型的源码。

可见此时 `pretrained_net` 最后的输出个数等于目标数据集的类别数 1000。所以我们应该将最后的 `fc` 成修改我们需要的输出类别数：

```
pretrained_net.fc = nn.Linear(512, 2)
print(pretrained_net.fc)
```

输出：

```
Linear(in_features=512, out_features=2, bias=True)
```

此时，`pretrained_net` 的 `fc` 层就被随机初始化了，但是其他层依然保存着预训练得到的参数。由于是在很大的 ImageNet 数据集上预训练的，所以参数已经足够好，因此一般只需使用较小的学习率来微调这些参数，而 `fc` 中的随机初始化参数一般需要更大的学习率从头训练。PyTorch 可以方便的对模型的不同部分设置不同的学习参数，我们在下面代码中将 `fc` 的学习率设为已经预训练过的部分的 10 倍。

```
output_params = list(map(id, pretrained_net.fc.parameters()))
feature_params = filter(lambda p: id(p) not in output_params, pretrained_net.parameters())

lr = 0.01
optimizer = optim.SGD([{'params': feature_params,
                      {'params': pretrained_net.fc.parameters(), 'lr': lr * 10}],
                      lr=lr, weight_decay=0.001)
```

9.2.1.3 微调模型

我们先定义一个使用微调的训练函数 `train_fine_tuning` 以便多次调用。

```
def train_fine_tuning(net, optimizer, batch_size=128, num_epochs=5):
    train_iter = DataLoader(ImageFolder(os.path.join(data_dir, 'hotdog/train')), transform,
                           batch_size, shuffle=True)
    test_iter = DataLoader(ImageFolder(os.path.join(data_dir, 'hotdog/test')), transform,
                          batch_size)
    loss = torch.nn.CrossEntropyLoss()
    d2l.train(train_iter, test_iter, net, loss, optimizer, device, num_epochs)
```

根据前面的设置，我们将以 10 倍的学习率从头训练目标模型的输出层参数。

```
train_fine_tuning(pretrained_net, optimizer)
```

输出：

```
training on  cuda
epoch 1, loss 3.1183, train acc 0.731, test acc 0.932, time 41.4 sec
epoch 2, loss 0.6471, train acc 0.829, test acc 0.869, time 25.6 sec
epoch 3, loss 0.0964, train acc 0.920, test acc 0.910, time 24.9 sec
epoch 4, loss 0.0659, train acc 0.922, test acc 0.936, time 25.2 sec
epoch 5, loss 0.0668, train acc 0.913, test acc 0.929, time 25.0 sec
```

作为对比，我们定义一个相同的模型，但将它的所有模型参数都初始化为随机值。由于整个模型都需要从头训练，我们可以使用较大的学习率。

```
scratch_net = models.resnet18(pretrained=False, num_classes=2)
lr = 0.1
optimizer = optim.SGD(scratch_net.parameters(), lr=lr, weight_decay=0.001)
train_fine_tuning(scratch_net, optimizer)
```

输出：

```
training on  cuda
epoch 1, loss 2.6686, train acc 0.582, test acc 0.556, time 25.3 sec
epoch 2, loss 0.2434, train acc 0.797, test acc 0.776, time 25.3 sec
epoch 3, loss 0.1251, train acc 0.845, test acc 0.802, time 24.9 sec
epoch 4, loss 0.0958, train acc 0.833, test acc 0.810, time 25.0 sec
epoch 5, loss 0.0757, train acc 0.836, test acc 0.780, time 24.9 sec
```

可以看到，微调的模型因为参数初始值更好，往往在相同迭代周期下取得更高的精度。

小结

- 迁移学习将从源数据集学到的知识迁移到目标数据集上。微调是迁移学习的一种常用技术。
- 目标模型复制了源模型上除了输出层外的所有模型设计及其参数，并基于目标数据集微调这些参数。而目标模型的输出层需要从头训练。
- 一般来说，微调参数会使用较小的学习率，而从头训练输出层可以使用较大的学习率。

注：除代码外本节与原书基本相同，[原书传送门](#)

9.3 目标检测和边界框

在前面的一些章节中，我们介绍了诸多用于图像分类的模型。在图像分类任务里，我们假设图像里只有一个主体目标，并关注如何识别该目标的类别。然而，很多时候图像里有多个我们感兴趣的目标，我们不仅想知道它们的类别，还想得到它们在图像中的具体位置。在计算机视觉里，我们将这类任务称为目标检测（object detection）或物体检测。

目标检测在多个领域中被广泛使用。例如，在无人驾驶里，我们需要通过识别拍摄到的视频图像里的车辆、行人、道路和障碍的位置来规划行进线路。机器人也常通过该任务来检测感兴趣的目标。安防领域则需要检测异常目标，如歹徒或者炸弹。

在接下来的几节里，我们将介绍目标检测里的多个深度学习模型。在此之前，让我们来了解目标位置这个概念。先导入实验所需的包或模块。

```
%matplotlib inline
from PIL import Image

import sys
sys.path.append('..')
import d2lzh_pytorch as d2l
```

下面加载本节将使用的示例图像。可以看到图像左边是一只狗，右边是一只猫。它们是这张图像里的两个主要目标。

```
d2l.set_figsize()
img = Image.open('..../img/catdog.jpg')
d2l.plt.imshow(img); # 加分号只显示图
```

9.3.1 边界框

在目标检测里，我们通常使用边界框（bounding box）来描述目标位置。边界框是一个矩形框，可以由矩形左上角的 x 和 y 轴坐标与右下角的 x 和 y 轴坐标确定。我们根据上面的图的坐标信息来定义图中狗和猫的边界框。图中的坐标原点在图像的左上角，原点往右和往下分别为 x 轴和 y 轴的正方向。

```
# bbox 是 bounding box 的缩写
dog_bbox, cat_bbox = [60, 45, 378, 516], [400, 112, 655, 493]
```

我们可以在图中将边界框画出来，以检查其是否准确。画之前，我们定义一个辅助函数 `bbox_to_rect`。它将边界框表示成 matplotlib 的边界框格式。

```
def bbox_to_rect(bbox, color): # 本函数已保存在 d2lzh_pytorch 中方便以后使用
    # 将边界框 (左上 x, 左上 y, 右下 x, 右下 y) 格式转换成 matplotlib 格式:
    # ((左上 x, 左上 y), 宽, 高)
    return d2l.plt.Rectangle(
        xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],
        fill=False, edgecolor=color, linewidth=2)
```

我们将边界框加载在图像上，可以看到目标的主要轮廓基本在框内。

```
fig = d2l.plt.imshow(img)
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
```

输出：

小结

- 在目标检测里不仅需要找出图像里面所有感兴趣的目标，而且要知道它们的位置。位置一般由矩形边界框来表示。

注：除代码外本节与原书基本相同，[原书传送门](#)

9.4 锚框

目标检测算法通常会在输入图像中采样大量的区域，然后判断这些区域中是否包含我们感兴趣的目标，并调整区域边缘从而更准确地预测目标的真实边界框（ground-truth bounding box）。不同的模型使用的区域采样方法可能不同。这里我们介绍其中的一种方法：它以每个像素为中心生成多个大小和宽高比（aspect ratio）不同的边界框。这些边界框被称为锚框（anchor box）。我们将在后面基于锚框实践目标检测。

首先，导入本节需要的包或模块。这里我们新引入了 `contrib` 包，并修改了 NumPy 的打印精度。由于 `NDArray` 的打印实际调用 NumPy 的打印函数，本节打印出的 `NDArray` 中的浮点数更简洁一些。

```
%matplotlib inline
import d2lzh as d2l
from mxnet import contrib, gluon, image, nd
import numpy as np
np.set_printoptions(2)
```

9.4.1 生成多个锚框

假设输入图像高为 h ，宽为 w 。我们分别以图像的每个像素为中心生成不同形状的锚框。设大小为 $s \in (0, 1]$ 且宽高比为 $r > 0$ ，那么锚框的宽和高将分别为 $ws\sqrt{r}$ 和 hs/\sqrt{r} 。当中心位置给定时，已知宽和高的锚框是确定的。

下面我们分别设定好一组大小 s_1, \dots, s_n 和一组宽高比 r_1, \dots, r_m 。如果以每个像素为中心时使用所有的大小与宽高比的组合，输入图像将一共得到 $whnm$ 个锚框。虽然这些锚框可能覆盖了所有的真实边界框，但计算复杂度容易过高。因此，我们通常只对包含 s_1 或 r_1 的大小与宽高比的组合感兴趣，即

$$(s_1, r_1), (s_1, r_2), \dots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \dots, (s_n, r_1).$$

也就是说，以相同像素为中心的锚框的数量为 $n + m - 1$ 。对于整个输入图像，我们将一共生成 $wh(n + m - 1)$ 个锚框。

以上生成锚框的方法已实现在 `MultiBoxPrior` 函数中。指定输入、一组大小和一组宽高比，该函数将返回输入的所有锚框。

```
img = image.imread('../img/catdog.jpg').asnumpy()
h, w = img.shape[0:2]
```

```
print(h, w)
X = nd.random.uniform(shape=(1, 3, h, w)) # 构造输入数据
Y = contrib.nd.MultiBoxPrior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
```

我们看到，返回锚框变量 y 的形状为（批量大小，锚框个数，4）。将锚框变量 y 的形状变为（图像高，图像宽，以相同像素为中心的锚框个数，4）后，我们就可以通过指定像素位置来获取所有以该像素为中心的锚框了。下面的例子里我们访问以（250, 250）为中心的第一个锚框。它有 4 个元素，分别是锚框左上角的 x 和 y 轴坐标和右下角的 x 和 y 轴坐标，其中 x 和 y 轴的坐标值分别已除以图像的宽和高，因此值域均为 0 和 1 之间。

```
boxes = Y.reshape((h, w, 5, 4))  
boxes[250, 250, 0, :]
```

为了描绘图像中以某个像素为中心的所有锚框，我们先定义 `show_bboxes` 函数以便在图像上画出多个边界框。

本函数已保存在 `d2lzh` 包中方便以后使用

```
def show_bboxes(axes, bboxes, labels=None, colors=None):
    def _make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj
```

```
labels = _make_list(labels)
colors = _make_list(colors, ['b', 'g', 'r', 'm', 'c'])
for i, bbox in enumerate(bboxes):
    color = colors[i % len(colors)]
    rect = d2l.bbox_to_rect(bbox.asnumpy(), color)
    axes.add_patch(rect)
    if labels and len(labels) > i:
        text_color = 'k' if color == 'w' else 'w'
        axes.text(rect.xy[0], rect.xy[1], labels[i],
                  va='center', ha='center', fontsize=9, color=text_color,
                  bbox=dict(facecolor=color, lw=0))
```

刚刚我们看到，变量 `boxes` 中 x 和 y 轴的坐标值分别已除以图像的宽和高。在绘图时，我们需要恢复锚框的原始坐标值，并因此定义了变量 `bbox_scale`。现在，我们可以画出图像中以 (250, 250) 为中心的所有锚框了。可以看到，大小为 0.75 且宽高比为 1 的锚框较好地覆盖了图像中的狗。

```
d2l.set_figsize()
bbox_scale = nd.array((w, h, w, h))
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,
            ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
             's=0.75, r=0.5'])
```

9.4.2 交并比

我们刚刚提到某个锚框较好地覆盖了图像中的狗。如果该目标的真实边界框已知，这里的“较好”该如何量化呢？一种直观的方法是衡量锚框和真实边界框之间的相似度。我们知道，Jaccard 系数（Jaccard index）可以衡量两个集合的相似度。给定集合 \mathcal{A} 和 \mathcal{B} ，它们的 Jaccard 系数即二者交集大小除以二者并集大小：

$$J(\mathcal{A}, \mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}.$$

实际上，我们可以把边界框内的像素区域看成是像素的集合。如此一来，我们可以用两个边界框的像素集合的 Jaccard 系数衡量这两个边界框的相似度。当衡量两个边界框的相似度时，我们通常将 Jaccard 系数称为交并比（Intersection over Union，IoU），即两个边界框相交面积与相并面积之比，如图 9.2 所示。交并比的取值范围在 0 和 1 之间：0 表示两个边界框无重合像素，1 表示两个边界框相等。

交并比是两个边界框相交面积与相并面积之比

在本节的剩余部分，我们将使用交并比来衡量锚框与真实边界框以及锚框与锚框之间的相似度。

9.4.3 标注训练集的锚框

在训练集中，我们将每个锚框视为一个训练样本。为了训练目标检测模型，我们需要为每个锚框标注两类标签：一是锚框所含目标的类别，简称类别；二是真实边界框相对锚框的偏移量，简称偏移量（offset）。在目标检测时，我们首先生成多个锚框，然后为每个锚框预测类别以及偏移量，接着根据预测的偏移量调整锚框位置从而得到预测边界框，最后筛选需要输出的预测边界框。

我们知道，在目标检测的训练集中，每个图像已标注了真实边界框的位置以及所含目标的类别。在生成锚框之后，我们主要依据与锚框相似的真实边界框的位置和类别信息为锚框标注。那么，该如何为锚框分配与其相似的真实边界框呢？

假设图像中锚框分别为 A_1, A_2, \dots, A_{n_a} ，真实边界框分别为 B_1, B_2, \dots, B_{n_b} ，且 $n_a \geq n_b$ 。定义矩阵 $\mathbf{X} \in \mathbb{R}^{n_a \times n_b}$ ，其中第 i 行第 j 列的元素 x_{ij} 为锚框 A_i 与真实边界框 B_j 的交并比。首先，我们找出矩阵 \mathbf{X} 中最大元素，并将该元素的行索引与列索引分别记为 i_1, j_1 。我们为锚框 A_{i_1} 分配真实边界框 B_{j_1} 。显然，锚框 A_{i_1} 和真实边界框 B_{j_1} 在所有的“锚框—真实边界框”的配对中相似度最高。接下来，将矩阵 \mathbf{X} 中第 i_1 行和第 j_1 列上的所有元素丢弃。找出矩阵 \mathbf{X} 中剩余的最大元素，并将该元素的行索引与列索引分别记为 i_2, j_2 。我们为锚框 A_{i_2} 分配真实边界框 B_{j_2} ，再将矩阵 \mathbf{X} 中第 i_2 行和第 j_2 列上的所有元素丢弃。此时矩阵 \mathbf{X} 中已有两行两列的元素被丢弃。依此类推，直到矩阵 \mathbf{X} 中所有 n_b 列元素全部被丢弃。这个时候，我们已为 n_b 个锚框各分配了一个真实边界框。接下来，我们只遍历剩余的 $n_a - n_b$ 个锚框：给定其中的锚框 A_i ，根据矩阵 \mathbf{X} 的第 i 行找到与 A_i 交并比最大的真实边界框 B_j ，且只有当该交并比大于预先设定的阈值时，才为锚框 A_i 分配真实边界框 B_j 。

如图 9.3 (左) 所示，假设矩阵 \mathbf{X} 中最大值为 x_{23} ，我们将为锚框 A_2 分配真实边界框 B_3 。然后，丢弃矩阵中第 2 行和第 3 列的所有元素，找出剩余阴影部分的最大元素 x_{71} ，为锚框 A_7 分配真实边界框 B_1 。接着如图 9.3 (中) 所示，丢弃矩阵中第 7 行和第 1 列的所有元素，找出剩余阴影部分的最大元素 x_{54} ，为锚框 A_5 分配真实边界框 B_4 。最后如图 9.3 (右) 所示，丢弃矩阵中第 5 行和第 4 列的所有元素，找出剩余阴影部分的最大元素 x_{92} ，为锚框 A_9 分配真实边界框 B_2 。之后，我们只需遍历除去 A_2, A_5, A_7, A_9 的剩余锚框，并根据阈值判断是否为剩余锚框分配真实边界框。

为锚框分配真实边界框

现在我们可以标注锚框的类别和偏移量了。如果一个锚框 A 被分配了真实边界框 B ，将锚框 A 的类别设为 B 的类别，并根据 B 和 A 的中心坐标的相对位置以及两个框的相对大小为锚框 A 标注偏移量。由于数据集中各个框的位置和大小各异，因此这些相对位置和相对大小通常需要一些特殊变换，才能使偏移量的分布更均匀从而更容易拟合。设锚框 A 及其被分配的真实边界框 B 的中心坐标分别为 (x_a, y_a) 和 (x_b, y_b) ， A 和 B 的宽分别为 w_a 和 w_b ，高分别为 h_a 和 h_b ，一个常用的技巧是将 A 的偏移量标注为

$$\left(\frac{\frac{x_b - x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b - y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log \frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log \frac{h_b}{h_a} - \mu_h}{\sigma_h} \right),$$

其中常数的默认值为 $\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1, \sigma_w = \sigma_h = 0.2$ 。如果一个锚框没有被分配真实边界框，我们只需将该锚框的类别设为背景。类别为背景的锚

框通常被称为负类锚框，其余则被称为正类锚框。

下面演示一个具体的例子。我们为读取的图像中的猫和狗定义真实边界框，其中第一个元素为类别（0 为狗，1 为猫），剩余 4 个元素分别为左上角的 x 和 y 轴坐标以及右下角的 x 和 y 轴坐标（值域在 0 到 1 之间）。这里通过左上角和右下角的坐标构造了 5 个需要标注的锚框，分别记为 A_0, \dots, A_4 （程序中索引从 0 开始）。先画出这些锚框与真实边界框在图像中的位置。

```
ground_truth = nd.array([[0, 0.1, 0.08, 0.52, 0.92],
                         [1, 0.55, 0.2, 0.9, 0.88]]))

anchors = nd.array([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                     [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                     [0.57, 0.3, 0.92, 0.9]])
```



```
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```

我们可以通过 `contrib.nd` 模块中的 `MultiBoxTarget` 函数来为锚框标注类别和偏移量。该函数将背景类别设为 0，并令从零开始的目标类别的整数索引自加 1（1 为狗，2 为猫）。我们通过 `expand_dims` 函数为锚框和真实边界框添加样本维，并构造形状为（批量大小，包括背景的类别个数，锚框数）的任意预测结果。

```
labels = contrib.nd.MultiBoxTarget(anchors.expand_dims(axis=0),
                                    ground_truth.expand_dims(axis=0),
                                    nd.zeros((1, 3, 5)))
```

返回的结果里有 3 项，均为 `NDArray`。第三项表示为锚框标注的类别。

```
labels[2]
```

我们根据锚框与真实边界框在图像中的位置来分析这些标注的类别。首先，在所有的“锚框—真实边界框”的配对中，锚框 A_4 与猫的真实边界框的交并比最大，因此锚框 A_4 的类别标注为猫。不考虑锚框 A_4 或猫的真实边界框，在剩余的“锚框—真实边界框”的配对中，最大交并比的配对为锚框 A_1 和狗的真实边界框，因此锚框 A_1 的类别标注为狗。接下来遍历未标注的剩余 3 个锚框：与锚框 A_0 交并比最大的真实边界框的类别为狗，但交并比小于阈值（默认为 0.5），因此类别标注为背景；与锚框 A_2 交并比最大的真实边界框的类别为猫，且交并比大于阈值，因此类别标注为猫；与锚框 A_3 交并比最大的真实边界框的类别为猫，但交并比小于阈值，因此类别标注为背景。

返回值的第二项为掩码（mask）变量，形状为（批量大小，锚框个数的四倍）。掩码变量中的元素与每个锚框的 4 个偏移量一一对应。由于我们不关心对背景的检测，有关负类的偏移量不应影响目标函数。通过按元素乘法，掩码变量中的 0 可以在计算目标函数之前过滤掉负类的偏移量。

`labels[1]`

返回的第一项是为每个锚框标注的四个偏移量，其中负类锚框的偏移量标注为 0。

`labels[0]`

9.4.4 输出预测边界框

在模型预测阶段，我们先为图像生成多个锚框，并为这些锚框一一预测类别和偏移量。随后，我们根据锚框及其预测偏移量得到预测边界框。当锚框数量较多时，同一个目标上可能会输出较多相似的预测边界框。为了使结果更加简洁，我们可以移除相似的预测边界框。常用的方法叫作非极大值抑制（non-maximum suppression，NMS）。

我们来描述一下非极大值抑制的工作原理。对于一个预测边界框 B ，模型会计算各个类别的预测概率。设其中最大的预测概率为 p ，该概率所对应的类别即 B 的预测类别。我们也将 p 称为预测边界框 B 的置信度。在同一图像上，我们将预测类别非背景的预测边界框按置信度从高到低排序，得到列表 L 。从 L 中选取置信度最高的预测边界框 B_1 作为基准，将所有与 B_1 的交并比大于某阈值的非基准预测边界框从 L 中移除。这里的阈值是预先设定的超参数。此时， L 保留了置信度最高的预测边界框并移除了与其相似的其他预测边界框。接下来，从 L 中选取置信度第二高的预测边界框 B_2 作为基准，将所有与 B_2 的交并比大于某阈值的非基准预测边界框从 L 中移除。重复这一过程，直到 L 中所有的预测边界框都曾作为基准。此时 L 中任意一对预测边界框的交并比都小于阈值。最终，输出列表 L 中的所有预测边界框。

下面来看一个具体的例子。先构造 4 个锚框。简单起见，我们假设预测偏移量全是 0：预测边界框即锚框。最后，我们构造每个类别的预测概率。

```
anchors = nd.array([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                    [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
offset_preds = nd.array([0] * anchors.size)
cls_probs = nd.array([[0] * 4, # 背景的预测概率
                      [0.9, 0.8, 0.7, 0.1], # 狗的预测概率
                      [0.1, 0.2, 0.3, 0.9]]) # 猫的预测概率
```

在图像上打印预测边界框和它们的置信度。

```
fig = d2l=plt.imshow(img)
show_bboxes(fig.axes, anchors * bbox_scale,
            ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```

我们使用 contrib.nd 模块的 MultiBoxDetection 函数来执行非极大值抑制并设置阈值为 0.5。这里为 NDArray 输入都增加了样本维。我们看到，返回的结果的形状为(批量大小, 锚框个数, 6)。其中每一行的 6 个元素代表同一个预测边界框的输出信息。第一个元素是索引从 0 开始计数的预测类别(0 为狗, 1 为猫)，其中-1 表示背景或在非极大值抑制中被移除。第二个元素是预测边界框的置信度。剩余的 4 个元素分别是预测边界框左上角的 x 和 y 轴坐标以及右下角的 x 和 y 轴坐标(值域在 0 到 1 之间)。

```
output = contrib.ndarray.MultiBoxDetection(
    cls_probs.expand_dims(axis=0), offset_preds.expand_dims(axis=0),
    anchors.expand_dims(axis=0), nms_threshold=0.5)
output
```

我们移除掉类别为-1 的预测边界框，并可视化非极大值抑制保留的结果。

```
fig = d2l=plt.imshow(img)
for i in output[0].asnumpy():
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [nd.array(i[2:]) * bbox_scale], label)
```

实践中，我们可以在执行非极大值抑制前将置信度较低的预测边界框移除，从而减小非极大值抑制的计算量。我们还可以筛选非极大值抑制的输出，例如，只保留其中置信度较高的结果作为最终输出。

小结

- 以每个像素为中心，生成多个大小和宽高比不同的锚框。
- 交并比是两个边界框相交面积与相并面积之比。
- 在训练集中，为每个锚框标注两类标签：一是锚框所含目标的类别；二是真实边界框相对锚框的偏移量。
- 预测时，可以使用非极大值抑制来移除相似的预测边界框，从而令结果简洁。

练习

- 改变 `MultiBoxPrior` 函数中 `sizes` 和 `ratios` 的取值，观察生成的锚框的变化。
- 构造交并比为 0.5 的两个边界框，观察它们的重合度。
- 按本节定义的为锚框标注偏移量的方法（常数采用默认值），验证偏移量 `labels[0]` 的输出结果。
- 修改“标注训练集的锚框”与“输出预测边界框”两小节中的变量 `anchors`，结果有什么变化？

10.1 词嵌入 (word2vec)

注：个人觉得本节和下一节写得过于简洁，对于初学者来说可能比较难懂。
所以强烈推荐读一读博客[Word2Vec-知其然知其所以然](#)。

自然语言是一套用来表达含义的复杂系统。在这套系统中，词是表义的基本单元。顾名思义，词向量是用来表示词的向量，也可被认为是词的特征向量或表征。把词映射为实数域向量的技术也叫词嵌入 (word embedding)。近年来，词嵌入已逐渐成为自然语言处理的基础知识。

10.1.1 为何不采用 one-hot 向量

我们在 6.4 节（循环神经网络的从零开始实现）中使用 one-hot 向量表示词（字符为词）。回忆一下，假设词典中不同词的数量（词典大小）为 N ，每个词可以和从 0 到 $N - 1$ 的连续整数一一对应。这些与词对应的整数叫作词的索引。假设一个词的索引为 i ，为了得到该词的 one-hot 向量表示，我们创建一个全 0 的长为 N 的向量，并将其第 i 位设成 1。这样一来，每个词就表示成了一个长度为 N 的向量，可以直接被神经网络使用。

虽然 one-hot 词向量构造起来很容易，但通常并不是一个好选择。一个主要的原因是，one-hot 词向量无法准确表达不同词之间的相似度，如我们常常使用的余弦相似度。对于向量 $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ ，它们的余弦相似度是它们之间夹角的余弦值

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \in [-1, 1].$$

由于任何两个不同词的 one-hot 向量的余弦相似度都为 0，多个不同词之间的相似度难以通过 one-hot 向量准确地体现出来。

word2vec 工具的提出正是为了解决上面这个问题 [1]。它将每个词表示成一个定长的向量，并使得这些向量能较好地表达不同词之间的相似和类比关系。word2vec 工具包含了两个模型，即跳字模型 (skip-gram) [2] 和连续词袋模型 (continuous bag of words, CBOW) [3]。接下来让我们分别介绍这两个模型以及它们的训练方法。

10.1.2 跳字模型

跳字模型假设基于某个词来生成它在文本序列周围的词。举个例子，假设文本序列是 “the”“man”“loves”“his”“son”。以 “loves” 作为中心词，设背景窗口大小为 2。如图 10.1 所示，跳字模型所关心的是，给定中心词 “loves”，生成与它距离不超过 2 个词的背景词 “the”“man”“his”“son”的条件概率，即

$$P(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} \mid \text{"loves"}).$$

假设给定中心词的情况下，背景词的生成是相互独立的，那么上式可以改写成

$$P(\text{"the"} \mid \text{"loves"}) \cdot P(\text{"man"} \mid \text{"loves"}) \cdot P(\text{"his"} \mid \text{"loves"}) \cdot P(\text{"son"} \mid \text{"loves"}).$$

图 10.1 跳字模型关心给定中心词生成背景词的条件概率

在跳字模型中，每个词被表示成两个 d 维向量，用来计算条件概率。假设这个词在词典中索引为 i ，当它为中心词时向量表示为 $\mathbf{v}_i \in \mathbb{R}^d$ ，而为背景词时向量表示为 $\mathbf{u}_i \in \mathbb{R}^d$ 。设中心词 w_c 在词典中索引为 c ，背景词 w_o 在词典中索引为 o ，给定中心词生成背景词的条件概率可以通过对向量内积做 softmax 运算而得到：

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)},$$

其中词典索引集 $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}| - 1\}$ 。假设给定一个长度为 T 的文本序列，设时间步 t 的词为 $w^{(t)}$ 。假设给定中心词的情况下背景词的生成相互独立，当背景窗口大小为 m 时，跳字模型的似然函数即给定任一中心词生成所有背景词的概率

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)} \mid w^{(t)}),$$

这里小于 1 和大于 T 的时间步可以忽略。

10.1.2.1 训练跳字模型

跳字模型的参数是每个词所对应的中心词向量和背景词向量。训练中我们通过最大化似然函数来学习模型参数，即最大似然估计。这等价于最小化以下损失函数：

$$-\sum_{t=1}^T \sum_{-m \leq j \leq m, j \neq 0} \log P(w^{(t+j)} \mid w^{(t)}).$$

如果使用随机梯度下降，那么在每一次迭代里我们随机采样一个较短的子序列来计算有关该子序列的损失，然后计算梯度来更新模型参数。梯度计算的关键是条件概率的对数有关中心词向量和背景词向量的梯度。根据定义，首先看到

$$\log P(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right)$$

通过微分，我们可以得到上式中 \mathbf{v}_c 的梯度

$$\begin{aligned}
\frac{\partial \log P(w_o | w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c) \mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \\
&= \mathbf{u}_o - \sum_{j \in \mathcal{V}} \left(\frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \right) \mathbf{u}_j \\
&= \mathbf{u}_o - \sum_{j \in \mathcal{V}} P(w_j | w_c) \mathbf{u}_j.
\end{aligned}$$

它的计算需要词典中所有词以 w_c 为中心词的条件概率。有关其他词向量的梯度同理可得。

训练结束后，对于词典中的任一索引为 i 的词，我们均得到该词作为中心词和背景词的两组词向量 \mathbf{v}_i 和 \mathbf{u}_i 。在自然语言处理应用中，一般使用跳字模型的中心词向量作为词的表征向量。

10.1.3 连续词袋模型

连续词袋模型与跳字模型类似。与跳字模型最大的不同在于，连续词袋模型假设基于某中心词在文本序列前后的背景词来生成该中心词。在同样的文本序列“the”“man”“loves”“his”“son”里，以“loves”作为中心词，且背景窗口大小为 2 时，连续词袋模型关心的是，给定背景词“the”“man”“his”“son”生成中心词“loves”的条件概率（如图 10.2 所示），也就是

$$P(\text{"loves"} | \text{"the"}, \text{"man"}, \text{"his"}, \text{"son"}).$$

图 10.2 连续词袋模型关心给定背景词生成中心词的条件概率

因为连续词袋模型的背景词有多个，我们将这些背景词向量取平均，然后使用和跳字模型一样的方法来计算条件概率。设 $\mathbf{v}_i \in \mathbb{R}^d$ 和 $\mathbf{u}_i \in \mathbb{R}^d$ 分别表示词典中索引为 i 的词作为背景词和中心词的向量（注意符号的含义与跳字模型中的相反）。设中心词 w_c 在词典中索引为 c ，背景词 $w_{o_1}, \dots, w_{o_{2m}}$ 在词典中索引为 o_1, \dots, o_{2m} ，那么给定背景词生成中心词的条件概率

$$P(w_c | w_{o_1}, \dots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m} \mathbf{u}_c^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m} \mathbf{u}_i^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})\right)}.$$

为了让符号更加简单，我们记 $\mathcal{W}_o = \{w_{o_1}, \dots, w_{o_{2m}}\}$ ，且 $\bar{\mathbf{v}}_o = (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}}) / (2m)$ ，那么上式可以简写成

$$P(w_c | \mathcal{W}_o) = \frac{\exp(\mathbf{u}_c^\top \bar{\mathbf{v}}_o)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)}.$$

给定一个长度为 T 的文本序列，设时间步 t 的词为 $w^{(t)}$ ，背景窗口大小为 m 。连续词袋模型的似然函数是由背景词生成任一中心词的概率

$$\prod_{t=1}^T P(w^{(t)} \mid w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}).$$

10.1.3.1 训练连续词袋模型

训练连续词袋模型同训练跳字模型基本一致。连续词袋模型的最大似然估计等价于最小化损失函数

$$-\sum_{t=1}^T \log P(w^{(t)} \mid w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}).$$

注意到

$$\log P(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o) \right).$$

通过微分，我们可以计算出上式中条件概率的对数有关任一背景词向量 \mathbf{v}_{o_i} ($i = 1, \dots, 2m$) 的梯度

$$\frac{\partial \log P(w_c \mid \mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \bar{\mathbf{v}}_o) \mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)} \right) = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} P(w_j \mid \mathcal{W}_o) \mathbf{u}_j \right).$$

有关其他词向量的梯度同理可得。同跳字模型不一样的一点在于，我们一般使用连续词袋模型的背景词向量作为词的表征向量。

小结

- 词向量是用来表示词的向量。把词映射为实数域向量的技术也叫词嵌入。
- word2vec 包含跳字模型和连续词袋模型。跳字模型假设基于中心词来生成背景词。连续词袋模型假设基于背景词来生成中心词。

参考文献

- [1] word2vec 工具。 <https://code.google.com/archive/p/word2vec/>
- [2] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems (pp. 3111-3119).

- [3] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

注：本节与原书完全相同，[原书传送门](#)

10.2 近似训练

回忆上一节的内容。跳字模型的核心在于使用 softmax 运算得到给定中心词 w_c 来生成背景词 w_o 的条件概率

$$P(w_o | w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}.$$

该条件概率相应的对数损失

$$-\log P(w_o | w_c) = -\mathbf{u}_o^\top \mathbf{v}_c + \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right).$$

由于 softmax 运算考虑了背景词可能是词典 \mathcal{V} 中的任一词，以上损失包含了词典大小数目的项的累加。在一节中我们看到，不论是跳字模型还是连续词袋模型，由于条件概率使用了 softmax 运算，每一步的梯度计算都包含词典大小数目的项的累加。对于含几十万或上百万词的较大词典，每次的梯度计算开销可能过大。为了降低该计算复杂度，本节将介绍两种近似训练方法，即负采样（negative sampling）或层序 softmax（hierarchical softmax）。由于跳字模型和连续词袋模型类似，本节仅以跳字模型为例介绍这两种方法。

10.2.1 负采样

负采样修改了原来的目标函数。给定中心词 w_c 的一个背景窗口，我们把背景词 w_o 出现在该背景窗口看作一个事件，并将该事件的概率计算为

$$P(D = 1 | w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c),$$

其中的 σ 函数与 sigmoid 激活函数的定义相同：

$$\sigma(x) = \frac{1}{1 + \exp(-x)}.$$

我们先考虑最大化文本序列中所有该事件的联合概率来训练词向量。具体来说，给定一个长度为 T 的文本序列，设时间步 t 的词为 $w^{(t)}$ 且背景窗口大小为 m ，考虑最大化联合概率

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(D = 1 | w^{(t)}, w^{(t+j)}).$$

然而，以上模型中包含的事件仅考虑了正类样本。这导致当所有词向量相等且值为无穷大时，以上的联合概率才被最大化为 1。很明显，这样的词向量毫无意义。负采样

通过采样并添加负类样本使目标函数更有意义。设背景词 w_o 出现在中心词 w_c 的一个背景窗口为事件 P , 我们根据分布 $P(w)$ 采样 K 个未出现在该背景窗口中的词, 即噪声词。设噪声词 w_k ($k = 1, \dots, K$) 不出现在中心词 w_c 的该背景窗口为事件 N_k 。假设同时含有正类样本和负类样本的事件 P, N_1, \dots, N_K 相互独立, 负采样将以上需要最大化的仅考虑正类样本的联合概率改写为

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)} | w^{(t)}),$$

其中条件概率被近似表示为

$$P(w^{(t+j)} | w^{(t)}) = P(D = 1 | w^{(t)}, w^{(t+j)}) \prod_{k=1, w_k \sim P(w)}^K P(D = 0 | w^{(t)}, w_k).$$

设文本序列中时间步 t 的词 $w^{(t)}$ 在词典中的索引为 i_t , 噪声词 w_k 在词典中的索引为 h_k 。有关以上条件概率的对数损失为

$$\begin{aligned} -\log P(w^{(t+j)} | w^{(t)}) &= -\log P(D = 1 | w^{(t)}, w^{(t+j)}) - \sum_{k=1, w_k \sim P(w)}^K \log P(D = 0 | w^{(t)}, w_k) \\ &= -\log \sigma(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}) - \sum_{k=1, w_k \sim P(w)}^K \log (1 - \sigma(\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t})) \\ &= -\log \sigma(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}) - \sum_{k=1, w_k \sim P(w)}^K \log \sigma(-\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}). \end{aligned}$$

现在, 训练中每一步的梯度计算开销不再与词典大小相关, 而与 K 线性相关。当 K 取较小的常数时, 负采样在每一步的梯度计算开销较小。

10.2.2 层序 softmax

层序 softmax 是另一种近似训练法。它使用了二叉树这一数据结构, 树的每个叶结点代表词典 \mathcal{V} 中的每个词。

图 10.3 层序 softmax。二叉树的每个叶结点代表着词典的每个词

假设 $L(w)$ 为从二叉树的根结点到词 w 的叶结点的路径 (包括根结点和叶结点) 上的结点数。设 $n(w, j)$ 为该路径上第 j 个结点, 并设该结点的背景词向量为 $\mathbf{u}_{n(w,j)}$ 。以图 10.3 为例, $L(w_3) = 4$ 。层序 softmax 将跳字模型中的条件概率近似表示为

$$P(w_o | w_c) = \prod_{j=1}^{L(w_o)-1} \sigma([\![n(w_o, j+1) = \text{leftChild}(n(w_o, j))]\!] \cdot \mathbf{u}_{n(w_o,j)}^\top \mathbf{v}_c),$$

其中 σ 函数与 3.8 节（多层感知机）中 sigmoid 激活函数的定义相同， $\text{leftChild}(n)$ 是结点 n 的左子结点：如果判断 x 为真， $\llbracket x \rrbracket = 1$ ；反之 $\llbracket x \rrbracket = -1$ 。让我们计算图 10.3 中给定词 w_c 生成词 w_3 的条件概率。我们需要将 w_c 的词向量 \mathbf{v}_c 和根结点到 w_3 路径上的非叶结点向量一一求内积。由于在二叉树中由根结点到叶结点 w_3 的路径上需要向左、向右再向左地遍历（图 10.3 中加粗的路径），我们得到

$$P(w_3 | w_c) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_c).$$

由于 $\sigma(x) + \sigma(-x) = 1$ ，给定中心词 w_c 生成词典 \mathcal{V} 中任一词的条件概率之和为 1 这一条件也将满足：

$$\sum_{w \in \mathcal{V}} P(w | w_c) = 1.$$

此外，由于 $L(w_o) - 1$ 的数量级为 $\mathcal{O}(\log_2 |\mathcal{V}|)$ ，当词典 \mathcal{V} 很大时，层序 softmax 在训练中每一步的梯度计算开销相较于使用近似训练时大幅降低。

小结

- 负采样通过考虑同时含有正类样本和负类样本的相互独立事件来构造损失函数。其训练中每一步的梯度计算开销与采样的噪声词的个数线性相关。
- 层序 softmax 使用了二叉树，并根据根结点到叶结点的路径来构造损失函数。其训练中每一步的梯度计算开销与词典大小的对数相关。

注：本节与原书完全相同，[原书传送门](#)

10.3 word2vec 的实现

本节是对前两节内容的实践。我们以 10.1 节（词嵌入 word2vec）中的跳字模型和 10.2 节（近似训练）中的负采样为例，介绍在语料库上训练词嵌入模型的实现。我们还会介绍一些实现中的技巧，如二次采样（subsampling）。

首先导入实验所需的包或模块。

```
import collections
import math
import random
import sys
import time
import os
import numpy as np
import torch
from torch import nn
import torch.utils.data as Data

sys.path.append("..")
import d2lzh_pytorch as d2l
print(torch.__version__)
```

10.3.1 处理数据集

PTB (Penn Tree Bank) 是一个常用的小型语料库 [1]。它采样自《华尔街日报》的文章，包括训练集、验证集和测试集。我们将在 PTB 训练集上训练词嵌入模型。该数据集的每一行作为一个句子。句子中的每个词由空格隔开。

确保 `ptb.train.txt` 已经放在了文件夹`../data/ptb`下。

```
assert 'ptb.train.txt' in os.listdir("../data/ptb")

with open('../data/ptb/ptb.train.txt', 'r') as f:
    lines = f.readlines()
    # st 是 sentence 的缩写
    raw_dataset = [st.split() for st in lines]

'# sentences: %d' % len(raw_dataset) # 输出 '# sentences: 42068'
```

对于数据集的前 3 个句子，打印每个句子的词数和前 5 个词。这个数据集中句尾符为“<eos>”，生僻词全用“<unk>”表示，数字则被替换成“N”。

```
for st in raw_dataset[:3]:
    print('# tokens:', len(st), st[:5])
```

输出：

```
# tokens: 24 ['aer', 'banknote', 'berlitz', 'calloway', 'centrust']
# tokens: 15 ['pierre', '<unk>', 'N', 'years', 'old']
# tokens: 11 ['mr.', '<unk>', 'is', 'chairman', 'of']
```

10.3.1.1 建立词语索引

为了计算简单，我们只保留在数据集中至少出现 5 次的词。

```
# tk 是 token 的缩写
counter = collections.Counter([tk for st in raw_dataset for tk in st])
counter = dict(filter(lambda x: x[1] >= 5, counter.items()))
```

然后将词映射到整数索引。

```
idx_to_token = [tk for tk, _ in counter.items()]
token_to_idx = {tk: idx for idx, tk in enumerate(idx_to_token)}
dataset = [[token_to_idx[tk] for tk in st if tk in token_to_idx]
           for st in raw_dataset]
num_tokens = sum([len(st) for st in dataset])
'# tokens: %d' % num_tokens # 输出 '# tokens: 887100'
```

10.3.1.2 二次采样

文本数据中一般会出现一些高频词，如英文中的“the”“a”和“in”。通常来说，在一个背景窗口中，一个词（如“chip”）和较低频词（如“microprocessor”）同时出现比和较高频词（如“the”）同时出现对训练词嵌入模型更有益。因此，训练词嵌入模型时可以对词进行二次采样 [2]。具体来说，数据集中每个被索引词 w_i 将有一定概率被丢弃，该丢弃概率为

$$P(w_i) = \max \left(1 - \sqrt{\frac{t}{f(w_i)}}, 0 \right),$$

其中 $f(w_i)$ 是数据集中词 w_i 的个数与总词数之比，常数 t 是一个超参数（实验中设为 10^{-4} ）。可见，只有当 $f(w_i) > t$ 时，我们才有可能在二次采样中丢弃词 w_i ，并且越高频的词被丢弃的概率越大。

```
def discard(idx):
    return random.uniform(0, 1) < 1 - math.sqrt(
        1e-4 / counter[idx_to_token[idx]] * num_tokens)

subsampled_dataset = [[tk for tk in st if not discard(tk)] for st in dataset]
'# tokens: %d' % sum([len(st) for st in subsampled_dataset]) # '# tokens: 375875'
```

可以看到，二次采样后我们去掉了一半左右的词。下面比较一个词在二次采样前后出现在数据集中的次数。可见高频词“the”的采样率不足 $1/20$ 。

```
def compare_counts(token):
    return '# %s: before=%d, after=%d' % (token, sum(
        [st.count(token_to_idx[token]) for st in dataset]), sum(
        [st.count(token_to_idx[token]) for st in subsampled_dataset]))

compare_counts('the') # '# the: before=50770, after=2013'
```

但低频词“join”则完整地保留了下来。

```
compare_counts('join') # '# join: before=45, after=45'
```

10.3.1.3 提取中心词和背景词

我们将与中心词距离不超过背景窗口大小的词作为它的背景词。下面定义函数提取出所有中心词和它们的背景词。它每次在整数 1 和 `max_window_size`（最大背景窗口）之间随机均匀采样一个整数作为背景窗口大小。

```
def get_centers_and_contexts(dataset, max_window_size):
    centers, contexts = [], []
    for st in dataset:
        if len(st) < 2: # 每个句子至少要有 2 个词才可能组成一对“中心词-背景词”
            continue
        centers += st
        for center_i in range(len(st)):
```

```

window_size = random.randint(1, max_window_size)
indices = list(range(max(0, center_i - window_size),
                     min(len(st), center_i + 1 + window_size)))
indices.remove(center_i) # 将中心词排除在背景词之外
contexts.append([st[idx] for idx in indices])
return centers, contexts

```

下面我们创建一个人工数据集，其中含有词数分别为 7 和 3 的两个句子。设最大背景窗口为 2，打印所有中心词和它们的背景词。

```

tiny_dataset = [list(range(7)), list(range(7, 10))]
print('dataset', tiny_dataset)
for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):
    print('center', center, 'has contexts', context)

```

输出：

```

dataset [[0, 1, 2, 3, 4, 5, 6], [7, 8, 9]]
center 0 has contexts [1, 2]
center 1 has contexts [0, 2, 3]
center 2 has contexts [1, 3]
center 3 has contexts [2, 4]
center 4 has contexts [3, 5]
center 5 has contexts [3, 4, 6]
center 6 has contexts [4, 5]
center 7 has contexts [8]
center 8 has contexts [7, 9]
center 9 has contexts [7, 8]

```

实验中，我们设最大背景窗口大小为 5。下面提取数据集中所有的中心词及其背景词。

```
all_centers, all_contexts = get_centers_and_contexts(subsampled_dataset, 5)
```

10.3.2 负采样

我们使用负采样来进行近似训练。对于一对中心词和背景词，我们随机采样 K 个噪声词（实验中设 $K = 5$ ）。根据 word2vec 论文的建议，噪声词采样概率 $P(w)$ 设为 w 词频与总词频之比的 0.75 次方 [2]。

```

def get_negatives(all_contexts, sampling_weights, K):
    all_negatives, neg_candidates, i = [], [], 0
    population = list(range(len(sampling_weights)))
    for contexts in all_contexts:
        negatives = []
        while len(negatives) < len(contexts) * K:
            if i == len(neg_candidates):
                # 根据每个词的权重 (sampling_weights) 随机生成 k 个词的索引作为噪声词。
                # 为了高效计算，可以将 k 设得稍大一点
                i, neg_candidates = 0, random.choices(
                    population, sampling_weights, k=int(1e5))
            neg, i = neg_candidates[i], i + 1
            # 噪声词不能是背景词
            if neg not in set(contexts):
                negatives.append(neg)
        all_negatives.append(negatives)
    return all_negatives

sampling_weights = [counter[w]**0.75 for w in idx_to_token]
all_negatives = get_negatives(all_contexts, sampling_weights, 5)

```

10.3.3 读取数据

我们从数据集中提取所有中心词 `all_centers`，以及每个中心词对应的背景词 `all_contexts` 和噪声词 `all_negatives`。我们先定义一个 `Dataset` 类。

```

class MyDataset(torch.utils.data.Dataset):
    def __init__(self, centers, contexts, negatives):
        assert len(centers) == len(contexts) == len(negatives)
        self.centers = centers
        self.contexts = contexts
        self.negatives = negatives

    def __getitem__(self, index):
        return (self.centers[index], self.contexts[index], self.negatives[index])

```

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

我们将通过随机小批量来读取它们。在一个小批量数据中，第 i 个样本包括一个中心词以及它所对应的 n_i 个背景词和 m_i 个噪声词。由于每个样本的背景窗口大小可能不一样，其中背景词与噪声词个数之和 $n_i + m_i$ 也会不同。在构造小批量时，我们将每个样本的背景词和噪声词连结在一起，并添加填充项 0 直至连结后的长度相同，即长度均为 $\max_i n_i + m_i$ (`max_len` 变量)。为了避免填充项对损失函数计算的影响，我们构造了掩码变量 `masks`，其每一个元素分别与连结后的背景词和噪声词 `contexts_negatives` 中的元素一一对应。当 `contexts_negatives` 变量中的某个元素为填充项时，相同位置的掩码变量 `masks` 中的元素取 0，否则取 1。为了区分正类和负类，我们还需要将 `contexts_negatives` 变量中的背景词和噪声词区分开来。依据掩码变量的构造思路，我们只需创建与 `contexts_negatives` 变量形状相同的标签变量 `labels`，并将与背景词（正类）对应的元素设 1，其余清 0。

下面我们实现这个小批量读取函数 `batchify`。它的小批量输入 `data` 是一个长度为批量大小的列表，其中每个元素分别包含中心词 `center`、背景词 `context` 和噪声词 `negative`。该函数返回的小批量数据符合我们需要的格式，例如，包含了掩码变量。

```
def batchify(data):
    """用作 DataLoader 的参数 collate_fn: 输入是个长为 batchsize 的 list,
    list 中的每个元素都是 Dataset 类调用 __getitem__ 得到的结果
    """

    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, masks, labels = [], [], [], []
    for center, context, negative in data:
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [context + negative + [0] * (max_len - cur_len)]
        masks += [[1] * cur_len + [0] * (max_len - cur_len)]
        labels += [[1] * len(context) + [0] * (max_len - len(context))]

    return (torch.tensor(centers).view(-1, 1), torch.tensor(contexts_negatives),
            torch.tensor(masks), torch.tensor(labels))
```

我们用刚刚定义的 `batchify` 函数指定 `DataLoader` 实例中小批量的读取方式，然后打印读取的第一个批量中各个变量的形状。

```
batch_size = 512
num_workers = 0 if sys.platform.startswith('win32') else 4
```

```
dataset = MyDataset(all_centers,
                     all_contexts,
                     all_negatives)
data_iter = Data.DataLoader(dataset, batch_size, shuffle=True,
                           collate_fn=batchify,
                           num_workers=num_workers)
for batch in data_iter:
    for name, data in zip(['centers', 'contexts_negatives', 'masks',
                           'labels'], batch):
        print(name, 'shape:', data.shape)
    break
```

输出：

```
centers shape: torch.Size([512, 1])
contexts_negatives shape: torch.Size([512, 60])
masks shape: torch.Size([512, 60])
labels shape: torch.Size([512, 60])
```

10.3.4 跳字模型

我们将通过使用嵌入层和小批量乘法来实现跳字模型。它们也常常用于实现其他自然语言处理的应用。

10.3.4.1 嵌入层

获取词嵌入的层称为嵌入层，在 PyTorch 中可以通过创建 `nn.Embedding` 实例得到。嵌入层的权重是一个矩阵，其行数为词典大小 (`num_embeddings`)，列数为每个词向量的维度 (`embedding_dim`)。我们设词典大小为 20，词向量的维度为 4。

```
embed = nn.Embedding(num_embeddings=20, embedding_dim=4)
embed.weight
```

输出：

```
Parameter containing:
tensor([-0.4689,  0.2420,  0.9826, -1.3280],
```

```

[-0.6690,  1.2385, -1.7482,  0.2986],
[ 0.1193,  0.1554,  0.5038, -0.3619],
[-0.0347, -0.2806,  0.3854, -0.8600],
[-0.6479, -1.1424, -1.1920,  0.3922],
[ 0.6334, -0.0703,  0.0830, -0.4782],
[ 0.1712,  0.8098, -1.2208,  0.4169],
[-0.9925,  0.9383, -0.3808, -0.1242],
[-0.3762,  1.9276,  0.6279, -0.6391],
[-0.8518,  2.0105,  1.8484, -0.5646],
[-1.0699, -1.0822, -0.6945, -0.7321],
[ 0.4806, -0.5945,  1.0795,  0.1062],
[-1.5377,  1.0420,  0.4325,  0.1098],
[-0.8438, -1.4104, -0.9700, -0.4889],
[-1.9745, -0.3092,  0.6398, -0.4368],
[ 0.0484, -0.8516, -0.4955, -0.1363],
[-2.6301, -0.7091,  2.2116, -0.1363],
[-0.2025,  0.8037,  0.4906,  1.5929],
[-0.6745, -0.8791, -0.9220, -0.8125],
[ 0.2450,  1.9456,  0.1257, -0.3728]], requires_grad=True)

```

嵌入层的输入为词的索引。输入一个词的索引 i , 嵌入层返回权重矩阵的第 i 行作为它的词向量。下面我们将形状为 (2, 3) 的索引输入进嵌入层, 由于词向量的维度为 4, 我们得到形状为 (2, 3, 4) 的词向量。

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.long)
embed(x)
```

输出:

```

tensor([[[ -0.6690,   1.2385, -1.7482,   0.2986],
         [ 0.1193,   0.1554,  0.5038, -0.3619],
         [-0.0347, -0.2806,  0.3854, -0.8600]],

        [[[ -0.6479, -1.1424, -1.1920,  0.3922],
          [ 0.6334, -0.0703,  0.0830, -0.4782],
          [ 0.1712,  0.8098, -1.2208,  0.4169]]], grad_fn=<EmbeddingBackward>)

```

10.3.4.2 小批量乘法

我们可以使用小批量乘法运算 `bmm` 对两个小批量中的矩阵一一做乘法。假设第一个小批量中包含 n 个形状为 $a \times b$ 的矩阵 $\mathbf{X}_1, \dots, \mathbf{X}_n$, 第二个小批量中包含 n 个形状为 $b \times c$ 的矩阵 $\mathbf{Y}_1, \dots, \mathbf{Y}_n$ 。这两个小批量的矩阵乘法输出为 n 个形状为 $a \times c$ 的矩阵 $\mathbf{X}_1\mathbf{Y}_1, \dots, \mathbf{X}_n\mathbf{Y}_n$ 。因此, 给定两个形状分别为 (n, a, b) 和 (n, b, c) 的 `Tensor`, 小批量乘法输出的形状为 (n, a, c) 。

```
X = torch.ones((2, 1, 4))
Y = torch.ones((2, 4, 6))
torch.bmm(X, Y).shape
```

输出:

```
torch.Size([2, 1, 6])
```

10.3.4.3 跳字模型前向计算

在前向计算中, 跳字模型的输入包含中心词索引 `center` 以及连结的背景词与噪声词索引 `contexts_and_negatives`。其中 `center` 变量的形状为 (批量大小, 1), 而 `contexts_and_negatives` 变量的形状为 (批量大小, `max_len`)。这两个变量先通过词嵌入层分别由词索引变换为词向量, 再通过小批量乘法得到形状为 (批量大小, 1, `max_len`) 的输出。输出中的每个元素是中心词向量与背景词向量或噪声词向量的内积。

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = torch.bmm(v, u.permute(0, 2, 1))
    return pred
```

10.3.5 训练模型

在训练词嵌入模型之前, 我们需要定义模型的损失函数。

10.3.5.1 二元交叉熵损失函数

根据负采样中损失函数的定义, 我们可以使用二元交叉熵损失函数, 下面定义 `SigmoidBinaryCrossEntropyLoss`。

```

class SigmoidBinaryCrossEntropyLoss(nn.Module):
    def __init__(self): # none mean sum
        super(SigmoidBinaryCrossEntropyLoss, self).__init__()
    def forward(self, inputs, targets, mask=None):
        """
            input - Tensor shape: (batch_size, len)
            target - Tensor of the same shape as input
        """
        inputs, targets, mask = inputs.float(), targets.float(), mask.float()
        res = nn.functional.binary_cross_entropy_with_logits(inputs, targets, reduction='mean')
        return res.mean(dim=1)

loss = SigmoidBinaryCrossEntropyLoss()

```

值得一提的是，我们可以通过掩码变量指定小批量中参与损失函数计算的部分预测值和标签：当掩码为 1 时，相应位置的预测值和标签将参与损失函数的计算；当掩码为 0 时，相应位置的预测值和标签则不参与损失函数的计算。我们之前提到，掩码变量可用于避免填充项对损失函数计算的影响。

```

pred = torch.tensor([[1.5, 0.3, -1, 2], [1.1, -0.6, 2.2, 0.4]])
# 标签变量 label 中的 1 和 0 分别代表背景词和噪声词
label = torch.tensor([[1, 0, 0, 0], [1, 1, 0, 0]])
mask = torch.tensor([[1, 1, 1, 1], [1, 1, 1, 0]]) # 掩码变量
loss(pred, label, mask) * mask.shape[1] / mask.float().sum(dim=1)

```

输出：

```
tensor([0.8740, 1.2100])
```

作为比较，下面将从零开始实现二元交叉熵损失函数的计算，并根据掩码变量 `mask` 计算掩码为 1 的预测值和标签的损失。

```

def sigmd(x):
    return -math.log(1 / (1 + math.exp(-x)))

print('%.4f' % ((sigmd(1.5) + sigmd(-0.3) + sigmd(1) + sigmd(-2)) / 4)) # 注意 1-sigmoid
print('%.4f' % ((sigmd(1.1) + sigmd(-0.6) + sigmd(-2.2)) / 3))

```

输出：

```
0.8740
1.2100
```

10.3.5.2 初始化模型参数

我们分别构造中心词和背景词的嵌入层，并将超参数词向量维度 `embed_size` 设置成 100。

```
embed_size = 100
net = nn.Sequential(
    nn.Embedding(num_embeddings=len(idx_to_token), embedding_dim=embed_size),
    nn.Embedding(num_embeddings=len(idx_to_token), embedding_dim=embed_size)
)
```

10.3.5.3 定义训练函数

下面定义训练函数。由于填充项的存在，与之前的训练函数相比，损失函数的计算稍有不同。

```
def train(net, lr, num_epochs):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print("train on", device)
    net = net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    for epoch in range(num_epochs):
        start, l_sum, n = time.time(), 0.0, 0
        for batch in data_iter:
            center, context_negative, mask, label = [d.to(device) for d in batch]

            pred = skip_gram(center, context_negative, net[0], net[1])

            # 使用掩码变量 mask 来避免填充项对损失函数计算的影响
            l = (loss(pred.view(label.shape), label, mask) *
                 mask.shape[1] / mask.float().sum(dim=1).mean()) # 一个 batch 的平均
            optimizer.zero_grad()
            l.backward()
```

```

        optimizer.step()
        l_sum += l.cpu().item()
        n += 1
        print('epoch %d, loss %.2f, time %.2fs'
              % (epoch + 1, l_sum / n, time.time() - start))
    
```

现在我们就可以使用负采样训练跳字模型了。

```
train(net, 0.01, 10)
```

输出：

```

train on cpu
epoch 1, loss 1.97, time 74.53s
epoch 2, loss 0.62, time 81.85s
epoch 3, loss 0.45, time 74.49s
epoch 4, loss 0.39, time 72.04s
epoch 5, loss 0.37, time 72.21s
epoch 6, loss 0.35, time 71.81s
epoch 7, loss 0.34, time 72.00s
epoch 8, loss 0.33, time 74.45s
epoch 9, loss 0.32, time 72.08s
epoch 10, loss 0.32, time 72.05s

```

10.3.6 应用词嵌入模型

训练好词嵌入模型之后，我们可以根据两个词向量的余弦相似度表示词与词之间在语义上的相似度。可以看到，使用训练得到的词嵌入模型时，与词“chip”语义最接近的词大多与芯片有关。

```

def get_similar_tokens(query_token, k, embed):
    W = embed.weight.data
    x = W[token_to_idx[query_token]]
    # 添加的 1e-9 是为了数值稳定性
    cos = torch.matmul(W, x) / (torch.sum(W * W, dim=1) * torch.sum(x * x) + 1e-9).sqrt()
    _, topk = torch.topk(cos, k=k+1)
    topk = topk.cpu().numpy()
    for i in topk[1:]: # 除去输入词

```

```
print('cosine sim=%.3f: %s' % (cos[i], (idx_to_token[i])))

get_similar_tokens('chip', 3, net[0])
```

输出：

```
cosine sim=0.478: hard-disk
cosine sim=0.446: intel
cosine sim=0.440: drives
```

小结

- 可以使用 PyTorch 通过负采样训练跳字模型。
- 二次采样试图尽可能减轻高频词对训练词嵌入模型的影响。
- 可以将长度不同的样本填充至长度相同的小批量，并通过掩码变量区分非填充和填充，然后只令非填充参与损失函数的计算。

参考文献

- [1] Penn Tree Bank. <https://catalog.ldc.upenn.edu/LDC99T42>
- [2] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems (pp. 3111-3119).
-

注：本节除代码外与原书基本相同，[原书传送门](#)

10.4 子词嵌入 (fastText)

英语单词通常有其内部结构和形成方式。例如，我们可以从“dog”“dogs”和“dogcatcher”的字面上推测它们的关系。这些词都有同一个词根“dog”，但使用不同的后缀来改变词的含义。而且，这个关联可以推广至其他词汇。例如，“dog”和“dogs”的关系如同“cat”和“cats”的关系，“boy”和“boyfriend”的关系如同“girl”和“girlfriend”的关系。这一特点并非为英语所独有。在法语和西班牙语中，很多动词根据场景不同有40多种不同的形态，而在芬兰语中，一个名词可能有15种以上的形态。事实上，构词学 (morphology) 作为语言学的一个重要分支，研究的正是词的内部结构和形成方式。

在 word2vec 中，我们并没有直接利用构词学中的信息。无论是在跳字模型还是连续词袋模型中，我们都将形态不同的单词用不同的向量来表示。例如，“dog”和“dogs”分别用两个不同的向量表示，而模型中并未直接表达这两个向量之间的关系。鉴于此，fastText 提出了子词嵌入 (subword embedding) 的方法，从而试图将构词信息引入 word2vec 中的跳字模型 [1]。

在 fastText 中，每个中心词被表示成子词的集合。下面我们用单词“where”作为例子来了解子词是如何产生的。首先，我们在单词的首尾分别添加特殊字符“<”和“>”以区分作为前后缀的子词。然后，将单词当成一个由字符构成的序列来提取 n 元语法。例如，当 $n = 3$ 时，我们得到所有长度为 3 的子词：“<wh>”“whe”“her”“ere”“<re>”以及特殊子词“<where>”。

在 fastText 中，对于一个词 w ，我们将它所有长度在 $3 \sim 6$ 的子词和特殊子词的并集记为 \mathcal{G}_w 。那么词典则是所有词的子词集合的并集。假设词典中子词 g 的向量为 \mathbf{z}_g ，那么跳字模型中词 w 的作为中心词的向量 \mathbf{v}_w 则表示成

$$\mathbf{v}_w = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g.$$

fastText 的其余部分同跳字模型一致，不在此重复。可以看到，与跳字模型相比，fastText 中词典规模更大，造成模型参数更多，同时一个词的向量需要对所有子词向量求和，继而导致计算复杂度更高。但与此同时，较生僻的复杂单词，甚至是词典中没有的单词，可能会从同它结构类似的其他词那里获取更好的词向量表示。

小结

- fastText 提出了子词嵌入方法。它在 word2vec 中的跳字模型的基础上，将中心词向量表示成单词的子词向量之和。
- 子词嵌入利用构词上的规律，通常可以提升生僻词表示的质量。

参考文献

- [1] Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2016). Enriching word vectors with subword information. arXiv preprint arXiv:1607.04606.
-

注：本节与原书完全相同，[原书传送门](#)

10.5 全局向量的词嵌入 (GloVe)

让我们先回顾一下 word2vec 中的跳字模型。将跳字模型中使用 softmax 运算表达的条件概率 $P(w_j | w_i)$ 记作 q_{ij} , 即

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)},$$

其中 \mathbf{v}_i 和 \mathbf{u}_i 分别是索引为 i 的词 w_i 作为中心词和背景词时的向量表示, $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}| - 1\}$ 为词典索引集。

对于词 w_i , 它在数据集中可能多次出现。我们将每一次以它作为中心词的所有背景词全部汇总并保留重复元素, 记作多重集 (multiset) \mathcal{C}_i 。一个元素在多重集中的个数称为该元素的重数 (multiplicity)。举例来说, 假设词 w_i 在数据集中出现 2 次: 文本序列中以这 2 个 w_i 作为中心词的背景窗口分别包含背景词索引 2, 1, 5, 2 和 2, 3, 2, 1。那么多重集 $\mathcal{C}_i = \{1, 1, 2, 2, 2, 2, 3, 5\}$, 其中元素 1 的重数为 2, 元素 2 的重数为 4, 元素 3 和 5 的重数均为 1。将多重集 \mathcal{C}_i 中元素 j 的重数记作 x_{ij} : 它表示了整个数据集中所有以 w_i 为 中心词的背景窗口中词 w_j 的个数。那么, 跳字模型的损失函数还可以用另一种方式表达:

$$-\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} x_{ij} \log q_{ij}.$$

我们将数据集中所有以词 w_i 为 中心词的背景词的数量之和 $|\mathcal{C}_i|$ 记为 x_i , 并将以 w_i 为 中心词生成背景词 w_j 的条件概率 x_{ij}/x_i 记作 p_{ij} 。我们可以进一步改写跳字模型的损失函数为

$$-\sum_{i \in \mathcal{V}} x_i \sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}.$$

上式中, $-\sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}$ 计算的是以 w_i 为 中心词的背景词条件概率分布 p_{ij} 和模型预测的条件概率分布 q_{ij} 的交叉熵, 且损失函数使用所有以词 w_i 为 中心词的背景词的数量之和来加权。最小化上式中的损失函数会令预测的条件概率分布尽可能接近真实的条件概率分布。

然而, 作为常用损失函数的一种, 交叉熵损失函数有时并不是好的选择。一方面, 正如我们在 10.2 节 (近似训练) 中所提到的, 令模型预测 q_{ij} 成为合法概率分布的代价是它在分母中基于整个词典的累加项。这很容易带来过大的计算开销。另一方面, 词典中往往有大量生僻词, 它们在数据集中出现的次数极少。而有关大量生僻词的条件概率分布在交叉熵损失函数中的最终预测往往并不准确。

10.5.1 GloVe 模型

鉴于此，作为在 word2vec 之后提出的词嵌入模型，GloVe 模型采用了平方损失，并基于该损失对跳字模型做了 3 点改动 [1]：

1. 使用非概率分布的变量 $p'_{ij} = x_{ij}$ 和 $q'_{ij} = \exp(\mathbf{u}_j^\top \mathbf{v}_i)$ ，并对它们取对数。因此，平方损失项是 $(\log p'_{ij} - \log q'_{ij})^2 = (\mathbf{u}_j^\top \mathbf{v}_i - \log x_{ij})^2$ 。
2. 为每个词 w_i 增加两个为标量的模型参数：中心词偏差项 b_i 和背景词偏差项 c_j 。
3. 将每个损失项的权重替换成函数 $h(x_{ij})$ 。权重函数 $h(x)$ 是值域在 $[0, 1]$ 的单调递增函数。

如此一来，GloVe 模型的目标是最小化损失函数

$$\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} h(x_{ij}) (\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log x_{ij})^2.$$

其中权重函数 $h(x)$ 的一个建议选择是：当 $x < c$ 时（如 $c = 100$ ），令 $h(x) = (x/c)^\alpha$ （如 $\alpha = 0.75$ ），反之令 $h(x) = 1$ 。因为 $h(0) = 0$ ，所以对于 $x_{ij} = 0$ 的平方损失项可以直接忽略。当使用小批量随机梯度下降来训练时，每个时间步我们随机采样小批量非零 x_{ij} ，然后计算梯度来迭代模型参数。这些非零 x_{ij} 是预先基于整个数据集计算得到的，包含了数据集的全局统计信息。因此，GloVe 模型的命名取“全局向量”（Global Vectors）之意。

需要强调的是，如果词 w_i 出现在词 w_j 的背景窗口里，那么词 w_j 也会出现在词 w_i 的背景窗口里。也就是说， $x_{ij} = x_{ji}$ 。不同于 word2vec 中拟合的是非对称的条件概率 p_{ij} ，GloVe 模型拟合的是对称的 $\log x_{ij}$ 。因此，任意词的中心词向量和背景词向量在 GloVe 模型中是等价的。但由于初始化值的不同，同一个词最终学习到的两组词向量可能不同。当学习得到所有词向量以后，GloVe 模型使用中心词向量与背景词向量之和作为该词的最终词向量。

10.5.2 从条件概率比值理解 GloVe 模型

我们还可以从另外一个角度来理解 GloVe 模型。沿用本节前面的符号， $P(w_j | w_i)$ 表示数据集中以 w_i 为中心词生成背景词 w_j 的条件概率，并记作 p_{ij} 。作为源于某大型语料库的真实例子，以下列举了两组分别以“ice”（冰）和“steam”（蒸汽）为中心词的条件概率以及它们之间的比值 [1]：

$w_k =$	“solid”	“gas”	“water”	“fashion”
$p_1 = P(w_k “ice”)$	0.00019	0.000066	0.003	0.000017

$w_k =$	“solid”	“gas”	“water”	“fashion”
$p_2 = P(w_k \mid \text{“steam”})$	0.000022	0.00078	0.0022	0.000018
p_1/p_2	8.9	0.085	1.36	0.96

我们可以观察到以下现象。

- 对于与 “ice” 相关而与 “steam” 不相关的词 w_k , 如 $w_k = \text{“solid”}$ (固体), 我们期望条件概率比值较大, 如上表最后一行中的值 8.9;
- 对于与 “ice” 不相关而与 “steam” 相关的词 w_k , 如 $w_k = \text{“gas”}$ (气体), 我们期望条件概率比值较小, 如上表最后一行中的值 0.085;
- 对于与 “ice” 和 “steam” 都相关的词 w_k , 如 $w_k = \text{“water”}$ (水), 我们期望条件概率比值接近 1, 如上表最后一行中的值 1.36;
- 对于与 “ice” 和 “steam” 都不相关的词 w_k , 如 $w_k = \text{“fashion”}$ (时尚), 我们期望条件概率比值接近 1, 如上表最后一行中的值 0.96。

由此可见, 条件概率比值能比较直观地表达词与词之间的关系。我们可以构造一个词向量函数使它能有效拟合条件概率比值。我们知道, 任意一个这样的比值需要 3 个词 w_i 、 w_j 和 w_k 。以 w_i 作为中心词的条件概率比值为 p_{ij}/p_{ik} 。我们可以找一个函数, 它使用词向量来拟合这个条件概率比值

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) \approx \frac{p_{ij}}{p_{ik}}.$$

这里函数 f 可能的设计并不唯一, 我们只需考虑一种较为合理的可能性。注意到条件概率比值是一个标量, 我们可以将 f 限制为一个标量函数: $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = f((\mathbf{u}_j - \mathbf{u}_k)^\top \mathbf{v}_i)$ 。交换索引 j 和 k 后可以看到函数 f 应该满足 $f(x)f(-x) = 1$, 因此一种可能是 $f(x) = \exp(x)$, 于是

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\exp(\mathbf{u}_k^\top \mathbf{v}_i)} \approx \frac{p_{ij}}{p_{ik}}.$$

满足最右边约等号的一种可能是 $\exp(\mathbf{u}_j^\top \mathbf{v}_i) \approx \alpha p_{ij}$, 这里 α 是一个常数。考虑到 $p_{ij} = x_{ij}/x_i$, 取对数后 $\mathbf{u}_j^\top \mathbf{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$ 。我们使用额外的偏差项来拟合 $-\log \alpha + \log x_i$, 例如, 中心词偏差项 b_i 和背景词偏差项 c_j :

$$\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j \approx \log(x_{ij}).$$

对上式左右两边取平方误差并加权, 我们可以得到 GloVe 模型的损失函数。

小结

- 在有些情况下，交叉熵损失函数有劣势。GloVe 模型采用了平方损失，并通过词向量拟合预先基于整个数据集计算得到的全局统计信息。
- 任意词的中心词向量和背景词向量在 GloVe 模型中是等价的。

参考文献

- [1] Pennington, J., Socher, R., & Manning, C. (2014). Glove: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP) (pp. 1532-1543).
-

注：本节与原书完全相同，[原书传送门](#)

10.6 求近义词和类比词

在 10.3 节 (word2vec 的实现) 中, 我们在小规模数据集上训练了一个 word2vec 词嵌入模型, 并通过词向量的余弦相似度搜索近义词。实际中, 在大规模语料上预训练的词向量常常可以应用到下游自然语言处理任务中。本节将演示如何用这些预训练的词向量来求近义词和类比词。我们还将在后面两节中继续应用预训练的词向量。

10.6.1 使用预训练的词向量

基于 PyTorch 的关于自然语言处理的常用包有官方的[torchtext](#)以及第三方的[pytorch-nlp](#)等等。你可以使用 `pip` 很方便地按照它们, 例如命令行执行

```
pip install torchtext
```

详情请参见其 README。

本节我们使用 `torchtext` 进行练习。下面查看它目前提供的预训练词嵌入的名称。

```
import torch
import torchtext.vocab as vocab

vocab.pretrained_aliases.keys()
```

输出:

```
dict_keys(['charngram.100d', 'fasttext.en.300d', 'fasttext.simple.300d', 'glove.42B.300d'])
```

下面查看该 `glove` 词嵌入提供了哪些预训练的模型。每个模型的词向量维度可能不同, 或是在不同数据集上预训练得到的。

```
[key for key in vocab.pretrained_aliases.keys()
    if "glove" in key]
```

输出:

```
['glove.42B.300d',
 'glove.840B.300d',
 'glove.twitter.27B.25d',
 'glove.twitter.27B.50d',
 'glove.twitter.27B.100d',
```

```
'glove.twitter.27B.200d',
'glove.6B.50d',
'glove.6B.100d',
'glove.6B.200d',
'glove.6B.300d']
```

预训练的 GloVe 模型的命名规范大致是“模型.（数据集.）数据集词数.词向量维度”。更多信息可以参考 GloVe 和 fastText 的项目网站 [1,2]。下面我们使用基于维基百科子集预训练的 50 维 GloVe 词向量。第一次创建预训练词向量实例时会自动下载相应的词向量到 `cache` 指定文件夹（默认为`.vector_cache`），因此需要联网。

```
cache_dir = "/Users/tangshusen/Datasets/glove"
# glove = vocab.pretrained_aliases["glove.6B.50d"](cache=cache_dir)
glove = vocab.GloVe(name='6B', dim=50, cache=cache_dir) # 与上面等价
```

返回的实例主要有以下三个属性：
* `stoi`: 词到索引的字典；
* `itos`: 一个列表，索引到词的映射；
* `vectors`: 词向量。

打印词典大小。其中含有 40 万个词。

```
print(" 一共包含%d个词。" % len(glove.stoi))
```

输出：

一共包含400000个词。

我们可以通过词来获取它在词典中的索引，也可以通过索引获取词。

```
glove.stoi['beautiful'], glove.itos[3366] # (3366, 'beautiful')
```

10.6.2 应用预训练词向量

下面我们以 GloVe 模型为例，展示预训练词向量的应用。

10.6.2.1 求近义词

这里重新实现 10.3 节（word2vec 的实现）中介绍过的使用余弦相似度来搜索近义词的算法。为了在求类比词时重用其中的求 k 近邻（ k -nearest neighbors）的逻辑，我们将这部分逻辑单独封装在 `knn` 函数中。

```
def knn(W, x, k):
    # 添加的 1e-9 是为了数值稳定性
    cos = torch.matmul(W, x.view((-1,))) / (
        (torch.sum(W * W, dim=1) + 1e-9).sqrt() * torch.sum(x * x).sqrt())
    _, topk = torch.topk(cos, k=k)
    topk = topk.cpu().numpy()
    return topk, [cos[i].item() for i in topk]
```

然后，我们通过预训练词向量实例 `embed` 来搜索近义词。

```
def get_similar_tokens(query_token, k, embed):
    topk, cos = knn(embed.vectors,
                    embed.vectors[embed.stoi[query_token]], k+1)
    for i, c in zip(topk[1:], cos[1:]):  # 除去输入词
        print('cosine sim=% .3f: %s' % (c, (embed.itos[i])))
```

已创建的预训练词向量实例 `glove_6b50d` 的词典中含 40 万个词和 1 个特殊的未知词。除去输入词和未知词，我们从中搜索与“chip”语义最相近的 3 个词。

```
get_similar_tokens('chip', 3, glove)
```

输出：

```
cosine sim=0.856: chips
cosine sim=0.749: intel
cosine sim=0.749: electronics
```

接下来查找“baby”和“beautiful”的近义词。

```
get_similar_tokens('baby', 3, glove)
```

输出：

```
cosine sim=0.839: babies
cosine sim=0.800: boy
cosine sim=0.792: girl
```

```
get_similar_tokens('beautiful', 3, glove)
```

输出：

```
cosine sim=0.921: lovely
cosine sim=0.893: gorgeous
cosine sim=0.830: wonderful
```

10.6.2.2 求类比词

除了求近义词以外，我们还可以使用预训练词向量求词与词之间的类比关系。例如，“man”（男人）：“woman”（女人） :: “son”（儿子）：“daughter”（女儿）是一个类比例子：“man”之于“woman”相当于“son”之于“daughter”。求类比词问题可以定义为：对于类比关系中的 4 个词 $a : b :: c : d$ ，给定前 3 个词 a 、 b 和 c ，求 d 。设词 w 的词向量为 $\text{vec}(w)$ 。求类比词的思路是，搜索与 $\text{vec}(c) + \text{vec}(b) - \text{vec}(a)$ 的结果向量最相似的词向量。

```
def get_analogy(token_a, token_b, token_c, embed):
    vecs = [embed.vectors[embed.stoi[t]]]
    for t in [token_a, token_b, token_c]:
        x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = knn(embed.vectors, x, 1)
    return embed.itos[topk[0]]
```

验证一下“男-女”类比。

```
get_analogy('man', 'woman', 'son', glove) # 'daughter'
```

“首都-国家”类比：“beijing”（北京）之于“china”（中国）相当于“tokyo”（东京）之于什么？答案应该是“japan”（日本）。

```
get_analogy('beijing', 'china', 'tokyo', glove) # 'japan'
```

“形容词-形容词最高级”类比：“bad”（坏的）之于“worst”（最坏的）相当于“big”（大的）之于什么？答案应该是“biggest”（最大的）。

```
get_analogy('bad', 'worst', 'big', glove) # 'biggest'
```

“动词一般时-动词过去时”类比：“do”（做）之于“did”（做过）相当于“go”（去）之于什么？答案应该是“went”（去过）。

```
get_analogy('do', 'did', 'go', glove) # 'went'
```

小结

- 在大规模语料上预训练的词向量常常可以应用于下游自然语言处理任务中。
- 可以应用预训练的词向量求近义词和类比词。

参考文献

- [1] GloVe 项目网站。 <https://nlp.stanford.edu/projects/glove/>
 - [2] fastText 项目网站。 <https://fasttext.cc/>
-

注：本节除代码外与原书基本相同，[原书传送门](#)

10.7 文本情感分类：使用循环神经网络

文本分类是自然语言处理的一个常见任务，它把一段不定长的文本序列变换为文本的类别。本节关注它的一个子问题：使用文本情感分类来分析文本作者的情绪。这个问题也叫情感分析，并有着广泛的应用。例如，我们可以分析用户对产品的评论并统计用户的满意度，或者分析用户对市场行情的情绪并用以预测接下来的行情。

同搜索近义词和类比词一样，文本分类也属于词嵌入的下游应用。在本节中，我们将应用预训练的词向量和含多个隐藏层的双向循环神经网络，来判断一段不定长的文本序列中包含的是正面还是负面的情绪。

在实验开始前，导入所需的包或模块。

```
import collections
import os
import random
import tarfile
import torch
from torch import nn
import torchtext.vocab as Vocab
import torch.utils.data as Data

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l

os.environ["CUDA_VISIBLE_DEVICES"] = "0"
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

DATA_ROOT = "/S1/CSCL/tangss/Datasets"
```

10.7.1 文本情感分类数据

我们使用斯坦福的 IMDb 数据集 (Stanford's Large Movie Review Dataset) 作为文本情感分类的数据集 [1]。这个数据集分为训练和测试用的两个数据集，分别包含 25,000 条从 IMDb 下载的关于电影的评论。在每个数据集中，标签为“正面”和“负面”的评论数量相等。

10.7.1.1 读取数据

首先[下载](#)这个数据集到 DATA_ROOT 路径下，然后解压。

```
fname = os.path.join(DATA_ROOT, "aclImdb_v1.tar.gz")
if not os.path.exists(os.path.join(DATA_ROOT, "aclImdb")):
    print("从压缩包解压...")
    with tarfile.open(fname, 'r') as f:
        f.extractall(DATA_ROOT)
```

接下来，读取训练数据集和测试数据集。每个样本是一条评论及其对应的标签：1 表示“正面”，0 表示“负面”。

```
from tqdm import tqdm
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def read_imdb(folder='train', data_root="/S1/CSCL/tangss/Datasets/aclImdb"):
    data = []
    for label in ['pos', 'neg']:
        folder_name = os.path.join(data_root, folder, label)
        for file in tqdm(os.listdir(folder_name)):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '').lower()
            data.append([review, 1 if label == 'pos' else 0])
    random.shuffle(data)
    return data

train_data, test_data = read_imdb('train'), read_imdb('test')
```

10.7.1.2 预处理数据

我们需要对每条评论做分词，从而得到分好词的评论。这里定义的 get_tokenized_imdb 函数使用最简单的方法：基于空格进行分词。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def get_tokenized_imdb(data):
    """
    data: list of [string, label]
    """
    ...
```

```
def tokenizer(text):
    return [tok.lower() for tok in text.split(' ')]
return [tokenizer(review) for review, _ in data]
```

现在，我们可以根据分好词的训练数据集来创建词典了。我们在这里过滤掉了出现次数少于 5 的词。

```
# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def get_vocab_imdb(data):
    tokenized_data = get_tokenized_imdb(data)
    counter = collections.Counter([tk for st in tokenized_data for tk in st])
    return Vocab.Vocab(counter, min_freq=5)

vocab = get_vocab_imdb(train_data)
'# words in vocab:', len(vocab)
```

输出：

```
('# words in vocab:', 46151)
```

因为每条评论长度不一致所以不能直接组合成小批量，我们定义 `preprocess_imdb` 函数对每条评论进行分词，并通过词典转换成词索引，然后通过截断或者补 0 来将每条评论长度固定成 500。

```
# 本函数已保存在 d2lzh_torch 包中方便以后使用
def preprocess_imdb(data, vocab):
    max_l = 500 # 将每条评论通过截断或者补 0，使得长度变成 500

    def pad(x):
        return x[:max_l] if len(x) > max_l else x + [0] * (max_l - len(x))

    tokenized_data = get_tokenized_imdb(data)
    features = torch.tensor([pad([vocab.stoi[word] for word in words]) for words in tokenized_data])
    labels = torch.tensor([score for _, score in data])
    return features, labels
```

10.7.1.3 创建数据迭代器

现在，我们创建数据迭代器。每次迭代将返回一个小批量的数据。

```
batch_size = 64
train_set = Data.TensorDataset(*preprocess_imdb(train_data, vocab))
test_set = Data.TensorDataset(*preprocess_imdb(test_data, vocab))
train_iter = Data.DataLoader(train_set, batch_size, shuffle=True)
test_iter = Data.DataLoader(test_set, batch_size)
```

打印第一个小批量数据的形状以及训练集中小批量的个数。

```
for X, y in train_iter:
    print('X', X.shape, 'y', y.shape)
    break
'#batches:', len(train_iter)
```

输出：

```
X torch.Size([64, 500]) y torch.Size([64])
'#batches:', 391)
```

10.7.2 使用循环神经网络的模型

在这个模型中，每个词先通过嵌入层得到特征向量。然后，我们使用双向循环神经网络对特征序列进一步编码得到序列信息。最后，我们将编码的序列信息通过全连接层变换为输出。具体来说，我们可以将双向长短期记忆在最初时间步和最终时间步的隐藏状态连结，作为特征序列的表征传递给输出层分类。在下面实现的 BiRNN 类中，`Embedding` 实例即嵌入层，`LSTM` 实例即为序列编码的隐藏层，`Linear` 实例即生成分类结果的输出层。

```
class BiRNN(nn.Module):
    def __init__(self, vocab, embed_size, num_hiddens, num_layers):
        super(BiRNN, self).__init__()
        self.embedding = nn.Embedding(len(vocab), embed_size)
        # bidirectional 设为 True 即得到双向循环神经网络
        self.encoder = nn.LSTM(input_size=embed_size,
                              hidden_size=num_hiddens,
                              num_layers=num_layers,
```

```

        bidirectional=True)

# 初始时间步和最终时间步的隐藏状态作为全连接层输入
self.decoder = nn.Linear(4*num_hiddens, 2)

def forward(self, inputs):
    # inputs 的形状是 (批量大小, 词数), 因为 LSTM 需要将序列长度 (seq_len) 作为第一
    # 再提取词特征, 输出形状为 (词数, 批量大小, 词向量维度)
    embeddings = self.embedding(inputs.permute(1, 0))
    # rnn.LSTM 只传入输入 embeddings, 因此只返回最后一层的隐藏层在各时间步的隐藏状态
    # outputs 形状是 (词数, 批量大小, 2 * 隐藏单元个数)
    outputs, _ = self.encoder(embeddings) # output, (h, c)
    # 连结初始时间步和最终时间步的隐藏状态作为全连接层输入。它的形状为
    # (批量大小, 4 * 隐藏单元个数)。
    encoding = torch.cat((outputs[0], outputs[-1]), -1)
    outs = self.decoder(encoding)
    return outs

```

创建一个含两个隐藏层的双向循环神经网络。

```

embed_size, num_hiddens, num_layers = 100, 100, 2
net = BiRNN(vocab, embed_size, num_hiddens, num_layers)

```

10.7.2.1 加载预训练的词向量

由于情感分类的训练数据集并不是很大，为应对过拟合，我们将直接使用在更大规模语料上预训练的词向量作为每个词的特征向量。这里，我们为词典 `vocab` 中的每个词加载 100 维的 GloVe 词向量。

```
glove_vocab = Vocab.GloVe(name='6B', dim=100, cache=os.path.join(DATA_ROOT, "glove"))
```

然后，我们将用这些词向量作为评论中每个词的特征向量。注意，预训练词向量的维度需要与创建的模型中的嵌入层输出大小 `embed_size` 一致。此外，在训练中我们不再更新这些词向量。

```

# 本函数已保存在 d2lzh_torch 包中方便以后使用
def load_pretrained_embedding(words, pretrained_vocab):
    """从预训练好的 vocab 中提取出 words 对应的词向量"""
    embed = torch.zeros(len(words), pretrained_vocab.vectors[0].shape[0]) # 初始化为

```

```

oov_count = 0 # out of vocabulary
for i, word in enumerate(words):
    try:
        idx = pretrained_vocab.stoi[word]
        embed[i, :] = pretrained_vocab.vectors[idx]
    except KeyError:
        oov_count += 0
if oov_count > 0:
    print("There are %d oov words.")
return embed

net.embedding.weight.data.copy_(
    load_pretrained_embedding(vocab.itos, glove_vocab))
net.embedding.weight.requires_grad = False # 直接加载预训练好的，所以不需要更新它

```

10.7.2.2 训练并评价模型

这时候就可以开始训练模型了。

```

lr, num_epochs = 0.01, 5
# 要过滤掉不计算梯度的 embedding 参数
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, net.parameters()), lr=lr)
loss = nn.CrossEntropyLoss()
d2l.train(train_iter, test_iter, net, loss, optimizer, device, num_epochs)

```

输出：

```

training on cuda
epoch 1, loss 0.5759, train acc 0.666, test acc 0.832, time 250.8 sec
epoch 2, loss 0.1785, train acc 0.842, test acc 0.852, time 253.3 sec
epoch 3, loss 0.1042, train acc 0.866, test acc 0.856, time 253.7 sec
epoch 4, loss 0.0682, train acc 0.888, test acc 0.868, time 254.2 sec
epoch 5, loss 0.0483, train acc 0.901, test acc 0.862, time 251.4 sec

```

最后，定义预测函数。

```

# 本函数已保存在 d2lzh_pytorch 包中方便以后使用
def predict_sentiment(net, vocab, sentence):

```

```
"""sentence 是词语的列表"""
device = list(net.parameters())[0].device
sentence = torch.tensor([vocab.stoi[word] for word in sentence], device=device)
label = torch.argmax(net(sentence.view((1, -1))), dim=1)
return 'positive' if label.item() == 1 else 'negative'
```

下面使用训练好的模型对两个简单句子的情感进行分类。

```
predict_sentiment(net, vocab, ['this', 'movie', 'is', 'so', 'great']) # positive
predict_sentiment(net, vocab, ['this', 'movie', 'is', 'so', 'bad']) # negative
```

小结

- 文本分类把一段不定长的文本序列变换为文本的类别。它属于词嵌入的下游应用。
- 可以应用预训练的词向量和循环神经网络对文本的情感进行分类。

参考文献

[1] Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011, June). Learning word vectors for sentiment analysis. In Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1 (pp. 142-150). Association for Computational Linguistics.

注：本节除代码外与原书基本相同，[原书传送门](#)

10.8 文本情感分类：使用卷积神经网络（textCNN）

在“卷积神经网络”一章中我们探究了如何使用二维卷积神经网络来处理二维图像数据。在之前的语言模型和文本分类任务中，我们将文本数据看作是只有一个维度的时间序列，并很自然地使用循环神经网络来表征这样的数据。其实，我们也可以将文本当作一维图像，从而可以用一维卷积神经网络来捕捉临近词之间的关联。本节将介绍将卷积神经网络应用到文本分析的开创性工作之一：textCNN [1]。

首先导入实验所需的包和模块。

```
import os
import torch
from torch import nn
import torchtext.vocab as Vocab
import torch.utils.data as Data
import torch.nn.functional as F

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l

os.environ["CUDA_VISIBLE_DEVICES"] = "0"
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

DATA_ROOT = "/S1/CSCL/tangss/Datasets"
```

10.8.1 一维卷积层

在介绍模型前我们先来解释一维卷积层的工作原理。与二维卷积层一样，一维卷积层使用一维的互相关运算。在一维互相关运算中，卷积窗口从输入数组的最左方开始，按从左往右的顺序，依次在输入数组上滑动。当卷积窗口滑动到某一位置时，窗口中的输入子数组与核数组按元素相乘并求和，得到输出数组中相应位置的元素。如图 10.4 所示，输入是一个宽为 7 的一维数组，核数组的宽为 2。可以看到输出的宽度为 $7 - 2 + 1 = 6$ ，且第一个元素是由输入的最左边的宽为 2 的子数组与核数组按元素相乘后再相加得到的： $0 \times 1 + 1 \times 2 = 2$ 。

图 10.4 一维互相关运算

下面我们将一维互相关运算实现在 `corr1d` 函数里。它接受输入数组 `X` 和核数组 `K`，并输出数组 `Y`。

```
def corr1d(X, K):
    w = K.shape[0]
    Y = torch.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i:i + w] * K).sum()
    return Y
```

让我们复现图 10.4 中一维互相关运算的结果。

```
X, K = torch.tensor([0, 1, 2, 3, 4, 5, 6]), torch.tensor([1, 2])
corr1d(X, K)
```

输出：

```
tensor([ 2.,  5.,  8., 11., 14., 17.])
```

多输入通道的一维互相关运算也与多输入通道的二维互相关运算类似：在每个通道上，将核与相应的输入做一维互相关运算，并将通道之间的结果相加得到输出结果。图 10.5 展示了含 3 个输入通道的一维互相关运算，其中阴影部分为第一个输出元素及其计算所使用的输入和核数组元素： $0 \times 1 + 1 \times 2 + 1 \times 3 + 2 \times 4 + 2 \times (-1) + 3 \times (-3) = 2$ 。

图 10.5 含 3 个输入通道的一维互相关运算

让我们复现图 10.5 中多输入通道的一维互相关运算的结果。

```
def corr1d_multi_in(X, K):
    # 首先沿着 X 和 K 的第 0 维（通道维）遍历并计算一维互相关结果。然后将所有结果堆叠起来
    return torch.stack([corr1d(x, k) for x, k in zip(X, K)]).sum(dim=0)
```

```
X = torch.tensor([[0, 1, 2, 3, 4, 5, 6],
                  [1, 2, 3, 4, 5, 6, 7],
                  [2, 3, 4, 5, 6, 7, 8]])
K = torch.tensor([[1, 2], [3, 4], [-1, -3]])
corr1d_multi_in(X, K)
```

输出：

```
tensor([ 2.,  8., 14., 20., 26., 32.])
```

由二维互相关运算的定义可知，多输入通道的一维互相关运算可以看作单输入通道的二维互相关运算。如图 10.6 所示，我们也可以将图 10.5 中多输入通道的一维互相关运算以等价的单输入通道的二维互相关运算呈现。这里核的高等于输入的高。图 10.6 中的阴影部分为第一个输出元素及其计算所使用的输入和核数组元素： $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$ 。

图 10.6 单输入通道的二维互相关运算

图 10.4 和图 10.5 中的输出都只有一个通道。我们在 5.3 节（多输入通道和多输出通道）一节中介绍了如何在二维卷积层中指定多个输出通道。类似地，我们也可以在一维卷积层指定多个输出通道，从而拓展卷积层中的模型参数。

10.8.2 时序最大池化层

类似地，我们有一维池化层。textCNN 中使用的时序最大池化（max-over-time pooling）层实际上对应一维全局最大池化层：假设输入包含多个通道，各通道由不同时间步上的数值组成，各通道的输出即该通道所有时间步中最大的数值。因此，时序最大池化层的输入在各个通道上的时间步数可以不同。

为提升计算性能，我们常常将不同长度的时序样本组成一个小批量，并通过在较短序列后附加特殊字符（如 0）令批量中各时序样本长度相同。这些人为添加的特殊字符当然是无意义的。由于时序最大池化的主要目的是抓取时序中最重要的特征，它通常能使模型不受人为添加字符的影响。

由于 PyTorch 没有自带全局的最大池化层，所以类似 5.8 节我们可以通过普通的池化来实现全局池化。

```
class GlobalMaxPool1d(nn.Module):
    def __init__(self):
        super(GlobalMaxPool1d, self).__init__()
    def forward(self, x):
        # x shape: (batch_size, channel, seq_len)
        # return shape: (batch_size, channel, 1)
        return F.max_pool1d(x, kernel_size=x.shape[2])
```

10.8.3 读取和预处理 IMDb 数据集

我们依然使用和上一节中相同的 IMDb 数据集做情感分析。以下读取和预处理数据集的步骤与上一节中的相同。

```
batch_size = 64
train_data = d2l.read_imdb('train', data_root=os.path.join(DATA_ROOT, "aclImdb"))
```

```

test_data = d2l.read_imdb('test', data_root=os.path.join(DATA_ROOT, "aclImdb"))
vocab = d2l.get_vocab_imdb(train_data)
train_set = Data.TensorDataset(*d2l.preprocess_imdb(train_data, vocab))
test_set = Data.TensorDataset(*d2l.preprocess_imdb(test_data, vocab))
train_iter = Data.DataLoader(train_set, batch_size, shuffle=True)
test_iter = Data.DataLoader(test_set, batch_size)

```

10.8.4 textCNN 模型

textCNN 模型主要使用了一维卷积层和时序最大池化层。假设输入的文本序列由 n 个词组成，每个词用 d 维的词向量表示。那么输入样本的宽为 n ，高为 1，输入通道数为 d 。textCNN 的计算主要分为以下几步。

1. 定义多个一维卷积核，并使用这些卷积核对输入分别做卷积计算。宽度不同的卷积核可能会捕捉到不同个数的相邻词的相关性。
2. 对输出的所有通道分别做时序最大池化，再将这些通道的池化输出值连结为向量。
3. 通过全连接层将连结后的向量变换为有关各类别的输出。这一步可以使用丢弃层应对过拟合。

图 10.7 用一个例子解释了 textCNN 的设计。这里的输入是一个有 11 个词的句子，每个词用 6 维词向量表示。因此输入序列的宽为 11，输入通道数为 6。给定 2 个一维卷积核，核宽分别为 2 和 4，输出通道数分别设为 4 和 5。因此，一维卷积计算后，4 个输出通道的宽为 $11 - 2 + 1 = 10$ ，而其他 5 个通道的宽为 $11 - 4 + 1 = 8$ 。尽管每个通道的宽不同，我们依然可以对各个通道做时序最大池化，并将 9 个通道的池化输出连结成一个 9 维向量。最终，使用全连接将 9 维向量变换为 2 维输出，即正面情感和负面情感的预测。

图 10.7 textCNN 的设计

下面我们来实现 textCNN 模型。与上一节相比，除了用一维卷积层替换循环神经网络外，这里我们还使用了两个嵌入层，一个的权重固定，另一个则参与训练。

```

class TextCNN(nn.Module):
    def __init__(self, vocab, embed_size, kernel_sizes, num_channels):
        super(TextCNN, self).__init__()
        self.embedding = nn.Embedding(len(vocab), embed_size)
        # 不参与训练的嵌入层
        self.constant_embedding = nn.Embedding(len(vocab), embed_size)
        self.dropout = nn.Dropout(0.5)

```

```

    self.decoder = nn.Linear(sum(num_channels), 2)
    # 时序最大池化层没有权重，所以可以共用一个实例
    self.pool = GlobalMaxPool1d()
    self.convs = nn.ModuleList() # 创建多个一维卷积层
    for c, k in zip(num_channels, kernel_sizes):
        self.convs.append(nn.Conv1d(in_channels = 2*embed_size,
                                   out_channels = c,
                                   kernel_size = k))

def forward(self, inputs):
    # 将两个形状是 (批量大小, 词数, 词向量维度) 的嵌入层的输出按词向量连结
    embeddings = torch.cat((
        self.embedding(inputs),
        self.constant_embedding(inputs)), dim=2) # (batch, seq_len, 2*embed_size)
    # 根据 Conv1D 要求的输入格式，将词向量维，即一维卷积层的通道维（即词向量那一维）,
    embeddings = embeddings.permute(0, 2, 1)
    # 对于每个一维卷积层，在时序最大池化后会得到一个形状为 (批量大小, 通道大小, 1) 的
    # Tensor。使用 flatten 函数去掉最后一维，然后在通道维上连结
    encoding = torch.cat([self.pool(F.relu(conv(embeddings))).squeeze(-1) for conv in self.convs])
    # 应用丢弃法后使用全连接层得到输出
    outputs = self.decoder(self.dropout(encoding))
    return outputs

```

创建一个 TextCNN 实例。它有 3 个卷积层，它们的核宽分别为 3、4 和 5，输出通道数均为 100。

```

embed_size, kernel_sizes, num_channels = 100, [3, 4, 5], [100, 100, 100]
net = TextCNN(vocab, embed_size, kernel_sizes, num_channels)

```

10.8.4.1 加载预训练的词向量

同上一节一样，加载预训练的 100 维 GloVe 词向量，并分别初始化嵌入层 embedding 和 constant_embedding，前者参与训练，而后者权重固定。

```

glove_vocab = Vocab.GloVe(name='6B', dim=100,
                           cache=os.path.join(DATA_ROOT, "glove"))
net.embedding.weight.data.copy_()

```

```
d2l.load_pretrained_embedding(vocab.itos, glove_vocab))
net.constant_embedding.weight.data.copy_(
    d2l.load_pretrained_embedding(vocab.itos, glove_vocab))
net.constant_embedding.weight.requires_grad = False
```

10.8.4.2 训练并评价模型

现在就可以训练模型了。

```
lr, num_epochs = 0.001, 5
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, net.parameters()), lr=lr)
loss = nn.CrossEntropyLoss()
d2l.train(train_iter, test_iter, net, loss, optimizer, device, num_epochs)
```

输出：

```
training on  cuda
epoch 1, loss 0.4858, train acc 0.758, test acc 0.832, time 42.8 sec
epoch 2, loss 0.1598, train acc 0.863, test acc 0.868, time 42.3 sec
epoch 3, loss 0.0694, train acc 0.917, test acc 0.876, time 42.3 sec
epoch 4, loss 0.0301, train acc 0.956, test acc 0.871, time 42.4 sec
epoch 5, loss 0.0131, train acc 0.979, test acc 0.865, time 42.3 sec
```

下面，我们使用训练好的模型对两个简单句子的情感进行分类。

```
d2l.predict_sentiment(net, vocab, ['this', 'movie', 'is', 'so', 'great']) # positive
d2l.predict_sentiment(net, vocab, ['this', 'movie', 'is', 'so', 'bad']) # negative
```

小结

- 可以使用一维卷积来表征时序数据。
- 多输入通道的一维互相关运算可以看作单输入通道的二维互相关运算。
- 时序最大池化层的输入在各个通道上的时间步数可以不同。
- textCNN 主要使用了一维卷积层和时序最大池化层。

参考文献

- [1] Kim, Y. (2014). Convolutional neural networks for sentence classification. arXiv preprint arXiv:1408.5882.
-

注：本节除代码外与原书基本相同，[原书传送门](#)

10.9 编码器—解码器 (seq2seq)

我们已经在前两节中表征并变换成了不定长的输入序列。但在自然语言处理的很多应用中，输入和输出都可以是不定长序列。以机器翻译为例，输入可以是一段不定长的英语文本序列，输出可以是一段不定长的法语文本序列，例如

英语输入：“They”、“are”、“watching”、“.”

法语输出：“Ils”、“regardent”、“.”

当输入和输出都是不定长序列时，我们可以使用编码器—解码器 (encoder-decoder) [1] 或者 seq2seq 模型 [2]。这两个模型本质上都用到了两个循环神经网络，分别叫做编码器和解码器。编码器用来分析输入序列，解码器用来生成输出序列。

图 10.8 描述了使用编码器—解码器将上述英语句子翻译成法语句子的一种方法。在训练数据集中，我们可以在每个句子后附上特殊符号 “<eos>” (end of sequence) 以表示序列的终止。编码器每个时间步的输入依次为英语句子中的单词、标点和特殊符号 “<eos>”。图 10.8 中使用了编码器在最终时间步的隐藏状态作为输入句子的表征或编码信息。解码器在各个时间步中使用输入句子的编码信息和上个时间步的输出以及隐藏状态作为输入。我们希望解码器在各个时间步能正确依次输出翻译后的法语单词、标点和特殊符号 “<eos>”。需要注意的是，解码器在最初时间步的输入用到了一个表示序列开始的特殊符号 “<bos>” (beginning of sequence)。

图 10.8 使用编码器—解码器将句子由英语翻译成法语。编码器和解码器分别为循环神经网络

接下来，我们分别介绍编码器和解码器的定义。

10.9.1 编码器

编码器的作用是把一个不定长的输入序列转换成一个定长的背景变量 \mathbf{c} ，并在该背景变量中编码输入序列信息。常用的编码器是循环神经网络。

让我们考虑批量大小为 1 的时序数据样本。假设输入序列是 x_1, \dots, x_T ，例如 x_i 是输入句子中的第 i 个词。在时间步 t ，循环神经网络将输入 x_t 的特征向量 \mathbf{x}_t 和上个时间步的隐藏状态 \mathbf{h}_{t-1} 变换为当前时间步的隐藏状态 \mathbf{h}_t 。我们可以用函数 f 表达循环神经网络隐藏层的变换：

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}).$$

接下来，编码器通过自定义函数 q 将各个时间步的隐藏状态变换为背景变量

$$\mathbf{c} = q(\mathbf{h}_1, \dots, \mathbf{h}_T).$$

例如，当选择 $q(\mathbf{h}_1, \dots, \mathbf{h}_T) = \mathbf{h}_T$ 时，背景变量是输入序列最终时间步的隐藏状态 \mathbf{h}_T 。

以上描述的编码器是一个单向的循环神经网络，每个时间步的隐藏状态只取决于该时间步及之前的输入子序列。我们也可以使用双向循环神经网络构造编码器。在这种情况下，编码器每个时间步的隐藏状态同时取决于该时间步之前和之后的子序列（包括当前时间步的输入），并编码了整个序列的信息。

10.9.2 解码器

刚刚已经介绍，编码器输出的背景变量 \mathbf{c} 编码了整个输入序列 x_1, \dots, x_T 的信息。给定训练样本中的输出序列 $y_1, y_2, \dots, y_{T'}$ ，对每个时间步 t' （符号与输入序列或编码器的时间步 t 有区别），解码器输出 $y_{t'}$ 的条件概率将基于之前的输出序列 $y_1, \dots, y_{t'-1}$ 和背景变量 \mathbf{c} ，即 $P(y_{t'} | y_1, \dots, y_{t'-1}, \mathbf{c})$ 。

为此，我们可以使用另一个循环神经网络作为解码器。在输出序列的时间步 t' ，解码器将上一时间步的输出 $y_{t'-1}$ 以及背景变量 \mathbf{c} 作为输入，并将它们与上一时间步的隐藏状态 $\mathbf{s}_{t'-1}$ 变换为当前时间步的隐藏状态 $\mathbf{s}_{t'}$ 。因此，我们可以用函数 g 表达解码器隐藏层的变换：

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}).$$

有了解码器的隐藏状态后，我们可以使用自定义的输出层和 softmax 运算来计算 $P(y_{t'} | y_1, \dots, y_{t'-1}, \mathbf{c})$ ，例如，基于当前时间步的解码器隐藏状态 $\mathbf{s}_{t'}$ 、上一时间步的输出 $y_{t'-1}$ 以及背景变量 \mathbf{c} 来计算当前时间步输出 $y_{t'}$ 的概率分布。

10.9.3 训练模型

根据最大似然估计，我们可以最大化输出序列基于输入序列的条件概率

$$\begin{aligned} P(y_1, \dots, y_{T'} | x_1, \dots, x_T) &= \prod_{t'=1}^{T'} P(y_{t'} | y_1, \dots, y_{t'-1}, x_1, \dots, x_T) \\ &= \prod_{t'=1}^{T'} P(y_{t'} | y_1, \dots, y_{t'-1}, \mathbf{c}), \end{aligned}$$

并得到该输出序列的损失

$$-\log P(y_1, \dots, y_{T'} | x_1, \dots, x_T) = -\sum_{t'=1}^{T'} \log P(y_{t'} | y_1, \dots, y_{t'-1}, \mathbf{c}),$$

在模型训练中，所有输出序列损失的均值通常作为需要最小化的损失函数。在图 10.8 所描述的模型预测中，我们需要将解码器在上一个时间步的输出作为当前时间步的输入。与此不同，在训练中我们也可以将标签序列（训练集的真实输出序列）在上一个时间步的标签作为解码器在当前时间步的输入。这叫作强制教学（teacher forcing）。

小结

- 编码器-解码器 (seq2seq) 可以输入并输出不定长的序列。
- 编码器—解码器使用了两个循环神经网络。
- 在编码器—解码器的训练中，可以采用强制教学。

参考文献

- [1] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.
- [2] Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Advances in neural information processing systems (pp. 3104-3112).

注：本节与原书基本相同，[原书传送门](#)

10.10 束搜索

上一节介绍了如何训练输入和输出均为不定长序列的编码器—解码器。本节我们介绍如何使用编码器—解码器来预测不定长的序列。

上一节里已经提到，在准备训练数据集时，我们通常会在样本的输入序列和输出序列后面分别附上一个特殊符号“`<eos>`”表示序列的终止。我们在接下来的讨论中也将沿用上一节的全部数学符号。为了便于讨论，假设解码器的输出是一段文本序列。设输出文本词典 \mathcal{Y} （包含特殊符号“`<eos>`”）的大小为 $|\mathcal{Y}|$ ，输出序列的最大长度为 T' 。所有可能的输出序列一共有 $\mathcal{O}(|\mathcal{Y}|^{T'})$ 种。这些输出序列中所有特殊符号“`<eos>`”后面的子序列将被舍弃。

10.10.1 贪婪搜索

让我们先来看一个简单的解决方案：贪婪搜索（greedy search）。对于输出序列任一时间步 t' ，我们从 $|\mathcal{Y}|$ 个词中搜索出条件概率最大的词

$$y_{t'} = \operatorname{argmax}_{y \in \mathcal{Y}} P(y | y_1, \dots, y_{t'-1}, c)$$

作为输出。一旦搜索出“`<eos>`”符号，或者输出序列长度已经达到了最大长度 T' ，便完成输出。

我们在描述解码器时提到，基于输入序列生成输出序列的条件概率是 $\prod_{t'=1}^{T'} P(y_{t'} | y_1, \dots, y_{t'-1}, c)$ 。我们将该条件概率最大的输出序列称为最优输出序列。而贪婪搜索的主要问题是不能保证得到最优输出序列。

下面来看一个例子。假设输出词典里面有“`A`”“`B`”“`C`”和“`<eos>`”这 4 个词。图 10.9 中每个时间步下的 4 个数字分别代表了该时间步生成“`A`”“`B`”“`C`”和“`<eos>`”这 4 个词的条件概率。在每个时间步，贪婪搜索选取条件概率最大的词。因此，图 10.9 中将生成输出序列“`A`”“`B`”“`C`”“`<eos>`”。该输出序列的条件概率是 $0.5 \times 0.4 \times 0.4 \times 0.6 = 0.048$ 。

图 10.9 在每个时间步，贪婪搜索选取条件概率最大的词

接下来，观察图 10.10 演示的例子。与图 10.9 中不同，图 10.10 在时间步 2 中选取了条件概率第二大的词“`C`”。由于时间步 3 所基于的时间步 1 和 2 的输出子序列由图 10.9 中的“`A`”“`B`”变为了图 10.10 中的“`A`”“`C`”，图 10.10 中时间步 3 生成各个词的条件概率发生了变化。我们选取条件概率最大的词“`B`”。此时时间步 4 所基于的前 3 个时间步的输出子序列为“`A`”“`C`”“`B`”，与图 10.9 中的“`A`”“`B`”“`C`”不同。因此，图 10.10 中时间步 4 生成各个词的条件概率也与图 10.9 中的不同。我们发现，此时的输出序列“`A`”“`C`”“`B`”“`<eos>`”的条件概率是 $0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$ ，大于贪婪搜索得到的输出序列的条件概率。因此，贪婪搜索得到的输出序列“`A`”“`B`”“`C`”“`<eos>`”并非最优

输出序列。

图 10.10 在时间步 2 选取条件概率第二大的词 “C”

10.10.2 穷举搜索

如果目标是得到最优输出序列，我们可以考虑穷举搜索（exhaustive search）：穷举所有可能的输出序列，输出条件概率最大的序列。

虽然穷举搜索可以得到最优输出序列，但它的计算开销 $\mathcal{O}(|\mathcal{Y}|^{T'})$ 很容易过大。例如，当 $|\mathcal{Y}| = 10000$ 且 $T' = 10$ 时，我们将评估 $10000^{10} = 10^{40}$ 个序列：这几乎不可能完成。而贪婪搜索的计算开销是 $\mathcal{O}(|\mathcal{Y}| T')$ ，通常显著小于穷举搜索的计算开销。例如，当 $|\mathcal{Y}| = 10000$ 且 $T' = 10$ 时，我们只需评估 $10000 \times 10 = 10^5$ 个序列。

10.10.3 束搜索

束搜索（beam search）是对贪婪搜索的一个改进算法。它有一个束宽（beam size）超参数。我们将它设为 k 。在时间步 1 时，选取当前时间步条件概率最大的 k 个词，分别组成 k 个候选输出序列的首词。在之后的每个时间步，基于上个时间步的 k 个候选输出序列，从 $k |\mathcal{Y}|$ 个可能的输出序列中选取条件概率最大的 k 个，作为该时间步的候选输出序列。最终，我们从各个时间步的候选输出序列中筛选出包含特殊符号“<eos>”的序列，并将它们中所有特殊符号“<eos>”后面的子序列舍弃，得到最终候选输出序列的集合。

图 10.11 束搜索的过程。束宽为 2，输出序列最大长度为 3。候选输出序列有 A、C、AB、CE、ABD 和 CED

图 10.11 通过一个例子演示了束搜索的过程。假设输出序列的词典中只包含 5 个元素，即 $\mathcal{Y} = \{A, B, C, D, E\}$ ，且其中一个为特殊符号“<eos>”。设束搜索的束宽等于 2，输出序列最大长度为 3。在输出序列的时间步 1 时，假设条件概率 $P(y_1 | \mathbf{c})$ 最大的 2 个词为 A 和 C 。我们在时间步 2 时将对所有的 $y_2 \in \mathcal{Y}$ 都分别计算 $P(y_2 | A, \mathbf{c})$ 和 $P(y_2 | C, \mathbf{c})$ ，并从计算出的 10 个条件概率中取最大的 2 个，假设为 $P(B | A, \mathbf{c})$ 和 $P(E | C, \mathbf{c})$ 。那么，我们在时间步 3 时将对所有的 $y_3 \in \mathcal{Y}$ 都分别计算 $P(y_3 | A, B, \mathbf{c})$ 和 $P(y_3 | C, E, \mathbf{c})$ ，并从计算出的 10 个条件概率中取最大的 2 个，假设为 $P(D | A, B, \mathbf{c})$ 和 $P(D | C, E, \mathbf{c})$ 。如此一来，我们得到 6 个候选输出序列：(1) A ; (2) C ; (3) A, B ; (4) C, E ; (5) A, B, D 和 (6) C, E, D 。接下来，我们将根据这 6 个序列得出最终候选输出序列的集合。

在最终候选输出序列的集合中，我们取以下分数最高的序列作为输出序列：

$$\frac{1}{L^\alpha} \log P(y_1, \dots, y_L) = \frac{1}{L^\alpha} \sum_{t'=1}^L \log P(y_{t'} | y_1, \dots, y_{t'-1}, \mathbf{c}),$$

其中 L 为最终候选序列长度, α 一般可选为 0.75。分母上的 L^α 是为了惩罚较长序列在以上分数中较多的对数相加项。分析可知, 束搜索的计算开销为 $\mathcal{O}(k|\mathcal{Y}|T')$ 。这介于贪婪搜索和穷举搜索的计算开销之间。此外, 贪婪搜索可看作是束宽为 1 的束搜索。束搜索通过灵活的束宽 k 来权衡计算开销和搜索质量。

小结

- 预测不定长序列的方法包括贪婪搜索、穷举搜索和束搜索。
 - 束搜索通过灵活的束宽来权衡计算开销和搜索质量。
-

注: 本节与原书基本相同, [原书传送门](#)

10.11 注意力机制

在 10.9 节（编码器—解码器（seq2seq））里，解码器在各个时间步依赖相同的背景变量来获取输入序列信息。当编码器为循环神经网络时，背景变量来自它最终时间步的隐藏状态。

现在，让我们再次思考那一节提到的翻译例子：输入为英语序列 “They”“are”“watching”“.”，输出为法语序列 “Ils”“regardent”“.”。不难想到，解码器在生成输出序列中的每一个词时可能只需利用输入序列某一部分的信息。例如，在输出序列的时间步 1，解码器可以主要依赖 “They”“are”的信息来生成 “Ils”，在时间步 2 则主要使用来自 “watching”的编码信息生成 “regardent”，最后在时间步 3 则直接映射句号 “.”。这看上去就像是在解码器的每一时间步对输入序列中不同时间步的表征或编码信息分配不同的注意力一样。这也是注意力机制的由来 [1]。

仍然以循环神经网络为例，注意力机制通过对编码器所有时间步的隐藏状态做加权平均来得到背景变量。解码器在每一时间步调整这些权重，即注意力权重，从而能够在不同时间步分别关注输入序列中的不同部分并编码进相应时间步的背景变量。本节我们将讨论注意力机制是怎么工作的。

在 10.9 节（编码器—解码器（seq2seq））里我们区分了输入序列或编码器的索引 t 与输出序列或解码器的索引 t' 。该节中，解码器在时间步 t' 的隐藏状态 $\mathbf{s}_{t'} = g(\mathbf{y}_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1})$ ，其中 $\mathbf{y}_{t'-1}$ 是上一时间步 $t' - 1$ 的输出 $y_{t'-1}$ 的表征，且任一时间步 t' 使用相同的背景变量 \mathbf{c} 。但在注意力机制中，解码器的每一时间步将使用可变的背景变量。记 $\mathbf{c}_{t'}$ 是解码器在时间步 t' 的背景变量，那么解码器在该时间步的隐藏状态可以改写为

$$\mathbf{s}_{t'} = g(\mathbf{y}_{t'-1}, \mathbf{c}_{t'}, \mathbf{s}_{t'-1}).$$

这里的关键是如何计算背景变量 $\mathbf{c}_{t'}$ 和如何利用它来更新隐藏状态 $\mathbf{s}_{t'}$ 。下面将分别描述这两个关键点。

10.11.1 计算背景变量

我们先描述第一个关键点，即计算背景变量。图 10.12 描绘了注意力机制如何为解码器在时间步 2 计算背景变量。首先，函数 a 根据解码器在时间步 1 的隐藏状态和编码器在各个时间步的隐藏状态计算 softmax 运算的输入。softmax 运算输出概率分布并对编码器各个时间步的隐藏状态做加权平均，从而得到背景变量。

图 10.12 编码器—解码器上的注意力机制

具体来说，令编码器在时间步 t 的隐藏状态为 \mathbf{h}_t ，且总时间步数为 T 。那么解码器在时间步 t' 的背景变量为所有编码器隐藏状态的加权平均：

$$\mathbf{c}_{t'} = \sum_{t=1}^T \alpha_{t't} \mathbf{h}_t,$$

其中给定 t' 时，权重 $\alpha_{t't}$ 在 $t = 1, \dots, T$ 的值是一个概率分布。为了得到概率分布，我们可以使用 softmax 运算：

$$\alpha_{t't} = \frac{\exp(e_{t't})}{\sum_{k=1}^T \exp(e_{t'k})}, \quad t = 1, \dots, T.$$

现在，我们需要定义如何计算上式中 softmax 运算的输入 $e_{t't}$ 。由于 $e_{t't}$ 同时取决于解码器的时间步 t' 和编码器的时间步 t ，我们不妨以解码器在时间步 $t' - 1$ 的隐藏状态 $\mathbf{s}_{t'-1}$ 与编码器在时间步 t 的隐藏状态 \mathbf{h}_t 为输入，并通过函数 a 计算 $e_{t't}$ ：

$$e_{t't} = a(\mathbf{s}_{t'-1}, \mathbf{h}_t).$$

这里函数 a 有多种选择，如果两个输入向量长度相同，一个简单的选择是计算它们的内积 $a(\mathbf{s}, \mathbf{h}) = \mathbf{s}^\top \mathbf{h}$ 。而最早提出注意力机制的论文则将输入连结后通过含单隐藏层的多层感知机变换 [1]：

$$a(\mathbf{s}, \mathbf{h}) = \mathbf{v}^\top \tanh(\mathbf{W}_s \mathbf{s} + \mathbf{W}_h \mathbf{h}),$$

其中 \mathbf{v} 、 \mathbf{W}_s 、 \mathbf{W}_h 都是可以学习的模型参数。

10.11.1.1 矢量化计算

我们还可以对注意力机制采用更高效的矢量化计算。广义上，注意力机制的输入包括查询项以及一一对应的键项和值项，其中值项是需要加权平均的一组项。在加权平均中，值项的权重来自查询项以及与该值项对应的键项的计算。

在上面的例子中，查询项为解码器的隐藏状态，键项和值项均为编码器的隐藏状态。让我们考虑一个常见的简单情形，即编码器和解码器的隐藏单元个数均为 h ，且函数 $a(\mathbf{s}, \mathbf{h}) = \mathbf{s}^\top \mathbf{h}$ 。假设我们希望根据解码器单个隐藏状态 $\mathbf{s}_{t'-1} \in \mathbb{R}^h$ 和编码器所有隐藏状态 $\mathbf{h}_t \in \mathbb{R}^h, t = 1, \dots, T$ 来计算背景向量 $\mathbf{c}_{t'} \in \mathbb{R}^h$ 。我们可以将查询项矩阵 $\mathbf{Q} \in \mathbb{R}^{1 \times h}$ 设为 $\mathbf{s}_{t'-1}^\top$ ，并令键项矩阵 $\mathbf{K} \in \mathbb{R}^{T \times h}$ 和值项矩阵 $\mathbf{V} \in \mathbb{R}^{T \times h}$ 相同且第 t 行均为 \mathbf{h}_t^\top 。此时，我们只需要通过矢量化计算

$$\text{softmax}(\mathbf{Q} \mathbf{K}^\top) \mathbf{V}$$

即可算出转置后的背景向量 $\mathbf{c}_{t'}^\top$ 。当查询项矩阵 \mathbf{Q} 的行数为 n 时，上式将得到 n 行的输出矩阵。输出矩阵与查询项矩阵在相同行上一一对应。

10.11.2 更新隐藏状态

现在我们描述第二个关键点，即更新隐藏状态。以门控循环单元为例，在解码器中我们可以对 6.7 节（门控循环单元（GRU））中门控循环单元的设计稍作修改，从而变换上一时间步 $t' - 1$ 的输出 $\mathbf{y}_{t'-1}$ 、隐藏状态 $\mathbf{s}_{t'-1}$ 和当前时间步 t' 的含注意力机制的背景变量 $\mathbf{c}_{t'}$ [1]。解码器在时间步 t' 的隐藏状态为

$$\mathbf{s}_{t'} = \mathbf{z}_{t'} \odot \mathbf{s}_{t'-1} + (1 - \mathbf{z}_{t'}) \odot \tilde{\mathbf{s}}_{t'},$$

其中的重置门、更新门和候选隐藏状态分别为

$$\begin{aligned}\mathbf{r}_{t'} &= \sigma(\mathbf{W}_{yr}\mathbf{y}_{t'-1} + \mathbf{W}_{sr}\mathbf{s}_{t'-1} + \mathbf{W}_{cr}\mathbf{c}_{t'} + \mathbf{b}_r), \\ \mathbf{z}_{t'} &= \sigma(\mathbf{W}_{yz}\mathbf{y}_{t'-1} + \mathbf{W}_{sz}\mathbf{s}_{t'-1} + \mathbf{W}_{cz}\mathbf{c}_{t'} + \mathbf{b}_z), \\ \tilde{\mathbf{s}}_{t'} &= \tanh(\mathbf{W}_{ys}\mathbf{y}_{t'-1} + \mathbf{W}_{ss}(\mathbf{s}_{t'-1} \odot \mathbf{r}_{t'}) + \mathbf{W}_{cs}\mathbf{c}_{t'} + \mathbf{b}_s),\end{aligned}$$

其中含下标的 \mathbf{W} 和 \mathbf{b} 分别为门控循环单元的权重参数和偏差参数。

10.11.3 发展

本质上，注意力机制能够为表征中较有价值的部分分配较多的计算资源。这个有趣的想法自提出后得到了快速发展，特别是启发了依靠注意力机制来编码输入序列并解码出输出序列的变换器（Transformer）模型的设计 [2]。变换器抛弃了卷积神经网络和循环神经网络的架构。它在计算效率上比基于循环神经网络的编码器—解码器模型通常更具明显优势。含注意力机制的变换器的编码结构在后来的 BERT 预训练模型中得以应用并令后者大放异彩：微调后的模型在多达 11 项自然语言处理任务中取得了当时最先进的结果 [3]。不久后，同样是基于变换器设计的 GPT-2 模型于新收集的语料数据集预训练后，在 7 个未参与训练的语言模型数据集上均取得了当时最先进的结果 [4]。除了自然语言处理领域，注意力机制还被广泛用于图像分类、自动图像描述、唇语解读以及语音识别。

小结

- 可以在解码器的每个时间步使用不同的背景变量，并对输入序列中不同时间步编码的信息分配不同的注意力。
- 广义上，注意力机制的输入包括查询项以及一一对应的键项和值项。
- 注意力机制可以采用更为高效的矢量化计算。

参考文献

- [1] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.
- [2] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In Advances in Neural Information Processing Systems (pp. 5998-6008).
- [3] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
- [4] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever I. (2019). Language Models are Unsupervised Multitask Learners. OpenAI.

注：本节与原书基本相同，[原书传送门](#)

10.12 机器翻译

机器翻译是指将一段文本从一种语言自动翻译到另一种语言。因为一段文本序列在不同语言中的长度不一定相同，所以我们使用机器翻译为例来介绍编码器—解码器和注意力机制的应用。

10.12.1 读取和预处理数据

我们先定义一些特殊符号。其中“<pad>”（padding）符号用来添加在较短序列后，直到每个序列等长，而“<bos>”和“<eos>”符号分别表示序列的开始和结束。

```
import collections
import os
import io
import math
import torch
from torch import nn
import torch.nn.functional as F
import torchtext.vocab as Vocab
import torch.utils.data as Data

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l

PAD, BOS, EOS = '<pad>', '<bos>', '<eos>'
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

接着定义两个辅助函数对后面读取的数据进行预处理。

```
# 将一个序列中所有的词记录在 all_tokens 中以便之后构造词典，然后在该序列后面添加 PAD 直到
# 长度变为 max_seq_len，然后将序列保存在 all_seqs 中
def process_one_seq(seq_tokens, all_tokens, all_seqs, max_seq_len):
    all_tokens.extend(seq_tokens)
    seq_tokens += [EOS] + [PAD] * (max_seq_len - len(seq_tokens) - 1)
    all_seqs.append(seq_tokens)
```

```
# 使用所有的词来构造词典。并将所有序列中的词变换为词索引后构造 Tensor
def build_data(all_tokens, all_seqs):
    vocab = Vocab.Vocab(collections.Counter(all_tokens),
                         specials=[PAD, BOS, EOS])
    indices = [[vocab.stoi[w] for w in seq] for seq in all_seqs]
    return vocab, torch.tensor(indices)
```

为了演示方便，我们在这里使用一个很小的法语—英语数据集。在这个数据集里，每一行是一对法语句子和它对应的英语句子，中间使用'\\t'隔开。在读取数据时，我们在句末附上“<eos>”符号，并可能通过添加“<pad>”符号使每个序列的长度均为max_seq_len。我们为法语词和英语词分别创建词典。法语词的索引和英语词的索引相互独立。

```
def read_data(max_seq_len):
    # in 和 out 分别是 input 和 output 的缩写
    in_tokens, out_tokens, in_seqs, out_seqs = [], [], [], []
    with io.open('..../data/fr-en-small.txt') as f:
        lines = f.readlines()
        for line in lines:
            in_seq, out_seq = line.rstrip().split('\\t')
            in_seq_tokens, out_seq_tokens = in_seq.split(' '), out_seq.split(' ')
            if max(len(in_seq_tokens), len(out_seq_tokens)) > max_seq_len - 1:
                continue # 如果加上 EOS 后长于 max_seq_len，则忽略掉此样本
            process_one_seq(in_seq_tokens, in_tokens, in_seqs, max_seq_len)
            process_one_seq(out_seq_tokens, out_tokens, out_seqs, max_seq_len)
        in_vocab, in_data = build_data(in_tokens, in_seqs)
        out_vocab, out_data = build_data(out_tokens, out_seqs)
    return in_vocab, out_vocab, Data.TensorDataset(in_data, out_data)
```

将序列的最大长度设成 7，然后查看读取到的第一个样本。该样本分别包含法语词索引序列和英语词索引序列。

```
max_seq_len = 7
in_vocab, out_vocab, dataset = read_data(max_seq_len)
dataset[0]

输出:
(tensor([ 5,  4, 45,  3,  2,  0,  0]), tensor([ 8,  4, 27,  3,  2,  0,  0]))
```

10.12.2 含注意力机制的编码器—解码器

我们将使用含注意力机制的编码器—解码器来将一段简短的法语翻译成英语。下面我们将来介绍模型的实现。

10.12.2.1 编码器

在编码器中，我们将输入语言的词索引通过词嵌入层得到词的表征，然后输入到一个门控循环单元中。正如我们在 6.5 节（循环神经网络的简洁实现）中提到的，PyTorch 的 `nn.GRU` 实例在前向计算后也会分别返回输出和最终时间步的多层隐藏状态。其中的输出指的是最后一层的隐藏层在各个时间步的隐藏状态，并不涉及输出层计算。注意力机制将这些输出作为键项和值项。

```
class Encoder(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 drop_prob=0, **kwargs):
        super(Encoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(embed_size, num_hiddens, num_layers, dropout=drop_prob)

    def forward(self, inputs, state):
        # 输入形状是 (批量大小, 时间步数)。将输出互换样本维和时间步维
        embedding = self.embedding(inputs.long()).permute(1, 0, 2) # (seq_len, batch, _)
        return self.rnn(embedding, state)

    def begin_state(self):
        return None # 隐藏态初始化为 None 时 PyTorch 会自动初始化为 0
```

下面我们来创建一个批量大小为 4、时间步数为 7 的小批量序列输入。设门控循环单元的隐藏层数为 2，隐藏单元个数为 16。编码器对该输入执行前向计算后返回的输出形状为 (时间步数, 批量大小, 隐藏单元个数)。门控循环单元在最终时间步的多层隐藏状态的形状为 (隐藏层数, 批量大小, 隐藏单元个数)。对于门控循环单元来说，`state` 就是一个元素，即隐藏状态；如果使用长短期记忆，`state` 是一个元组，包含两个元素即隐藏状态和记忆细胞。

```
encoder = Encoder(vocab_size=10, embed_size=8, num_hiddens=16, num_layers=2)
output, state = encoder(torch.zeros((4, 7)), encoder.begin_state())
output.shape, state.shape # GRU 的 state 是 h, 而 LSTM 的是一个元组 (h, c)
```

输出：

```
(torch.Size([7, 4, 16]), torch.Size([2, 4, 16]))
```

10.12.2.2 注意力机制

我们将实现 10.11 节（注意力机制）中定义的函数 a : 将输入连结后通过含单隐藏层的多层感知机变换。其中隐藏层的输入是解码器的隐藏状态与编码器在所有时间步上隐藏状态的一一连结，且使用 \tanh 函数作为激活函数。输出层的输出个数为 1。两个 `Linear` 实例均不使用偏差。其中函数 a 定义里向量 v 的长度是一个超参数，即 `attention_size`。

```
def attention_model(input_size, attention_size):
    model = nn.Sequential(nn.Linear(input_size,
                                    attention_size, bias=False),
                          nn.Tanh(),
                          nn.Linear(attention_size, 1, bias=False))
    return model
```

注意力机制的输入包括查询项、键项和值项。设编码器和解码器的隐藏单元个数相同。这里的查询项为解码器在上一时间步的隐藏状态，形状为 (批量大小, 隐藏单元个数)；键项和值项均为编码器在所有时间步的隐藏状态，形状为 (时间步数, 批量大小, 隐藏单元个数)。注意力机制返回当前时间步的背景变量，形状为 (批量大小, 隐藏单元个数)。

```
def attention_forward(model, enc_states, dec_state):
    """
    enc_states: (时间步数, 批量大小, 隐藏单元个数)
    dec_state: (批量大小, 隐藏单元个数)
    """

    # 将解码器隐藏状态广播到和编码器隐藏状态形状相同后进行连结
    dec_states = dec_state.unsqueeze(dim=0).expand_as(enc_states)
    enc_and_dec_states = torch.cat((enc_states, dec_states), dim=2)
    e = model(enc_and_dec_states)  # 形状为 (时间步数, 批量大小, 1)
    alpha = F.softmax(e, dim=0)  # 在时间步维度做 softmax 运算
    return (alpha * enc_states).sum(dim=0)  # 返回背景变量
```

在下面的例子中，编码器的时间步数为 10，批量大小为 4，编码器和解码器的隐藏单元个数均为 8。注意力机制返回一个小批量的背景向量，每个背景向量的长度等于编码器的隐藏单元个数。因此输出的形状为 (4, 8)。

```
seq_len, batch_size, num_hiddens = 10, 4, 8
model = attention_model(2*num_hiddens, 10)
enc_states = torch.zeros((seq_len, batch_size, num_hiddens))
dec_state = torch.zeros((batch_size, num_hiddens))
attention_forward(model, enc_states, dec_state).shape # torch.Size([4, 8])
```

10.12.2.3 含注意力机制的解码器

我们直接将编码器在最终时间步的隐藏状态作为解码器的初始隐藏状态。这要求编码器和解码器的循环神经网络使用相同的隐藏层数和隐藏单元个数。

在解码器的前向计算中，我们先通过刚刚介绍的注意力机制计算得到当前时间步的背景向量。由于解码器的输入来自输出语言的词索引，我们将输入通过词嵌入层得到表征，然后和背景向量在特征维连结。我们将连结后的结果与上一时间步的隐藏状态通过门控循环单元计算出当前时间步的输出与隐藏状态。最后，我们将输出通过全连接层变换为有关各个输出词的预测，形状为 (批量大小, 输出词典大小)。

```
class Decoder(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 attention_size, drop_prob=0):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.attention = attention_model(2*num_hiddens, attention_size)
        # GRU 的输入包含 attention 输出的 c 和实际输入，所以尺寸是 2*embed_size
        self.rnn = nn.GRU(2*embed_size, num_hiddens, num_layers, dropout=drop_prob)
        self.out = nn.Linear(num_hiddens, vocab_size)

    def forward(self, cur_input, state, enc_states):
        """
        cur_input shape: (batch, )
        state shape: (num_layers, batch, num_hiddens)
        """
        # 使用注意力机制计算背景向量
        c = attention_forward(self.attention, enc_states, state[-1])
```

```

# 将嵌入后的输入和背景向量在特征维连结
input_and_c = torch.cat((self.embedding(cur_input), c), dim=1) # (批量大小, 2
# 为输入和背景向量的连结增加时间步维, 时间步个数为 1
output, state = self.rnn(input_and_c.unsqueeze(0), state)
# 移除时间步维, 输出形状为 (批量大小, 输出词典大小)
output = self.out(output).squeeze(dim=0)
return output, state

def begin_state(self, enc_state):
    # 直接将编码器最终时间步的隐藏状态作为解码器的初始隐藏状态
    return enc_state

```

10.12.3 训练模型

我们先实现 `batch_loss` 函数计算一个小批量的损失。解码器在最初时间步的输入是特殊字符 `BOS`。之后，解码器在某时间步的输入为样本输出序列在上一时间步的词，即强制教学。此外，同 10.3 节（word2vec 的实现）中的实现一样，我们在这里也使用掩码变量避免填充项对损失函数计算的影响。

```

def batch_loss(encoder, decoder, X, Y, loss):
    batch_size = X.shape[0]
    enc_state = encoder.begin_state()
    enc_outputs, enc_state = encoder(X, enc_state)
    # 初始化解码器的隐藏状态
    dec_state = decoder.begin_state(enc_state)
    # 解码器在最初时间步的输入是 BOS
    dec_input = torch.tensor([out_vocab.stoi[BOS]] * batch_size)
    # 我们将使用掩码变量 mask 来忽略掉标签为填充项 PAD 的损失
    mask, num_not_pad_tokens = torch.ones(batch_size,), 0
    l = torch.tensor([0.0])
    for y in Y.permute(1,0): # Y shape: (batch, seq_len)
        dec_output, dec_state = decoder(dec_input, dec_state, enc_outputs)
        l = l + (mask * loss(dec_output, y)).sum()
        dec_input = y # 使用强制教学
        num_not_pad_tokens += mask.sum().item()
    # 将 PAD 对应位置的掩码设成 0, 原文这里是 y != out_vocab.stoi[EOS], 感觉有误

```

```

    mask = mask * (y != out_vocab.stoi[PAD]).float()
    return l / num_not_pad_tokens

```

在训练函数中，我们需要同时迭代编码器和解码器的模型参数。

```

def train(encoder, decoder, dataset, lr, batch_size, num_epochs):
    enc_optimizer = torch.optim.Adam(encoder.parameters(), lr=lr)
    dec_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)

    loss = nn.CrossEntropyLoss(reduction='none')
    data_iter = Data.DataLoader(dataset, batch_size, shuffle=True)
    for epoch in range(num_epochs):
        l_sum = 0.0
        for X, Y in data_iter:
            enc_optimizer.zero_grad()
            dec_optimizer.zero_grad()
            l = batch_loss(encoder, decoder, X, Y, loss)
            l.backward()
            enc_optimizer.step()
            dec_optimizer.step()
            l_sum += l.item()
        if (epoch + 1) % 10 == 0:
            print("epoch %d, loss %.3f" % (epoch + 1, l_sum / len(data_iter)))

```

接下来，创建模型实例并设置超参数。然后，我们就可以训练模型了。

```

embed_size, num_hiddens, num_layers = 64, 64, 2
attention_size, drop_prob, lr, batch_size, num_epochs = 10, 0.5, 0.01, 2, 50
encoder = Encoder(len(in_vocab), embed_size, num_hiddens, num_layers,
                  drop_prob)
decoder = Decoder(len(out_vocab), embed_size, num_hiddens, num_layers,
                  attention_size, drop_prob)
train(encoder, decoder, dataset, lr, batch_size, num_epochs)

```

输出：

```

epoch 10, loss 0.441
epoch 20, loss 0.183

```

```
epoch 30, loss 0.100
epoch 40, loss 0.046
epoch 50, loss 0.025
```

10.12.4 预测不定长的序列

在 10.10 节（束搜索）中我们介绍了 3 种方法来生成解码器在每个时间步的输出。这里我们实现最简单的贪婪搜索。

```
def translate(encoder, decoder, input_seq, max_seq_len):
    in_tokens = input_seq.split(' ')
    in_tokens += [EOS] + [PAD] * (max_seq_len - len(in_tokens) - 1)
    enc_input = torch.tensor([[in_vocab.stoi[tk] for tk in in_tokens]]) # batch=1
    enc_state = encoder.begin_state()
    enc_output, enc_state = encoder(enc_input, enc_state)
    dec_input = torch.tensor([out_vocab.stoi[BOS]])
    dec_state = decoder.begin_state(enc_state)
    output_tokens = []
    for _ in range(max_seq_len):
        dec_output, dec_state = decoder(dec_input, dec_state, enc_output)
        pred = dec_output.argmax(dim=1)
        pred_token = out_vocab.itos[int(pred.item())]
        if pred_token == EOS: # 当任一时间步搜索出 EOS 时，输出序列即完成
            break
        else:
            output_tokens.append(pred_token)
            dec_input = pred
    return output_tokens
```

简单测试一下模型。输入法语句子“ils regardent.”，翻译后的英语句子应该是“they are watching.”。

```
input_seq = 'ils regardent .'
translate(encoder, decoder, input_seq, max_seq_len)
```

输出：

```
['they', 'are', 'watching', '.']
```

10.12.5 评价翻译结果

评价机器翻译结果通常使用 BLEU (Bilingual Evaluation Understudy) [1]。对于模型预测序列中任意的子序列，BLEU 考察这个子序列是否出现在标签序列中。

具体来说，设词数为 n 的子序列的精度为 p_n 。它是预测序列与标签序列匹配词数为 n 的子序列的数量与预测序列中词数为 n 的子序列的数量之比。举个例子，假设标签序列为 A, B, C, D, E, F ，预测序列为 A, B, B, C, D ，那么 $p_1 = 4/5$, $p_2 = 3/4$, $p_3 = 1/3$, $p_4 = 0$ 。设 $\text{len}_{\text{label}}$ 和 len_{pred} 分别为标签序列和预测序列的词数，那么，BLEU 的定义为

$$\exp \left(\min \left(0, 1 - \frac{\text{len}_{\text{label}}}{\text{len}_{\text{pred}}} \right) \right) \prod_{n=1}^k p_n^{1/2^n},$$

其中 k 是我们希望匹配的子序列的最大词数。可以看到当预测序列和标签序列完全一致时，BLEU 为 1。

因为匹配较长子序列比匹配较短子序列更难，BLEU 对匹配较长子序列的精度赋予了更大权重。例如，当 p_n 固定在 0.5 时，随着 n 的增大， $0.5^{1/2} \approx 0.7$, $0.5^{1/4} \approx 0.84$, $0.5^{1/8} \approx 0.92$, $0.5^{1/16} \approx 0.96$ 。另外，模型预测较短序列往往会得到较高 p_n 值。因此，上式中连乘项前面的系数是为了惩罚较短的输出而设的。举个例子，当 $k = 2$ 时，假设标签序列为 A, B, C, D, E, F ，而预测序列为 A, B 。虽然 $p_1 = p_2 = 1$ ，但惩罚系数 $\exp(1 - 6/2) \approx 0.14$ ，因此 BLEU 也接近 0.14。

下面来实现 BLEU 的计算。

```
def bleu(pred_tokens, label_tokens, k):
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
    for n in range(1, k + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[''.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[''.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[''.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score
```

接下来，定义一个辅助打印函数。

```
def score(input_seq, label_seq, k):  
    pred_tokens = translate(encoder, decoder, input_seq, max_seq_len)  
    label_tokens = label_seq.split(' ')  
    print('bleu %.3f, predict: %s' % (bleu(pred_tokens, label_tokens, k),  
                                      ' '.join(pred_tokens)))
```

预测正确则分数为 1。

```
score('ils regardent .', 'they are watching .', k=2)
```

输出：

```
bleu 1.000, predict: they are watching .
```

测试一个不在训练集中的样本。

```
score('ils sont canadiens .', 'they are canadian .', k=2)
```

输出：

```
bleu 0.658, predict: they are russian .
```

小结

- 可以将编码器—解码器和注意力机制应用于机器翻译中。
- BLEU 可以用来评价翻译结果。

参考文献

[1] Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002, July). BLEU: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting on association for computational linguistics (pp. 311-318). Association for Computational Linguistics.

[2] WMT. <http://www.statmt.org/wmt14/translation-task.html>

[3] Tatoeba Project. <http://www.manythings.org/anki/>

注：本节除代码外与原书基本相同，[原书传送门](#)