第 4 章　机器学习

第 5 节　强化学习

本节中我们将介绍 Agent 如何从成功与失败、回报与惩罚中进行学习。

1　强化学习

1.1　强化学习

1.1.1　强化学习[Sutton, R. and Barto, A,1998]

强化学习就是学习“做什么（即如何把当前的情境映射成动作）才能使得数值化的收益信号最大化”。学习者不会被告知应该采取什么动作，而是必须自己通过尝试去发现哪些动作会产生最丰厚的收益。在最有趣又最困难的案例中，动作往往影响的不仅仅是即时收益，也会影响下一个情境，从而影响随后的收益。试错和延迟收益这两个特征是强化学习两个最重要最显著的特征。

强化学习和现在机器学习领域中广泛使用的有监督学习不同，有监督学习是从外部监督者提供的带标注训练集中进行学习。每一个样本都是关于情境和标注的描述。所谓标注，即针对当前情境，系统应该做出的正确动作，也可将其看作对当前情景进行分类的所属类别标签。采用这种学习方式是为了让系统能够具备推断或泛化能力，能够响应不同的情境并做出正确的动作，哪怕这个情境并没有在训练集合中出现过。这是一种重要的学习方式，但是并不适用于从交互中学习这类问题。在交互问题中，我们不可能获得在所有情境下既正确又有代表性的动作示例。在一个未知领域，若想做到最好（收益最大），智能体必须要能够从自身的经验中学习。

强化学习也和机器学习研究者口中的无监督学习不同，无监督学习是一个典型的寻找未标注数据中隐含结构的过程。看上去所有的机器学习范式都可以被划分成有监督学习和无监督学习，但事实并非如此。尽管有人可能会认为强化学习也是一种无监督学习，因为它不依赖于每个样本所对应的正确行为（即标注），但强化学习的目的是最大化收益信号，而不是找出数据的隐含结构。通过智能体的经验揭示其结构在强化学习中当然是有益的，但这并不能解决最大化收益信号的强化学习问题。因此，我们认为强化学习是与有监督学习和无监督学习并列的第三种机器学习范式，当然也有可能存在其他的范式。

所有强化学习的智能体都有一个明确的目标，即能够感知环境的各个方面，并可以选择动作来影响它们所处的环境。此外，通常规定智能体从最开始就要工作，尽管其面对的环境具有很大的不确定性。当强化学习涉及规划时，它必须处理规划和实时动作选择之间的相互影响，以及如何获取和改善环境模型的问题。

1.1.1.1　强化学习要素

除了智能体和环境之外，强化学习系统有四个核心要素：策略、收益信号、价值函数以及（可选的）对环境建立的模型。

策略[Conti, E.et al.,]定义了学习智能体在特定时间的行为方式。简单地说，策略是环境状态到动作的映射。它对应于心理学中被称为“刺激-反应”的规则或关联关系。在某些情况下，策略可能是一个简单的函数或查询表，而在另一些情况下，它可能涉及大量的计算，例如搜索过程。策略本身是可以决定行为的，因此策略是强化学习智能体的核心。一般来说，策略可能是环境所在状态和智能体所采取的动作的随机函数。

收益信号定义了强化学习问题中的目标。在每一步中，环境向强化学习智能体发送一个称为收益的标

量数值。智能体的唯一目标是最大化长期总收益。也就是说，收益信号决定了，对于智能体来说何为好、何为坏。因此，收益信号是改变策略的主要基础。如果策略选择的动作导致了低收益，那么可能会改变策略，从而在未来的这种情况下选择一些其他的动作。一般来说，收益信号可能是环境状态和在此基础上所采取的动作的随机函数。

收益信号表明了在短时间内什么是好的，而价值函数则表示了从长远的角度看什么是好的。简单地说，一个状态的价位是一个智能体从这个状态开始，对将来累积的总收益的期望。尽管收益决定了环境状态直接、即时、内在的吸引力，但价值表示了接下来所有可能状态的长期期望。例如，某状态的即时收益可能很低，但它仍然可能具有很高的价值，因为之后定期会出现高收益的状态，反之亦然。用人打比方，收益就像即时的愉悦（高收益）和痛苦（低收益），而价值则是在当前的环境与特定状态下，对我们未来究竟有多愉悦或多不愉悦的更具有远见的判断。

从某种意义上来说，收益更加重要，而作为收益预测的价值次之。没有收益就没有价值，而评估价值的唯一目的就是获得更多的收益。然而，在制定和评估策略时，我们最关心的是价值。动作选择是基于对价值的判断做出的。我们寻求能带来最高价值而不是最高收益的状态的动作，因为这些动作从长远来看会为我们带来最大的累积收益。不幸的是，确定价值要比确定收益难得多。收益基本上是由环境直接给予的，但是价值必须综合评估，并根据智能体在整个过程中观察到的收益序列重新估计。事实上，价值评估方法才是几乎所有强化学习算法中最重要的组成部分。价值评估的核心作用可以说是我们在过去 60 年里所学到的关于强化学习的最重要的东西。

强化学习系统的第四个也是最后一个要素是对环境建立的模型。这是一种对环境的反应模式的模拟，或者更一般地说，它允许对外部环境的行为进行推断。例如，给定一个状态和动作，模型就可以预测外部环境的下一个状态和下一个收益。环境模型会被用于做规划。规划，就是在真正经历之前，先考虑未来可能发生的各种情境从而预先决定采取何种动作。使用环境模型和规划来解决强化学习问题的方法被称为有模型的方法。而简单的无模型的方法则是直接地试错，这与有目标地进行规划恰好相反。强化学习系统可以同时通过试错、学习环境模型并使用模型来进行规划。现代强化学习已经从低级的、试错式的学习延展到了高级的、深思熟虑的规划。

1.1.2 有限马尔可夫决策过程[Sutton, R. and Barto, A,1998]

1.1.2.1 “智能体-环境”交互接口

马尔可夫过程（MDP）是一种通过交互式学习来实现目标的理论框架。进行学习及实施决策的机器被称为智能体(agent)。智能体之外所有与其相互作用的事物都被称为环境(environment)。这些事物之间持续进行交互，智能体选择动作，环境对这些动作做出相应的响应，并向智能体呈现出新的状态。环境也会产生一个收益，通常是特定的数值，这就是智能体在动作选择过程中想要最大化的目标，如图 1.1 所示。

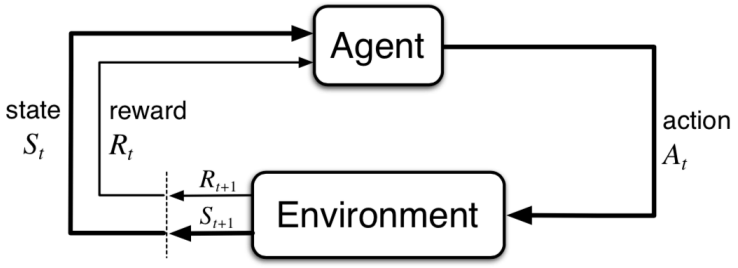


图 1.1 马尔可夫决策过程中的“智能体-环境”交互

更具体地说，在每个离散时刻$t = 0,1,2,3\cdots$，智能体和环境都发生了交互。在每个时刻 t，智能体观察到所在的环境状态的某种特征表达，$S_t \in S$，并且在此基础上选择一个动作，$A_t \in A(S)$。下一时刻，作为其

动作的结果，智能体接收到一个数值化的收益，$R_{t+1} \in \mathcal{R} \subset \mathbb{R}$，并进入一个新的状态$S_{t+1}$。从而，MDP 和智能体共同给出了一个序列或轨迹，类似这样：

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \cdots \tag{1.1}$$

在有限 MDP 中，状态、动作和收益的集合（S、A 和$\mathcal{R}$）都只有有限个元素。在这种情况下，随机变量R_t和S_t具有定义明确的离散概率分布，并且只依赖于前继状态和动作。也就是说，给定前继状态和动作的值时，这些随机变量的特定值，$s' \in S$和$\mathrm{r} \in \mathcal{R}$，在 t 时刻出现的概率是：

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{1.2}$$

对于任意s'，$\mathrm{s} \in S$，$\mathrm{r} \in \mathcal{R}$，以及$\mathrm{a} \in A(s)$。函数 p 定义了 MDP 的动态特性。公式中等号上的圆点提醒我们，这是一个定义（在这种情况下是函数p），而不是从之前定义推导出来的事实。动态函数$p: S \times \mathcal{R} \times A \to [0,1]$是有 4 个参数的普通的确定性函数。中间的“|”是表示条件概率的符号，但这里也只是提醒我们，函数 p 为每个 s 和 a 的选择都指定了一个概率分布，即

$$\sum_{s' \in S} \sum_{\mathrm{r} \in \mathcal{R}} p(s', r|s, a) = 1, \text{对所有 } \mathrm{s} \in S, \mathrm{a} \in A(s) \tag{1.3}$$

在马尔可夫决策过程中，由p给出的概率完全刻画了环境的动态特性。也就是说，S_t和R_t的每个可能的值出现的概率只取决于前一个状态S_{t-1}和前一个动作A_{t-1}，并且与更早之前的状态和动作完全无关。这个限制并不是针对决策过程，而是针对状态的。状态必须包括过去智能体和环境交互的方方面面的信息，这些信息会对未来产生一定影响。这样，状态就被认为具有马尔可夫性。

从四参数动态函数p中，我们可以计算出关于环境的任何其他信息，例如状态转移概率（我们将其表示为一个三参数函数$p: S \times S \times A \to [0,1]$）：

$$p(s'|s, a) \doteq Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{\mathrm{r} \in \mathcal{R}} p(s', r|s, a) \tag{1.4}$$

我们还可以定义“状态-动作”二元组的期望收益，并将其表示为一个双参数函数$r: S \times A \to \mathbb{R}$：

$$r(s, a) \doteq \mathrm{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{\mathrm{r} \in \mathcal{R}} \sum_{s' \in S} p(s', r|s, a) \tag{1.5}$$

和“状态-动作-后继状态”三元组的期望收益，并将其表示为一个三参数函数：$S \times A \times S \to \mathbb{R}$

$$r(s, a, s') \doteq \mathrm{E}[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{\mathrm{r} \in \mathcal{R}} r \frac{p(s', r|s, a)}{p(s'|s, a)} \tag{1.6}$$

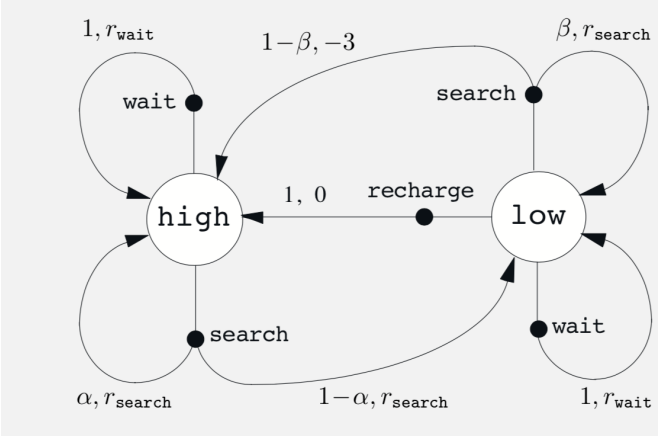
在本书中，我们经常会用到四参数函数p（式 1.2），但也会偶尔用其他函数。

MDP 框架非常抽象与灵活，并且能够以许多不同的方式应用到众多问题中。例如时间步长不需要是真实时间的固定间隔，也可以是决策和行动的任意的连贯阶段。动作可以是低级的控制，例如机器臂的发动机的电压，也可以是高级的决策，例如是否吃午餐或是否去研究院。同样地，状态可以采取多种表述形式。它们可以完全由低级感知决定，例如传感器的直接读数，也可以是更高级、更抽象的，比如对房间中的某个对象进行的符号性描述。状态的一些组成成分可以是基于过去感知的记忆，甚至也可以是完全主观的。例如，智能体可能不确定对象位置，也可能刚刚响应过某种特定的感知。类似地，一些动作可能是完全主观的或完全可计算的。例如，一些动作可以控制智能体选择考虑的内容，或者控制它把注意力集中在哪里。

一般来说，动作可以是任何我们想要做的决策，而状态则可以是任何对决策有所帮助的事情。

例 1 回收机器人可以完成在办公环境中收集废弃易拉罐的工作。它具有用于检测易拉罐的传感器，以及可以拿起易拉罐并放人机载箱中的臂和夹具，并由可充电电池供能。机器人的控制系统包括用于传感信息解释、导航和控制手臂与夹具的组件。强化学习智能体基于当前电池电量，做出如何搜索易拉罐的高级决策。举个简单的例子，假设只有两个可区分的充电水平，并组成一个很小的状态集合 S=｛high，low｝。在每个状态中，智能体可以决定是否应该：(1)在某段特定时间内主动搜索（search）易拉罐；(2)保持静止并等待（wait）易拉罐；或(3)直接回到基地充电（recharge）。当能量水平高（high）时，充电的动作是非常愚蠢的动作，所以我们不会把它加入这个状态对应的动作集合中。于是我们可以把动作集合表示为$\mathrm{A}(high) = \{\mathrm{search}，\ \mathrm{wait}\}$和$\mathrm{A}(low) = \{\mathrm{search}，\ \mathrm{wait}，\ \mathrm{recharge}\}$。

在大多数情况下，收益为零；但当机器人捡到一个空罐子时，收益就为正；或当电池完全耗尽时，收益就是一个非常大的负值。寻找罐子的最好方法是主动搜索，但这会耗尽机器人的电池，而等待则不会。每当机器人进行搜索时，电池都有被耗尽的可能性。耗尽时，机器人必须关闭系统并等待被救（产生低收益）。如果能量水平高，那么总是可以完成一段时间的主动搜索，而不用担心没电。以高能级开始进行一段时间的搜索后，其能量水平仍是高的概率为α，下降为低的概率为$1-\alpha$。另一方面，以低能级开始进行一段时间的搜索后，其能量水平仍是低（low）的概率为β，耗尽电池能量的概率为$1-\beta$。在后一种情况下，机器人需要人工救援，然后将电池重新充电至高水平。机器人收集的每个罐子都可作为一个单位收益，而每当机器人需要被救时，收益为-3。让r_{search}和 r_{wait}（$r_{search} > r_{wait}$）分别表示机器人在搜索和等待期间收集的期望数量（也就是期望收益）。最后，假设机器人在充电时不能收集罐子，并且在电池耗尽时也不能收集罐子。这个系统则是一个有限 MDP，我们可以写出其转移概率和期望收益，其动态变化如下表所示。



s	a	s'	$p(s'\|s,a)$	$r(s,a,s')$
high	search	high	α	r_{search}
high	search	low	$1-\alpha$	r_{search}
low	search	high	$1-\beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	r_{wait}
low	wait	high	0	r_{wait}
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	0

请注意，当前状态 s、动作a ∈ $A(s)$和后继状态s'的每一个可能的组合都在表中有对应的一行表示。另一种归纳有限 MDP 的有效方法就是转移图，如上图所示。图中有两种类型的节点：状态节点和动作节点。每个可能的状态都有一个状态节点（以状态命名的一个大空心圆），而每个“状态-动作”二元组都有一个动作节点（以动作命名的一个小实心圆，和指向状态节点的连线）。从状态 s 开始并执行动作 a，你将顺着连线从状态节点 s 到达动作节点(s，a)。然后环境做出响应，通过一个离开动作节点(s，a)的箭头，转移到下一个状态节点。每个箭头都对应着个三元组(s，s'，a)，其中s'是下一个状态。我们把每个箭头都标上一个转移概率$p(s'|s,a)$和转移的期望收益$r(s,a,s')$。请注意，离开一个动作节点的转移概率之和为 1。

1.1.2.2 目标和收益

在强化学习中，智能体的目标被形式化表征为一种特殊信号，称为收益，它通过环境传递给智能体。在每个时刻，收益都是一个单一标量数值，$R_t \in \mathbb{R}$。非正式地说，智能体的目标是最大化其收到的总收益。这意味着需要最大化的不是当前收益，而是长期的累积收益。我们可以将这种非正式想法清楚地表述为收益假设：

我们所有的“目标”或“目的”都可以归结为：最大化智能体接收到的标量信号（称之为收益）累积和的概率期望值。

使用收益信号来形式化目标是强化学习最显著的特征之一。

1.1.2.3 回报和分幕

知道智能体的目标就是最大限度地提高长期收益。把时刻 t 后接收的收益序列表示为R_{t+1}，R_{t+2}，R_{t+3}，…，我们寻求的是最大化期望回报，记为G_t，它被定义为收益序列的一些特定函数。在最简单的情况下，回报是收益的总和：

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_r \tag{1.7}$$

其中 T 是最终时刻。这种方法在有“最终时刻”这种概念的应用中是有意义的。在这类应用中，智能体和环境的交互能被自然地分成一系列子序列（每个序列都存在最终时刻），我们称每个子序列为幕(episodes)，例如一盘游戏、一次走迷宫的旅程或任何这类重复性的交互过程。每幕都以一种特殊状态结束，称之为终结状态。随后会重新从某个标准的起始状态或起始状态的分布中的某个状态样本开始。即使结束的方式不同，例如比赛的胜负，下一幕的开始状态与上一幕的结束方式完全无关。因此，这些幕可以被认为在同样的终结状态下结束，只是对不同的结果有不同的收益。具有这种分幕重复特性的任务称为分幕式任务。在分幕式任务中，我们有时需要区分非终结状态集，记为 S，和包含终结与终结状态的所有状态集，记作S^+。终结的时间 T 是一个随机变量，通常随着幕的不同而不同。

另一方面，在许多情况下，智能体-环境交互不一定能被自然地分为单独的幕，而是持续不断地发生。例如，我们很自然地就会想到一个连续的过程控制任务或者长期运行机器人的应用。我们称这些为持续性任务。回报公式（1.7）用于描述持续性任务时会出现问题，因为最终时刻 T= ∞，并且我们试图最大化的回报也很容易趋于无穷（例如，假设智能体在每个时刻都收到+1 的收益）。

我们需要引入一个额外概念，即折扣。根据这种方法，智能体尝试选择动作，使得它在未来收到的经过折扣系数加权后的（我们成为“折后”）收益总和是最大化的。特别地，它选择A_t来最大化期望折后回报：

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.8}$$

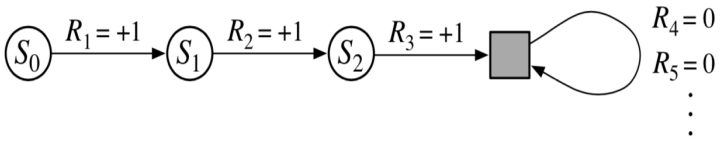
其中，γ是一个参数，0≤γ≤1，被称为折扣率。

折扣率决定了未来收益的现值：未来时刻 k 的收益值只有它的当前值的γ^{k-1}倍。如果γ＜1，那么只要

收益序列$\{R_k\}$有界，式（1.8）中的无限序列总和就是一个有限值。如果γ=0，那么智能体是“目光短浅的”，即只关心最大化当前收益。在这种情况下，其目标是学习如何选择A_t来最大化R_{t+1}。如果每个智能体的行为都碰巧只影响当前收益，而不是未来的回报，那么目光短浅的智能体可以通过单独最大化每个当前收益来最大化式（1.8）。但一般来说，最大化当前收益会减少未来的收益，以至于实际上的收益变少了。随着γ接近 1，折后回报将更多地考虑未来的收益，也就是说智能体变得有远见了。

1.1.2.4 分幕式和持续性任务的统一表示法

我们需要使用另一个约定来获得一个统一符号，它可以同时适用于分幕式和持续性任务。在一种情况中，我们将回报定义为有限项的总和，如式（1.7）所示；而在另一种情况中，我们将回报定义为无限项的总和，如式（1.8）所示。这两者可以通过一个方法进行统一，即把幕的终止当作一个特殊的吸收状态的入口，它只会转移到自己并且只产生零收益。例如，考虑状态转移图：



这里的方块表示与幕结束对应的吸收状态。从S_0开始，我们就会得到收益序列+1，+1，+1，0，0，0，…。总之，无论我们是计算前 T 个收益（这里=3）的总和，还是计算无限序列的全部总和，我们都能得到相同的回报。即使我们引入折扣，这也仍然成立。因此，一般来说，我们可以根据式（1.8）来定义回报。这个定义符合前述的省略幕编号的简化写法，并且考虑了$\gamma = 1$且加和依然存在的情况（例如，在分幕式任务中，所有幕都会在有限时长终止）。或者，我们也可以把回报表示为：

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \qquad (1.9)$$

并允许上式包括$\mathrm{T} = \infty$或$\gamma = 1$（但不是二者同时）的可能性。我们在本书的其他地方就会使用这些惯例来简化符号，并表达分幕式和持续性任务之间的紧密联系。

1.1.2.5 策略和价值函数

几乎所有的强化学习算法都涉及价值函数的计算。价值函数是状态（或状态与动作二元组）的函数，用来评估当前智能体在给定状态（或给定状态与动作）下有多好。这里“有多好”的概念是用未来预期的收益来定义的，或者更准确地说，就是回报的期望值。当然，智能体期望未来能得到的收益取决于智能体所选择的动作。因此，价值函数是与特定的行为方式相关的，我们称之为策略。

严格地说，策略是从状态到每个动作的选择概率之间的映射。如果智能体在时刻 t 选择了策略π，那么$\pi(a|s)$就是当$S_t = s$时$A_t = a$的概率。就像 p 一样，π就是一个普通的函数；$\pi(a|s)$中间的“|”只是提醒我们它为每个$\mathrm{s} \in \mathrm{S}$都定义了一个在$\mathrm{a} \in \mathrm{A}$上的概率分布。强化学习方法规定了智能体的策略如何随着其经验而发生变化。

练习：如果当前状态是S_t，并根据随机策略π选择动作，那么如何用π和四参数函数 p（式 1.2）来表示R_{t+1}的期望呢？

我们把策略π下状态 s 的价值函数记为$v_\pi(s)$，即从状态 s 开始，智能体按照策略π进行决策所获得的回报的概率期望值。对于 MDP，我们可以正式定义v_π为：

$$v_\pi(s) \doteq \mathrm{E}_\pi[G_t|S_t = s] = \mathrm{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} |S_t = s] \qquad (1.10)$$

其中，E_π表示在给定策略π时一个随机变量的期望值，t 可以是任意时刻。请注意，终止状态的价值始终为零。我们把函数v_π称为策略π的状态价值函数。

类似地，我们把策略π下在状态 s 时采取动作 a 的价值记为$q_\pi(s，a)$。这就是根据策略π，从状态 s 开始，执行动作 a 之后，所有可能的决策序列的期望回报：

$$q_\pi(s，a) \doteq E_\pi[G_t|S_t=s,A_t=a] = E_\pi[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t=s,A_t=a] \quad (1.11)$$

我们称q_π为策略π的动作价值函数。

1.1.2.6 最优策略和最优价值函数

解决一个强化学习任务就意味着要找出一个策略，使其能够在长期过程中获得大量收益。对于有限 MDP，我们可以通过比较价值函数精确地定义一个最优策略。在本质上，价值函数定义了策略上的一个偏序关系。即如果要说一个策略π与另一个策略π′相差不多甚至其更好，那么其所有状态上的期望回报都应该等于或大于π′的期望回报。也就是说，若对于所有的$s \in S$，$\pi \geq \pi'$，那么应当$v_\pi(s) \geq v_{\pi'}(s)$。总会存在至少一个策略不劣于其他所有的策略，这就是最优策略。尽管最优策略可能不止一个，我们还是用π，来表示所有这些最优策略。它们共享相同的状态价值函数，称之为最优状态价值函数，记为v_*，其定义为：对于任意$s \in S$：

$$v_*(s) \doteq \max_\pi v_\pi(s) \quad (1.12)$$

最优的策略也共享相同的最优动作价值函数，记为q_*，其定义为：对于任意$s \in S$，$a \in A$：

$$q_*(s,a) \doteq \max_\pi q_\pi(s，a) \quad (1.13)$$

对于“状态-动作”二元组$(s，a)$，这个函数给出了在状态 s 下，先采取动作 a，之后按照最优策略去决策的期望回报。因此，我们可以用v_*来表示q_*，如下所示：

$$q_*(s,a) = E[R_{t+1} + \gamma v_*(S_{t+1})|S_t=s,A_t=a] \quad (1.14)$$

1.1.3 Q 学习：离轨策略下的时序差分控制[Sutton, R.and Barto, A,1998]

离轨策略下的时序差分控制算法的提出是强化学习早期的一个重要突破。这一算法被称为 Q 学习[Watkins, C. J. C. H. (1989)]，其定义为：

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)\right] \quad (1.15)$$

在这里，待学习的动作价值函数 Q 采用了对最优动作价值函数q_*的直接近似作为学习目标，而与用于生成智能体决策序列轨迹的行动策略是什么无关。这大大简化了算法的分析，也很早就给出收敛性证明。当然，正在遵循的行动策略仍会产生影响，它可以决定哪些“状态-动作”二元组会被访问和更新。然而，只需要所有的“状态-动作”二元组可以持续更新，整个学习过程就能够正确地收敛。在一般情况下，任何方法要保证能找到最优的智能体行为的最低要求。基于这种假设以及步长参数序列的某个常用的随机近似条件，可以证明 Q 能以 1 的概率收敛到q_*。Q 学习算法的流程如下面框中所示。

Q 学习(离轨策略下的时序差分控制)算法，用于预测$\pi \approx \pi_*$

算法参数：步长$\alpha \in (0,1]$，很小的ε，$\varepsilon > 0$

对所有$s \in S^+$，$a \in A(s)$，任意初始化$Q(s,a)$，其中$Q(\text{终止状态},\cdot) = 0$

对每幕：

初始化 S

对幕中的每一步循环：

使用从 Q 得到的策略（例如ε-贪心），在 S 处选择 A

执行 A，观察到 R，S'

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$$

$S \leftarrow S'$

直到 S 是终止状态

规则（式 1.15）更新的是一个“状态-动作”二元组，因此顶部节点，也即更新过程的根节点，必须是一个代表动作的小实心圆点。更新也都来自于动作节点，即在下一个状态的所有可能的动作中找到价值最大的那个。因此回溯图底部的节点就是所有这些可能的动作节点。

接下来给出一个 q-learning 算法[web q]实现的 agent 走到终点的实例。环境设置为“-----T”，其中 T 为终点。Agent 只能左右移动直到到达终点。

```
import pandas as pd
import random
import time

#首先设置参数
tanlan = 0.9 #设置即贪婪率
efficiency = 0.1 #学习效率
reward_decrease = 0.8 #奖励递减值

# 定义 agent 目前所处的状态（位置）和环境
states = range(6) #设置状态集 0-6 即 agent 距离终点 6 个步长
actions = ['left','right'] #设置动作集，一维只有两个方向，即向左和向右
rewards = [0, 0, 0, 0, 0, 1] #设置奖励集， 即第 6 位时走到终点那里时，才给 agent 奖励

Q_table = pd.DataFrame(data=[[0 for _ in actions] for _ in states],index = states,columns = actions)# 创建一个表格
# DataFrame 函数创建一个表格，index 为行，这里用状态集赋值；columns 为列，这里用动作集赋值；Q 为某一时刻下状态 state 中采取动作 action 能获得收益的期望，即行为值



# 定义环境更新函数，实时更新且打印状态
    def updata_environment(state): #
        global states #状态设为全局变量，可以整个环境内引用
```

```
    environment = list('-----T') #环境设为一个 6 字符长的字符串
    if state != states[-1]: #如果不在最后一个位置
        environment[state] = 'u' #确定 agent 所处的位置
    print('\r{}'.format(''.join(environment)), end='')#打印
    time.sleep(0.1)#调用线程推迟执行该函数 0.1S
#agent 动作之后，定义下一状态函数
def get_next_state(state, action):
    global states
    # left,right,none = -1,+1,0
    if action == 'right' and state != states[-1]: #如果 agent 不在最后一个位置，则向终点移动一位
        next_state = state +1
    elif action == 'left' and state != states[0]: #如果 agent 不在最开始的一个位置，则远离终点移动一位(elif 表示否则如果,如果 if 步骤执行成功，那么 elif 步骤就不会执行）
        next_state = state -1    #如果 if 和 elif 都判断失败，则执行 else 语句
    else:
        next_state = state #否则+0，就是 agent 没动的意思
    return next_state #把下一状态值返回带出
#定义当前状态下合法的动作集合函数
def get_vaild_actions(state):
    global actions
    vaild_actions = set(actions)
    if state == states[-1]: #  如果 agent 在最后一个位置
        vaild_actions -= set(['right'])    #则不能再向右了
    if state == states[0]: #如果 agent 在最初的位置
        vaild_actions -= set(['left'])    #则不能再向左了
    return list(vaild_actions) #把当前状态值返回带出为一个列表格式

for i in range(13):    #i 在 0-13 内依次取值，这里的 i 是 episode，即从 action 开始到结束的一个过程
    current_state = 0 #设置 Q 估计表内的状态位置为 0
    # current_state = random.choice(states)
    updata_environment(current_state) #与绑定环境相关
    total_steps = 0

    while current_state != states[-1]:#  创建一个循环，直到 agent 到达最后一个位置，这里是重复状态动作的执行
        if (random.uniform(0,1) > tanlan) or ((Q_table.loc[current_state] == 0).all()): #random.uniform(0,1)表示在 0-1 内随机生成一个随机数，loc 通过行/列标签索引到状态矩阵，（iloc 通过行/列号索引矩阵这里没用到），all()表示索引到的状态矩阵与 0 的比对结果再作一次与运算
            current_action = random.choice(get_vaild_actions(current_state)) #random.choice 可以从定义（）里随机选取内容，并将选取结果放入赋值中返回
            #  该 if 语句意思为：如果 agent 的贪婪小于这个随机数或者索引到的估计状态为 0，则 agent 目前探索到的状态是正确的行动
```

```
            else:
                current_action = Q_table.loc[current_state].idxmax()#否则利用 agent 的渴望逼近正确的行动,idmax（）表示索引最大值
            next_state = get_next_state(current_state,current_action) #调用 agent 下一动作后的状态函数，与估计到的正确状态和动作绑定
            next_state_Q_values = Q_table.loc[next_state,get_vaild_actions(next_state)] # 得到现实 Q
            Q_table.loc[current_state, current_action] += efficiency*(rewards[next_state] + reward_decrease*next_state_Q_values.max() - Q_table.loc[current_state,current_action]) #根据贝尔曼方程更新 Q，Q 为某一时刻下状态 state 中采取动作 action 能获得收益的期望，Q_table.loc[current_state,current_action]为估计的 Q
            current_state = next_state
            updata_environment(current_state) #更新环境
            total_steps += 1 #总步骤加 1
        print('\rEpisode {}: total_steps = {}'.format(i, total_steps), end='')
        time.sleep(2)
        print('\r                                                                ',end='')

print('\nQ_table:')
print(Q_table)

 ‘，’
运行以上程序可以得到输出:
Episode{1,2,3,4,5,6,7,8,9,10,11,12,13}：total_steps={43,13,13,9,5,5,5,5,5,5,5,5,5}
Q_table:
    left       right
0    0.0    0.002646
1    0.0    0.017490
2    0.0    0.085685
```

习题

1. 什么是强化学习？
2. 强化学习和监督学习、无监督学习的区别是什么？
3. 强化学习适合解决什么样子的问题？

参考文献

Conti, E. , Madhavan, V. , Such, F. P. , Lehman, J. , & Clune, J. . (2017). Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents.

Sutton, R. , & Barto, A. . (1998). Reinforcement Learning:An Introduction. MIT Press.

Watkins, C. J. C. H. (1989). Learning from delayed rewards.

web q.https://blog.csdn.net/weixin_45776027/article/details/10645805

第 5 章　深度学习

人工神经网络为许多问题的研究提供了新的思路，特别是迅速发展的深度学习，能够发现高维数据中的复杂结构，取得比传统机器学习方法更好的结果，在图像识别、语音识别、机器视觉、自然语言理解等领域获得成功应用，解决了人工智能界很多年没有进展的问题。本章介绍深度学习及其应用。

5　深度学习

5.1　深度学习基本概念：

深度学习是为了解决表示学习难题而被提出的。本节介绍深度学习相关的基本概念，主要包括表示在学习、深度学习、端到端学习、多任务学习、迁移学习、深度神经网络、多层感知机和激活函数。

表示学习（representation learning）

机器学习旨在自动地学到从数据的表示（representation）到数据的标记（label）的映射。随着机器学习算法的日趋成熟，人们发现，在某些领域（如图像、语音、文本等），如何从数据中提取合适的表示成为整个任务的瓶颈所在，而数据表示的好坏直接影响后续学习任务（所谓 garbage in，garbage out）。与其依赖人类专家设计手工特征（难设计还不见得好用），表示学习希望能从数据中自动地学到从数据的原始形式到数据的表示之间的映射。

深度学习（deep learning，DL）

表示学习的理想很丰满，但实际中人们发现从数据的原始形式直接学得数据表示这件事很难。深度学习是目前最成功的表示学习方法，因此，目前国际表示学习大会（ICLR）的绝大部分论文都是关于深度学习的。深度学习是把表示学习的任务划分成几个小目标，先从数据的原始形式中先学习比较低级的表示，再从低级表示学得比较高级的表示。这样，每个小目标比较容易达到，综合起来我们就完成表示学习的任务。这类似于算法设计思想中的分治法（divide-and-conquer）。

迁移学习（transfer learning）

深度学习下的迁移学习旨在利用源任务数据辅助目标任务数据下的学习。迁移学习适用于源任务数据比目标任务数据多，并且源任务中学习得到的低层特征可以帮助目标任务的学习的情形。在计算机视觉领域，最常用的源任务数据是 ImageNet。对 ImageNet 预训练模型的利用通常有两种方式。1.固定特征提取器。用 ImageNet 预训练模型提取目标任务数据的高层特征。2.微调（fine-tuning）。以 ImageNet 预训练模型作为目标任务模型的初始化初始化权值，之后在目标任务数据上进行微调。

多任务学习（multi-task learning）

与其针对每个任务训练一个小网络，深度学习下的多任务学习旨在训练一个大网络以同时完成全部任务。这些任务中用于提取低层特征的层是共享的，之后产生分支，各任务拥有各自的若干层用于完成其任务。多任务学习适用于多个任务共享低层特征，并且各个任务的数据很相似的情况。

端到端学习（end-to-end learning）

深度学习下的端到端学习旨在通过一个深度神经网络直接学习从数据的原始形式到数据的标记的映射。端到端学习并不应该作为我们的一个追求目标，是否要采用端到端学习的一个重要考虑因素是：有没有足够的数据对应端到端的过程，以及我们有没有一些领域知识能够用于整个系统中的一些模块。

深度神经网络（deep neural networks，DNN）

深度学习目前几乎唯一行之有效的实现形式。简单的说，深度神经网络就是很深的神经网络。我们利用网络中逐层对特征进行加工的特性，逐渐从低级特征提取高级特征。除了深度神经网络之外，有学者在探索其他深度学习的实现形式，比如深度森林。

深度神经网络目前的成功取决于三大推动因素。1.大数据。当数据量小时，很难从数据中学得合适的表

示，而传统算法+特征工程往往能取得很好的效果；2.计算能力。大的数据和大的网络需要有足够的快的计算能力才能使得模型的应用成为可能。3.算法创新。现在很多算法设计关注在如何使网络更好地训练、更快地运行、取得更好的性能。

多层感知机（multi-layer perceptrons，MLP）

多层由全连接层组成的深度神经网络。多层感知机的最后一层全连接层实质上是一个线性分类器，而其他部分则是为这个线性分类器学习一个合适的数据表示，使倒数第二层的特征线性可分。

激活函数（activation function）

激活函数是神经网络的必要组成部分。如果没有激活函数，多次线性运算的堆叠仍然是一个线性运算，即不管用再多层，实质只起到了一层神经网络的作用。一个好的激活函数应满足以下性质。1.不会饱和。sigmoid 和 tanh 激活函数在两侧尾端会有饱和现象，这会使导数在这些区域接近零，从而阻碍网络的训练。2.零均值。ReLU 激活函数的输出均值不为零，这会影响网络的训练 。3.容易计算。

5.2 深度神经网络：

人工神经网络（Artificial Neural Network，ANN）是指一系列受生物学和神经科学启发的数学模型。这些模型主要是通过对人脑的神经元网络进行抽象，构建人工神经元，并按照一定拓扑结构来建立人工神经元之间的连接，来模拟生物神经网络。在人工智能领域，人工神经网络也常常简称为神经网络（Neural Network，NN）或神经模型（Neural Model）。

神经网络最早是作为一种主要的连接主义模型。20 世纪 80 年代中后期，最流行的一种连接主义模型是分布式并行处理（Parallel Distributed Processing，PDP）模型[McClelland et al.,1986]，其有 3 个主要特性：1）信息表示是分布式的（非局部的）；2）记忆和知识是存储在单元之间的连接上；3）通过逐渐改变单元之间的连接强度来学习新的知识。

连接主义的神经网络有着多种多样的网络结构以及学习方法，虽然早期模型强调模型的生物学合理性（Biological Plausibility），但后期更关注对某种特定认知能力的模拟，比如物体识别、语言理解等。尤其在引入误差反向传播来改进其学习能力之后，神经网络也越来越多地应用在各种机器学习任务上。随着训练数据的增多以及（并行）计算能力的增强，神经网络在很多机器学习任务上已经取得了很大的突破，特别是在语音、图像等感知信号的处理上，神经网络表现出了卓越的学习能力。

5.2.1 神经元：

生物学家在 20 世纪初就发现了生物神经元的结构。一个生物神经元通常具有多个树突和一条轴突。树突用来接收信息，轴突用来发送信息。当神经元所获得的输入信号的积累超过某个阈值时，它就处于兴奋状态，产生电脉冲。轴突尾端有许多末梢可以给其他神经元的树突产生连接（突触），并将电脉冲信号传递给其他神经元。

1943 年，心理学家 McCulloch 和数学家 Pitts 根据生物神经元的结构，提出了一种非常简单的神经元模型，MP 神经元[McCullochetal.,1943]。现代神经网络中的神经元和 MP 神经元的结构并无太多变化。不同的是，MP 神经元中的激活函数f为 0 或 1 的阶跃函数，而现代神经元中的激活函数通常要求是连续可导的函数。

假设一个神经元接收D个输入，$x_1, x_2, \cdots x_D$令向量$x = [x_1; x_2; \cdots; x_D]$来表示这组输入，并用净输入（Net Input）$z \in \mathcal{R}$表示一个神经元所获得的输入信号$x$的加权和，

$$\begin{aligned} z &= \sum_{d=1}^{D} w_d x_d + b \\ &= \omega^T x + b \end{aligned} \tag{5.1}$$

其中$\omega = [\omega_1; \omega_2; \cdots; \omega_D] \in R^D$是$D$维的权重向量，$b \in R$是偏置。

净输入z在经过一个非线性函数$f(\cdot)$后，得到神经元的活性值（Activation）a:

$$a = f(z) \tag{5.2}$$

其中非线性函数$f(\cdot)$称为激活函数（Activation Function）。神经元的典型结构如图 5.1 所示，不同的输入与不同的权重计算，得到线性映射结果后，通过激活函数得到非线性映射的结果。

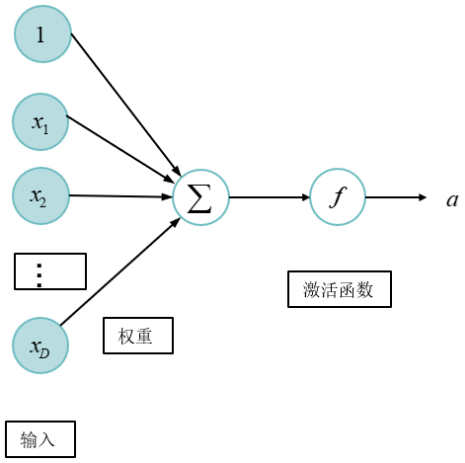


图 5.1 一个典型的神经元结构示例

5.2.2神经网络：

一个生物神经细胞的功能比较简单，而人工神经元只是生物神经细胞的理想化和简单实现，功能更加简单。要想模拟人脑的能力，单一的神经元是远远不够的，需要通过很多神经元一起协作来完成复杂的功能。这样通过一定的连接方式或信息传递方式进行协作的神经元可以看作一个网络，就是神经网络。到目前为止，研究者已经发明了各种各样的神经网络结构。目前常用的神经网络结构有以下三种：

（1）前馈神经网络：

前馈网络中各个神经元按接收信息的先后分为不同的组。每一组可以看作一个神经层。每一层中的神经元接收前一层神经元的输出，并输出到下一层神经元。整个网络中的信息是朝一个方向传播，没有反向的信息传播，可以用一个有向无环路图表示。前馈网络包括全连接前馈网络和卷积神经网络等。前馈网络可以看作一个函数，通过简单非线性函数的多次复合，实现输入空间到输出空间的复杂映射。这种网络结构简单，易于实现。

前馈神经网络（Feed forward NeuralNetwork，FNN）是最早发明的简单人工神经网络。前馈神经网络也经常称为多层感知器（Multi-Layer Perceptron，MLP）。但多层感知器的叫法并不是十分合理，因为前馈神经网络其实是由多层的 Logistic 回归模型（连续的非线性函数）组成，而不是由多层的感知器（不连续的非线性函数）组成[Bishop,2007]。

在前馈神经网络中，各神经元分别属于不同的层。每一层的神经元可以接收前一层神经元的信号，并产生信号输出到下一层。第 0 层称为输入层，最后一层称为输出层，其他中间层称为隐藏层。整个网络中无反馈，信号从输入层向输出层单向传播，可用一个有向无环图表示。图 5.2 给出了前馈神经网络的示例。

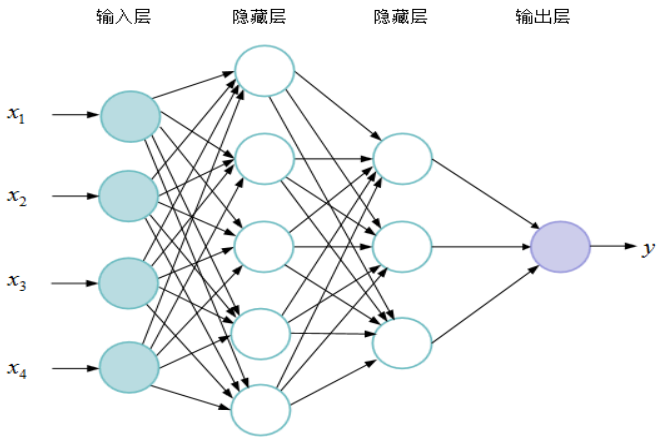


图 5.2 多层前馈神经网络

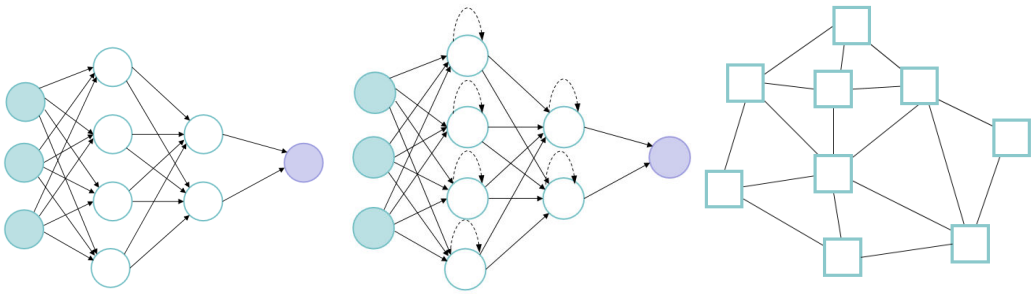
（2）记忆网络

记忆网络，也称为反馈网络，网络中的神经元不但可以接收其他神经元的信息，也可以接收自己的历史信息。和前馈网络相比，记忆网络中的神经元具有记忆功能，在不同的时刻具有不同的状态。记忆神经网络中的信息传播可以是单向或双向传递，因此可用一个有向循环图或无向图来表示。记忆网络包括循环神经网络、Hopfield 网络、玻尔兹曼机、受限玻尔兹曼机等。

记忆网络可以看作一个程序，具有更强的计算和记忆能力。为了增强记忆网络的记忆容量，可以引入外部记忆单元和读写机制，用来保存一些网络的中间状态，称为记忆增强神经网络（Memory Augmented Neural Network，MANN），比如神经图灵机[Graves et al.,2014]和记忆网络[Sukhbaatar et al.,2015]等。

（3）图网络

前馈网络和记忆网络的输入都可以表示为向量或向量序列。但实际应用中很多数据是图结构的数据，比如知识图谱、社交网络、分子（Molecular）网络等前馈网络和记忆网络很难处理图结构的数据。图网络是定义在图结构数据上的神经网络。图中每个节点都由一个或一组神经元构成。节点之间的连接可以是有向的，也可以是无向的。每个节点可以收到来自相邻节点或自身的信息。图网络是前馈网络和记忆网络的泛化，包含很多不同的实现方式，比如图卷积网络（Graph Convolutional Network，GCN）[Kipf et al.,2016]、图注意力网络（Graph AttentionNetwork，GAT）[Veličković et al.,2017]、消息传递神经网络（Message Passing Neural Network，MPNN）[Gilmer et al.,2017]等。图 5.3 给出了前馈网络、记忆网络和图网络的网络结构示例，其中圆形节点表示一个神经元，方形节点表示一组神经元。



(a)前馈网络　　(b)记忆网络　　(c)图网络

图 5.3 三种不同的网络结构示例

5.2.3激活函数：

激活函数在神经元中非常重要的。为了增强网络的表示能力和学习能力，激活函数需要具备以下几点性质：（1）连续并可导（允许少数点上不可导）的非线性函数。可导的激活函数可以直接利用数值优化的方法来学习网络参数。（2）激活函数及其导函数要尽可能的简单，有利于提高网络计算效率。（3）激活函数的导函数的值域要在一个合适的区间内，不能太大也不能太小，否则会影响训练的效率和稳定性。下面介绍几种在神经网络中常用的激活函数。

Sigmoid 型函数：

Sigmoid 型函数是指一类 S 型曲线函数，为两端饱和函数。常用的有 Logistic 函数和 Tanh 函数。

Logistic 函数：

Logistic 函数定义为

$$\sigma(x) = \frac{1}{1+exp(-x)} \tag{5.3}$$

Logistic 函数可以看成是一个“挤压”函数，把一个实数域的输入“挤压”到(0,1)。当输入值在 0 附近时，Sigmoid 型函数近似为线性函数；当输入值靠近两端时，对输入进行抑制。输入越小，越接近于 0；输入越大，越接近于 1。这样的特点也和生物神经元类似，对一些输入会产生兴奋（输出为 1），对另一些输入产生抑制（输出为 0）。和感知器使用的阶跃激活函数相比，Logistic 函数是连续可导的，其数学性质更好。

因为 Logistic 函数的性质，使得装备了 Logistic 激活函数的神经元具有以下两点性质：

1）其输出直接可以看作概率分布，使得神经网络可以更好地和统计学习模型进行结合。

2）其可以看作一个软性门（Soft Gate），用来控制其他神经元输出信息的数量。

Tanh 函数

Tanh 函数也是一种 Sigmoid 型函数。其定义为

$$\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)} \tag{5.4}$$

Tanh 函数可以看作放大并平移的 Logistic 函数，其值域是$(-1,1)$。

$$tanh(x) = 2\sigma(2x) - 1 \tag{5.5}$$

给出了 Logistic 函数和 Tanh 函数的形状。Tanh 函数的输出是零中心化的（Zero-Centered），而 Logistic 函数的输出恒大于 0。非零中心化的输出会使得其后一层的神经元的输入发生偏置偏移（BiasShift），并进一步使得梯度下降的收敛速度变慢。

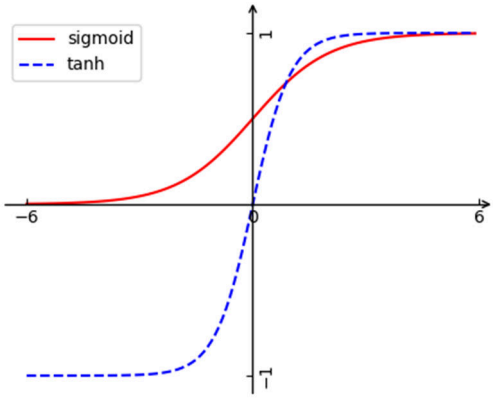


图 5.4 sigmoid 函数和 tanh 函数

ReLU 函数

ReLU（Rectified Linear Unit，修正线性单元）[Nair et al.,2010]，也叫 Rectifier 函数[Glorot et al.,2011]，是目前深度神经网络中经常使用的激活函数。ReLU 实际上是一个斜坡（ramp）函数，定义为

$$ReLU(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{5.6}$$

$$= max(0, x) \tag{5.7}$$

ReLU 的优点

采用 ReLU 的神经元只需要进行加、乘和比较的操作，计算上更加高效。ReLU 函数也被认为具有生物学合理性（Biological Plausibility），比如单侧抑制、宽兴奋边界（即兴奋程度可以非常高）。在生物神经网络中，同时处于兴奋状态的神经元非常稀疏。人脑中在同一时刻大概只有 1%~4%的神经元处于活跃状态。Sigmoid 型激活函数会导致一个非稀疏的神经网络，而 ReLU 却具有很好的稀疏性，大约 50%的神经元会处于激活状态。在优化方面，相比于 Sigmoid 型函数的两端饱和，ReLU 函数为左饱和函数，且在$x > 0$时导数为 1，在一定程度上缓解了神经网络的梯度消失问题，加速梯度下降的收敛速度。

ReLU 的缺点

ReLU 函数的输出是非零中心化的，给后一层的神经网络引入偏置偏移，会影响梯度下降的效率。ReLU 神经元指采用ReLU作为激活函数的神经元。此外，ReLU 神经元在训练时比较容易“死亡”。在训练时，如果参数在一次不恰当的更新后，第一个隐藏层中的某个 ReLU 神经元在所有的训练数据上都不能被激活，那么这个神经元自身参数的梯度永远都会是 0。在以后的训练过程中永远不能被激活。这种现象称为死亡 ReLU 问题（Dying ReLU Problem），并且也有可能会发生在其他隐藏层。在实际使用中，为了避免上述情况，有

几种 ReLU 的变种也会被广泛使用:

带泄露的 ReLU 函数

带泄露的 ReLU（LeakyReLU）在输入$\boldsymbol{x<0}$时，保持一个很小的梯度λ。这样当神经元非激活时也能有一个非零的梯度可以更新参数，避免永远不能被激活(Maas et al.,2013)。带泄露的 ReLU 的定义如下：

$$\begin{aligned}\text{LeakyReLU}(\mathbf{x}) &= \begin{cases}\mathbf{x} & \text{if } \mathbf{x}>\mathbf{0}\\ \boldsymbol{\gamma}\mathbf{x} & \text{if } \mathbf{x}\le\mathbf{0}\end{cases}\\ &= \mathbf{max}(\mathbf{0},\mathbf{x})+\boldsymbol{\gamma}\mathbf{min}(\mathbf{0},\mathbf{x})\end{aligned} \tag{5.8}$$

其中$\boldsymbol{\gamma}$是一个很小的常数，比如 0.01。当$\boldsymbol{\gamma<1}$时，带泄露的 ReLU 也可以写为

$$\text{LeakyReLU}(x)=\max(x,\gamma x) \tag{5.9}$$

相当于是一个比较简单的 maxout 单元。

带参数的 ReLU 函数

带参数的 ReLU（Parametric ReLU，PReLU）引入一个可学习的参数，不同神经元可以有不同的参数[He et al.,2015]。对于第i个神经元，其 PReLU 的定义为

$$\begin{aligned}\text{PReLU}_\mathbf{i}(\mathbf{x}) &= \begin{cases}\mathbf{x} & \text{if } \mathbf{x}>\mathbf{0}\\ \boldsymbol{\gamma}_\mathbf{i}\mathbf{x} & \text{if } \mathbf{x}\le\mathbf{0}\end{cases}\\ &= \mathbf{max}(\mathbf{0},\mathbf{x})+\boldsymbol{\gamma}_\mathbf{i}\mathbf{min}(\mathbf{0},\mathbf{x})\end{aligned} \tag{5.10}$$

其中 $\boldsymbol{\gamma_i}$ 为 $\boldsymbol{x \le 0}$ 时函数的斜率。因此，PReLU 是非饱和函数。如果$\boldsymbol{\gamma_i=0}$，那么 PReLU 就退化为 ReLU。如果$\boldsymbol{\gamma_i}$ 为一个很小的常数，则 PReLU 可以看作带泄露的 ReLU。PReLU 可以允许不同神经元具有不同的参数，也可以一组神经元共享一个参数。

ELU 函数

ELU（Exponential Linear Unit，指数线性单元）[Clevert et al., 2015] 是一个近似的零中心化的非线性函数，其定义为

$$\begin{aligned}\text{ELU(x)} &= \begin{cases}\text{x} & \text{if } \text{x}>0\\ \gamma(\exp(\text{x})-1) & \text{if } \text{x}\le 0\end{cases}\\ &= \max(0,\text{x})+\min(0,\gamma(\exp(\text{x})-1))\end{aligned} \tag{5.11}$$

其中 $\boldsymbol{\gamma \ge 0}$是一个超参数，决定$\boldsymbol{x \le 0}$时的饱和曲线，并调整输出均值在 0 附近。

Softplus 函数

Softplus 函数(Dugas et al.,2001)可以看作 Rectifier 函数的平滑版本，其定义为

$$Softplus(x)=log(1+exp(x)) \tag{5.12}$$

Softplus 函数其导数是 Logistic 函数。Softplus 函数虽然也具有单侧抑制、宽兴奋边界的特性，却没有稀疏激活性。图 5.5 给出了 ReLU、LeakyReLU、ELU 以及 Softplus 函数的示例。

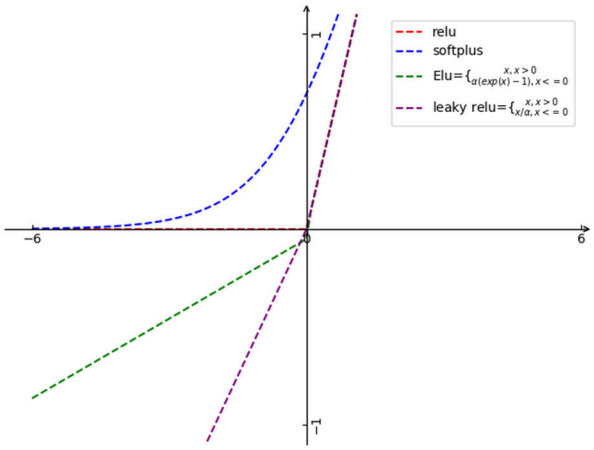


图 5.5 ReLU、LeakyReLU、ELU 及 Softplus 函数的示例

5.2.4参数初始化：

神经网络的参数学习是一个非凸优化问题。当使用梯度下降法来进行优化网络参数时，参数初始值

的选取十分关键，关系到网络的优化效率和泛化能力。参数初始化的方式通常有以下三种：

(1)预训练初始化：

不同的参数初始值会收敛到不同的局部最优解。虽然这些局部最优解在训练集上的损失比较接近，但是它们的泛化能力差异很大。一个好的初始值会使得网络收敛到一个泛化能力高的局部最优解。通常情况下，一个已经在大规模数据上训练过的模型可以提供一个好的参数初始值，这种初始化方法称为预训练初始化（Pre-trained Initialization）。预训练任务可以为监督学习或无监督学习任务。由于无监督学习任务更容易获取大规模的训练数据，因此被广泛采用。预训练初始化通常会提升模型泛化能力的一种解释是预训练任务起到正则化作用。预训练模型在目标任务上的学习过程也称为精调（Fine-Tuning）。

(2)随机初始化：

在线性模型的训练（比如感知器和 Logistic 回归）中，一般将参数全部初始化为 0。但是这在神经网络的训练中会存在一些问题。因为如果参数都为 0，在第一遍前向计算时，所有的隐藏层神经元的激活值都相同；在反向传播时，所有权重的更新也都相同，这样会导致隐藏层神经元没有区分性。这种现象也称为对称权重现象。为了打破这个平衡，比较好的方式是对每个参数都随机初始化（Random Initialization），使得不同神经元之间的区分性更好。

(3)固定值初始化：

对于一些特殊的参数，我们可以根据经验用一个特殊的固定值来进行初始化。比如偏置（Bias）通常用 0 来初始化，但是有时可以设置某些经验值以提高优化效率。在 LSTM 网络的遗忘门中，偏置通常初始化为 1 或 2，使得时序上的梯度变大。对于使用 ReLU 的神经元，有时也可以将偏置设为 0.01，使得 ReLU 神经元在训练初期更容易激活，从而获得一定的梯度来进行误差反向传播。

虽然预训练初始化通常具有更好的收敛性和泛化性，但是灵活性不够，不能在目标任务上任意地调整网络结构。因此，好的随机初始化方法对训练神经网络模型来说依然十分重要。随机初始化通常只应用在神经网络的权重矩阵上。这里我们介绍三类常用的随机初始化方法：基于固定方差的参数初始化、基于方差缩放的参数初始化和正交初始化方法。

基于固定方差的参数初始化

一种最简单的随机初始化方法是从一个固定均值（通常为 0）和方差σ^2的分布中采样来生成参数的初始值。基于固定方差的参数初始化方法主要有以下两种：

（1） 高斯分布初始化：使用一个高斯分布$\mathcal{N}(0,\sigma^2)$对每个参数进行随机初始化。

（2） 均匀分布初始化：在一个给定的区间$[-r,r]$内采用均匀分布来初始化参数。假设随机变量x在区间$[a,b]$内均匀分布，则其方差为

$$\mathrm{var}(x)=\frac{(b-a)^2}{12} \tag{5.13}$$

因此，若使用区间为 $[-r,r]$ 的均分分布来采样，并满足$var(x)=\sigma^2$时，则r的取值为

$$r=\sqrt{3\sigma^2} \tag{5.14}$$

在基于固定方差的随机初始化方法中，比较关键的是如何设置方差σ^2。如果参数范围取的太小，一是会导致神经元的输出过小，经过多层之后信号就慢慢消失了；二是还会使得 Sigmoid 型激活函数丢失非线性的能力。以 Sigmoid 型函数为例，在 0 附近基本上是近似线性的。这样多层神经网络的优势也就不存在了。如果参数范围取的太大，会导致输入状态过大。对于 Sigmoid 型激活函数来说，激活值变得饱和，梯度接近于 0，从而导致梯度消失问题。

为了降低固定方差对网络性能以及优化效率的影响，基于固定方差的随机初始化方法一般需要配合逐层归一化来使用

基于方差缩放的参数初始化

要高效地训练神经网络，给参数选取一个合适的随机初始化区间是非常重要的。一般而言，参数初始化的区间应该根据神经元的性质进行差异化的设置。如果一个神经元的输入连接很多，它的每个输入连接上的权重就应该小一些，以避免神经元的输出过大（当激活函数为 ReLU 时）或过饱和（当激活函数为 Sigmoid 函数时）。

初始化一个深度网络时，为了缓解梯度消失或爆炸问题，我们尽可能保持每个神经元的输入和输出的方差一致，根据神经元的连接数量来自适应地调整初始化分布的方差，这类方法称为方差缩放（Variance Scaling）。

He 初始化

当第l层神经元使用 ReLU 激活函数时，通常有一半的神经元输出为 0，因此其分布的方差也近似为使用恒等函数时的一半。这样，只考虑前向传播时，参数$w(l)i$的理想方差为

$$\mathrm{var}\left(w_i^{(l)}\right) = \frac{2}{M_{l-1}} \tag{5.15}$$

其中M_{l-1} 是第$l-1$层神经元个数。

因此当使用 ReLU 激活函数时，若采用高斯分布来初始化参数 $w_i^{(l)}$，其方差为$\frac{2}{M_{l-1}}$ ；若采用区间为 $[-r,r]$ 的均匀分布来初始化参数$w_i^{(l)}$ ，则$r=\sqrt{\frac{6}{M_{l-1}}}$。这种初始化方法称为 He 初始化[He et al., 2015]。

正交初始化

上面介绍的两种基于方差的初始化方法都是对权重矩阵中的每个参数进行独立采样。由于采样的随机性，采样出来的权重矩阵依然可能存在梯度消失或梯度爆炸问题。

假设一个L层的等宽线性网络（激活函数为恒等函数）为

$$y = W^{(L)}W^{(L-1)}\cdots W^{(1)}x \tag{5.16}$$

其中$W^{(l)} \in \mathbb{R}^{M\times M}(1 \le l \le L)$ 为神经网络的第 l 层权重矩阵。在反向传播中，误差项δ 的反向传播公式为$\delta^{(l-1)} = \left(W^{(l)}\right)^{\mathrm{T}}\delta^{(l)}$。为了避免梯度消失或梯度爆炸问题，我们希望误差项在反向传播中具有范数保持性（Norm-Preserving），即

$$\left\|\delta^{(l-1)}\right\|^2 = \left\|\delta^{(l)}\right\|^2 = \left\|\left(W^{(l)}\right)^{\mathrm{T}}\delta^{(l)}\right\|^2 \tag{5.17}$$

如果我们以均值为 0、方差为$\frac{1}{M}$ 的高斯分 布来随机生成权重矩阵$W^{(l)}$中每个元素的初始值，那么当$M \to \infty$时，范数保持性成立。但是当M不足够大时，这种对每个参数进行独立采样的初始化方式难以保证范数保持性。

因此，一种更加直接的方式是将$W^{(l)}$初始化为正交矩阵，即

$$W^{(l)}\left(W^{(l)}\right)^{\mathrm{T}} = I \tag{5.18}$$

这种方法称为正交初始化（Orthogonal Initialization）[Saxe et al., 2014]。正交初始化的具体实现过程可以分为两步：

1）用均值为 0、方差为 1 的高斯分布初始化一个矩阵；

2）将这个矩阵用奇异值分解得到两个正交矩阵，并使用其中之一作为权重矩阵。

根据正交矩阵的性质，这个线性网络在信息的前向传播过程和误差的反向传播过程中都具有范数保持性，从而可以避免在训练开始时就出现梯度消失或梯度爆炸现象。

当在非线性神经网络中应用正交初始化时，通常需要将正交矩阵乘以一个缩放系数ρ。比如当激活函数为 ReLU 时，激活函数在 0 附近的平均梯度可以近似为 0.5。为了保持范数不变，缩放系数ρ可以设置为$\sqrt{2}$。正交初始化通常用在循环神经网络中循环边上的权重矩阵上。

5.2.5反向传播算法：

本节我们将使用数学来描述正向传播和反向传播。具体来说，我们将以带 L2 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。

正向传播：

正向传播（forward propagation）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 ，假设输入是一个特征为$x \in R^d$的样本，且不考虑偏差项，那么中间变量

$$z = W^{(1)}x \tag{5.19}$$

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

$$h = \varphi(z) \tag{5.20}$$

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

$$o = W^{(2)}h \tag{5.21}$$

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

$$L = l(o, y) \tag{5.22}$$

根据$L2$范数正则化的定义，给定超参数λ，正则化项即,

$$s = \frac{\lambda}{2}(\|W^{(1)}\|_F^2 + \|W^{(2)}\|_F^2) \tag{5.23}$$

其中矩阵的 Frobenius 范数等价于将矩阵变平为向量后计算$L2$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。

我们通常绘制计算图（computational graph）来可视化运算符和变量在计算中的依赖关系。图 5.6 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。

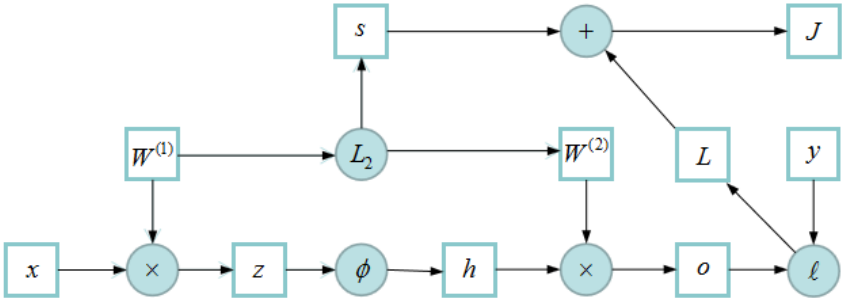


图 5.6 正向传播的计算图

反向传播：

假设采用随机梯度下降进行神经网络参数学习，给定一个样本(x, y)，将其输入到神经网络模型中，得到网络输出为$\hat{y}$。 假设损失函数为$l(y, \hat{y})$，要进行参数学习就需要计算损失函数关于每个参数的导数。在计算出每一层的误差项之后，我们就可以得到每一层参数的梯度。

因此，使用误差反向传播算法的前馈神经网络训练过程可以分为以下三步：

（1）前馈计算每一层的净输入$z^{(l)}$和激活值$a^{(l)}$，直到最后一层；

（2）反向传播计算每一层的误差项$\delta^{(l)}$；

（3）计算每一层参数的偏导数，并更新参数。

算法 5.1 给出使用反向传播算法的随机梯度下降训练过程：

算法 5.1 使用反向传播算法的随机梯度下降训练过程

Input:训练集$\mathcal{D}=\{x^{(n)},y^{(n)}\}_{n=1}^{N}$，验证集$\mathcal{V}$，学习率$\alpha$，正则化系数$\lambda$，网络层数$L$，神经元数量$M_l,1\le l\le L$.

随机初始化W,b;

repeat

对训练集$\mathcal{D}$中的样本随机重排序；

For $n=1\cdots N$ **do**

从训练集$\mathcal{D}$中选取样本$(x^{(n)},y^{(n)})$

前馈计算每一层的净输入$z^{(l)}$和激活值，直到最后一层；

反向传播计算每一层的误差

//计算每一层参数的导数

$$\forall l,\frac{\partial\mathcal{L}\left(y^{(n)},\hat{y}^{(n)}\right)}{\partial W^{(l)}}=\delta^{(l)}\left(a^{(l-1)}\right)^{\mathrm{T}};$$

$$\forall l,\frac{\partial\mathcal{L}\left(y^{(n)},\hat{y}^{(n)}\right)}{\partial b^{(l)}}=\delta^{(l)};$$

//更新参数

$$W^{(l)}\leftarrow W^{(l)}-\alpha\left(\delta^{(l)}\left(a^{(l-1)}\right)^{\mathrm{T}}+\lambda W^{(l)}\right);$$

$$b^{(l)}\leftarrow b^{(l)}-\alpha\delta^{(l)};$$

end

until 神经网络模型在验证集$\mathcal{V}$上的错误率不再下降；

输出W,b;

具体来说，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

$$\frac{\partial \mathrm{Z}}{\partial \mathrm{X}}=\mathrm{prod}(\frac{\partial \mathrm{Z}}{\partial \mathrm{Y}},\frac{\partial \mathrm{Y}}{\partial \mathrm{X}}) \tag{5.24}$$

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$W^{(1)}$和$W^{(2)}$，因此反向传播的目标是计算 $\frac{\partial J}{\partial W^{(1)}}$和$\frac{\partial J}{\partial W^{(2)}}$。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

$$\frac{\partial J}{\partial L}=1,\ \frac{\partial J}{\partial s}=1 \tag{5.25}$$

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

$$\frac{\partial J}{\partial o}=\mathrm{prod}(\frac{\partial J}{\partial L},\frac{\partial L}{\partial o})=\frac{\partial L}{\partial o} \tag{5.26}$$

接下来，计算正则项有关两个参数的梯度：

$$\frac{\partial s}{\partial W^{(1)}}=\lambda W^{(1)},\ \frac{\partial s}{\partial W^{(2)}}=\lambda W^{(2)} \tag{5.27}$$

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

$$\frac{\partial J}{\partial W^{(2)}} = \text{prod}(\frac{\partial J}{\partial o}, \frac{\partial o}{\partial W^{(2)}}) + \text{prod}(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial W^{(2)}}) = \frac{\partial L}{\partial o} h^T + \lambda W^{(2)} \tag{5.28}$$

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

$$\frac{\partial J}{\partial h} = \text{prod}(\frac{\partial J}{\partial o}, \frac{\partial o}{\partial h}) = W^{(2)T} \frac{\partial J}{\partial o} \tag{5.29}$$

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

$$\frac{\partial J}{\partial z} = \text{prod}(\frac{\partial J}{\partial h}, \frac{\partial h}{\partial z}) = \frac{\partial J}{\partial h} \odot \phi'(z) \tag{5.30}$$

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

$$\frac{\partial J}{\partial W^{(1)}} = \text{prod}(\frac{\partial J}{\partial z}, \frac{\partial z}{\partial W^{(1)}}) + \text{prod}(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial W^{(1)}}) = \frac{\partial J}{\partial z} x^T + \lambda W^{(1)} \tag{5.31}$$

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。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$s = \frac{\lambda}{2}(\|W^{(1)}\|_F^2 + \|W^{(2)}\|_F^2)$依赖模型参数$W^{(1)}$和$W^{(2)}$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$\frac{\partial J}{\partial W^{(1)}} = \frac{\partial J}{\partial o} h^T + \lambda W^{(2)}$ 的计算需要依赖隐藏层变量的当前值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。

5.3 深度模型的优化：

在训练模型时，我们会使用优化算法不断迭代模型参数以降低模型损失函数的值。当迭代终止时，模型的训练随之终止，此时的模型参数就是模型通过训练所学习到的参数。优化算法对深度学习十分重要。一方面，训练一个复杂的深度学习模型可能需要数小时、数日，甚至数周时间，而优化算法的表现直接影响模型的训练效率；另一方面，理解各种优化算法的原理以及其中超参数的意义将有助于我们更有针对性地调参，从而使深度学习模型表现更好。

优化与深度学习的关系：

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。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。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。

优化在深度学习中的挑战

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。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。

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$f(x) = x \cdot cos(\pi x), -1.0 \leq x \leq 2.0$,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。

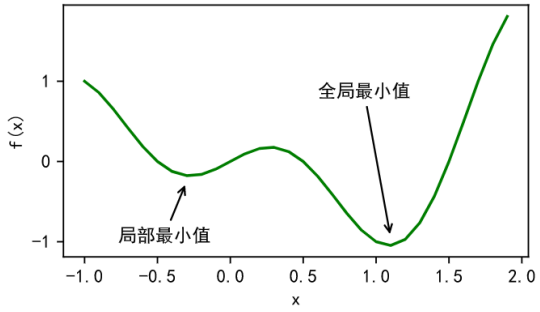


图 5.7 局部最小值示意图

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，由于目标函数解的梯度接近或变成 0，最终迭代的数值解可能只能令目标函数局部最小化而非全局最小化。

鞍点：

刚刚提到，梯度接近或变成 0 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$f(x) = x^3$。我们可找出该函数的鞍点位置

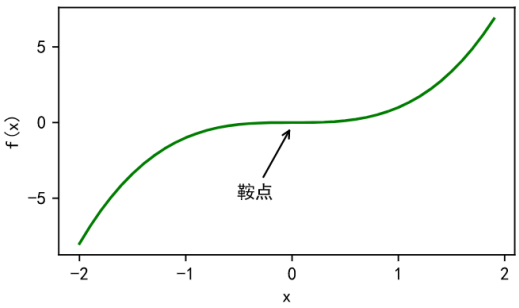


图 5.8 二维的鞍点示意图

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。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。

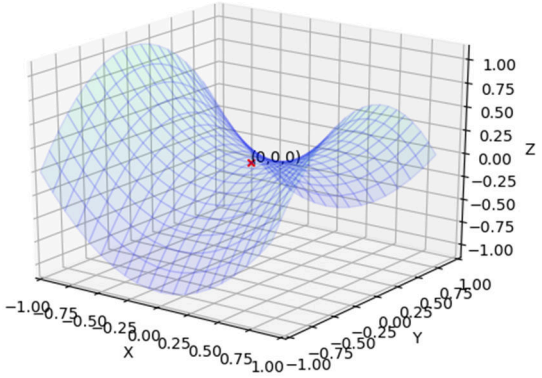


图 5.9 三维的鞍点示意图

5.3.1动态调整学习率

学习率是神经网络优化时的重要超参数。在梯度下降法中，学习率α的取值非常关键，如果过大就不会收敛，如果过小则收敛速度太慢。常用的学习率调整方法包括学习率衰减、学习率预热、周期性学习率调整以及一些自适应调整学习率的方法，比如 AdaGrad、RMSprop、AdaDelta 等。自适应学习率方法可以针对 每个参数设置不同的学习率。

学习率衰减：

从经验上看，学习率在一开始要保持大些来保证收敛速度，在收敛到最优点附近时要小些以避免来回振荡。比较简单的学习率调整可以通过学习率衰减（Learning Rate Decay）的方式来实现，也称为学习率退火（Learning Rate Annealing）。学习率衰减是按每次迭代（Iteration）进行，也可以按每m次迭代或每个回合（Epoch）进行。衰减率通常和总迭代次数相关。

不失一般性，这里的衰减方式设置为按迭代次数进行衰减。假设初始化学习率为$\alpha 0$，在第t次迭代时的学习率α_t。常见的衰减方法有以下几种：分段常数衰减（Piecewise Constant Decay）：即每经过$T_1, T_2, \dots, T_m$，次迭代将学习率衰减为原来的$\beta_1, \beta_2, \dots, \beta_m$倍，其中$T_m < 1, \beta_m < 1$为根据经验设的超参数。分段常数衰减也称为阶梯衰减（Step Decay）。

逆时衰减（Inverse Time Decay）：

$$\alpha_t = \alpha_0 \frac{1}{1+\beta \times t} \tag{5.32}$$

其中β为衰减率。

指数衰减（Exponential Decay）：

$$\boldsymbol{\alpha_t = \alpha_0 \beta^t} \tag{5.33}$$

其中β<1 为衰减率。

自然指数衰减（Natural Exponential Decay）：

$$\boldsymbol{\alpha_t = \alpha_0 \exp(-\beta \times t)} \tag{5.34}$$

其中β为衰减率。

余弦衰减（Cosine Decay）：

$$\boldsymbol{\alpha_t = \frac{1}{2}\alpha_0 (1 + cos(\frac{t\pi}{T}))} \tag{5.35}$$

其中T为总的迭代次数。

图 5.10 给出了不同衰减方法的示例（假设初始学习率为 1）。

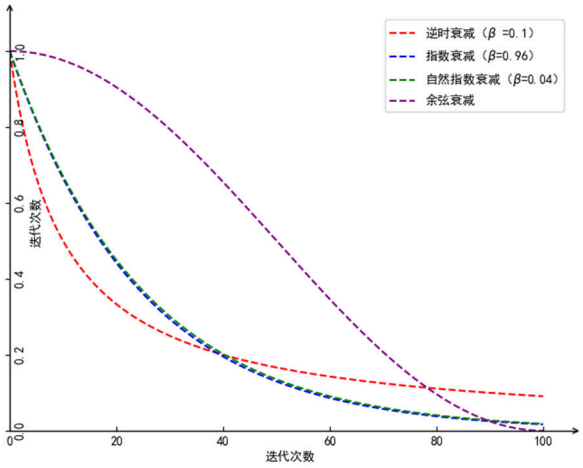


图 5.10 不同衰减方法示例

学习率预热：

在小批量梯度下降法中，当批量大小的设置比较大时，通常需要比较大的学习率。但在刚开始训练时，

由于参数是随机初始化的，梯度往往也比较大，再加上比较大的初始学习率，会使得训练不稳定。

为了提高训练稳定性，我们可以在最初几轮迭代时，采用比较小的学习率，等梯度下降到一定程度后再恢复到初始的学习率，这种方法称为学习率预热（Learning Rate Warmup）。一个常用的学习率预热方法是逐渐预热（Gradual Warmup）[Goyal et al.,2017]。假设预热的迭代次数为T'，初始学习率为α_0，在预热过程中，每次更新的学习率为

$$\alpha'_t = \frac{t}{T'}\alpha_0, \qquad 1 \le t \le T' \tag{5.36}$$

当预热过程结束，再选择一种学习率衰减方法来逐渐降低学习率。

周期性学习率调整：

为了使得梯度下降法能够逃离鞍点或尖锐最小值，一种经验性的方式是在训练过程中周期性地增大学习率。当参数处于尖锐最小值附近时，增大学习率有助于逃离尖锐最小值；当参数处于平坦最小值附近时，增大学习率依然有可能在该平坦最小值的吸引域（Basin of Attraction）内。因此，周期性地增大学习率虽然可能短期内损害优化过程，使得网络收敛的稳定性变差，但从长期来看有助于找到更好的局部最优解。

一种简单的方法是使用循环学习率（Cyclic Learning Rate）[Goyal et al.,2017]，即让学习率在一个区间内周期性地增大和缩小。通常可以使用线性缩放来调整学习率，称为三角循环学习率（Triangular Cyclic Learning Rate）。假设每个循环周期的长度相等都为$2\Delta T$，其中前ΔT步为学习率线性增大阶段，后ΔT步为学习率线性缩小阶段。在第t次迭代时，其所在的循环周期数m为：

$$m = \left[1 + \frac{t}{2\Delta T}\right] \tag{5.37}$$

其中$[\cdot]$表示“向下取整”函数。第t次迭代的学习率为：

$$\alpha_t = \alpha_{min}^m + (\alpha_{max}^m - \alpha_{min}^m)(max(0, 1-b)) \tag{5.38}$$

其中$\propto_{max}^m$ 和 $\propto_{min}^m$分别为第m个周期中学习率的上界和下界，可以随着m的增大而逐渐降低；$b \in [0,1]$的计算为

$$b = \left|\frac{t}{\Delta T} - 2m + 1\right| \tag{5.39}$$

5.3.2 自适应学习率算法

学习率是神经网络优化时的重要超参数。在梯度下降法中，学习率α的取值非常关键，如果过大就不会收敛，如果过小则收敛速度太慢。常用的学习率调整方法包括学习率衰减、学习率预热、周期性学习率调整以及一些自适应调整学习率的方法，比如 AdaGrad、RMSprop、AdaDelta 等。自适应学习率方法可以针对每个参数设置不同的学习率。

AdaGrad：

AdaGrad 算法，如算法 5.2 所示，独立地适应所有模型参数的学习率，缩放每个参数反比于其所有梯度历史平方值总和的平方根[Duchi et al.,2011)。具有损失最大偏导的参数相应地有一个快速下降的学习率，而具有小偏导的参数在学习率上有相对较小的下降。净效果是在参数空间中更为平缓的倾斜方向会取得更大的进步。在凸优化背景中，AdaGrad 算法具有一些令人满意的理论性质。然而，经验上已经发现，对于训练深度神经网络模型而言，从训练开始时积累梯度平方会导致有效学习率过早和过量的减小。AdaGrad 在某些深度学习模型上效果不错，但不是全部。

算法 5.2 AdaGrad 算法

Require:全局学习率ϵ

Require:初始参数θ

Require:小常数δ，为了数值稳定大约设为 10^{-7}

初始化梯度累计变量$r = 0$

while 没有达到停止准则 **do**

从训练集中采包含m个样本$\{x^{(1)},\ldots,x^{(m)}\}$的小批量，

计算梯度：$g \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta),y^{(i)})$

累积平方梯度：$r \leftarrow r + g \odot g$

计算更新：$\Delta\theta \leftarrow -\frac{\epsilon}{\delta+\sqrt{r}} \odot g$ （逐元素地应用除和求平方根）

应用更新：$\theta \leftarrow \theta + \Delta\theta$

end while

RMSProp：

RMSProp 算法[Hinton,2012)修改 AdaGrad 以在非凸设定下效果更好，改变梯度积累为指数加权的移动平均。AdaGrad 旨在应用于凸问题时快速收敛。当应用于非凸函数训练神经网络时，学习轨迹可能穿过了很多不同的结构，最终到达一个局部是凸碗的区域。AdaGrad 根据平方梯度的整个历史收缩学习率，可能使得学习率在达到这样的凸结构前就变得太小了。

RMSProp 使用指数衰减平均以丢弃遥远过去的历史，使其能够在找到凸碗状结构后快速收敛，它就像一个初始化于该碗状结构的 AdaGrad 算法实例。RMSProp 的标准形式如算法 5.3 所示，结合 Nesterov 动量的形式如算法 5.4 所示。相比于 AdaGrad，使用移动平均引入了一个新的超参数ρ，用来控制移动平均的长度范围。经验上，RMSProp 已被证明是一种有效且实用的深度神经网络优化算法。目前它是深度学习从业者经常采用的优化方法之一。

算法 5.3 RMSProp 算法

Require:全局学习率ϵ，衰减速率

Require:初始参数θ

Require:小常数δ，通常设为 10^{-6} （用于被小数除时的数值稳定）

初始化梯度累计变量$r = 0$

while 没有达到停止准则 **do**

从训练集中采包含m个样本$\{x^{(1)},\ldots,x^{(m)}\}$的小批量，对应目标为$y^i$

计算梯度：$g \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta),y^{(i)})$

累积平方梯度：$r \leftarrow pr + (1-p)g \odot g$

计算更新：$\Delta\theta \leftarrow -\frac{\epsilon}{\delta+\sqrt{r}} \odot g$ （$\frac{1}{\sqrt{\delta+r}}$逐元素应用）

应用更新：$\theta \leftarrow \theta + \Delta\theta$

end while

算法 5.4 使用 Nesterov 动量的 RMSProp 算法

Require:全局学习率ϵ，衰减速率，动量系数α

Require:初始参数θ，初始参数v

初始化梯度累计变量$r = 0$

while 没有达到停止准则 **do**

从训练集中采包含m个样本$\{x^{(1)},\ldots,x^{(m)}\}$的小批量，对应目标为$y^{(i)}$

计算临时更新：$\hat{\theta} \leftarrow \theta + \alpha v$

计算梯度：$g \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$

累积梯度：$r \leftarrow pr + (1-p)g \odot g$

计算速度更新：$v \leftarrow av - \frac{\epsilon}{\sqrt{r}} \odot g$ （$\frac{1}{\sqrt{r}}$逐元素应用）

应用更新：$\theta \leftarrow \theta + v$

end while

动量法：

动量（Momentum）是模拟物理中的概念。一个物体的动量指的是该物体在它运动方向上保持运动的趋势，是该物体的质量和速度的乘积。动量法（Momentum Method）是用之前积累动量来替代真正的梯度。每次迭代的梯度可以看作加速度。

在第t次迭代时，计算负梯度的“加权移动平均”作为参数的更新方向

$$\Delta\theta_t = \rho\Delta\theta_{t-1} - \alpha g_t = -\alpha \sum_{\tau=1}^{t} \rho^{t-\tau} g_\tau \tag{5.40}$$

其中ρ为动量因子，通常设为 0.9，α为学习率。这样，每个参数的实际更新差值取决于最近一段时间内梯度的加权平均值。当某个参数在最近一段时间内的梯度方向不一致时，其真实的参数更新幅度变小；相反，当在最近一段时间内的梯度方向都一致时，其真实的参数更新幅度变大，起到加速作用。一般而言，在迭代初期，梯度方向都比较一致，动量法会起到加速作用，可以更快地到达最优点。在迭代后期，梯度方向会不一致，在收敛值附近振荡，动量法会起到减速作用，增加稳定性。从某种角度来说，当前梯度叠加上部分的上次梯度，一定程度上可以近似看作二阶梯度。

Adam：

Adam 算法（Adaptive Moment Estimation Algorithm）[Kingma et al.,2015]可以看作动量法和 RMSprop 算法的结合，不但使用动量作为参数更新方向，而且可以自适应调整学习率。Adam 算法一方面计算梯度平方g_t^2的指数加权平均（和 RMSprop 算法类似），另一方面计算梯度g_t的指数加权平均（和动量法类似）。

$$M_t = \beta_1 M_{t-1} + \odot(1-\beta_1) g_t \tag{5.41}$$

$$G_t = \beta_2 G_{t-1} + (1-\beta_2) g_t \odot g_t \tag{5.42}$$

其中β_1和β_2分别为两个移动平均的衰减率，通常取值为$\beta_1 = 0., \beta_2 = 0.99$。我们可以把$M_t$和$G_t$分别看作梯度的均值（一阶矩）和未减去均值的方差（二阶矩）。假设$M_0 = 0, G_0 = 0$，那么在迭代初期M_t和G_t的值会比真实的均值和方差要小。特别是当β_1和β_2都接近于 1 时，偏差会很大。因此，需要对偏差进行修正。

$$\widehat{M}_t = \frac{M_t}{1-\beta_1^t} \tag{5.43}$$

$$\widehat{G}_t = \frac{G_t}{1-\beta_2^t} \tag{5.44}$$

Adam 算法的参数更新差值为

$$\Delta\theta_t = -\frac{\alpha}{\sqrt{\widehat{G}_t + \varepsilon}} \widehat{M}_t \tag{5.45}$$

在 Adam 中动量直接并入了梯度一阶矩（指数加权）的估计。其次，Adam 包括偏置修正，修正从原点初始化的一阶矩（动量项）和（非中心的）二阶矩的估计（算法 5.5）。Adam 通常被认为对超参数的选择相当鲁棒，尽管学习率有时需要从建议的默认修改。

算法 5.5 Adam 算法

Require:步长ϵ（建议默认为：0.001）

Require:矩估计的指数衰减速率，ρ_1和ρ_2在区间[0,1]内。（建议默认分别为 0.9 和 0.999）

Require:初始参数θ

初始化一阶和二阶矩变量$s=0, r=0$

初始化时间步$t=0$

while 没有达到停止准则 **do**

从训练集中采包含m个样本$\{x^{(1)}, \ldots, x^{(m)}\}$的小批量，对应目标为$y^{(i)}$

计算梯度：$g \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$

$t \leftarrow t+1$

更新有偏一阶矩估计：$s \leftarrow \rho_1 s + (1-\rho_1)g$

更新有偏二阶矩估计：$r \leftarrow \rho_2 r + (1-\rho_2)g \odot g$

修正一阶矩的偏差：$\hat{s} \leftarrow \frac{s}{1-p_1^t}$

修正二阶矩的偏差：$\hat{r} \leftarrow \frac{r}{1-p_2^t}$

计算更新：$\Delta\theta \leftarrow -\epsilon\frac{\hat{s}}{\sqrt{\hat{r}}+\delta}$ （逐元素应用操作）

应用更新：$\theta \leftarrow \theta + \Delta\theta$

end while

5.3.3正则化方法：

机器学习中的一个核心问题是设计不仅在训练数据上表现好，并且能在新输入上泛化好的算法。在机器学习中，许多策略显式地被设计来减少测试误差（可能会以增大训练误差为代价）。这些策略被统称为正则化。深度学习工作者可以使用许多不同形式的正则化策略。

我们将正则化定义为“对学习算法的修改——旨在减少泛化误差而不是训练误差”。目前有许多正则化策略。有些策略向机器学习模型添加限制参数值的额外约束。有些策略向目标函数增加额外项来对参数值进行软约束。如果我们细心选择，这些额外的约束和惩罚可以改善模型在测试集上的表现。有时侯，这些约束和惩罚被设计为编码特定类型的先验知识；其他时候，这些约束和惩罚被设计为偏好简单模型，以便提高泛化能力。有时，惩罚和约束对于确定欠定的问题是必要的。其他形式的正则化，如被称为集成的方法，则结合多个假说来解释训练数据。

在深度学习的背景下，大多数正则化策略都会对估计进行正则化。估计的正则化以偏差的增加换取方差的减少。一个有效的正则化是有利的“交易”，也就是能显著减少方差而不过度增加偏差。我们在讨论泛化和过拟合时，主要侧重模型族训练的 3 个情形：

（1）不包括真实的数据生成过程——对应欠拟合和含有偏差的情况，

（2）匹配真实数据生成过程，

（3）除了包括真实的数据生成过程，还包括许多其他可能的生成过程——方差（而不是偏差）主导的过拟合。正则化的目标是使模型从第三种情况转化为第二种情况.

深度学习算法通常应用于极为复杂的领域，如图像、音频序列和文本，本质上这些领域的真实生成过程涉及模拟整个宇宙。从某种程度上说，我们总是持方枘（数据生成过程）而欲内圆凿（我们的模型族）。我们可能会发现，或者说在实际的深度学习场景中我们几乎总是会发现，最好的拟合模型（从最小化泛化误差的意义上）是一个适当正则化的大型模型。

正则化通过限制模型复杂度，从而避免过拟合，提高泛化能力，比如引入约束、增加先验、提前停止等。在传统的机器学习中，提高泛化能力的方法主要是限制模型复杂度，比如采用l_1和l_2正则化等方式。而

在训练深度神经网络时，特别是在过度参数化（Over-Parameterization）时，l_1和l_2正则化的效果往往不如浅层机器学习模型中显著。过度参数化是指模型参数的数量远远大于训练数据的数量。因此训练深度学习模型时，往往还会使用其他的正则化方法，比如数据增强、提前停止、丢弃法、集成法等。

（1） 参数惩罚（参数正则化）

正则化在深度学习的出现前就已经被使用了数十年。线性模型，如线性回归和逻辑回归可以使用简单、直接、有效的正则化策略。许多正则化方法通过对目标函数J添加一个参数范数惩罚$\Omega(\theta)$，限制模型（如神经网络、线性回归或逻辑回归）的学习能力。我们将正则化后的目标函数记为$\tilde{J}$

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta) \tag{5.46}$$

其中$\alpha \in [0, \infty)$是权衡范数惩罚项 Ω 和标准目标函数$J(X; \theta)$相对贡献的超参数。将α设为 0 表示没有正则化。α越大，对应正则化惩罚越大。当我们的训练算法最小化正则化后的目标函数$\tilde{J}$时，它会降低原始目标J关于训练数据的误差并同时减小在某些衡量标准下参数θ（或参数子集）的规模。选择不同的参数范数 Ω 会偏好不同的解。

在神经网络中，参数包括每一层仿射变换的权重和偏置，我们通常只对权重做惩罚而不对偏置做正则惩罚。精确拟合偏置所需的数据通常比拟合权重少得多。每个权重会指定两个变量如何相互作用。我们需要在各种条件下观察这两个变量才能良好地拟合权重。而每个偏置仅控制一个单变量。这意味着，我们不对其进行正则化也不会导致太大的方差。另外，正则化偏置参数可能会导致明显的欠拟合。

因此，我们使用向量w表示所有应受范数惩罚影响的权重，而向量θ表示所有参数(包括w和无需正则化的参数)。在神经网络的情况下，有时希望对网络的每一层使用单独的惩罚，并分配不同的α系数。寻找合适的多个超参数的代价很大，因此为了减少搜索空间，我们会在所有层使用相同的权重衰减。

L2 参数正则化

L2 参数范数惩罚是最简单而又最常见的参数范数惩罚，通常被称为权重衰减（weight decay）.这个正则化策略通过向目标函数添加一个正则项$\Omega(\theta) = \frac{1}{2}\|w\|_2^2$，使权重更加接近原点 1。在其他学术圈，L2 也被称为岭回归或 Tikhonov 正则。我们可以通过研究正则化后目标函数的梯度，洞察一些权重衰减的正则化表现。为了简单起见，我们假定其中没有偏置参数，因此θ就是w。这样一个模型具有以下总的目标函数：

$$\tilde{J}(\omega; X, y) = \frac{\alpha}{2}\omega^T\omega + J(\omega; X, y) \tag{5.47}$$

与之对应的梯度为：

$$\nabla_\omega\tilde{J}(\omega; X, y) = \alpha\omega + \nabla_\omega J(\omega; X, y) \tag{5.48}$$

使用单步梯度下降更新权重，即执行以下更新：

$$\omega \leftarrow \omega - \epsilon(\alpha\omega + \nabla_\omega J(\omega; X, y) \tag{5.49}$$

换种写法就是：

$$\omega \leftarrow (1 - \epsilon\alpha)\omega - \epsilon\nabla_\omega J(\omega; X, y) \tag{5.50}$$

L1 参数正则化

L2 权重衰减是权重衰减最常见的形式，我们还可以使用其他的方法限制模型参数的规模。一个选择是使用 L1 正则化。形式地，对模型参数w的L_1正则化被定义为：

$$\Omega(\theta) = \|\omega\|_1 = \sum_i|\omega_i| \tag{5.51}$$

即各个参数的绝对值之和。接着我们将讨论L_1正则化对简单线性回归模型的影响，与分析L_2正则化时一样不考虑偏置参数。我们尤其感兴趣的是找出L_1和L_2正则化之间的差异。与L_2权重衰减类似，我们也可以通过缩放惩罚项Ω 的正超参数α来控制L_1权重衰减的强度。因此，正则化的目标函数$\tilde{J}(\omega; X, y)$如下所示：

$$\tilde{J}(\omega; X, y) = \alpha\|\omega\|_1 + J(\omega; X, y) \tag{5.52}$$

对应的梯度(实际上是次梯度)：

$$\nabla_\omega\tilde{J}(\omega; X, y) = \alpha sign(\omega) + \nabla_\omega J(\omega; X, y) \tag{5.53}$$

其中$sign(w)$只是简单地取w各个元素的正负号。

观察上式，我们立刻发现L_1的正则化效果与L_2大不一样。具体来说，我们可以看到正则化对梯度的影响不再是线性地缩放每个ω_i；而是添加了一项与$sign(\omega_i)$同号的常数。使用这种形式的梯度之后，我们不一定能得到$J(\omega;X,y)$二次近似的直接算术解（L_2正则化时可以）。

相比L_2正则化，L_1正则化会产生更稀疏（sparse）的解。此处稀疏性指的是最优值中的一些参数为 0。和L_2正则化相比，L_1正则化的稀疏性具有本质的不同。由L_1正则化导出的稀疏性质已经被广泛地用于特征选择（feature selection）机制。特征选择从可用的特征子集选择出有意义的特征，化简机器学习问题。著名的 LASSO[Tibshirani,1995]（Least Absolute Shrinkage and Selection Operator）模型将L_1惩罚和线性模型结合，并使用最小二乘代价函数。L_1惩罚使部分子集的权重为零，表明相应的特征可以被安全地忽略。

数据增强（数据正则化）

让机器学习模型泛化得更好的最好办法是使用更多的数据进行训练。当然，在实践中，我们拥有的数据量是很有限的。解决这个问题的一种方法是创建假数据并添加到训练集中。对于一些机器学习任务，创建新的假数据相当简单。

对分类来说这种方法是最简单的。分类器需要一个复杂的高维输入x，并用单个类别标识y概括x。这意味着分类面临的一个主要任务是要对各种各样的变换保持不变。我们可以轻易通过转换训练集中的x来生成新的(x,y)对。

这种方法对于其他许多任务来说并不那么容易。例如，除非我们已经解决了密度估计问题，否则在密度估计任务中生成新的假数据是很困难的。数据集增强对一个具体的分类问题来说是特别有效的方法：对象识别。图像是高维的并包括各种巨大的变化因素，其中有许多可以轻易地模拟。即使模型已使用卷积和池化技术对部分平移保持不变，沿训练图像每个方向平移几个像素的操作通常可以大大改善泛化。许多其他操作如旋转图像或缩放图像也已被证明非常有效。

我们必须要小心，不能使用会改变类别的转换。例如，光学字符识别任务需要认识到“b”和“d”以及“6”和“9”的区别，所以对这些任务来说，水平翻转和旋转180°并不是合适的数据集增强方式。能保持我们希望的分类不变，但不容易执行的转换也是存在的。例如，平面外绕轴转动难以通过简单的几何运算在输入像素上实现。

数据集增强对语音识别任务也是有效的[Jaitly and Hinton,2013]。在神经网络的输入层注入噪声[Sietsma and Dow,1991]也可以被看作是数据增强的一种方式。对于许多分类甚至一些回归任务而言，即使小的随机噪声被加到输入，任务仍应该是能够被解决的。然而，神经网络被证明对噪声不是非常健壮[Tang and Eliasmith,2010]。改善神经网络健壮性的方法之一是简单地将随机噪声添加到输入再进行训练。输入噪声注入是一些无监督学习算法的一部分，如去噪自编码器[Vincent et al.,2008a]。向隐藏单元施加噪声也是可行的，这可以被看作在多个抽象层上进行的数据集增强。Poole et al.(2014)最近表明，噪声的幅度被细心调整后，该方法是非常高效的。我们将在介绍一个强大的正则化策略 Dropout，该策略可以被看作是通过与噪声相乘构建新输入的过程。

在比较机器学习基准测试的结果时，考虑其采取的数据集增强是很重要的。通常情况下，人工设计的数据集增强方案可以大大减少机器学习技术的泛化误差。将一个机器学习算法的性能与另一个进行对比时，对照实验是必要的。在比较机器学习算法 A 和机器学习算法 B 时，应该确保这两个算法使用同一人工设计的数据集增强方案。假设算法 A 在没有数据集增强时表现不佳，而 B 结合大量人工转换的数据后表现良好。在这样的情况下，很可能是合成转化引起了性能改进，而不是机器学习算法 B 比算法 A 更好。有时候，确定实验是否已经适当控制需要主观判断。例如，向输入注入噪声的机器学习算法是执行数据集增强的一种形式。通常，普适操作（例如，向输入添加高斯噪声）被认为是机器学习算法的一部分，而特定于一个应用领域（如随机地裁剪图像）的操作被认为是独立的预处理步骤。

集成方法（模型正则化）

Bagging（bootstrap aggregating）是通过结合几个模型降低泛化误差的技术[Breiman,1994]。主要想法

是分别训练几个不同的模型，然后让所有模型表决测试样例的输出。这是机器学习中常规策略的一个例子，被称为模型平均（model averaging）。采用这种策略的技术被称为集成方法。模型平均（model averaging）奏效的原因是不同的模型通常不会在测试集上产生完全相同的误差。

假设我们有 k 个回归模型。假设每个模型在每个例子上的误差是ϵ_i，这个误差服从零均值方差为$E[\epsilon_i^2] = v$且协方差为$E[\epsilon_i \epsilon_j] = c$的多维正态分布。通过所有集成模型的平均预测所得误差是$\frac{1}{k}\sum_i(\epsilon_i)$。集成预测器平方误差的期望是

$$\mathbb{E}\left[(\frac{1}{k}\sum_i(\epsilon_i))^2\right] = \frac{1}{k^2}\mathbb{E}\left[\left(\frac{1}{k}\right)\sum_i\left(\epsilon_i^2 + \sum_{j\neq i}\epsilon_i\,\epsilon_j\right)\right] = \frac{1}{k}v + \frac{k-1}{k}c \qquad (5.54)$$

在误差完全相关即$c = v$的情况下，均方误差减少到 v，所以模型平均没有任何帮助。在错误完全不相关即$c = 0$的情况下，该集成平方误差的期望仅为$\frac{1}{k}v$。这意味着集成平方误差的期望会随着集成规模增大而线性减小。换言之，平均上，集成至少与它的任何成员表现得一样好，并且如果成员的误差是独立的，集成将显著地比其成员表现得更好。

不同的集成方法以不同的方式构建集成模型。例如，集成的每个成员可以使用不同的算法和目标函数训练成完全不同的模型。Bagging 是一种允许重复多次使用同一种模型、训练算法和目标函数的方法。

具体来说，Bagging 涉及构造 k 个不同的数据集。每个数据集从原始数据集中重复采样构成，和原始数据集具有相同数量的样例。这意味着，每个数据集以高概率缺少一些来自原始数据集的例子，还包含若干重复的例子（如果所得训练集与原始数据集大小相同，那所得数据集中大概有原始数据集 2/3 实例）。模型 i 在数据集 i 上训练。每个数据集所含样本的差异导致了训练模型之间的差异。

神经网络能找到足够多的不同的解，意味着他们可以从模型平均中受益(即使所有模型都在同一数据集上训练)。神经网络中随机初始化的差异、小批量的随机选择、超参数的差异或不同输出的非确定性实现往往足以使得集成中的不同成员具有部分独立的误差。

不是所有构建集成的技术都是为了让集成模型比单一模型更加正则化。例如，一种被称为 Boosting 的技术[Freund and Schapire,1996b,a]构建比单个模型容量更高的集成模型。通过向集成逐步添加神经网络，Boosting 已经被应用于构建神经网络的集成[Schwenk and Bengio,1998]。通过逐渐增加神经网络的隐藏单元，Boosting 也可以将单个神经网络解释为一个集成。

提前停止（训练正则化）

当训练有足够的表示能力甚至会过拟合的大模型时，我们经常观察到，训练误差会随着时间的推移逐渐降低但验证集的误差会再次上升。这意味着我们只要返回使验证集误差最低的参数设置，就可以获得验证集误差更低的模型（并且因此有希望获得更好的测试误差）。在每次验证集误差有所改善后，我们存储模型参数的副本。当训练算法终止时，我们返回这些参数而不是最新的参数。当验证集上的误差在事先指定的循环次数内没有进一步改善时，算法就会终止。此过程在算法 5.6 中有更正式的说明。这种策略被称为提前终止（early stopping）。这可能是深度学习中最常用的正则化形式。它的流行主要是因为有效性和简单性。

算法 5.6 用于确定最佳训练时间量的提前终止元算法。这种元算法是一种通用策略，可以很好地在各种训练算法和各种量化验证集误差的方法上工作。

令n为评估间隔的步数。

令p为“耐心（patient）”，即观察到较坏的验证集表现p次后终止。

令θ_0为初始参数。

$\theta \leftarrow \theta_0$

$i \leftarrow 0$

$j \leftarrow 0$

$v \leftarrow \infty$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

while $j < p$ **do**

运行训练算法 n 步，更新θ

$i \leftarrow i + n$

$v' \leftarrow ValidationSetError(\theta)$

if $v' < v$ then

$j \leftarrow 0$

$\theta^* \leftarrow \theta$

$i^* \leftarrow -i$

$v \leftarrow -v'$

else

$j \leftarrow j + 1$

end if

end while

最佳参数为$\boldsymbol{\theta^i}$，最佳训练步数为$\boldsymbol{i^*}$

我们可以认为提前终止是非常高效的超参数选择算法。按照这种观点，训练步数仅是另一个超参数。在提前终止的情况下，我们通过控制拟合训练集的步数来控制模型的有效容量。大多数超参数的选择必须使用高代价的猜测和检查过程，我们需要在训练开始时猜测一个超参数，然后运行几个步骤检查它的训练效果。“训练时间”是唯一只要跑一次训练就能尝试很多值的超参数。通过提前终止自动选择超参数的唯一显著的代价是训练期间要定期评估验证集。在理想情况下，这可以并行在与主训练过程分离的机器上，或独立的 CPU，或独立的 GPU 上完成。如果没有这些额外的资源，可以使用比训练集小的验证集或较不频繁地评估验证集来减小评估代价，较粗略地估算取得最佳的训练时间。

另一个提前终止的额外代价是需要保持最佳的参数副本。这种代价一般是可忽略的，因为可以将它储存在较慢较大的存储器上（例如，在 GPU 内存中训练，但将最佳参数存储在主存储器或磁盘驱动器上）。由于最佳参数的写入很少发生而且从不在训练过程中读取，这些偶发的慢写入对总训练时间的影响不大。

提前终止是一种非常不显眼的正则化形式，它几乎不需要改变基本训练过程、目标函数或一组允许的参数值。这意味着，无需破坏学习动态就能很容易地使用提前终止。相对于权重衰减，必须小心不能使用太多的权重衰减，以防网络陷入不良局部极小点(对应于病态的小权重)。

Dropout(训练正则化)

Dropout[Srivastava et al.,2014]提供了正则化一大类模型的方法，计算方便但功能强大。在第一种近似下，Dropout 可以被认为是集成大量深层神经网络的实用 Bagging 方法。Bagging 涉及训练多个模型，并在每个测试样本上评估多个模型。当每个模型都是一个很大的神经网络时，这似乎是不切实际的，因为训练和评估这样的网络需要花费很多运行时间和内存。通常我们只能集成五至十个神经网络，如 Szegedy et

al.(2014a)集成了六个神经网络赢得 ILSVRC，超过这个数量就会迅速变得难以处理。Dropout 提供了一种廉价的 Bagging 集成近似，能够训练和评估指数级数量的神经网络。

具体而言，Dropout 训练的集成包括所有从基础网络除去非输出单元后形成的子网络，如图 5.11 所示。最先进的神经网络基于一系列仿射变换和非线性变换，我们只需将一些单元的输出乘零就能有效地删除一个单元。这个过程需要对模型（如径向基函数网络，单元的状态和参考值之间存在一定区别）进行一些修改。为了简单起见，我们在这里提出乘零的简单 Dropout 算法，但是它被简单修改后，可以与从网络中移除单元的其他操作结合使用。

回想一下 Bagging 学习，我们定义 k 个不同的模型，从训练集有放回采样构造 k 个不同的数据集，然后在训练集 i 上训练模型 i。Dropout 的目标是在指数级数量的神经网络上近似这个过程。具体来说，在训练中使用 Dropout 时，我们会使用基于小批量产生较小步长的学习算法，如随机梯度下降等。我们每次在小批量中加载一个样本，然后随机抽样应用于网络中所有输入和隐藏单元的不同二值掩码。对于每个单元，掩码是独立采样的。掩码值为 1 的采样概率（导致包含一个单元）是训练开始前一个固定的超参数。它不是模型当前参数值或输入样本的函数。通常在每一个小批量训练的神经网络中，一个输入单元被包括的概率为 0.8，一个隐藏单元被包括的概率为 0.5。然后，我们运行和之前一样的前向传播、反向传播以及学习更新。图 5.12 说明了在 Dropout 下的前向传播。

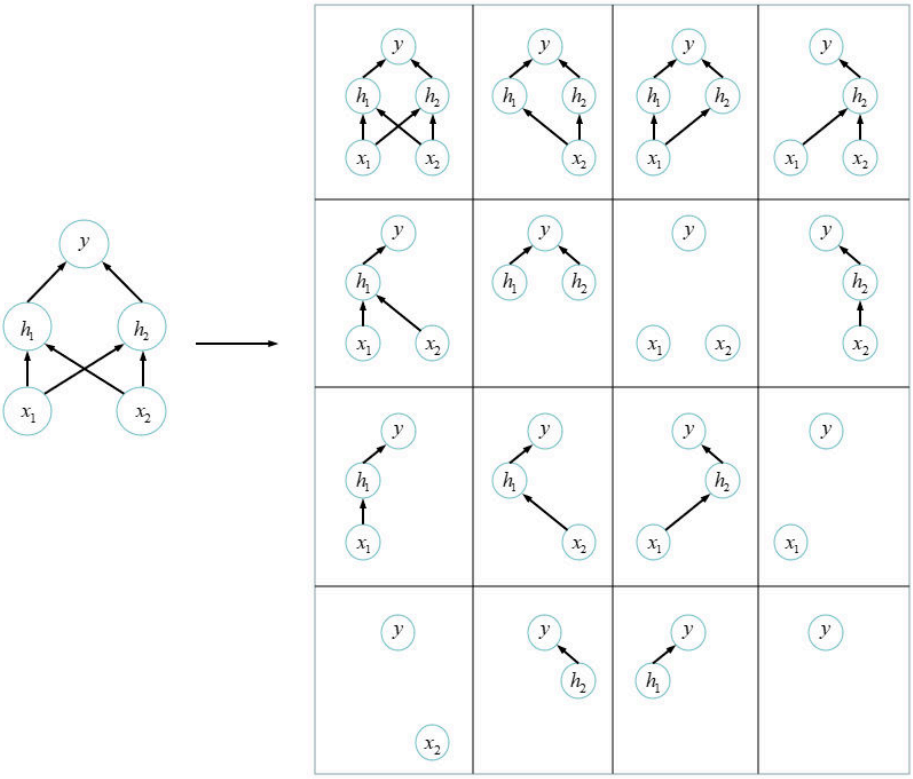


图 5.11 Dropout 训练由所有子网络组成的集成，其中子网络通过从基本网络中删除非输出单元构建

我们从具有两个可见单元和两个隐藏单元的基本网络开始。这四个单元有十六个可能的子集。右图展示了从原始网络中丢弃不同的单元子集而形成的所有十六个子网络。在这个小例子中，所得到的大部分网络没有输入单元或没有从输入连接到输出的路径。当层较宽时，丢弃所有从输入到输出的可能路径的概率变小，所以这个问题不太可能在出现层较宽的网络中。

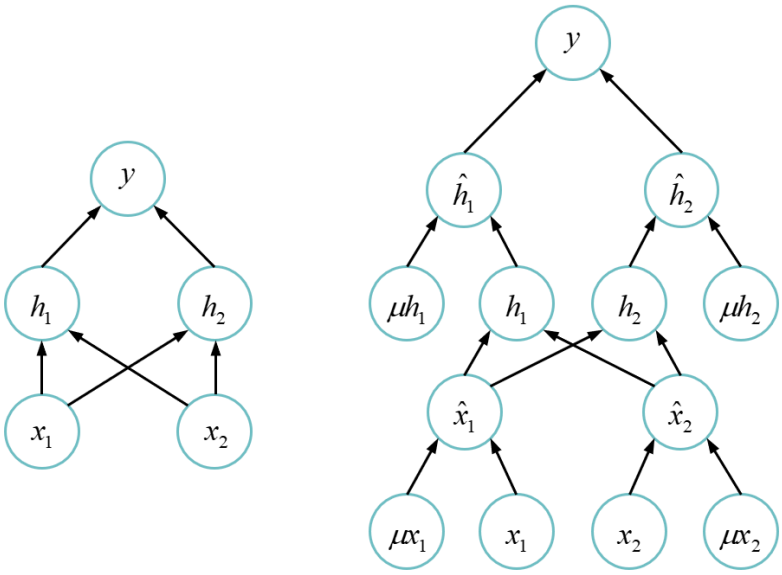


图 5.12 在使用 Dropout 的前馈网络中前向传播的示例。(顶部)在此示例中，我们使用具有两个输入单元，具有两个隐藏单元的隐藏层以及一个输出单元的前馈网络。(底部)为了执行具有 Dropout 的前向传播，我们随机地对向量μ进行采样，其中网络中的每个输入或隐藏单元对应一项。μ中的每项都是二值的且独立于其他项采样。超参数的采样概率为 1，隐藏层的采样概率通常为 0.5，输入的采样概率通常为 0.8。网络中的每个单元乘以相应的掩码，然后正常地继续沿着网络的其余部分前向传播。这相当于从图 5.11 中随机选择一个子网络并沿着前向传播。

Srivastava et al.(2014)显示，Dropout 比其他标准的计算开销小的正则化方法（如权重衰减、过滤器范数约束和稀疏激活的正则化）更有效。Dropout 也可以与其他形式的正则化合并，得到进一步的提升。计算方便是 Dropout 的一个优点。训练过程中使用 Dropout 产生 n 个随机二进制数与状态相乘，每个样本每次更新只需 O(n)的计算复杂度。根据实现，也可能需要 O(n)的存储空间来持续保存这些二进制数（直到反向传播阶段）。使用训练好的模型推断时，计算每个样本的代价与不使用 Dropout 是一样的，尽管我们必须在开始运行推断前将权重除以 2。

Dropout 的另一个显著优点是不怎么限制适用的模型或训练过程。几乎在所有使用分布式表示且可以用随机梯度下降训练的模型上都表现很好。包括前馈神经网络、概率模型，如受限玻尔兹曼机[Srivastava et al.,2014]，以及循环神经网络[Bayer and Osendorfer,2014; Pascanu et al.,2014a]。许多效果差不多的其他正则化策略对模型结构的限制更严格。

虽然 Dropout 在特定模型上每一步的代价是微不足道的，但在一个完整的系统上使用 Dropout 的代价可能非常显著。因为 Dropout 是一个正则化技术，它减少了模型的有效容量。为了抵消这种影响，我们必须增大模型规模。不出意外的话，使用 Dropout 时最佳验证集的误差会低很多，但这是以更大的模型和更多训练算法的迭代次数为代价换来的。对于非常大的数据集，正则化带来的泛化误差减少得很小。在这些情况下，使用 Dropout 和更大模型的计算代价可能超过正则化带来的好处。

5.4 卷积神经网络：

5.4.1计算机视觉解决的基本问题

简单来说，计算机视觉解决的主要问题是：给出一张二维图像，计算机视觉系统必须识别出图像中的对象及其特征，如形状、纹理、颜色、大小、空间排列等，从而尽可能完整地描述该图像。计算机视觉基于大量不同任务，并组合在一起实现高度复杂的应用。计算机视觉中最常见的任务是图像和视频识别与分类，目标检测、识别和追踪，人体关键点检测和场景文字识别。

图像分类

图像分类是计算机视觉中重要的基础问题。后面提到的其他任务也是以它为基础的。举几个典型的例

子：人脸识别、图片鉴黄、相册根据人物自动分类等

语义分割

它将整个图像分成像素组，然后对像素组进行标记和分类。语义分割试图在语义上理解图像中每个像素是什么（人、车、狗、树）。

实例分割

除了语义分割之外，实例分割将不同类型的实例进行分类，比如用 5 种不同颜色来标记 5 辆汽车。我们会看到多个重叠物体和不同背景的复杂景象，我们不仅需要将这些不同的对象进行分类，而且还要确定对象的边界、差异和彼此之间的关系！

视频分类

与图像分类不同的是，分类的对象不再是静止的图像，而是一个由多帧图像构成的、包含语音数据、包含运动信息等的视频对象，因此理解视频需要获得更多的上下文信息，不仅要理解每帧图像是什么、包含什么，还需要结合不同帧，知道上下文的关联信息。

人体关键点检测

体关键点检测，通过人体关键节点的组合和追踪来识别人的运动和行为，对于描述人体姿态，预测人体行为至关重要。

场景文字识别

很多照片中都有一些文字信息，这对理解图像有重要的作用。场景文字识别是在图像背景复杂、分辨率低下、字体多样、分布随意等情况下，将图像信息转化为文字序列的过程。停车场、收费站的车牌识别就是典型的应用场景。

目标检测

目标检测任务的目标是给定一张图像或是一个视频帧，让计算机找出其中所有目标的位置，并给出每个目标的具体类别。

目标识别

目标识别与目标检测略有不同，尽管它们使用类似的技术。给出一个特定对象，目标识别的目标是在图像中找出该对象的实例。这并不是分类，而是确定该对象是否出现在图像中，如果出现，则执行定位。搜索包含某公司 logo 的图像就是一个例子。另一个例子是监控安防摄像头拍摄的实时图像以识别某个人的面部。

目标追踪

目标追踪旨在追踪随着时间不断移动的对象，它使用连续视频帧作为输入。该功能对于机器人来说是必要的，以守门员机器人举例，它们需要执行从追球到挡球等各种任务。目标追踪对于自动驾驶汽车而言同样重要，它可以实现高级空间推理和路径规划。类似地，目标追踪在多人追踪系统中也很有用，包括用于理解用户行为的系统（如零售店的计算机视觉系统），以及在游戏中监控足球或篮球运动员的系统。

执行目标追踪的一种相对直接的方式是，对视频序列中的每张图像执行目标追踪并对比每个对象实例，以确定它们的移动轨迹。该方法的缺陷是为每张图像执行目标检测通常成本高昂。另一种替换方式仅需捕捉被追踪对象一次（通常是该对象出现的第一次），然后在不明确识别该对象的情况下在后续图像中辨别它的移动轨迹。最后，目标追踪方法未必就能检测出对象，它可以在不知道追踪对象是什么的情况下，仅查看目标的移动轨迹。

计算机视觉在日常生活中的应用场景

计算机视觉的应用场景非常广泛，下面列举几个生活中常见的应用场景。

1. 门禁、支付宝上的人脸识别
2. 停车场、收费站的车牌识别
3. 上传图片或视频到网站时的风险识别
4. 抖音上的各种道具（需要先识别出人脸的位置）

5.4.2卷积神经网络背景

卷积神经网络最早主要是用来处理图像信息。在用全连接前馈网络来处理图像时，存在以下两个问题：

（1）参数太多：如果输入图像大小为 100×100×3（即图像高度为 100，宽度为 100 以及 RGB3 个颜色通道），在全连接前馈网络中，第一个隐藏层的每个神经元到输入层都有 100×100×3=30000 个互相独立的连接，每个连接都对应一个权重参数。随着隐藏层神经元数量的增多，参数的规模也会急剧增加。这会导致整个神经网络的训练效率非常低，也很容易出现过拟合。

（2）局部不变性特征：自然图像中的物体都具有局部不变性特征，比如尺度缩放、平移、旋转等操作不影响其语义信息。而全连接前馈网络很难提取这些局部不变性特征，一般需要进行数据增强来提高性能。

卷积神经网络是受生物学上感受野机制的启发而提出的。感受野（Receptive Field）机制主要是指听觉、视觉等神经系统中一些神经元的特性，即神经元只接受其所支配的刺激区域内的信号。在视觉神经系统中，视觉皮层中的神经细胞的输出依赖于视网膜上的光感受器。视网膜上的光感受器受刺激兴奋时，将神经冲动信号传到视觉皮层，但不是所有视觉皮层中的神经元都会接受这些信号。一个神经元的感受野是指视网膜上的特定区域，只有这个区域内的刺激才能够激活该神经元。

目前的卷积神经网络是由卷积层、汇聚层和全连接层交叉堆叠而成的前馈神经网络。全连接层一般在卷积网络的最顶层。卷积神经网络有三个结构上的特性：局部连接、权重共享以及汇聚。这些特性使得卷积神经网络具有一定程度上的平移、缩放和旋转不变性。和前馈神经网络相比，卷积神经网络的参数更少。

卷积神经网络主要使用在图像和视频分析的各种任务（比如图像分类、人脸识别、物体识别、图像分割等）上，其准确率一般也远远超出了其他的神经网络模型。近年来卷积神经网络也广泛地应用到自然语言处理、推荐系统等领域。

5.4.3卷积

卷积是对两个实变函数的一种数学运算。卷积的特点是局部连接、权值共享，目的是提取图像特征。我们从两个可能会用到的函数的例子出发。假设我们正在用激光传感器追踪一艘宇宙飞船的位置。我们的激光传感器给出一个单独的输出$x(t)$，表示宇宙飞船在时刻t的位置。x和t都是实值的，这意味着我们可以在任意时刻从传感器中读出飞船的位置。

现在假设我们的传感器受到一定程度的噪声干扰。为了得到飞船位置的低噪声估计，我们对得到的测量结果进行平均。显然，时间上越近的测量结果越相关，所以我们采用一种加权平均的方法，对于最近的测量结果赋予更高的权重。我们可以采用一个加权函数$w(a)$来实现，其中a表示测量结果距当前时刻的时间间隔。如果我们对任意时刻都采用这种加权平均的操作，就得到了一个新的对于飞船位置的平滑估计函数s：

$$s(t) = \int x(a)w(t-a)da \tag{5.55}$$

这种运算就叫做卷积（convolution）。卷积运算通常用星号表示：

$$s(t) = (x * w)(t) \tag{5.56}$$

在我们的例子中，w必须是一个有效的概率密度函数，否则输出就不再是一个加权平均。另外，在参数为负值时，w的取值必须为 0，否则它会预测到未来，这不是我们能够推测得了的。但这些限制仅仅是对我们这个例子来说。通常，卷积被定义在满足上述积分式的任意函数上，并且也可能被用于加权平均以外的目的。

在卷积网络的术语中，卷积的第一个参数（在这个例子中，函数x）通常叫做输入（input），第二个参数（函数w）叫做核函数（kernel function）。输出有时被称作特征映射（feature map）。

在本例中，激光传感器在每个瞬间反馈测量结果的想法是不切实际的。一般地，当我们用计算机处理数据时，时间会被离散化，传感器会定期地反馈数据。所以在我们的例子中，假设传感器每秒反馈一次测量结果是比较现实的。这样，时刻t只能取整数值。如果我们假设x和w都定义在整数时刻t上，就可以定义离散形式的卷积：

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{5.57}$$

在机器学习的应用中，输入通常是多维数组的数据，而核通常是由学习算法优化得到的多维数组的参数。我们把这些多维数组叫做张量。因为在输入与核中的每一个元素都必须明确地分开存储，我们通常假设在存储了数值的有限点集以外，这些函数的值都为零。这意味着在实际操作中，我们可以通过对有限个数组元素的求和来实现无限求和。

最后，我们经常一次在多个维度上进行卷积运算。例如，如果把一张二维的图像I作为输入，我们也许也想要使用一个二维的核K：

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{5.58}$$

卷积是可交换的(commutative)，我们可以等价地写作：

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{5.59}$$

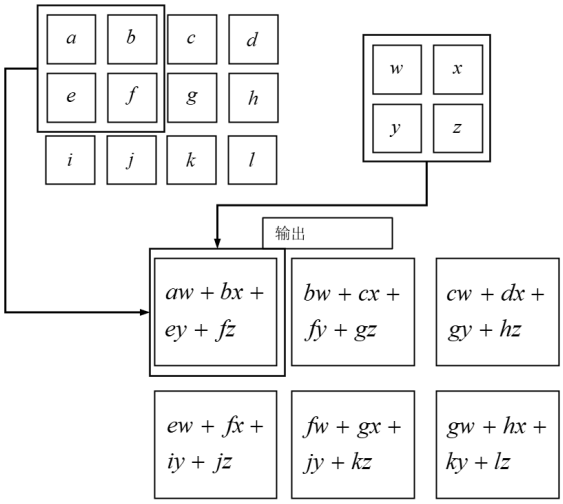


图 5.13 一个 2 维卷积的例子（没有对核进行翻转）

我们限制只对核完全处在图像中的位置进行输出，在一些上下文中称为“有效”卷积。我们用画有箭头的盒子来说明输出张量的左上角元素是如何通过对输入张量相应的左上角区域应用核进行卷积得到的。

5.4.4池化：

卷积网络中一个典型层包含三级（如图 9.7 所示）。在第一级中，这一层并行地计算多个卷积产生一组线性激活响应。在第二级中，每一个线性激活响应将会通过一个非线性的激活函数，例如整流线性激活函数。这一级有时也被称为检测阶段 ector stage）。在第三级中，我们使用池化函数（pooling function）来进步调整这一层的输出。

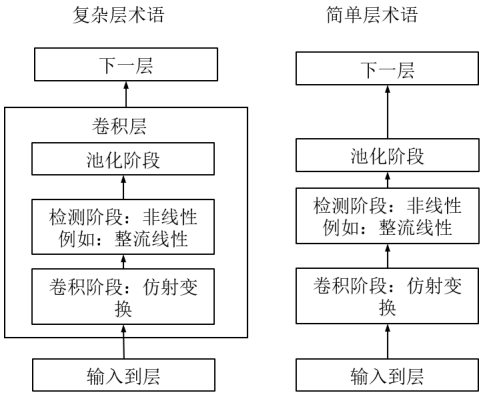


图 5.14 一个典型卷积神经网络层的组件。有两组常用的术语用于描述这些层。(左)在这组术语中，卷积网络被视为少量相对复杂的层，每层具有许多“级”。在这组术语中，核张量与网络层之间存在一一对应关系。在本书中，我们通常使用这组术语。(右)在这组术语中，卷积网络被视为更多数量的简单层；每一个处理步骤都被认为是一个独立的层。这意味着不是每一“层”都有参数。

池化函数使用某一位置的相邻输出的总体统计特征来代替网络在该位置的输出。池化的目的在于减小 feature map 尺寸、防止过拟合、引入平移不变性。例如，最大池化（max pooling）函数(Zhou and Chellappa,1988)给出相邻矩形区域内的最大值。其他常用的池化函数包括相邻矩形区域内的平均值、L2 范数以及基于据中心像素距离的加权平均函数。

不管采用什么样的池化函数，当输入作出少量平移时，池化能够帮助输入的表示近似不变（invariant）。对于平移的不变性是指当我们对输入进行少量平移时，经过池化函数后的大多数输出并不会发生改变。图 5.14 用了一个例子来说明这是如何实现的。局部平移不变性是一个很有用的性质，尤其是当我们关心某个特征是否出现而不关心它出现的具体位置时。例如，当判定一张图像中是否包含人脸时，我们并不需要知道眼睛的精确像素位置，我们只需要知道有一只眼睛在脸的左边，有一只在右边就行了。但在一些其他领域，保存特征的具体位置却很重要。例如当我们想要寻找一个由两条边相交而成的拐角时，我们就需要很好地保存边的位置来判定它们是否相交。

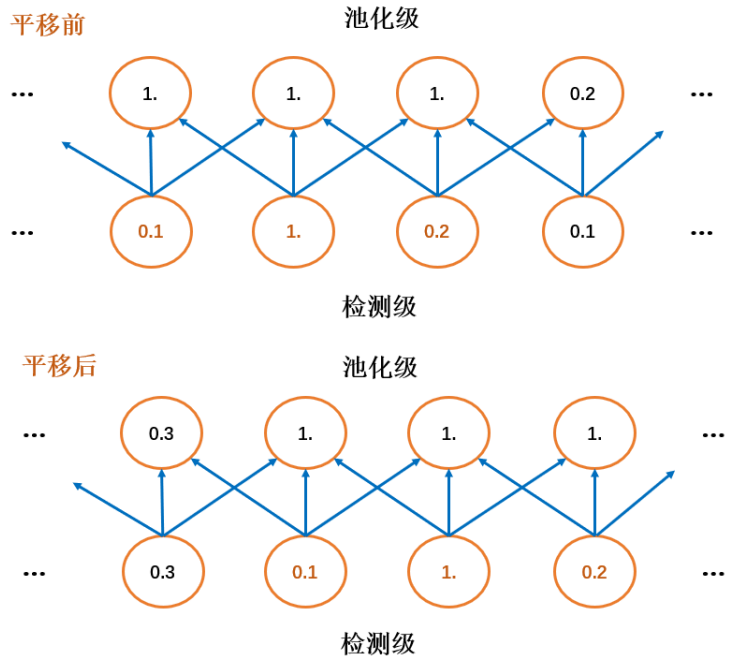


图 5.15 最大池化引入了不变性。(上)卷积层中间输出的视图。下面一行显示非线性的输出。上面一行显示最大池化的输出，每个池的宽度为三个像素并且池化区域的步幅为一个像素。(下)相同网络的视图，不过对输入右移了一个像素。下面一行的所有值都发生了改变，但上面一行只有一半的值发生了改变，这是因为最大池化单元只对周围的最大值比较敏感，而不是对精确的位置。

使用池化可以看作是增加了一个无限强的先验：这一层学得的函数必须具有对少量平移的不变性。当这个假设成立时，池化可以极大地提高网络的统计效率。对空间区域进行池化产生了平移不变性，但当我们对分离参数的卷积的输出进行池化时，特征能够学得应该对于哪种变换具有不变性。

因为池化综合了全部邻居的反馈，这使得池化单元少于探测单元成为可能，我们可以通过综合池化区域的k个像素的统计特征而不是单个像素来实现。这种方法提高了网络的计算效率，因为下一层少了约k倍的输入。当下一层的参数数目是关于那一层输入大小的函数时（例如当下一层是全连接的基于矩阵乘法的网络层时），这种对于输入规模的减小也可以提高统计效率并且减少对于参数的存储需求。

在很多任务中，池化对于处理不同大小的输入具有重要作用。例如我们想对不同大小的图像进行分类时，分类层的输入必须是固定的大小，而这通常通过调整池化区域的偏置大小来实现，这样分类层总是能接收到相同数量的统计特征而不管最初的输入大小了。例如，最终的池化层可能会输出四组综合统计特征，每组对应着图像的一个象限，而与图像的大小无关。

5.4.4结构化输出

卷积神经网络可以用于输出高维的结构化对象，而不仅仅是预测分类任务的类标签或回归任务的实数

值。通常这个对象只是一个张量，由标准卷积层产生。例如，模型可以产生张量S，其中$S_{i,j,k}$是网络的输入像素(j,k)属于类i的概率。这允许模型标记图像中的每个像素，并绘制沿着单个对象轮廓的精确掩模。

经常出现的一个问题是输出平面可能比输入平面要小。用于对图像中单个对象分类的常用结构中，网络空间维数的最大减少来源于使用大步幅的池化层。为了产生与输入大小相似的输出映射，我们可以避免把池化放在一起[Jain et al.,2007]。另一种策略是单纯地产生一张低分辨率的标签网格[Pinheiro and Collobert,2014,2015]。最后，原则上可以使用具有单位步幅的池化操作。

对图像逐个像素标记的一种策略是先产生图像标签的原始猜测，然后使用相邻像素之间的交互来修正该原始猜测。重复这个修正步骤数次对应于在每一步使用相同的卷积，该卷积在深层网络的最后几层之间共享权重[Jain et al.,2007]。这使得在层之间共享参数的连续的卷积层所执行的一系列运算，形成了一种特殊的循环神经网络[Pinheiro and Collobert,2014,2015]。

一旦对每个像素都进行了预测，我们就可以使用各种方法来进一步处理这些预测，以便获得图像在区域上的分割[Briggman et al.,2009;Turaga et al.,2010;Farabet et al.,2013]。一般的想法是假设大片相连的像素倾向于对应着相同的标签。图模型可以描述相邻像素间的概率关系。或者，卷积网络可以被训练来最大化地近似图模型的训练目标[Ning et al.,2005;Thompson et al.,2014]。

5.4.5典型的卷积神经网络

LeNet-5

LeNet-5[LeCun et al.,1998]虽然提出的时间比较早，但它是一个非常成功的神经网络模型。基于 LeNet-5 的手写数字识别系统在 20 世纪 90 年代被美国很多银行使用，用来识别支票上面的手写数字。

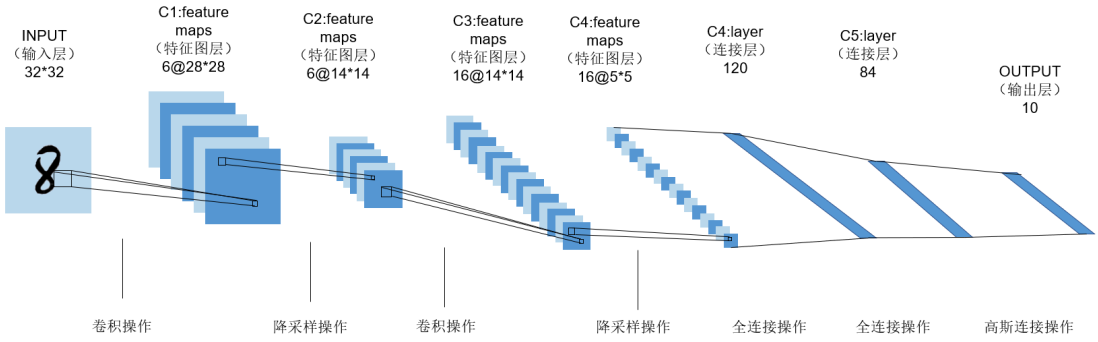


图 5.16 LeNet 网络结构(参照[LeCun et al.,1998])

LeNet-5 共有 7 层，接受输入图像大小为$32 \times 32 = 1024$，输出对应 10 个类别的得分。LeNet-5 中的每一层结构如下：

（1）$C1$层是卷积层，使用 6 个5×5的卷积核，得到 6 组大小为$28 \times 28 = 784$的特征映射。因此，$C1$层的神经元数量为$6 \times 784 = 4704$，可训练参数数量为$6 \times 25 + 6 = 156$，连接数为$156 \times 784 = 122304$（包括偏置在内，下同）。

（2）$S2$层为汇聚层，采样窗口为2×2，使用平均汇聚，并使用一个如公式(5.27)的非线性函数。神经元个数为$6 \times 14 \times 14 = 1176$，可训练参数数量为$6 \times (1 + 1) = 12$，连接数为$6 \times 196 \times (4 + 1) = 5880$。

（3）C3 层为卷积层。LeNet-5 中用一个连接表来定义输入和输出特征映射之间的依赖关系，连接表参见公式(5.40)。如图 5.11 所示，共使用60 个5×5的卷积核，得到 16 组大小为10×10的特征映射。如果不使用连接表，则需要 96 个5×5的卷积核。神经元数量为$16 \times 100 = 1600$，可训练参数数量为$(60 \times 25) + 16 = 1516$，连接数为$100 \times 1516 = 151600$。

（4）S4 层是一个汇聚层，采样窗口为2×2，得到 16 个5×5大小的特征映射，可训练参数数量为$16 \times 2 = 32$，连接数为$16 \times 25 \times (4 + 1) = 2000$。

（5）C5 层是一个卷积层，使用$120 \times 16 = 1920$ 个5×5的卷积核，得到 120 组大小为1×1的特征映射。C5 层的神经元数量为 120，可训练参数数量为$1920 \times 25 + 120 = 48120$，连接数为$120 \times (16 \times 25 + 1) = 48120$。

（6）F6 层是一个全连接层，有 84 个神经元，可训练参数数量为$84 \times (120 + 1) = 10164$。连接数和可训练参数个数相同，为 10164。

（7）输出层：输出层由 10 个径向基函数（Radial Basis Function，RBF）组成。

AlexNet

AlexNet[Krizhevsky et al.,2012]是第一个现代深度卷积网络模型，其首次使用了很多现代深度卷积网络的技术方法，比如使用 GPU 进行并行训练，采用了 ReLU 作为非线性激活函数，使用 Dropout 防止过拟合，使用数据增强来提高模型准确率等。AlexNet 赢得了 2012 年 ImageNet 图像分类竞赛的冠军。

AlexNet 的结构如图 5.17 所示，包括 5 个卷积层、3 个汇聚层和 3 个全连接层（其中最后一层是使用 Softmax 函数的输出层）。因为网络规模超出了当时的单个 GPU 的内存限制，AlexNet 将网络拆为两半，分别放在两个 GPU 上，GPU 间只在某些层（比如第 3 层）进行通信。

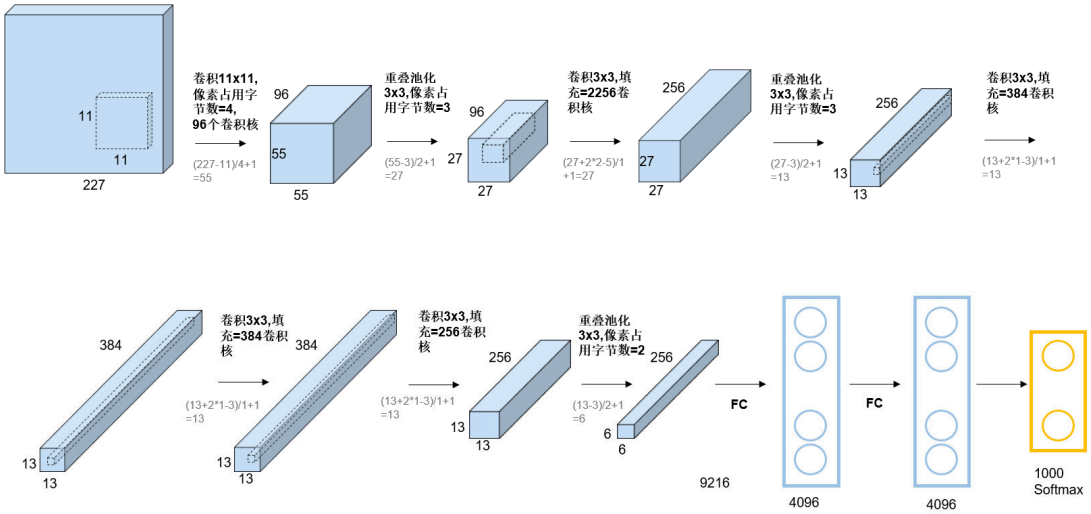


图 5.17 AlexNet 网络结构图(参照[Krizhevsky et al.,2012])

AlexNet 的输入为$224 \times 224 \times 3$的图像，输出为 1000 个类别的条件概率，具体结构如下：

（1）第一个卷积层，使用两个大小为$11 \times 11 \times 3 \times 48$的卷积核，步长$S = 4$，零填充$P = 3$，得到两个大小为$55 \times 55 \times 48$的特征映射组。

（2）第一个汇聚层，使用大小为3×3的最大汇聚操作，步长$S = 2$，得到两个$27 \times 27 \times 48$的特征映射组。

（3）第二个卷积层，使用两个大小为$5 \times 5 \times 48 \times 128$的卷积核，步长$S = 1$，零填充$P = 2$，得到两个大小为$27 \times 27 \times 128$的特征映射组。

（4）第二个汇聚层，使用大小为3×3的最大汇聚操作，步长$S = 2$，得到两个大小为$13 \times 13 \times 128$的特征映射组。

（5）第三个卷积层为两个路径的融合，使用一个大小为$3 \times 3 \times 256 \times 384$的卷积核，步长$S = 1$，零填充$P$=1，得到两个大小为$13 \times 13 \times 192$的特征映射组。

（6）第四个卷积层，使用两个大小为$3 \times 3 \times 192 \times 192$的卷积核，步长$S = 1$，零填充$P = 1$，得到两个大小为$13 \times 13 \times 192$的特征映射组。

（7）第五个卷积层，使用两个大小为$3 \times 3 \times 192 \times 128$的卷积核，步长$S = 1$，零填充$P = 1$，得到两个大小为$13 \times 13 \times 128$的特征映射组。

（8）第三个汇聚层，使用大小为3×3的最大汇聚操作，步长$S = 2$，得到两个大小为$6 \times 6 \times 128$的特征映射组。

（9）三个全连接层，神经元数量分别为 4096、4096 和 1000。

此外，AlexNet 还在前两个汇聚层之后进行了局部响应归一化（Loca lResponse Normalization，LRN）

以增强模型的泛化能力。

ResNet

残差网络（Residual Network，ResNet）通过给非线性的卷积层增加直连边（Shortcut Connection）（也称为残差连接（Residual Connection））的方式来提高信息的传播效率。

假设在一个深度网络中，我们期望一个非线性单元（可以为一层或多层的卷积层）$f(x;\theta)$去逼近一个目标函数为$h(x)$。如果将目标函数拆分成两部分：恒等函数（Identity Function）x和残差函数（Residue Function）$h(x)-x$。

$$h(x)=x+(h(x)-x) \tag{5.60}$$

根据通用近似定理，一个由神经网络构成的非线性单元有足够的能力来近似逼近原始目标函数或残差函数，但实际中后者更容易学习[He et al.,2016]。因此，原来的优化问题可以转换为：让非线性单元$f(x;\theta)$去近似残差函数$h(x)-x$，并用$f(x,\theta)+x$去逼近$h(x)$

下图给出了一个典型的残差单元示例。残差单元由多个级联的（等宽）卷积层和一个跨层的直连边组成，再经过 ReLU 激活后得到输出。残差网络就是将很多个残差单元串联起来构成的一个非常深的网络。和残差网络类似的还有 Highway Network[Srivastava et al.,2015]。

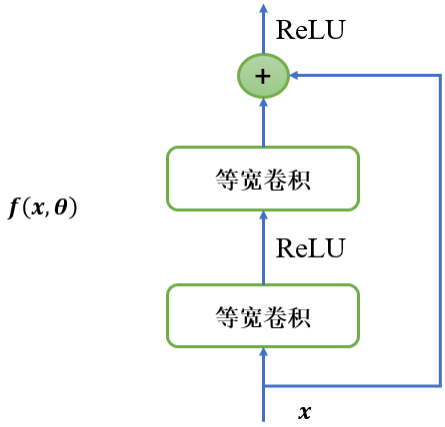


图 5.18 一个简单的残差单元网络

5.5 循环神经网络：

5.5.1序列模型和循环神经网络

序列模型能够应用在许多领域，例如：语音识别、音乐发生器、情感分类、DNA 序列分析、机器翻译、视频动作识别、命名实体识别。语音识别：输入和输出数据都是序列数据,X 是按时序播放的音频片段,输出 Y 是一系列单词.音乐生成：只有输出时序列数据,输入数据可以是空集，也可以是单一整数(指代音乐风格).

在前馈神经网络中，信息的传递是单向的，这种限制虽然使得网络变得更容易学习，但在一定程度上也减弱了神经网络模型的能力。在生物神经网络中，神经元之间的连接关系要复杂得多。前馈神经网络可以看作一个复杂的函数，每次输入都是独立的，即网络的输出只依赖于当前的输入。但是在很多现实任务中，网络的输出不仅和当前时刻的输入相关，也和其过去一段时间的输出相关。比如一个有限状态自动机，其下一个时刻的状态（输出）不仅仅和当前输入相关，也和当前状态（上一个时刻的输出）相关。此外，前馈网络难以处理时序数据，比如视频、语音、文本等。时序数据的长度一般是不固定的，而前馈神经网络要求输入和输出的维数都是固定的，不能任意改变。因此，当处理这一类和时序数据相关的问题时，就需要一种能力更强的模型。

循环神经网络（Recurrent Neural Network，RNN）是一类具有短期记忆能力的神经网络。在循环神经网络中，神经元不但可以接受其他神经元的信息，也可以接受自身的信息，形成具有环路的网络结构。就像卷积网络是专门用于处理网格化数据X（如一个图像）的神经网络，循环神经网络是专门用于处理序列

$x^{(1)},\ldots,x^{(\tau)}$的神经网络。正如卷积网络可以很容易地扩展到具有很大宽度和高度的图像，以及处理大小可变的图像，循环网络可以扩展到更长的序列（比不基于序列的特化网络长得多）。大多数循环网络也能处理可变长度的序列。

和前馈神经网络相比，循环神经网络更加符合生物神经网络的结构。循环神经网络已经被广泛应用在语音识别、语言模型以及自然语言生成等任务上。循环神经网络的参数学习可以通过随时间反向传播算法[Werbos,1990]来学习。随时间反向传播算法即按照时间的逆序将错误信息一步步地往前传递。当输入序列比较长时，会存在梯度爆炸和消失问题[Bengio et al.,1994; Hochreiter et al.,1997,2001]，也称长程依赖问题。为了解决这个问题，人们对循环神经网络进行了很多的改进，其中最有效的改进方式引入门控机制（Gating Mechanism）。循环神经网络可以很容易地扩展到两种更广义的记忆网络模型：递归神经网络和图网络。

5.5.2序列模型的计算图

计算图是形式化一组计算结构的方式，如那些涉及将输入和参数映射到输出和损失的计算。我们对展开（unfolding）递归或循环计算得到的重复结构进行解释，这些重复结构通常对应于一个事件链。展开（unfolding）这个计算图将导致深度网络结构中的参数共享。

例如，考虑动态系统的经典形式

$$s^{(t)} = f(s^{(t-1)};\theta) \tag{5.61}$$

其中$s^{(t)}$称为系统的状态。

s在时刻 t 的定义需要参考时刻 t−1 时同样的定义，因此上式是循环的。对有限时间步τ，$\tau-1$次应用这个定义可以展开这个图。例如$\tau=3$，我们对式(5.58)展开，可以得到：

$$s^{(3)} = f\left(s^{(2)};\theta\right) \tag{5.62}$$

$$= f(f\left(s^{(1)};\theta\right);\theta) \tag{5.63}$$

以这种方式重复应用定义，展开等式，就能得到不涉及循环的表达。现在我们可以使用传统的有向无环计算图呈现这样的表达.上式的展开计算图如图 5.19 所示。

图 5.19 循环神经网络展开的计算图。每个节点表示在某个时刻t的状态，并且函数f将t处的状态映射到$t+1$处的状态。所有时间步都使用相同的参数（用于参数化f的相同θ值）。

作为另一个例子，让我们考虑由外部信号$x^{(t)}$驱动的动态系统，

$$s^{(t)} = f\left(s^{(t-1)},x^{(t)};\theta\right) \tag{5.64}$$

我们可以看到，当前状态包含了整个过去序列的信息。

循环神经网络可以通过许多不同的方式建立。就像几乎所有函数都可以被认为是前馈网络，本质上任何涉及循环的函数都可以被认为是一个循环神经网络。很多循环神经网络公式定义隐藏单元的值，为了表明状态是网络的隐藏单元，我们使用变量 h 代表状态重写上式：

$$h^{(t)} = f\left(h^{(t-1)},x^{(t)};\theta\right) \tag{5.65}$$

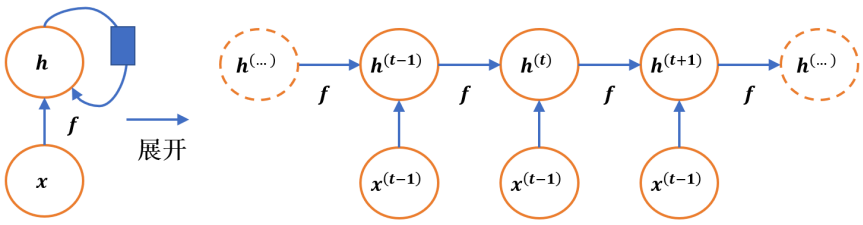


图 5.20 没有输出的循环网络。此循环网络只处理来自输入x的信息，将其合并到经过时间向前传播的状态h。(左)回路原理图。黑色方块表示单个时间步的延迟。(右)同一网络被视为展开的计算图，其中每个节点现在与一个特定的时间实例相关联。

当训练循环网络根据过去预测未来时，网络通常要学会使用$h^{(t)}$作为过去序列（直到 t）与任务相关方面的有损摘要。此摘要一般而言一定是有损的，因为其映射任意长度的序列$(x^{(t)},x^{(t-1)},x^{(t-2)},\ldots,x^{(2)},x^{(1)})$

到一固定长度的向量$h^{(t)}$。根据不同的训练准则，摘要可能选择性地精确保留过去序列的某些方面。例如，如果在统计语言建模中使用的 RNN，通常给定前一个词预测下一个词，可能没有必要存储时刻t前输入序列中的所有信息；而仅仅存储足够预测句子其余部分的信息。

我们可以用一个函数$g^{(t)}$代表经t步展开后的循环：

$$h^{(t)} = g^{(t)}\left(x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(2)}, x^{(1)}\right) \tag{5.66}$$

$$= f\left(h^{(t-1)}, x^{(t)}; \theta\right) \tag{5.67}$$

函数$g(t)$将全部的过去序列$x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(2)}, x^{(1)}$作为输入来生成当前状态，但是展开的循环架构允许我们将$g(t)$分解为函数 f 的重复应用。因此，展开过程引入两个主要优点：

1.无论序列的长度，学成的模型始终具有相同的输入大小，因为它指定的是从一种状态到另一种状态的转移，而不是在可变长度的历史状态上操作。

2.我们可以在每个时间步使用相同参数的相同转移函数f。

这两个因素使得学习在所有时间步和所有序列长度上操作单一的模型f是可能的，而不需要在所有可能时间步学习独立的模型$g(t)$。学习单一的共享模型允许泛化到没有见过的序列长度（没有出现在训练集中），并且估计模型所需的训练样本远远少于不带参数共享的模型。

5.5.3普通循环神经网络

基于上一节中的图展开和参数共享的思想，我们可以设计各种循环神经网络。RNN 是一种特殊的神经网络结构,它是根据人的认知是基于过往的经验和记忆这一观点提出的.它与 DNN,CNN 不同的是:它不仅考虑前一时刻的输入,而且赋予了网络对前面的内容的一种记忆功能.

RNN 之所以称为循环神经网路，即一个序列当前的输出与前面的输出也有关。具体的表现形式为网络会对前面的信息进行记忆并应用于当前输出的计算中，即隐藏层之间的节点不再无连接而是有连接的，并且隐藏层的输入不仅包括输入层的输出还包括上一时刻隐藏层的输出。

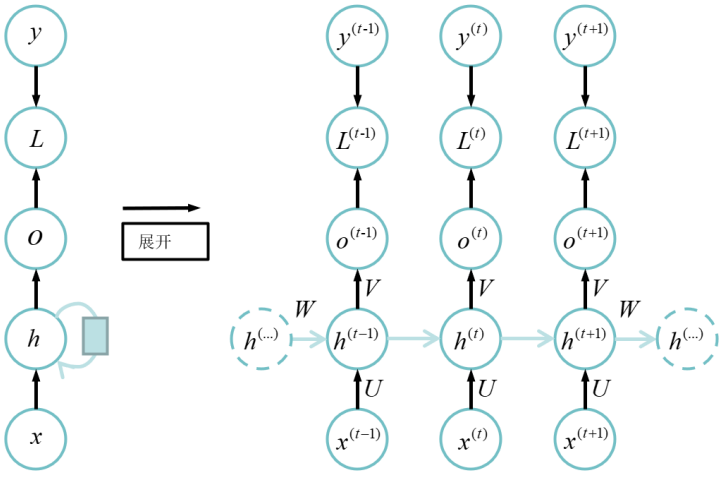


图 5.21 计算循环网络(将x值的输入序列映射到输出值o的对应序列)训练损失的计算图。损失L衡量每个o与相应的训练目标y的距离。当使用$softmax$输出时，我们假设o是未归一化的对数概率。损失 L 内部计算$\hat{y} = softmax(o)$，并将其与目标y比较。RNN 输入到隐藏的连接由权重矩阵U参数化，隐藏到隐藏的循环连接由权重矩阵W参数化以及隐藏到输出的连接由权重矩阵V参数化。(左)使用循环连接绘制的 RNN 和它的损失。(右)同一网络被视为展开的计算图，其中每个节点现在与一个特定的时间实例相关联。

上图是一个标准的 RNN 结构图，图中每个箭头代表做一次变换，也就是说箭头连接带有权值。左侧是折叠起来的样子，右侧是展开的样子，左侧中h旁边的箭头代表此结构中的“循环“体现在隐层。在展开结构中我们可以观察到，在标准的 RNN 结构中，隐层的神经元之间也是带有权值的。也就是说，随着序列的不断推进，前面的隐层将会影响后面的隐层。图中o代表输出，y代表样本给出的确定值，L代表损失函数，我们可以看到，“损失”也是随着序列的推荐而不断积累的。

除了上述特点之外，标准 RNN 的还有以下特点：

1. 权值参数共享，图中的W全是相同的，U和V也一样。
2. 每一个输入值都只与它本身的那条路线建立权连接，不会和别的神经元连接。
3. 隐藏状态可以理解为:$h = f(现有的输入 + 过去记忆总结)$

Rnn 的几种类型：

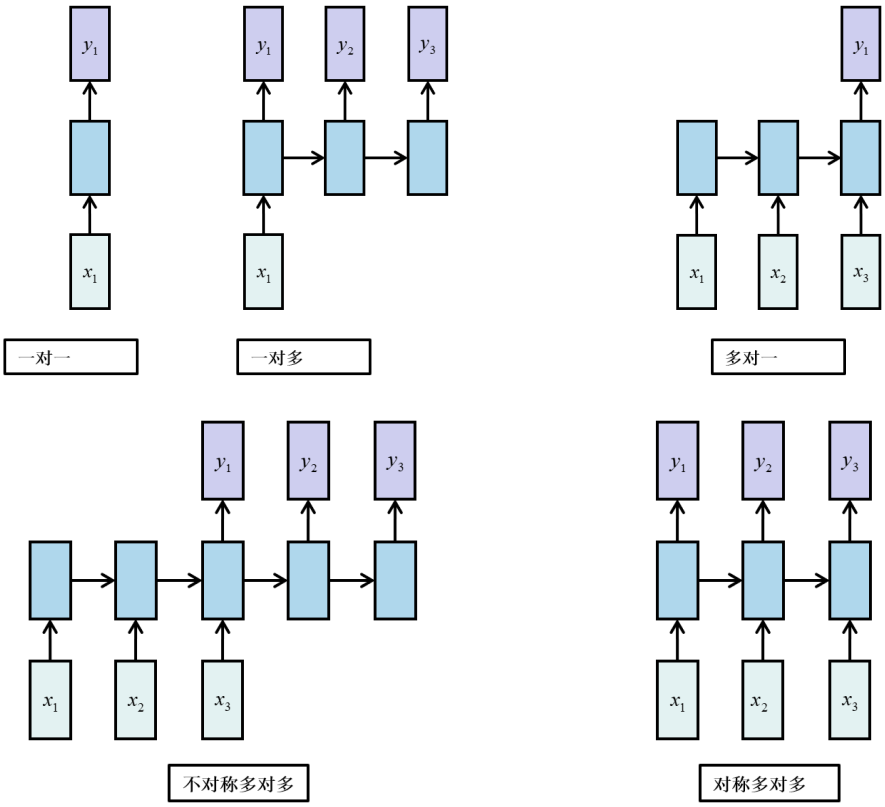


图 5.22 多种不同类型的 RNN 结构示意图

RNN 有一对一，一对多，多对一，多对多几种不同的网络类型，如图 5.21 所示。

一对多 RNN 通常用于图像描述，输入图像，输出一段描述性的文字；

多对一 RNN 用于文本情感分析，输入描述的文字，输出感情类别；

多对多 RNN 用于机器翻译。

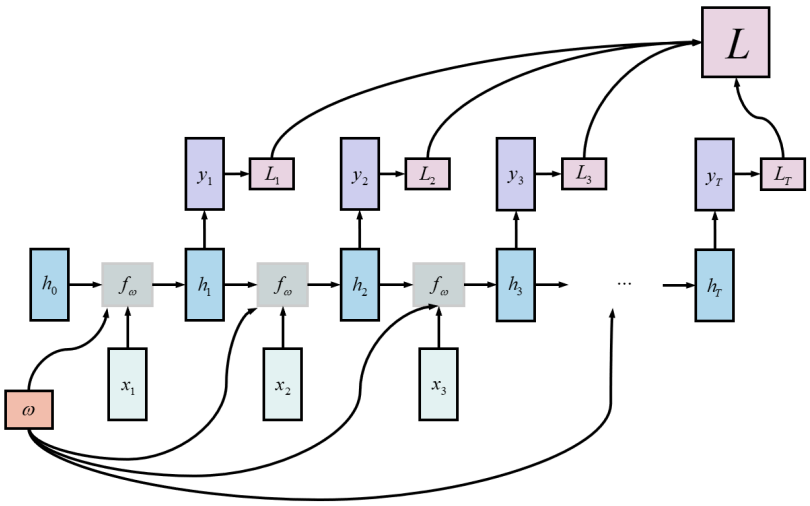


图 5.23 多对多 RNN 的计算图

梯度随时间反向传播：

现在我们研究图 5.23 中 RNN 的前向传播公式。这个图没有指定隐藏单元的激活函数。我们假设使用双曲正切激活函数。此外，图中没有明确指定何种形式的输出和损失函数。我们假定输出是离散的，如用于预测词或字符的 RNN。表示离散变量的常规方式是把输出 o 作为每个离散变量可能值的非标准化对数概率。然后，我们可以应用 softmax 函数后续处理后，获得标准化后概率的输出向量$\hat{y}$。RNN 从特定的初始状态$h^{(0)}$开始前向传播。从$t=1$到$t=\tau$的每个时间步，我们应用以下更新方程：

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t-1)} \tag{5.68}$$

$$h^{(t)} = \tanh(a^{(t)}) \tag{5.69}$$

$$o^{(t)} = c + Vh^{(t)} \tag{5.70}$$

$$\hat{y}^{(t)} = softmax(o^{(t)}) \tag{5.71}$$

其中的参数的偏置向量b和c连同权重矩阵U、V和W，分别对应于输入到隐藏、隐藏到输出和隐藏到隐藏的连接。这个循环网络将一个输入序列映射到相同长度的输出序列。与x序列配对的y的总损失就是所有时间步的损失之和。例如，$L^{(t)}$为给定的$x^{(1)},\dots,x^{(t)}$后$y^{(t)}$的负对数似然，则

$$L(\{x^{(1)},\dots,x^{(t)}\},\{y^{(1)},\dots,y^{(t)}\}) = \sum_t L^{(t)} \tag{5.72}$$

$$= -\sum_t \log p_{model}(y^{(t)}|\{x^{(1)},\dots,x^{(t)}\}) \tag{5.73}$$

其中$p_{model}(y^{(t)}|\{x^{(1)},\dots,x^{(t)}\})$需要读取模型输出向量$\hat{y}^{(t)}$中对应于$y^{(t)}$的项。关于各个参数计算这个损失函数的梯度是计算成本很高的操作。梯度计算涉及执行一次前向传播（如在图 10.3 展开图中从左到右的传播），接着是由右到左的反向传播。运行时间是$O(\tau)$，并且不能通过并行化来降低，因为前向传播图是固有循序的;每个时间步只能一前一后地计算。前向传播中的各个状态必须保存，直到它们反向传播中被再次使用，因此内存代价也是$O(\tau)$。应用于展开图且代价为$O(\tau)$的反向传播算法称为通过时间反向传播（back-propagation through time,BPTT）。

BPTT（back-propagation through time）算法是常用的训练 RNN 的方法，其实本质还是 BP 算法，只不过 RNN 处理时间序列数据，所以要基于时间反向传播，故叫随时间反向传播。BPTT 的中心思想和 BP 算法相同，沿着需要优化的参数的负梯度方向不断寻找更优的点直至收敛。综上所述，BPTT 算法本质还是 BP 算法，BP 算法本质还是梯度下降法，那么求各个参数的梯度便成了此算法的核心。

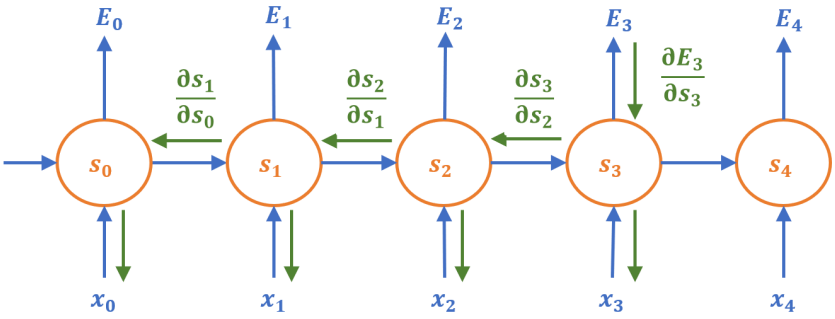


图 5.24 RNN 的梯度通过时间反向传播算法示意图

要寻优的参数有三个，分别是U,V,W。与 BP 算法不同的是，其中是W和是U两个参数的寻优过程需要追溯之前的历史数据，参数是V相对简单只需关注目前，那么我们就来先求解参数V的偏导数。

$$\frac{\partial L^{(t)}}{\partial V} = \frac{\partial L^{(t)}}{\partial o^{(t)}} \cdot \frac{\partial o^{(t)}}{\partial V} \tag{5.74}$$

这个式子看起来简单但是求解起来很容易出错，因为其中嵌套着激活函数函数，是复合函数的求道过程。RNN 的损失也是会随着时间累加的，所以不能只求 t 时刻的偏导。

$$L = \sum_{t=1}^{n} L^{(t)} \tag{5.75}$$

$$\frac{\partial L}{\partial V} = \sum_{t=1}^{n} \frac{\partial L^{(t)}}{\partial o^{(t)}} \cdot \frac{\partial o^{(t)}}{\partial V} \tag{5.76}$$

W和U的偏导的求解由于需要涉及到历史数据，其偏导求起来相对复杂，我们先假设只有三个时刻，那么在第三个时刻L对W的偏导数为：

$$\frac{\partial L^{(3)}}{\partial W}=\frac{\partial L^{(3)}}{\partial o^{(3)}}\frac{\partial o^{(3)}}{\partial h^{(3)}}\frac{\partial h^{(3)}}{\partial W}+\frac{\partial L^{(3)}}{\partial o^{(3)}}\frac{\partial o^{(3)}}{\partial h^{(3)}}\frac{\partial h^{(3)}}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial W}+\frac{\partial L^{(3)}}{\partial o^{(3)}}\frac{\partial o^{(3)}}{\partial h^{(3)}}\frac{\partial h^{(3)}}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial h^{(1)}}\frac{\partial h^{(1)}}{\partial W} \tag{5.77}$$

相应的，L在第三个时刻对U的偏导数为：

$$\frac{\partial L^{(3)}}{\partial U}=\frac{\partial L^{(3)}}{\partial o^{(3)}}\frac{\partial o^{(3)}}{\partial h^{(3)}}\frac{\partial h^{(3)}}{\partial U}+\frac{\partial L^{(3)}}{\partial o^{(3)}}\frac{\partial o^{(3)}}{\partial h^{(3)}}\frac{\partial h^{(3)}}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial U}+\frac{\partial L^{(3)}}{\partial o^{(3)}}\frac{\partial o^{(3)}}{\partial h^{(3)}}\frac{\partial h^{(3)}}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial h^{(1)}}\frac{\partial h^{(1)}}{\partial U} \tag{5.78}$$

可以观察到，在某个时刻的对或是的偏导数，需要追溯这个时刻之前所有时刻的信息，这还仅仅是一个时刻的偏导数，上面说过损失也是会累加的，那么整个损失函数对和的偏导数将会非常繁琐。虽然如此但好在规律还是有迹可循，我们根据上面两个式子可以写出 L 在时刻对和偏导数的通式：

$$\frac{\partial L^{(t)}}{\partial W}=\sum_{k=0}^{t}\frac{\partial L^{(t)}}{\partial o^{(t)}}\frac{\partial o^{(t)}}{\partial h^{(t)}}\left(\prod_{j=k+1}^{t}\frac{\partial h^{(j)}}{\partial h^{(j-1)}}\right)\frac{\partial h^{(k)}}{\partial W} \tag{5.79}$$

$$\frac{\partial L^{(t)}}{\partial U}=\sum_{k=0}^{t}\frac{\partial L^{(t)}}{\partial o^{(t)}}\frac{\partial o^{(t)}}{\partial h^{(t)}}\left(\prod_{j=k+1}^{t}\frac{\partial h^{(j)}}{\partial h^{(j-1)}}\right)\frac{\partial h^{(k)}}{\partial U} \tag{5.80}$$

整体的偏导公式就是将其按时刻再一一加起来。前面说过激活函数是嵌套在里面的，如果我们把激活函数放进去，拿出中间累乘的那部分：

$$\frac{\partial L^{(t)}}{\partial U}=\sum_{k=0}^{t}\frac{\partial L^{(t)}}{\partial o^{(t)}}\frac{\partial o^{(t)}}{\partial h^{(t)}}\left(\prod_{j=k+1}^{t}\frac{\partial h^{(j)}}{\partial h^{(j-1)}}\right)\frac{\partial h^{(k)}}{\partial U} \tag{5.81}$$

我们会发现累乘会导致激活函数导数的累乘，进而会导致梯度消失和梯度爆炸现象的发生。下一节会介绍梯度消失和梯度爆炸及其解决办法。

5.5.4LSTM长短时记忆神经网络

由于循环神经网络经常使用非线性激活函数为 Logistic 函数或 Tanh 函数作为非线性激活函数，其导数值都小于 1，并且权重矩阵$\|U\|$也不会太大，因此如果时间间隔$t-k$过大，$\delta_{t,k}$会趋向于 0，因而经常会出现梯度消失问题。

虽然简单循环网络理论上可以建立长时间间隔的状态之间的依赖关系，但是由于梯度爆炸或消失问题，实际上只能学习到短期的依赖关系。这样，如果时刻t的输出y^t依赖于时刻k的输入x_k，当间隔$t-k$比较大时，简单神经网络很难建模这种长距离的依赖关系，称为长程依赖问题（Long-Term Dependencies Problem）。

为了避免梯度爆炸或消失问题，一种最直接的方式就是选取合适的参数，同时使用非饱和的激活函数，尽量使得$diag(f'(z))U^T\approx 1$，这种方式需要足够的人工调参经验，限制了模型的广泛应用。比较有效的方式是通过改进模型或优化方法来缓解循环网络的梯度爆炸和梯度消失问题。

梯度爆炸

一般而言，循环网络的梯度爆炸问题比较容易解决，一般通过权重衰减或梯度截断来避免。权重衰减是通过给参数增加 $\ell 1$ 或 $\ell 2$ 范数的正则化项来限制参数的取值，从而使得$\gamma\leq 1$。梯度截断是另一种有效的启发式方法，当梯度的模大于一定阈值时，就将它截断成为一个较小的数。梯度消失梯度消失是循环网络的主要问题。除了使用一些优化技巧外，更有效的方式就是改变模型，如让$U=I$，同时$\frac{\partial h_t}{\partial h_{t-1}}=I$为单位矩阵，即

$$h_t=h_{t-1}+g(x_t;\theta) \tag{5.82}$$

其中$g(\cdot)$是一个非线性函数，θ为参数。

上式中，h_t和h_{t-1}之间为线性依赖关系，且权重系数为 1，这样就不存在梯度爆炸或消失问题。但是，这种改变也丢失了神经元在反馈边上的非线性激活的性质，因此也降低了模型的表示能力。

为了避免这个缺点，我们可以采用一种更加有效的改进策略：

$$h_t=h_{t-1}+g(x_t,h_{t-1};\theta) \tag{5.83}$$

这样h_t和h_{t-1}之间为既有线性关系，也有非线性关系，并且可以缓解梯度消失问题。但这种改进依然存在两个问题：

（1）梯度爆炸问题：

令$z_k = Uh_{k-1} + Wx_k + b$为在第$k$时刻函数$g(\cdot)$的输入，在计算误差项$\delta_{t,k} = \frac{\partial \mathcal{L}_t}{\partial z_k}$时，梯度可能会过大，从而导致梯度爆炸问题。

（2）记忆容量（Memory Capacity）问题：

随着h_t不断累积存储新的输入信息，会发生饱和现象。假设$g(\cdot)$为 Logistic 函数，则随着时间t的增长，h_t会变得越来越大，从而导致h变得饱和。也就是说，隐状态h_t可以存储的信息是有限的，随着记忆单元存储的内容越来越多，其丢失的信息也越来越多。为了解决这两个问题，可以通过引入门控机制来进一步改进模型。

基于门控的循环神经网络：

为了改善循环神经网络的长程依赖问题，一种非常好的解决方案是在上式的基础门控机制来控制信息的累积速度，包括有选择地加入新的信息，并有选择地遗忘之前累积的信息。这一类网络可以称为基于门控的循环神经网络（Gated RNN）。本节中，主要介绍两种基于门控的循环神经网络：长短期记忆网络和门控循环单元网络。

长短期神经网络：

长短期记忆网络（Long Short-Term Memory Network，LSTM）[Gers et al.,2000;Hochreiter et al.,1997]循环神经网络的一个变体，可以有效地解决简单循环神经网络的梯度爆炸或消失问题。

LSTM 网络主要改进在以下两个方面：

新的内部状态

LSTM 网络引入一个新的内部状态（internal state）$c_t \in \mathcal{R}^D$专门进行线性的循环信息传递，同时（非线性地）输出信息给隐藏层的外部状态$h_t \in \mathcal{R}^D$。内部状态$c_t ct$通过下面公式计算：

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{5.84}$$

$$h_t = o_t \odot \tanh(c_t) \tag{5.85}$$

其中$f_t \in [0,1]^D$、$i_t \in [0,1]^D$和$o_t \in [0,1]^D$为三个门（gate）来控制信息传递的路径；$\odot$为向量元素乘积；c_{t-1}为上一时刻的记忆单元；$\tilde{c}_t \in \mathbb{R}^D$是通过非线性函数得到的候选状态：

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{5.86}$$

在每个时刻t，LSTM 网络的内部状态c_t记录了到当前时刻为止的历史信息。

门控机制

在数字电路中，门（gate）为一个二值变量{0,1}，0 代表关闭状态，不许任何信息通过；1 代表开放状态，允许所有信息通过。

LSTM 网络引入门控机制（Gating Mechanism）来控制信息传递的路径。公式中三个“门”分别为输入门i_t、遗忘门f_t和输出门o_t。这三个门的作用为

（1）遗忘门f_t控制上一个时刻的内部状态c_{t-1}需要遗忘多少信息。

（2）输入门i_t控制当前时刻的候选状态$\tilde{c}_t$有多少信息需要保存。

（3）输出门o_t控制当前时刻的内部状态c_t有多少信息需要输出给外部状态h_t。

当$f_t = 0,\ i_t = 1$时，记忆单元将历史信息清空，并将候选状态向量$\tilde{c}_t$写入。但此时记忆单元c_t依然和上一时刻的历史信息相关。当$f_t = 1, i_t = 0$时，记忆单元将复制上一时刻的内容，不写入新的信息。

LSTM 网络中的“门”是一种“软”门，取值在(0,1)之间，表示以一定的比例允许信息通过。三个门的计算方式为：

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{5.87}$$

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{5.88}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{5.89}$$

其中$\sigma(\cdot)$为 Logistic 函数，其输出区间为(0,1)，x_t为当前时刻的输入，h_{t-1}为上一时刻的外部状态。

图 5.24 给出了 LSTM 网络的循环单元结构，其计算过程为：

1）首先利用上一时刻的外部状态h_{t-1}和当前时刻的输入x_t，计算出三个门，以及候选状态$\tilde{c}_t$；

2）结合遗忘门f_t和输入门i_t来更新记忆单元c_t；

3）结合输出门o_t，将内部状态的信息传递给外部状态h_t。

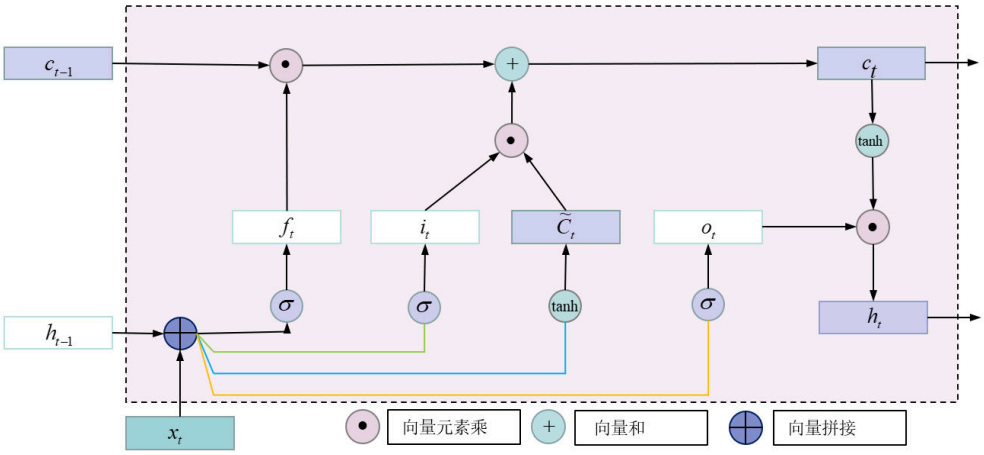


图 5.25 LSTM 网络的循环单元结构

通过 LSTM 循环单元，整个网络可以建立较长距离的时序依赖关系,上述公式可以简洁地描述为:

$$\begin{bmatrix} \tilde{c}_t \\ o_t \\ i_t \\ f_t \end{bmatrix} = \begin{bmatrix} \tanh \\ \sigma \\ \sigma \\ \sigma \end{bmatrix} \left(W \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} + b \right) \tag{5.90}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{5.91}$$

$$h_t = o_t \odot \tanh(c_t) \tag{5.92}$$

其中$x_t \in \mathcal{R}^M$为当前时刻的输入，$W \in \mathcal{R}^{4D \times (D+M)}$和$b \in \mathcal{R}^{4D}$为网络参数.

记忆

循环神经网络中的隐状态 h 存储了历史信息，可以看作一种记忆（Memory）。在简单循环网络中，隐状态每个时刻都会被重写，因此可以看作一种短期记忆（Short-Term Memory）。在神经网络中，长期记忆（Long-Term Memory）可以看作网络参数，隐含了从训练数据中学到的经验，其更新周期要远远慢于短期记忆。而在 LSTM 网络中，记忆单元 c 可以在某个时刻捕捉到某个关键信息，并有能力将此关键信息保存一定的时间间隔。记忆单元 c 中保存信息的生命周期要长于短期记忆 h，但又远远短于长期记忆，因此称为长短期记忆（Long Short-Term Memory）。长短期记忆是指长的“短期记忆”.

一般在深度网络参数学习时，参数初始化的值一般都比较小。但是在训练 LSTM 网络时，过小的值会使得遗忘门的值比较小。这意味着前一时刻的信息大部分都丢失了，这样网络很难捕捉到长距离的依赖信息。并且相邻时间间隔的梯度会非常小，这会导致梯度弥散问题。因此遗忘的参数初始值一般都设得比较大，其偏置向量b_f设为 1 或 2。

5.5.5深层循环神经网络

堆叠循环神经网络

一种常见的增加循环神经网络深度的做法是将多个循环网络堆叠起来，称为堆叠循环神经网络（Stacked Recurrent Neural Network，SRNN）。一个堆叠的简单循环网络（Stacked SRN）也称为循环多层感知器（Recurrent MultiLayer Perceptron，RMLP）(Parlos et al.,1991)。图 5.26 给出了按时间展开的堆叠循环神经网络。第l层网络的输入是第$l-1$层网络的输出。我们定义$\boldsymbol{h}_t^{(l)}$为在时刻t时第l层的隐状态

$$h_t^{(l)} = f\left(U^{(l)}h_{t-1}^{(l)} + W^{(l)}h_t^{(l-1)} + b^{(l)}\right) \tag{5.93}$$

其中$U^{(l)}$、$W^{(l)}$和$b^{(l)}$为权重矩阵和偏置向量，$h_t^{(0)} = x_t$。

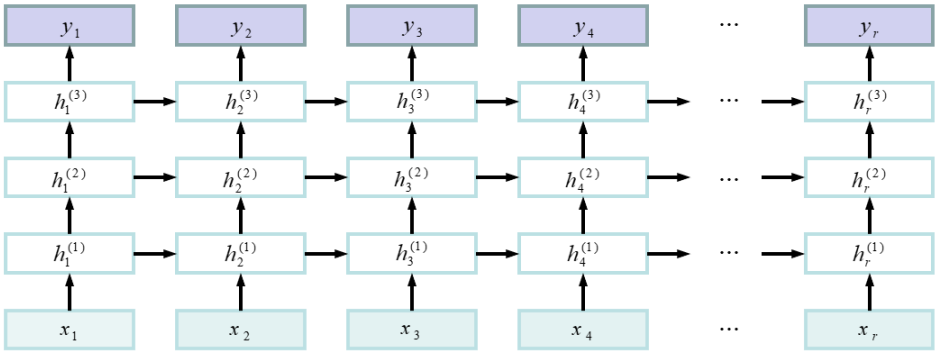


图 5.26 按时间展开的堆叠神经网络

双向循环神经网络

在有些任务中，一个时刻的输出不但和过去时刻的信息有关，也和后续时刻的信息有关。比如给定一个句子，其中一个词的词性由它的上下文决定，即包含左右两边的信息。因此，在这些任务中，我们可以增加一个按照时间的逆序来传递信息的网络层，来增强网络的能力。

双向循环神经网络（Bidirectional Recurrent Neural Network，Bi-RNN）由两层循环神经网络组成，它们的输入相同，只是信息传递的方向不同。假设第 1 层按时间顺序，第 2 层按时间逆序，在时刻t时的隐状态定义为 $h_t^{(1)}$和 $h_t^{(2)}$，则

$$h_t^{(1)} = f\left(U^{(1)}h_{t-1}^{(1)} + W^{(1)}x_t + b^{(1)}\right) \tag{5.94}$$

$$h_t^{(2)} = f\left(U^{(2)}h_{t+1}^{(2)} + W^{(2)}x_t + b^{(2)}\right) \tag{5.95}$$

$$\mathrm{h_t} = \mathrm{h_t^{(1)}} \oplus \mathrm{h_t^{(2)}} \tag{5.96}$$

其中⊕为向量拼接操作。图 5.27 给出了按时间展开的双向循环神经网络。

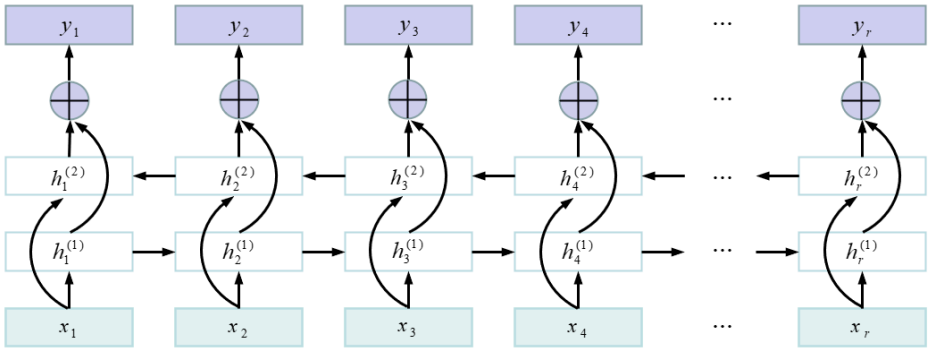


图 5.27 按时间展开的双向循环神经网络

5.6 深度学习实验：

5.6.1实验一：四种梯度下降

（1）随机梯度下降

导入环境

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

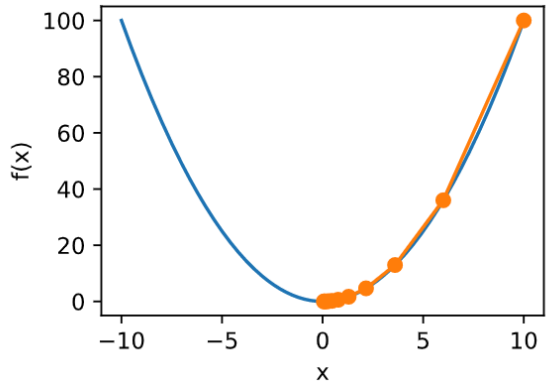
1）一维梯度下降

```
def gd(eta):
    x = 10
    results = [x]
    for i in range(10):
        x -= eta * 2 * x
        results.append(x)
    print('epoch 10, x:', x)
    return results

res = gd(0.2)
```

输出：epoch10,x:0.06046617599999997

画出轨迹图：

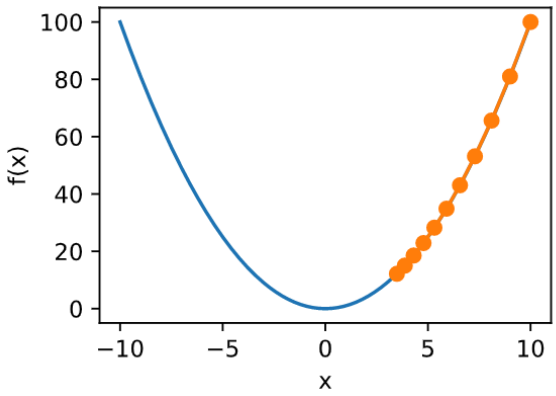
```
def show_trace(res):
    n = max(abs(min(res)), abs(max(res)), 10)
    f_line = np.arange(-n, n, 0.1)
    d2l.set_figsize()
    d2l.plt.plot(f_line, [x * x for x in f_line])
    d2l.plt.plot(res, [x * x for x in res], '-o')
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('f(x)')

show_trace(res)
```



调整学习率：

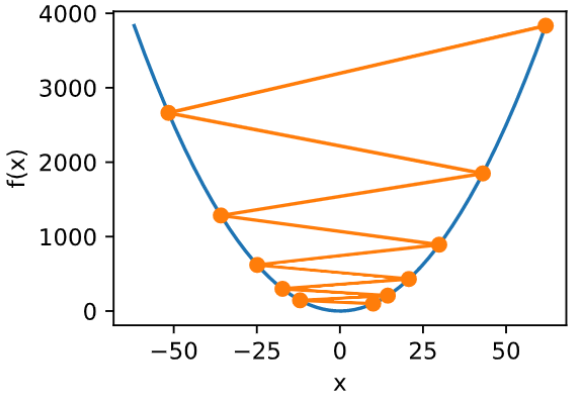
```
show_trace(gd(0.05))
```

输出：epoch10,x:3.4867844009999995




```
show_trace(gd(1.1))
```

输出：epoch10,x:61.917364224000096



2）多维梯度下降：

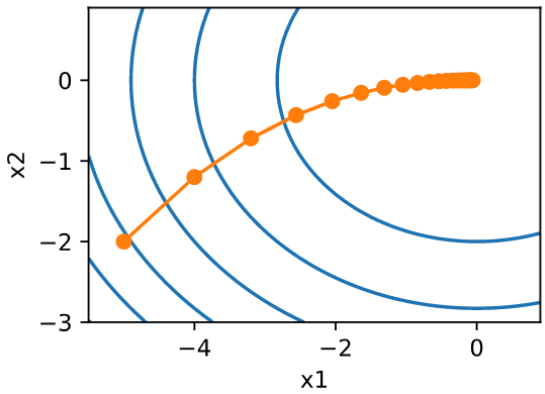
训练与画图：

```
def train_2d(trainer):
    x1, x2, s1, s2 = -5, -2, 0, 0    # s1 和 s2 是自变量状态
    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):
    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')
```

设置目标函数并调用训练过程：

```
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))
```

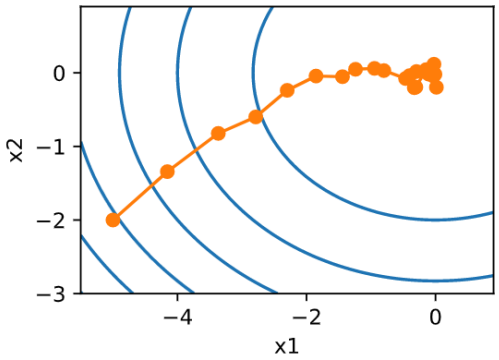
输出：epoch20,x1-0.057646,x2-0.000073



3）随机梯度下降：

```
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：epoch20,x1-0.008833,x2-0.019536



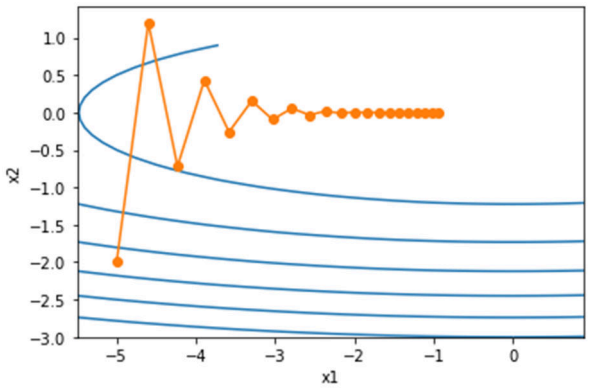
4）梯度消失

导入要调用的包：

```
import sys
sys.path.append("..")
import d2lzh_pytorch as d2l
import torch
eta = 0.4
```

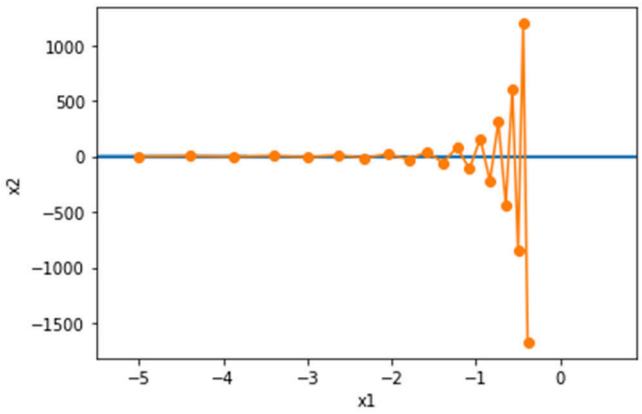
画出轨迹：

```
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))
```



5）梯度爆炸

```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```



（2）小批量随机梯度下降

导入包：

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

1）读取数据

```
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) # 标准化
    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()
```

输出：torch.Size([1500,5])

2）从零开始

```
def sgd(params, states, hyperparams):
    for p in params:
        p.data -= hyperparams['lr'] * p.grad.data
```

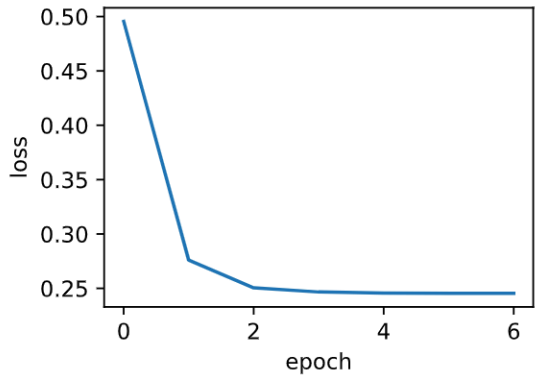
设置损失函数，实现小批量随机梯度下降：

```
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)), dtype=torch.float32), 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')
```

调用小批量随机梯度下降：

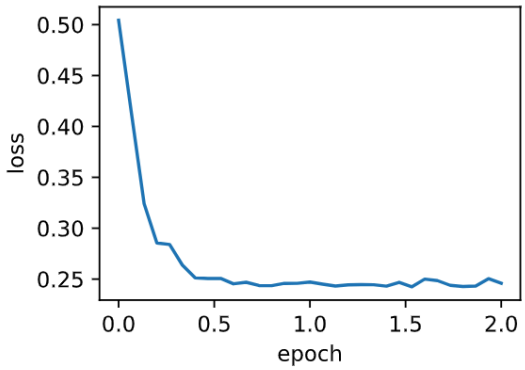
```
def train_sgd(lr, batch_size, num_epochs=2):
    train_ch7(sgd, None, {'lr': lr}, features, labels, batch_size, num_epochs)

train_sgd(1, 1500, 6)
```

输出：loss:0.245416,0.013962sec per epoch



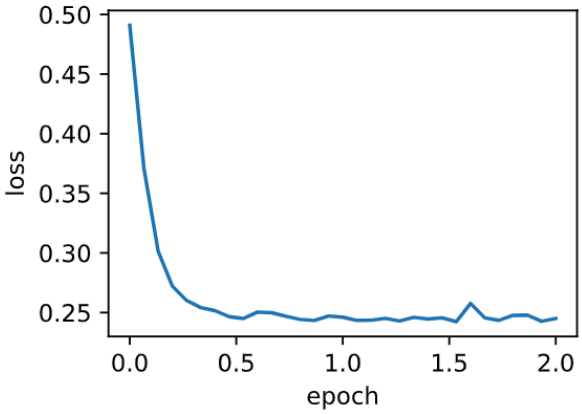
输出：loss:0.245993,0.546791sec per epoch

```
train_sgd(0.005, 1)
```




```
train_sgd(0.05, 10)
```

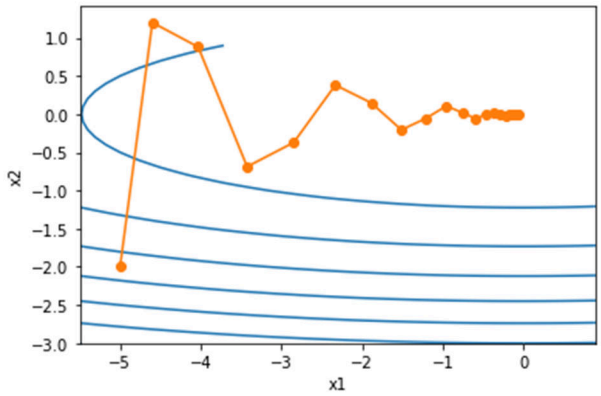
输出：loss:0.245057,0.076257sec per epoch



（3）动量法

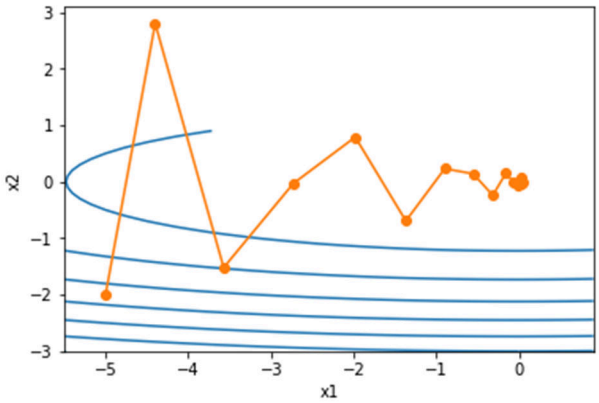
```
def momentum_2d(x1, x2, v1, v2):
    v1 = gamma * v1 + eta * 0.2 * x1
    v2 = gamma * v2 + eta * 4 * x2
    return x1 - v1, x2 - v2, v1, v2

eta, gamma = 0.4, 0.5
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

输出：epoch20,x1-0.062843,x20.001202




```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

输出：epoch20,x10.007188,x20.002553



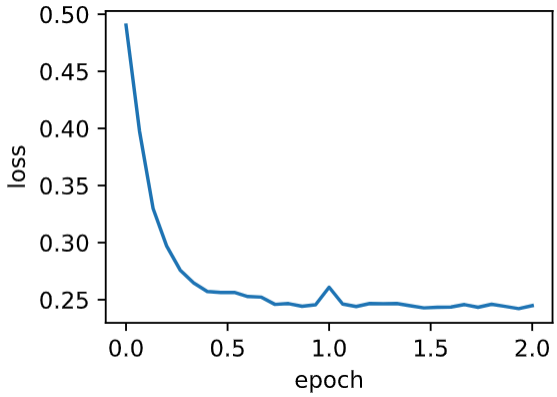
从零开始实现：

```
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
```

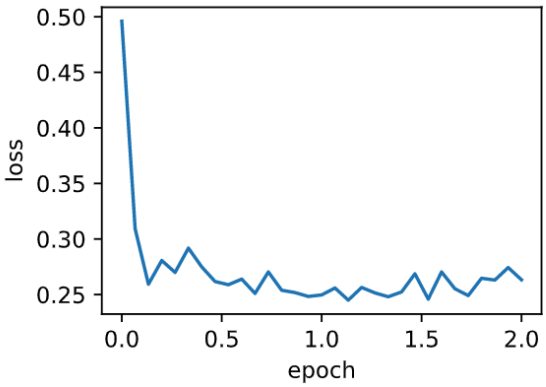
```
d2l.train_ch7(sgd_momentum, init_momentum_states(),{'lr': 0.02, 'momentum': 0.5}, features,
```

loss:0.244809,0.083222sec per epoch




```
d2l.train_ch7(sgd_momentum, init_momentum_states(), {'lr': 0.02, 'momentum': 0.9}, features,
```

loss:0.263299,0.080856sec per epoch



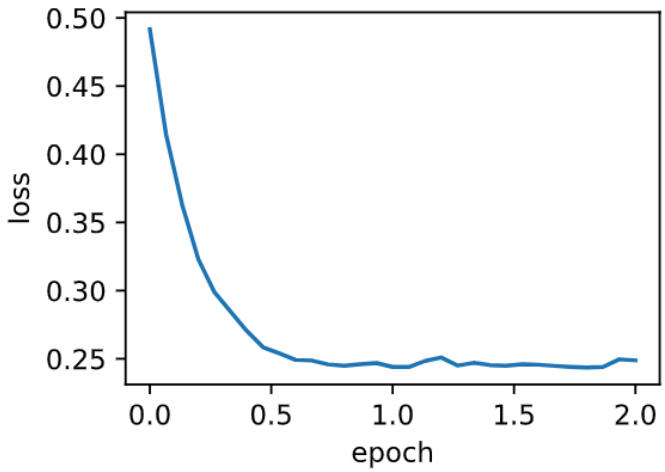
（4）Adam法

从零开始

```
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(1,
dtype=torch.float32)
    s_w, s_b = torch.zeros((features.shape[1], 1), dtype=torch.float32), torch.zeros(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

d2l.train_ch7(adam, init_adam_states(), {'lr': 0.01, 't': 1}, features, labels)
```

loss:0.248828,0.106953secperepoch



5.6.2实验二：cnn

1）卷积

```
import torch
from torch import nn

def corr2d(X, K):
    h, w = K.shape
    X, K = X.float(), K.float()
    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

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

# 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：
Step5,loss2.032
Step10,loss0.278
Step15,loss0.045
Step20,loss0.009

2）填充和步幅

填充：

```
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:])  # 排除不关心的前两维：批量和通道

# 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
```

步幅：

```
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
print(comp_conv2d(conv2d, X).shape)

conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
print(comp_conv2d(conv2d, X).shape)
```

输出：
torch.Size([4,4])
torch.Size([2,2])

3）多通道

a)多输入通道：

```
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.corr2d(X[i, :, :], K[i, :, :])
    return res

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.]])

b)多输出通道：

```
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])
K = torch.stack([K, K + 1, K + 2])
print(K.shape)
print(corr2d_multi_in_out(X, K))
```

输出：

torch.Size([3,2,2,2])
tensor([[[56.,72.],
[104.,120.]],

[[76.,100.],
[148.,172.]],

[[96.,128.],
[192.,224.]]])

4）池化

a)二维最大池化，平均池化

```
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
X = torch.tensor([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
print(pool2d(X, (2, 2)))
print(pool2d(X, (2, 2), 'avg'))
```

输出：

tensor([[4.,5.],
[7.,8.]])
tensor([[2.,3.],
[5.,6.]])

b)池化的填充和步幅

```
X = torch.arange(16, dtype=torch.float).view((1, 1, 4, 4))
print(X)

pool2d = nn.MaxPool2d(3)
print(pool2d(X))

pool2d = nn.MaxPool2d(3, padding=1, stride=2)
print(pool2d(X))

pool2d = nn.MaxPool2d((2, 4), padding=(1, 2), stride=(2, 3))
print(pool2d(X))
```

输出：

tensor([[[[0.,1.,2.,3.],

[4.,5.,6.,7.],
[8.,9.,10.,11.],
[12.,13.,14.,15.]]]])

tensor([[[[10.]]]])

tensor([[[[5.,7.],
[13.,15.]]]])

tensor([[[[1.,3.],
[9.,11.],
[13.,15.]]]])

c)池化的多通道

```
X = torch.cat((X, X + 1), dim=1)
print(X)

pool2d = nn.MaxPool2d(3, padding=1, stride=2)
print(pool2d(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.]]]])

tensor([[[[5.,7.],
[13.,15.]],

[[6.,8.],
[14.,16.]]]])

5）Lenet实现

导入包：

```
import os
import time
import torch
from torch import nn, optim

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')
```

a)模型搭建

```
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
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)
)
)
```

b)数据获取与模型训练

```
def train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs): #训练模型
    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, time.time() - start))

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)
```

```
    batch_size = 256
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size) #获取数据

    def evaluate_accuracy(data_iter, net, device=None): #评价指标
        if device is None and isinstance(net, torch.nn.Module):
            device = list(net.parameters())[0].device
        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().sum().cpu().item()
                    net.train() #  改回训练模式
                else:
                    if('is_training' in net.__code__.co_varnames): #  如果有 is_training 这个参数
                        acc_sum += (net(X, is_training=False).argmax(dim=1) ==
y).float().sum().item()
                    else:
                        acc_sum += (net(X).argmax(dim=1) == y).float().sum().item()
                n += y.shape[0]
        return acc_sum / n
```

输出：

```
Training on cpu
epoch1,loss1.8024,trainacc0.333,testacc0.581,time29.3sec
epoch2,loss0.4725,trainacc0.634,testacc0.687,time30.3sec
epoch3,loss0.2605,trainacc0.711,testacc0.718,time30.4sec
epoch4,loss0.1743,trainacc0.739,testacc0.736,time30.3sec
epoch5,loss0.1290,trainacc0.754,testacc0.758,time32.4sec
```

6）Alexnet实现

导入使用的包：

```
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')
```

a)模型搭建

```
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, padding
            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。除了最后卷积层外，进一步增大输出通道数。
             #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(),
```

```
            # 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

net = AlexNet()
```

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)))
```

b)数据获取

```
    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)
        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,
transform=transform)
        mnist_test = torchvision.datasets.FashionMNIST(root=root, train=False, download=True,
transform=transform)

        train_iter = torch.utils.data.DataLoader(mnist_train, batch_size=batch_size, shuffle=True,
num_workers=4)
        test_iter = torch.utils.data.DataLoader(mnist_test, batch_size=batch_size, shuffle=False,
num_workers=4)
        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)
```

c)模型训练

```
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
epoch1,loss0.0047,trainacc0.770,testacc0.865,time128.3sec
epoch2,loss0.0025,trainacc0.879,testacc0.889,time128.8sec
epoch3,loss0.0022,trainacc0.898,testacc0.901,time130.4sec
epoch4,loss0.0019,trainacc0.908,testacc0.900,time131.4sec
epoch5,loss0.0018,trainacc0.913,testacc0.902,time129.9sec
```

5.6.3实验三：rnn

1）基础rnn实现

a)导入要用的包

```
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()
```

b)搭建模型

设置模型参数

```
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)

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)
```

搭建模型：

```
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, 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
```

c)模型训练和预测

```
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)    # 前向计算不需要传入模型参数
        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])
model = RNNModel(rnn_layer, vocab_size).to(device)
predict_rnn_pytorch('分开', 10, model, vocab_size, device, idx_to_char, char_to_idx)
```

```
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, device)
        for X, Y in data_iter:
            if state is not None:
                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))
```

```
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)
```

输出结果：

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 我不多再想 我不要
 - 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 坏坏的让我疯狂的
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 你的完美主义 太彻底 让我
 - 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 靠着我的肩膀 你 在我
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 习武之人切记 仁者无敌
 - 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过外滩 消失的 旧
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 我已无能为力再提起
 - 不分开 让我面到你 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 我不 我不 我不
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 让我叫带你 你你的
 - 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 我跟了这节奏 后知后觉

2）lstm实现

a)读取数据集

```
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()
```

b)初始化参数

```
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)
    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), requires_grad=True))

    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),
requires_grad=True)
    return nn.ParameterList([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])
```

c)搭建模型

```
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))

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] = params
    (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)
```

d)训练模型并创作歌词

```
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']

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)
```

输出结果：

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 我不要这我 我不

- 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 我不要这我 我不要这我
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 说给怎么么 有你在空 你在在空 在你的空
- 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 我该了这生活 我
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 你想我难难头 一话看人对落我一望望我 我不那这
- 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 后知后觉 我该

课后习题：

1. 使用不同的目标函数，观察梯度下降和随机梯度下降中自变量的迭代轨迹。
2. 在二维梯度下降中尝试不同的学习率，观察并分析实验结果。
3. 修改批量大小和学习率，观察目标函数值的下降速度和每个迭代周期的耗时。
4. 使用其他动量超参数和学习率的组合，观察并分析实验结果。
5. 实验中尝试不同的初始学习率，结果有什么变化？
6. 有人说 Adam 是 RMSProp 和动量法的结合，为什么？
7. 比较使用丢弃法和权重衰减的效果，如果同时使用二者，实验结果如何？
8. 是否可以将线性回归或 softmax 回归中所有权重参数都初始化为相同值？
9. 如何构造一个全连接层来检测物体边缘？
10. 尝试基于 LeNet 构造更复杂的网络来提高分类准确率。如调整卷积窗口大小、通道数，使用不同的学习率、激活函数、参数初始化方法和梯度下降优化方法，增加迭代周期，
11. 最大池化和平均池化在作用上有哪些区别？
12. AlexNet 对 Fashion-Mnist 数据集可能过于复杂，有什么简化网络的方法？要保证准确率不降低，但是能缩短训练时间。
13. 你还能想到哪些采样小批量时序数据的方法？
14. 在相同条件下，比较门控循环单元和不带门控的循环神经网络的运行时间。
15. 除了堆叠循环神经网络，还有什么结构可以增加循环神经网络的深度？
16. 分析卷积神经网络中用 1×1 的卷积核的作用。

参考文献

Bishop C M, 2007. Pattern recognition and machine learning[M]. 5th edition. Springer. Clevert D A, Unterthiner T, Hochreiter S, 2015. Fast and accurate deep network learning by exponential linear units (elus)[J]. arXiv preprint arXiv:1511.07289.

Graves A, Wayne G, Danihelka I, 2014. Neural turing machines[J]. arXiv preprint arXiv:1410.5401.

Sukhbaatar S, Weston J, Fergus R, et al., 2015. End-to-end memory networks[C]//Advances in Neural Information Processing Systems. 2431-2439.

Kipf T N, Welling M, 2016. Semi-supervised classification with graph convolutional networks[J]. arXiv preprint arXiv:1609.02907.

Veličković P, Cucurull G, Casanova A, et al., 2017. Graph attention networks[J]. arXiv preprint arXiv:1710.10903.

Gilmer J, Schoenholz S S, Riley P F, et al., 2017. Neural message passing for quantum chemistry [J]. arXiv preprint arXiv:1704.01212.

Nair V, Hinton G E, 2010. Rectified linear units improve restricted boltzmann machines[C]// Proceedings of the International Conference on Machine Learning. 807-814.

Glorot X, Bengio Y, 2010. Understanding the difficulty of training deep feedforward neural networks[C]//Proceedings of International conference on artificial intelligence and statistics. 249- 256.

Maas A L, Hannun A Y, Ng A Y, 2013. Rectifier nonlinearities improve neural network acoustic models[C]//Proceedings of the International Conference on Machine Learning.

He K, Zhang X, Ren S, et al., 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification[C]//Proceedings of the IEEE International Conference on Computer Vision. 1026-1034.

Clevert D A, Unterthiner T, Hochreiter S, 2015. Fast and accurate deep network learning by exponential linear units (elus)[J]. arXiv preprint arXiv:1511.07289.

Dugas C, Bengio Y, Bélisle F, et al., 2001. Incorporating second-order functional knowledge for better option pricing[J]. Advances in Neural Information Processing Systems:472-478.

Goyal P, Dollár P, Girshick R, et al., 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour[J]. arXiv preprint arXiv:1706.02677.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research. 261

Hinton, G. E. (2012). Tutorial on deep learning. IPAM Graduate Summer School: Deep Learning, Feature Learning. 262

Kingma D, Ba J, 2015. Adam: A method for stochastic optimization[C]//Proceedings of International Conference on Learning Representations.

Jaitly, N. and Hinton, G. E. (2013). Vocal tract length perturbation (VTLP) improves speech recognition. In ICML'2013 . 207

Sietsma, J. and Dow, R. (1991). Creating artificial neural networks that generalize. Neural Networks, 4(1), 67–79. 207

Tang, Y. and Eliasmith, C. (2010). Deep networks for robust visual recognition. In Proceedings of the 27th International Conference on Machine Learning, June 21-24, 2010, Haifa, Israel. 207

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008a). Extracting and composing robust features with denoising autoencoders. In ICM (1a), pages 1096–1103. 207

Poole, B., Sohl-Dickstein, J., and Ganguli, S. (2014). Analyzing noise in autoencoders and deep networks. CoRR, abs/1406.1831. 207

Breiman, L. (1994). Bagging predictors. Machine Learning, 24(2), 123–140. 220

Freund, Y. and Schapire, R. E. (1996a). Experiments with a new boosting algorithm. In Machine Learning: Proceedings of Thirteenth International Conference, pages 148–156, USA. ACM. 222

Freund, Y. and Schapire, R. E. (1996b). Game theory, on-line prediction and boosting. In Proceedings of the Ninth Annual Conference on Computational Learning Theory, pages 325– 332. 222

Schwenk, H. and Bengio, Y. (1998). Training methods for adaptive boosting of neural networks. In M. Jordan, M. Kearns, and S.

Solla, editors, Advances in Neural Information Processing Systems 10 (NIPS'97), pages 647–653. MIT Press. 222

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15, 1929–1958. 222, 227, 228, 229, 574

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014a). Going deeper with convolutions. Technical report, arXiv:1409.4842. 20, 21, 174, 222, 231, 278, 295

Bayer, J. and Osendorfer, C. (2014). Learning stochastic recurrent networks. ArXiv e-prints. 228

Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2014a). How to construct deep recurrent neural networks. In ICLR. 17, 228, 340, 341, 349, 392

Zhou, Y. and Chellappa, R. (1988). Computation of optical flow using a neural network. In Neural Networks, 1988., IEEE International Conference on, pages 71–78. IEEE. 290

Jain, V., Murray, J. F., Roth, F., Turaga, S., Zhigulin, V., Briggman, K. L., Helmstaedter, M. N., Denk, W., and Seung, H. S. (2007). Supervised learning of image restoration with convolutional networks. In Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on, pages 1–8. IEEE. 306

Pinheiro, P. H. O. and Collobert, R. (2014). Recurrent convolutional neural networks for scene labeling. In ICML'2014 . 306

Pinheiro, P. H. O. and Collobert, R. (2015). From image-level to pixel-level labeling with convolutional networks. In Conference on Computer Vision and Pattern Recognition (CVPR). 306

Briggman, K., Denk, W., Seung, S., Helmstaedter, M. N., and Turaga, S. C. (2009). Maximin affinity learning of image segmentation. In NIPS'2009 , pages 1865–1873. 306

Turaga, S. C., Murray, J. F., Jain, V., Roth, F., Helmstaedter, M., Briggman, K., Denk, W., and Seung, H. S. (2010). Convolutional networks can learn to generate affinity graphs for image segmentation. Neural Computation, 22, 511–538. 306

Farabet, C., Couprie, C., Najman, L., and LeCun, Y. (2013). Learning hierarchical features for scene labeling. IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(8), 1915–1929. 22, 174, 306

Neal, R. M. (2005). Estimating ratios of normalizing constants using linked importance sampling. 536, 537

Thompson, J., Jain, A., LeCun, Y., and Bregler, C. (2014). Joint training of a convolutional network and a graphical model for human pose estimation. In NIPS'2014 . 306

LeCun Y, Bottou L, Bengio Y, et al., 1998. Gradient-based learning applied to document recognition [J]. Proceedings of the IEEE, 86(11):2278-2324.

Krizhevsky A, Sutskever I, Hinton G E, 2012. ImageNet classification with deep convolutional neural networks[C]//Advances in Neural Information Processing Systems 25. 1106-1114.

Srivastava R K, Greff K, Schmidhuber J, 2015. Highway networks[J]. arXiv preprint arXiv:1505.00387.

Werbos P J, 1990. Backpropagation through time: what it does and how to do it[J]. Proceedings of the IEEE, 78(10):1550-1560

Hochreiter S, Schmidhuber J, 1997. Flat minima[J]. Neural Computation, 9(1):1-42.

Younger A S, Hochreiter S, Conwell P R, 2001. Meta-learning with backpropagation[C]// Proceedings of International Joint Conference on Neural Networks: volume 3. IEEE.

Parlos A, Atiya A, Chong K, et al., 1991. Recurrent multilayer perceptron for nonlinear system identification[C]//International Joint Conference on Neural Networks: volume 2. IEEE: 537-540. Pollack J B, 1990. Recursive distributed representations[J]. Artificial Intelligence, 46(1):77-105.

Gers F A, Schmidhuber J, Cummins F, 2000. Learning to forget: Continual prediction with lstm [J]. Neural Computation.

Hochreiter S, Schmidhuber J, 1997. Long short-term memory[J]. Neural computation, 9(8):1735- 1780

Aston Zhang, Mu Li, Zachary C. Lipton ,Alexander J. Smola(2020).动手学深度学习.中国：人民邮电出版社,109-110,253-257, 149-177 ,226-228, 241-247

Ian Goodfellow, Yoshua Bengio, Aaron Courville(2017).深度学习(中文版).中国：人民邮电出版社,198-203 ,207 ,210-213, 220-225, 256-262 ,282-284, 290-291, 320-323

邱锡鹏(2020).神经网络与深度学习.中国：机械工业出版社，83-85, 87-89, 95, 118-120, 122, 143-146, 158-160, 161-164, 168-

169, 171-172