

深度强化学习（初稿）

Deep Reinforcement Learning

王树森 张志华 著

《深度强化学习》2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

《深度强化学习》 2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

前言

在 2015 年发生了两件大事：DQN 在 Atari 电子游戏上超越了人类水平，AlphaGo 在围棋游戏中击败了职业棋手樊麾。自此开始，深度强化学习受到空前的关注并成为 AI 领域的研究热点，新的方法如雨后春笋般出现，在各种任务上不断刷新纪录。深度强化学习是值得深入研究的领域，但是入门却非常困难。一方面，强化学习的数学原理复杂，远甚于深度学习；另一方面，深度强化学习发展迅猛，而又脉络复杂，外行很难理清头绪，找到其中的基础和前沿。本书作者在刚涉足该领域的時候，亦被此等问题困扰，学习进度缓慢。写作本书的目的就是解释清楚深度强化学习的原理，帮助读者深入理解其中的数学原理，建立完整的知识体系，培养科研和创新能力。

本书作者通过阅读大量文献，并对文献做梳理，将教学内容与学生反馈相结合，写成本书。本书面向的对象是有一定机器学习基础的学生，特别是有志从事科研工作的研究生。阅读本书，相当于阅读多篇经典论文，并掌握其中的核心思想和数学原理。本书作者没有照搬论文内容，而是提取论文的主要思想，再按照本书整体思路和结构重新做推导、表述。与原始论文相比，本书在细节方面予以简化、甚至纠正。读者如果发现本书内容在细节上与原始论文有出入，不必感到疑惑，也不必质疑本书的正确性。

现在市面上已有多本强化学习的教材，那么本书与其他教材的区别在哪里呢？传统的强化学习书籍知识体系完整，但其中多数内容在今天已经不太重要，而当今最重要的技术却没有被囊括。较新的深度强化学习教材几乎都偏重编程实践，而对方法和原理的解释比较欠缺，对数学推导采用完全回避的态度；这是情有可原的，因为想把代码讲解清楚容易，而想把方法和原理讲解清楚却很困难。本书的独特之处在于有系统地讲解深度强化学习，不回避数学原理，而是用通俗的语言解释数学原理。为了将方法和原理解释清楚，作者精心制作了超过一百张插图，让模型和数学变得直观。本书尽量剔除一切不必要的概念，只保留最有用的内容，争取做到每一个章节都值得阅读。

为了降低阅读的难度，本书尽量避免一切不必要的数学公式，可是书中仍然有大量的公式。强化学习方法几乎都来自于严格的数学推导，每种方法的本质往往在于一两个数学公式。不完全理解数学公式，不可能彻底深入理解强化学习方法。如果你理解每个公式是怎么来的，那么算法的流程就一目了然。本书不会绕开必要数学公式，但会尽量解释清楚，绝对不会“空降公式”。

本书假设读者完全不懂强化学习，但是要求读者了解机器学习的基础知识，比如优化、目标函数、正则、梯度等基本概念。读者可以不熟悉深度学习的技术细节，但是应当知晓深度学习的“常识”，知道神经网络的全连接层、卷积层、Sigmoid 激活函数、Softmax 激活函数的用途。如果读者几乎不懂深度学习，也可以阅读本书，但是会在一定程度上影响阅读和理解。

市面上讲解深度强化学习代码的书籍已经很多，本书就不花大量篇幅讲解编程实现，而是给出伪代码。再者，有的读者熟悉 TensorFlow，而有的读者偏好 PyTorch，一本书没有办法同时照顾两个群体。用 TensorFlow 和 PyTorch 讲解深度强化学习的书籍在市面上

都能找到，不论读者喜欢哪种，都能找到相应书籍，对本书起补充作用。读者并没有必要阅读讲解源代码的书籍，因为源代码及其讲解都很容易在互联网上搜索到。比如，要是读者想要搜索 DDPG（深度确定策略梯度方法）的 TensorFlow 实现，只需要在互联网上搜索“DDPG+TensorFlow”，就能找到源代码及其讲解。有了本书的基础知识，读者可以轻松看懂源代码。

王树森

2021 年 3 月 9 日

《深度强化学习》2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

常用符号

符号	中文	英文
S 或 s	状态	state
A 或 a	动作	action
R 或 r	奖励	reward
U 或 u	回报	return
γ	折扣率	discount factor
\mathcal{S}	状态空间	state space
\mathcal{A}	动作空间	action space
$\pi(a s)$	随机策略函数	stochastic policy function
$\mu(s)$	确定策略函数	deterministic policy function
$p(s' s, a)$	状态转移函数	state-transition function
$Q_\pi(s, a)$	动作价值函数	action-value function
$Q_*(s, a)$	最优动作价值函数	optimal action-value function
$V_\pi(s)$	状态价值函数	state-value function
$V_*(s)$	最优状态价值函数	optimal state-value function
$D_\pi(s)$	优势函数	advantage function
$D_*(s)$	最优优势函数	optimal advantage function
$\pi(a s; \theta)$	随机策略网络	stochastic policy network
$\mu(s; \theta)$	确定策略网络	deterministic policy network
$Q(s, a; \mathbf{w})$	深度 Q 网络	deep Q network (DQN)
$q(s, a; \mathbf{w})$	价值网络	value network

《深度强化学习》2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

目录

1 概率论基础与蒙特卡洛	1
1.1 概率论基础	1
1.2 蒙特卡洛	4
2 深度学习基础	13
2.1 线性模型	13
2.2 非线性核方法	19
2.3 神经网络	19
2.4 反向传播和梯度下降	22
3 马尔可夫决策过程 (MDP)	25
3.1 基本概念	25
3.2 随机性的来源	29
3.3 回报与折扣回报	31
3.4 价值函数	33
3.5 策略学习和价值学习	35
3.6 实验环境	36
4 DQN 与 Q 学习	39
4.1 DQN	39
4.2 时间差分 (TD) 算法	41
4.3 用 TD 训练 DQN	44
4.4 Q 学习算法	47
4.5 同策略 (On-policy) 与异策略 (Off-policy)	49
5 SARSA 算法	51
5.1 表格形式的 SARSA	51
5.2 神经网络形式的 SARSA	54
5.3 多步 TD 目标	56
5.4 蒙特卡洛与自举	58
6 价值学习高级技巧	63
6.1 经验回放	63
6.2 高估问题及解决方法	67
6.3 对决网络 (Dueling Network)	73
6.4 噪声网络	77

7 策略梯度方法	81
7.1 策略网络	81
7.2 策略学习的目标函数	83
7.3 策略梯度定理的证明	85
7.4 REINFORCE	91
7.5 Actor-Critic	94
8 带基线的策略梯度方法	101
8.1 策略梯度中的基线	101
8.2 带基线的 REINFORCE 算法	104
8.3 Advantage Actor-Critic (A2C)	107
8.4 证明带基线的策略梯度定理	111
9 策略学习高级技巧	113
9.1 Trust Region Policy Optimization (TRPO)	113
9.2 熵正则 (Entropy Regularization)	118
10 连续控制	123
10.1 离散控制与连续控制的区别	123
10.2 确定策略梯度 (DPG)	124
10.3 双延时确定策略梯度 (TD3)	129
10.4 随机高斯策略	133
11 对状态的不完全观测	139
11.1 不完全观测问题	139
11.2 循环神经网络 (RNN)	141
11.3 RNN 作为策略网络	143
12 并行计算	145
12.1 并行计算基础	145
12.2 同步与异步	151
12.3 并行强化学习	154
13 多智能体系统	159
13.1 多智能体系统的设定	159
13.2 多智能体系统的概念	161
13.3 实验环境	164
14 合作关系设定下的多智能体强化学习	169
14.1 合作关系设定下的策略学习	170
14.2 合作设定下的多智能体 A2C	171

《深度强化学习》2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

14.3 三种架构	175
15 非合作关系设定下的多智能体强化学习	183
15.1 非合作关系设定下的策略学习	184
15.2 非合作设定下的多智能体 A2C	187
15.3 三种架构	190
15.4 连续控制与 MADDPG	194
16 注意力机制与多智能体强化学习	201
16.1 自注意力机制	201
16.2 自注意力在中心化训练中的应用	205
17 模仿学习	211
17.1 行为克隆	211
17.2 逆向强化学习	215
17.3 生成判别模仿学习 (GAIL)	218
18 AlphaGo 与蒙特卡洛树搜索	225
18.1 动作、状态、策略网络、价值网络	225
18.2 蒙特卡洛树搜索 (MCTS)	227
18.3 训练策略网络和价值网络	232
A 贝尔曼方程	237

《深度强化学习》 2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

第一章 概率论基础与蒙特卡洛

本章首先回顾一些概率论基础知识，然后介绍蒙特卡洛。蒙特卡洛是一类随机算法的总称，它是很多强化学习算法的关键要素。

1.1 概率论基础

强化学习中会经常用到两个概念：**随机变量、观测值**。随机变量是一个不确定量，它的值取决于一个随机事件的结果。比如抛一枚硬币，正面朝上记为 0，反面朝上记为 1。抛硬币是个随机事件，抛硬币的结果记为随机变量 X ，用大写字母表示。随机变量 X 有两种可能的取值：可能是 0，也可能是 1。抛硬币之前， X 是未知的，而且带有随机性。抛硬币之后，我们会观测到硬币哪一面朝上，此时随机变量 X 就有了观测值，记作 x 。举个例子，如果重复抛硬币 4 次，得到了 4 个观测值：

$$x_1 = 1, \quad x_2 = 1, \quad x_3 = 0, \quad x_4 = 1.$$

这四个观测值只是数字而已，没有随机性。本书用大写字母表示随机变量，小写字母表示观测值，避免造成混淆。

强化学习会反复用到**概率质量函数** (Probability Mass Function, PMF) 或**概率密度函数** (Probability Density Function, PDF)，意思是随机变量 X 在确定的取值点 x 的可能性。

- 概率质量函数 (PMF) 描述一个**离散概率分布**——即变量的取值范围 \mathcal{X} 是个离散的集合。在抛硬币的例子中，随机变量 X 的取值范围是集合 $\mathcal{X} = \{0, 1\}$ 。 X 的概率质量函数是

$$p(0) = 0.5, \quad p(1) = 0.5.$$

公式的意思随机变量取值 0 和 1 的概率都是 0.5。见图 1.1(左) 的例子。概率质量函数有这样的性质：

$$\sum_{x \in \mathcal{X}} p(x) = 1.$$

- 当考虑连续随机变量时，我们用概率密度函数 (PDF)。正态分布是最常见的一种**连续概率分布**。随机变量 X 的取值范围是所有实数 \mathbb{R} 。正态分布的概率密度函数是

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \cdot \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

此处的 μ 是均值， σ 是标准差。图 1.1(右) 的例子说明 X 在均值附近取值的可能性大，在远离均值的地方取值的可能性小。设 \mathcal{X} 为变量 X 的取值范围。概率密度函数有这样的性质：

$$\int_{\mathcal{X}} p(x) dx = 1.$$

函数 $f(X)$ 的**期望**是这样定义的。设 $p(X)$ 为 X 的概率密度函数 (或概率质量函数)。

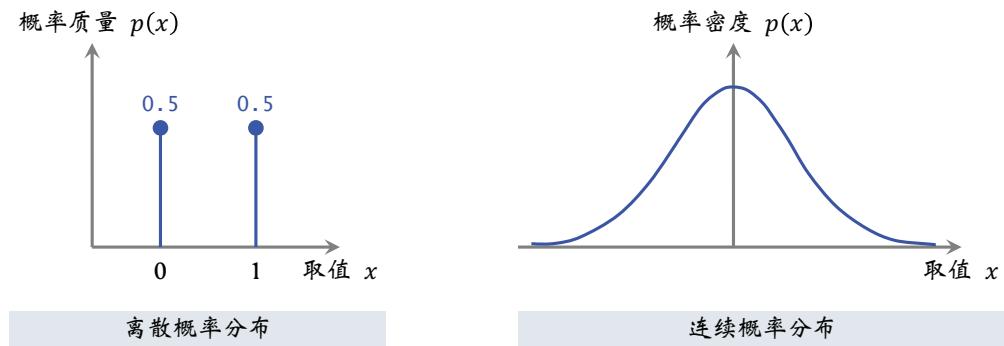


图 1.1: 左图是抛硬币的例子。右图是均值为零的正态分布。

对于离散概率分布, $f(X)$ 的期望是

$$\mathbb{E}_{X \sim p(\cdot)}[f(X)] = \sum_{x \in \mathcal{X}} p(x) \cdot f(x).$$

对于连续概率分布, $f(X)$ 的期望是

$$\mathbb{E}_{X \sim p(\cdot)}[f(X)] = \int_{\mathcal{X}} p(x) \cdot f(x) dx.$$

设 $g(X, Y)$ 为二元函数。如果对 $g(X, Y)$ 关于随机变量 X 求期望, 那么会消掉 X , 得到的结果是 Y 的函数。举个例子, 设随机变量 X 的取值范围是 $\mathcal{X} = [0, 10]$, 概率密度函数是 $p(x) = \frac{1}{10}$ 。设 $g(X, Y) = 2XY$, 那么 $g(X, Y)$ 关于 X 的期望等于

$$\begin{aligned} \mathbb{E}_{X \sim p(\cdot)}[g(X, Y)] &= \int_{\mathcal{X}} g(x, Y) \cdot p(x) dx \\ &= \int_0^{10} 2xY \cdot \frac{1}{10} dx \\ &= 10Y. \end{aligned}$$

这个例子说明期望如何消掉函数 $g(X, Y)$ 中的变量 X 。

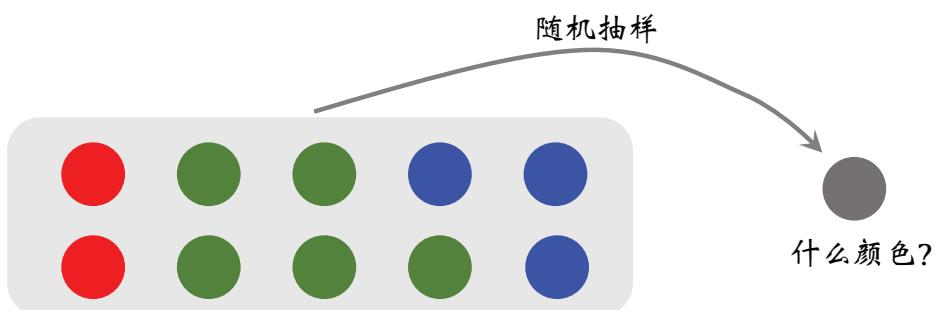


图 1.2: 箱子里有 10 个球。2 个是红色, 5 个是绿色, 3 个是蓝色。

强化学习中常用到**随机抽样**, 此处给一个直观的解释。如图 1.2 所示, 箱子里有 10 个球, 其中 2 个是红色, 5 个是绿色, 3 个是蓝色。我现在把箱子摇一摇, 把手伸进箱子里, 闭着眼睛摸出来一个球。当我睁开眼睛, 就观测到球的颜色, 比如红色。这个过程叫做随机抽样, 本轮随机抽样的结果是红色。如果把抽到的球放回, 可以无限次重复随机抽样, 得到多个观测值。

请读者注意随机变量与观测值的区别。在我摸出一个球之前, 随机抽样的颜色是随

1.1 概率论基础

机变量，记作 X ，它有三种可能的取值——红色、绿色、蓝色。当我摸到球之后，我观测到了颜色“ $x = \text{红}$ ”，这是 X 的一个观测值。注意，观测值“ $x = \text{红}$ ”没有随机性，而变量 X 有随机性。

可以用计算机程序做随机抽样。假设箱子里有很多个球，红色球占 20%，绿色球占 50%，蓝色球占 30%。如果我随机摸一个球，那么抽到的球服从这样一个离散概率分布：

$$p(\text{红}) = 0.2, \quad p(\text{绿}) = 0.5, \quad p(\text{蓝}) = 0.3.$$

下面的 Python 代码按照概率质量 p 做随机抽样，重复 100 次，输出抽样的结果。

```
from numpy.random import choice
samples = choice(['R', 'G', 'B'],
                 size=100,
                 p=[0.2, 0.5, 0.3])
print(samples)
```

随机变量的取值范围是集合 {R, G, B}。
重复抽样 100 次，函数返回长度为 100 的数组。
R, G, B 三种颜色的球被选中的概率分别是 0.2, 0.5, 0.3。

```
['B', 'R', 'R', 'G', 'B', 'B', 'G', 'G', 'G', 'R', 'R', 'G', 'G', 'B', 'R', 'G', 'G', 'B',
 'B', 'G', 'B', 'G', 'G', 'R', 'G', 'B', 'R', 'G', 'G', 'R', 'G', 'R', 'G', 'G', 'G', 'G',
 'G', 'B', 'G', 'B', 'R', 'R', 'G', 'G', 'B', 'B', 'G', 'R', 'R', 'B', 'G', 'G', 'G', 'B', 'B',
 'G', 'G', 'G', 'B', 'B', 'G', 'B', 'G', 'G', 'B', 'G', 'R', 'B', 'G', 'R', 'B', 'B', 'G',
 'G', 'G', 'R', 'G', 'R', 'G', 'G', 'G', 'G', 'G', 'B', 'G', 'B', 'G', 'G', 'R', 'G', 'R', 'B']
```

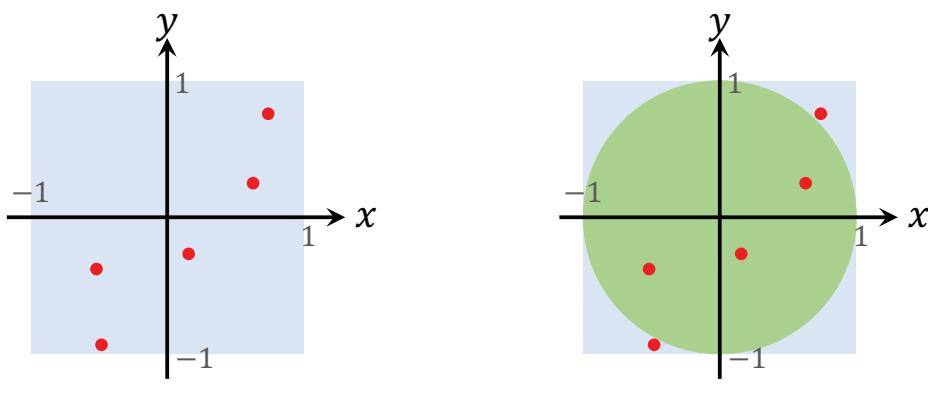
1.2 蒙特卡洛

蒙特卡洛 (Monte Carlo) 是一大类随机算法 (Randomized Algorithms) 的总称，它们通过随机样本来估算真实值。本节用几个例子讲解蒙特卡洛算法。

1.2.1 例一：近似 π 值

我们都知道 π 约等于 3.1415927。现在假装我们不知道 π ，而是要想办法近似估算 π 值。假设我们有（伪）随机数生成器，我们能不能用随机样本来近似 π 呢？这一小节使用蒙特卡洛近似 π 值。

假设我们有一个（伪）随机数生成器，可以均匀生成 -1 到 $+1$ 之间的数。每次生成两个随机数，一个作为 x ，另一个作为 y 。于是每次就生成了一个平面坐标系中的点 (x, y) ；见图 1.3(左)。因为 x 和 y 都是在 $[-1, 1]$ 区间上均匀分布，所以 $[-1, 1] \times [-1, 1]$ 这个正方形内的点被抽到的概率是相同的。我们重复抽样 n 次，得到了 n 个正方形内的点。



从蓝色正方形中做随机抽样，
得到 n 个红色的点。

抽到的红色的点可能落在绿色
的圆内部，也可能落在外部。

图 1.3: 通过抽样来近似 π 值。

如图 1.3(右) 所示，蓝色正方形里面包含一个绿色的圆，圆心是 $(0, 0)$ ，半径等于 1。刚才随机生成的 n 个点有些落在圆外面，有些落在圆里面。请问一个点落在圆里面的概率有多大呢？由于抽样是均匀的，因此这个概率显然是圆的面积与正方形面积之比。正方形的面积是边长的平方，即 $a_1 = 2^2 = 4$ 。圆的面积是 π 乘以半径的平方，即 $a_2 = \pi \cdot 1^2 = \pi$ 。那么一个点落在圆里面的概率就是

$$p = \frac{a_2}{a_1} = \frac{\pi}{4}.$$

设我们随机抽样了 n 个点，设圆内的点的数量为随机变量 M 。很显然， M 的期望等于

$$\mathbb{E}[M] = pn = \frac{\pi n}{4}.$$

注意，这只是期望，并不是实际发生的结果。如果你抽 $n = 5$ 个点，那么期望有 $\mathbb{E}[M] = \frac{5\pi}{4}$ 个点落在圆内。但实际观测值 m 可能等于 0、1、2、3、4、5 中的任何一个。

给定一个点的坐标 (x, y) , 该如何判断该点是否在圆内呢? 已知圆心在原点, 半径等于 1, 我们用一下圆的方程。如果 (x, y) 满足:

$$x^2 + y^2 \leq 1,$$

则说明 (x, y) 落在圆里面; 反之, 点就在圆外面。

我们均匀随机抽样得到 n 个点, 通过圆的方程对每个点做判别, 发现有 m 个点落在圆里面。假如 n 非常大, 那么随机变量 M 的真实观测值 m 就会非常接近期望 $\mathbb{E}[M] = \frac{\pi n}{4}$:

$$m \approx \frac{\pi n}{4}.$$

对公式做个变换, 得到:

$$\pi \approx \frac{4m}{n}.$$

我们可以依据这个公式做编程实现。下面是伪代码:

1. 初始化 $m = 0$ 。用户指定样本数量 n 的大小。 n 越大, 精度越高, 但是计算量越大。
2. 把下面的步骤重复 n 次:
 - (a). 从区间 $[-1, 1]$ 上均匀随机抽样得到 x ; 再做一次均匀随机抽样, 得到 y 。
 - (b). 如果 $x^2 + y^2 \leq 1$, 那么 $m \leftarrow m + 1$ 。
3. 返回 $\frac{4m}{n}$ 作为对 π 的估计。

大数定律保证了蒙特卡洛的正确性: 当 n 趋于无穷, $\frac{4m}{n}$ 趋于 π 。其实还能进一步用概率不等式分析误差的上界。使用 Bernstein 不等式, 可以证明出这个结论:

$$\left| \frac{4m}{n} - \pi \right| = O\left(\frac{1}{\sqrt{n}}\right).$$

这个不等式说明 $\frac{4m}{n}$ (即对 π 的估计) 会收敛到 π , 收敛率是 $\frac{1}{\sqrt{n}}$ 。然而这个收敛率并不快: 样本数量 n 增加一万倍, 精度才能提高一百倍。

1.2.2 例二: 估算阴影部分面积

图 1.4 中有正方形、圆、扇形, 几个形状相交。请估算阴影部分面积。这个问题常见于初中数学竞赛。假如你不会微积分, 也不会几何的奇技淫巧, 你是否有办法近似估算阴影部分面积呢? 用蒙特卡洛可以很容易解决这个问题。

图 1.5 中绿色圆的圆心是 $(1, 1)$, 半径等于 1; 蓝色扇形的圆心是 $(0, 0)$, 半径等于 2。阴影区域内的点 (x, y) 在绿色的圆中, 而不在蓝色的扇形中。

- 利用圆的方程可以判定点 (x, y) 是否在绿色圆里面。如果 (x, y) 满足方程

$$(x - 1)^2 + (y - 1)^2 \leq 1, \quad (1.1)$$

则说明 (x, y) 在绿色圆里面。

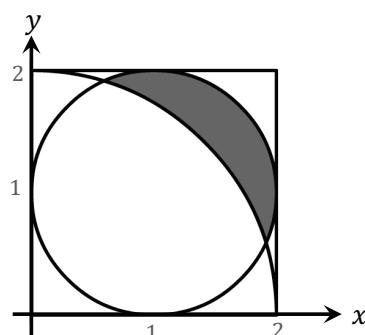


图 1.4: 估算阴影部分面积。

- 利用扇形的方程可以判定点 (x, y) 是否在蓝色扇形外面。如果点 (x, y) 满足方程

$$x^2 + y^2 > 2^2, \quad (1.2)$$

则说明 (x, y) 在蓝色扇形外面。

如果一个点同时满足方程 (1.1) 和 (1.2)，那么这个点一定在阴影区域内。从 $[0, 2] \times [0, 2]$ 这个正方形中做随机抽样，得到 n 个点。然后用两个方程筛选落在阴影部分的点。

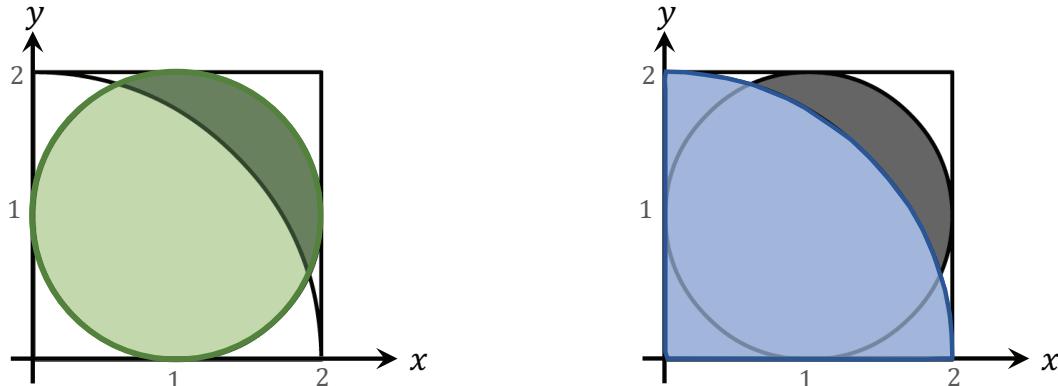


图 1.5：如果一个点在阴影部分，那么它在左边绿的的圆中，而在右边蓝色的扇形中。

我们在正方形 $[0, 2] \times [0, 2]$ 中随机均匀抽样，得到的点有一定概率会落在阴影部分。我们来计算这个概率。正方形的边长等于 2，所以面积 $a_1 = 4$ 。设阴影部分面积为 a_2 。那么点落在阴影部分概率是

$$p = \frac{a_2}{a_1} = \frac{a_2}{4}.$$

我们从正方形中随机抽 n 个点，设有 M 个点落在阴影部分内 (M 是个随机变量)。每个点落在阴影部分的概率是 p ，所以 M 的期望等于

$$\mathbb{E}[M] = np = \frac{na_2}{4}.$$

用方程 (1.1) 和 (1.2) 对 n 个点做筛选，发现实际上有 m 个点落在阴影部分内 (m 是随机变量 M 的观测值)。如果 n 很大，那么 m 会比较接近期望 $\mathbb{E}[M] = \frac{na_2}{4}$ ，即

$$m \approx \frac{na_2}{4}.$$

把等式变换一下，得到：

$$a_2 \approx \frac{4m}{n}.$$

这个公式就是对阴影部分面积的估计。我们可以依据这个公式做编程实现。下面是伪代码：

- 初始化 $m = 0$ 。用户指定样本数量 n 的大小。 n 越大，精度越高，但是计算量越大。
- 把下面的步骤重复 n 次：
 - 从区间 $[0, 2]$ 上均匀随机抽样得到 x ；再做一次均匀随机抽样，得到 y 。
 - 如果 $(x - 1)^2 + (y - 1)^2 \leq 1$ 和 $x^2 + y^2 > 4$ 两个不等式都成立，那么让 $m \leftarrow m + 1$ 。

3. 返回 $\frac{4m}{n}$ 作为对阴影部分面积的估计。

1.2.3 例三：近似定积分

近似求积分是蒙特卡洛最重要的应用之一，在科学和工程中有广泛的应用。举个例子，给定一个函数：

$$f(x) = \frac{1}{1 + (\sin x) \cdot (\ln x)^2},$$

要求计算 f 在区间 0.8 到 3 上的定积分：

$$I = \int_{0.8}^3 f(x) dx.$$

有很多科学和工程问题需要计算定积分，而函数 $f(x)$ 可能很复杂，求定积分会很困难，甚至有可能不存在解析解。如果求解析解很困难，或者解析解不存在，则可以用蒙特卡洛近似计算数值解。

一元函数的定积分是相对比较简单的问题。一元函数的意思是变量 x 是个标量。给定一元函数 $f(x)$ ，求函数在 a 到 b 区间上的定积分：

$$I = \int_a^b f(x) dx.$$

蒙特卡洛方法通过下面的步骤近似定积分：

1. 在区间 $[a, b]$ 上做随机抽样，得到 n 个样本，记作： x_1, \dots, x_n 。样本数量 n 由用户自己定， n 越大，计算量越大，近似越准确。
2. 对函数值 $f(x_1), \dots, f(x_n)$ 求平均，再乘以区间长度 $b - a$ ：

$$q_n = (b - a) \cdot \frac{1}{n} \sum_{i=1}^n f(x_i).$$

3. 返回 q_n 作为定积分 I 的估计值。

多元函数的定积分要复杂一些。设 $f : \mathbb{R}^d \rightarrow \mathbb{R}$ 是一个多元函数，变量 \mathbf{x} 是 d 维向量。要求计算 f 在集合 Ω 上的定积分：

$$I = \int_{\Omega} f(\mathbf{x}) d\mathbf{x}.$$

蒙特卡洛方法通过下面的步骤近似定积分：

1. 在集合 Ω 上做均匀随机抽样，得到 n 个样本，记作向量 $\mathbf{x}_1, \dots, \mathbf{x}_n$ 。样本数量 n 由用户自己定， n 越大，计算量越大，近似越准确。
2. 计算集合 Ω 的体积：

$$v = \int_{\Omega} d\mathbf{x}.$$

3. 对函数值 $f(\mathbf{x}_1), \dots, f(\mathbf{x}_n)$ 求平均，再乘以 Ω 体积 v ：

$$q_n = v \cdot \frac{1}{n} \sum_{i=1}^n f(\mathbf{x}_i). \tag{1.3}$$

4. 返回 q_n 作为定积分 I 的估计值。

注意，算法第二步需要求 Ω 的体积。如果 Ω 是长方体、球体等规则形状，那么可以解析

地算出体积 v 。可是如果 Ω 是不规则形状，那么就需要定积分求 Ω 的体积 v ，这是比较困难的。可以用类似于上一小节“求阴影部分面积”的方法近似计算体积 v 。

举例讲解多元函数的蒙特卡洛积分：这个例子中被积分的函数是二元函数：

$$f(x, y) = \begin{cases} 1, & \text{if } x^2 + y^2 \leq 1; \\ 0, & \text{otherwise.} \end{cases} \quad (1.4)$$

直观地说，如果点 (x, y) 落在右图的绿色圆内，那么函数值就是 1；否则函数值就是 0。定义集合 $\Omega = [-1, 1] \times [-1, 1]$ ，即右图中蓝色的正方形，它的面积是 $v = 4$ 。定积分

$$I = \int_{\Omega} f(x, y) dx dy$$

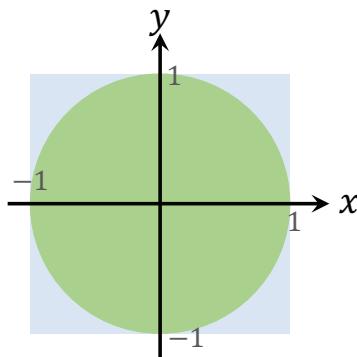


图 1.6：用蒙特卡洛积分近似 π 。

等于多少呢？很显然，定积分等于圆的面积，即 $\pi \cdot 1^2 = \pi$ 。因此，定积分 $I = \pi$ 。用蒙特卡洛求出 I ，就得到了 π 。从集合 $\Omega = [-1, 1] \times [-1, 1]$ 上均匀随机抽样 n 个点，记作 $(x_1, y_1), \dots, (x_n, y_n)$ 。应用公式 (1.3)，可得

$$q_n = v \cdot \frac{1}{n} \sum_{i=1}^n f(x_i, y_i) = \frac{4}{n} \sum_{i=1}^n f(x_i, y_i). \quad (1.5)$$

把 q_n 作为对定积分 $I = \pi$ 的近似。这与第 1.2.1 小节近似 π 的算法完全相同，区别在于此处的算法是从另一个角度推导出的。

1.2.4 例四：近似期望

蒙特卡洛还可以用来近似期望，这在整本书中会反复应用。定义 X 是 d 维随机变量，它的取值范围是集合 $\Omega \subset \mathbb{R}^d$ 。函数 $p(\mathbf{x}) = \mathbb{P}(X = \mathbf{x})$ 是 X 的概率密度函数，它描述变量 X 在取值点 \mathbf{x} 附近的可能性。设 $f : \Omega \mapsto \mathbb{R}$ 是任意的多元函数，它关于变量 X 的期望是：

$$\mathbb{E}_{X \sim p(\cdot)} [f(X)] = \int_{\Omega} p(\mathbf{x}) \cdot f(\mathbf{x}) d\mathbf{x}.$$

由于期望是定积分，所以可以按照上一小节的方法，用蒙特卡洛求定积分。上一小节在集合 Ω 上做**均匀抽样**，用得到的样本近似上面的定积分。

下面介绍一种更好的算法。既然我们知道概率密度函数 $p(\mathbf{x})$ ，我们最好是按照 $p(\mathbf{x})$ 做**非均匀抽样**，而不是均匀抽样。按照 $p(\mathbf{x})$ 做非均匀抽样，可以比均匀抽样有更快的收敛。具体步骤如下：

1. 按照概率密度函数 $p(\mathbf{x})$ ，在集合 Ω 上做**非均匀随机抽样**，得到 n 个样本，记作向量 $\mathbf{x}_1, \dots, \mathbf{x}_n \sim p(\cdot)$ 。样本数量 n 由用户自己定， n 越大，计算量越大，近似越准确。
2. 对函数值 $f(\mathbf{x}_1), \dots, f(\mathbf{x}_n)$ 求平均：

$$q_n = \frac{1}{n} \sum_{i=1}^n f(\mathbf{x}_i).$$

1.2 蒙特卡洛

3. 返回 q_n 作为期望 $\mathbb{E}_{X \sim p(\cdot)}[f(X)]$ 的估计值。

注 如果按照上述方式做编程实现，需要储存函数值 $f(\mathbf{x}_1), \dots, f(\mathbf{x}_n)$ 。用如下的方式做编程实现，可以减小内存开销。初始化 $q_0 = 0$ 。从 $t = 1$ 到 n ，依次计算

$$q_t = (1 - \frac{1}{t}) \cdot q_{t-1} + \frac{1}{t} \cdot f(\mathbf{x}_t). \quad (1.6)$$

不难证明，这样得到的 q_n 等于 $\frac{1}{n} \sum_{i=1}^n f(\mathbf{x}_i)$ 。这样无需存储所有的 $f(\mathbf{x}_1), \dots, f(\mathbf{x}_n)$ 。可以进一步把公式 (1.6) 中的 $\frac{1}{t}$ 替换成 α_t ，得到公式：

$$q_t = (1 - \alpha_t) \cdot q_{t-1} + \alpha_t \cdot f(\mathbf{x}_t).$$

这个公式叫做 Robbins-Monro 算法，其中 α_t 称为学习步长或学习率。只要 α_t 满足下面的性质，就能保证算法的正确性：

$$\lim_{n \rightarrow \infty} \sum_{t=1}^n \alpha_t = \infty \quad \text{和} \quad \lim_{n \rightarrow \infty} \sum_{t=1}^n \alpha_t^2 < \infty.$$

很显然， $\alpha_t = \frac{1}{t}$ 满足上述性质。Robbins-Monro 算法可以应用在 Q 学习算法中。

1.2.5 例五：随机梯度

蒙特卡洛近似期望在机器学习中的一个应用是**随机梯度**。设随机变量 X 为一个数据点，设 \mathbf{w} 为神经网络的参数。设 $p(x) = \mathbb{P}(X = x)$ 为随机变量 X 的概率密度函数。定义损失函数 $L(X; \mathbf{w})$ 。它的值越小，意味着模型对 X 的预测越准确；反之，它的值越大，则意味着模型对 X 的预测越离谱。因此，我们希望调整神经网络的参数 \mathbf{w} ，使得损失函数的期望尽量小。神经网络的训练可以定义为这样的优化问题：

$$\min_{\mathbf{w}} \mathbb{E}_{X \sim p(\cdot)} [L(X; \mathbf{w})]. \quad (1.7)$$

目标函数 $\mathbb{E}_X [L(X; \mathbf{w})]$ 关于 \mathbf{w} 的梯度是：

$$\mathbf{g} \triangleq \nabla_{\mathbf{w}} \mathbb{E}_{X \sim p(\cdot)} [L(X; \mathbf{w})] = \mathbb{E}_{X \sim p(\cdot)} [\nabla_{\mathbf{w}} L(X; \mathbf{w})].$$

可以做梯度下降更新 \mathbf{w} ，以减小目标函数 $\mathbb{E}_X [L(X; \mathbf{w})]$ ：

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \mathbf{g}.$$

此处的 α 被称作学习率 (Learning Rate)。直接计算梯度 \mathbf{g} 通常会比较慢。为了加速计算，可以对期望

$$\mathbf{g} = \mathbb{E}_{X \sim p(\cdot)} [\nabla_{\mathbf{w}} L(X; \mathbf{w})]$$

做蒙特卡洛近似，把得到的近似梯度 $\tilde{\mathbf{g}}$ 称作随机梯度 (Stochastic Gradient)，用 $\tilde{\mathbf{g}}$ 代替 \mathbf{g} 来更新 \mathbf{w} 。

1. 根据概率密度函数 $p(x)$ 做随机抽样，得到 b 个样本，记作 $\tilde{x}_1, \dots, \tilde{x}_b$ 。
2. 计算梯度 $\nabla_{\mathbf{w}} L(\tilde{x}_j; \mathbf{w})$ ， $\forall j = 1, \dots, b$ 。对它们求平均：

$$\tilde{\mathbf{g}} = \frac{1}{b} \sum_{j=1}^b \nabla_{\mathbf{w}} L(\tilde{x}_j; \mathbf{w}).$$

$\tilde{\mathbf{g}}$ 被称作随机梯度，它是 \mathbf{g} 的蒙特卡洛近似。

3. 做随机梯度下降更新 \mathbf{w} :

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \tilde{\mathbf{g}}.$$

蒙特卡洛用的样本数量 b 称作批量大小 (Batch Size)，通常是一个比较小的整数，比如 1、8、16、32。

在机器学习中，随机变量 X 通常服从下面这个离散概率分布。已经收集到一个数据集 $\{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ ，定义概率质量函数为：

$$p(\mathbf{x}_i) = \mathbb{P}(X = \mathbf{x}_i) = \frac{1}{n}, \quad \forall i = 1, \dots, n.$$

这个概率分布的意思是随机变量 X 的取值是 n 个数据点中的一个，概率都是 $\frac{1}{n}$ 。那么随机梯度下降每一轮都从集合 $\{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ 中均匀随机抽取 b 个样本。

 第一章 习题 

1. 设 X 是离散随机变量，取值范围是集合 $\mathcal{X} = \{1, 2, 3\}$ 。定义概率质量函数：

$$p(1) = \mathbb{P}(X = 1) = 0.4,$$

$$p(2) = \mathbb{P}(X = 2) = 0.1,$$

$$p(3) = \mathbb{P}(X = 3) = 0.5.$$

定义函数 $f(x) = 2x^2 + 3$ 。请计算 $\mathbb{E}_{X \sim p(\cdot)}[f(X)]$ 。

2. 设 X 服从均值为 $\mu = 1$ 、标准差 $\sigma = 2$ 的一元正态分布。定义函数 $f(x) = 2x + 10 \ln|x| + 3$ 。请设计蒙特卡洛算法，并编程计算 $\mathbb{E}_X[f(X)]$ 。

3. Bernstein 概率不等式是这样定义的。设 Z_1, \dots, Z_n 为独立的随机变量，它们的概率密度函数是任意的，但是它们必须满足三个条件：

- 变量的期望为零： $\mathbb{E}[Z_1] = \dots = \mathbb{E}[Z_n] = 0$ 。
- 变量是有界的：存在 $b > 0$ ，使得 $|Z_i| \leq b$, $\forall i = 1, \dots, n$ 。
- 变量的方差是有界的：存在 $v > 0$ ，使得 $\mathbb{E}[Z_i^2] \leq v$, $\forall i = 1, \dots, n$ 。

那么有这样的概率不等式：

$$\mathbb{P}\left(\left|\frac{1}{n} \sum_{i=1}^n Z_i\right| \geq \epsilon\right) \leq \exp\left(-\frac{\epsilon^2 n/2}{v + \epsilon b/3}\right).$$

公式 (1.5) 算出的 q_n 是 π 的蒙特卡洛近似。请用 Bernstein 不等式证明：

$$\left|q_n - \pi\right| = O\left(\frac{1}{\sqrt{n}}\right) \quad \text{以很高的概率成立。}$$

(提示：设 (X_i, Y_i) 是从正方形 $[-1, 1] \times [-1, 1]$ 中随机抽取的点。二元函数 f 在公式 (1.4) 中定义。设 $Z_i = 4f(X_i, Y_i) - \pi$ ，它是个均值为零的随机变量。)

4. 初始化 $q_0 = 0$ 。让 t 从 1 增长到 n ，依次计算

$$q_t = \left(1 - \frac{1}{t}\right) \cdot q_{t-1} + \frac{1}{t} \cdot f(\mathbf{x}_t).$$

请证明上述迭代得到的结果 q_n 等于 $\frac{1}{n} \sum_{i=1}^n f(\mathbf{x}_i)$ 。

《深度强化学习》2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

第二章 深度学习基础

本书假设读者有一定的机器学习基础，了解向量、矩阵、优化、梯度、梯度下降等基础知识。本章只是帮助读者查漏补缺，并由此熟悉本书的语言和符号。

2.1 线性模型

线性模型 (Linear Models) 是一类最简单的机器学习模型，常被用于简单的机器学习任务。可以将线性模型视为单层的神经网络。本节用最小二乘回归、逻辑斯蒂回归 (logistic regression)、Softmax 分类器这三种模型解决回归、二分类、多分类问题。

2.1.1 线性回归

以房价预测问题为例讲解回归 (Regression)。一个房屋有 d 个属性 (Attributes 或 Features)，比如面积、建造年份、离地铁站的距离。把一个房屋的 d 个属性记作向量：

$$\mathbf{x} = [x_1, x_2, \dots, x_d]^T.$$

本书中的向量 \mathbf{x} (除非它的转置 \mathbf{x}^T) 表示为列向量，记作粗体小写字母，以区分标量 (实数)。问题的目标是基于房屋的属性 $\mathbf{x} \in \mathbb{R}^d$ 预测其价格。

有多种方法对房价预测问题建模。最简单方法是使用如下线性模型：

$$f(\mathbf{x}; \mathbf{w}, b) \triangleq \mathbf{x}^T \mathbf{w} + b.$$

这里 $\mathbf{w} \in \mathbb{R}^d$ 和 $b \in \mathbb{R}$ 是模型的参数 (Parameters)。线性模型 $f(\mathbf{x}; \mathbf{w}, b)$ 的输出就是对房价的预测；输出既依赖于房屋的特征 \mathbf{x} ，也依赖于参数 \mathbf{w} 和 b 。很多书和论文将 \mathbf{w} 称作权重 (Weights)，将 b 称作偏移量 (Bias 或 Intercept)，原因是这样的：可以将 f 的定义 $\mathbf{x}^T \mathbf{w} + b$ 展开，得到

$$f(\mathbf{x}; \mathbf{w}, b) \triangleq w_1 x_1 + w_2 x_2 + \dots + w_d x_d + b.$$

如果 x_1 是房屋的面积，那么 w_1 就是房屋面积在房价中的权重。 w_1 越大，说明房价与面积的相关性越强；这就是为什么 \mathbf{w} 被称为权重。可以把偏移量 b 视作市面上房价的均值或者中位数，它与房屋的具体属性无关。

线性模型 $f(\mathbf{x}; \mathbf{w}, b)$ 依赖于参数 \mathbf{w} 和 b ；只有确定了 \mathbf{w} 和 b ，我们才能利用线性模型做预测。该怎么样获得 \mathbf{w} 和 b 呢？可以用历史数据来训练模型，得到参数 \mathbf{w}^* 和 b^* ，然后就可以用线性模型做预测：

$$f(\mathbf{x}; \mathbf{w}^*, b^*) \triangleq \mathbf{x}^T \mathbf{w}^* + b^*$$

卖家和中介可以用这个训练好的模型 f 给待售房屋定价。对于一个待售的房屋，首先找到它的面积、建造年份等属性，表示成向量 \mathbf{x}' ，然后把它输入 f ，得到

$$\hat{y}' = f(\mathbf{x}'; \mathbf{w}^*, b^*),$$

把它作为对该房屋价格的预测。

下面用最小二乘回归方法 (Least Squares Regression) 为例, 讲解如何训练线性模型 $f(\mathbf{x}; \mathbf{w}, b)$ 。训练有以下几个要点:

- **第一, 准备训练数据。** 收集到近期的 n 个房屋的属性和卖价, 作为训练数据集。把训练集记作 $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ 。向量 $\mathbf{x}_i \in \mathbb{R}^d$ 表示第 i 个房屋的所有属性, 标量 y_i 表示该房屋的成交价格。
- **第二, 把训练描述成优化问题。** 模型对第 i 个房屋价格的预测是 $\hat{y}_i = f(\mathbf{x}_i; \mathbf{w}, b)$, 而这个房屋的真实成交价格是 y_i 。我们希望 \hat{y}_i 尽量接近 y_i , 所以希望平方误差 $(\hat{y}_i - y_i)^2$ 越小越好。定义损失函数:

$$L(\mathbf{w}, b) = \frac{1}{2n} \sum_{i=1}^n [f(\mathbf{x}_i; \mathbf{w}, b) - y_i]^2.$$

我们希望找到 \mathbf{w} 和 b 使得损失函数尽量小, 也就是让模型的预测尽量准确。定义下面的优化模型:

$$\min_{\mathbf{w}, b} L(\mathbf{w}, b) + R(\mathbf{w}).$$

这个优化模型叫做最小二乘回归 (Least Squares Regression)。模型中的参数 \mathbf{w} 和 b 在此处叫做优化变量。 $L(\mathbf{w}, b) + R(\mathbf{w})$ 是目标函数。 $R(\mathbf{w})$ 是正则项 (Regularizer), 比如:

$$R(\mathbf{w}) = \lambda \|\mathbf{w}\|_2^2 \quad \text{或} \quad R(\mathbf{w}) = \lambda \|\mathbf{w}\|_1.$$

把优化问题的最优解记作:

$$(\mathbf{w}^*, b^*) = \underset{\mathbf{w}, b}{\operatorname{argmin}} L(\mathbf{w}, b) + R(\mathbf{w}).$$

请注意 \min 与 argmin 的区别。

- **第三, 用数值优化算法求解模型。** 在建立优化模型之后, 需要寻找最优解 (\mathbf{w}^*, b^*) 。通常随机初始化 (或全零初始化) \mathbf{w} 和 b , 然后用共轭梯度下降、随机梯度下降等优化算法迭代更新 \mathbf{w} 和 b 。

2.1.2 逻辑斯蒂回归

上一小节介绍了回归问题, 其中的预测目标 y 是连续变量, 比如房价就是连续数值。本小节研究二分类问题 (Binary Classification), 其中的预测目标 y 不是连续变量, 而是二元变量, 要么等于 0, 要么等于 1。本小节用逻辑斯蒂回归 (Logistic Regression) 解决二元分类问题¹。

以疾病检测为例讲解二元分类问题。为了初步排查癌症, 需要做血检, 血检中有 d 项指标, 包括白细胞数量、含氧量、以及多种激素含量。一份血液样本的检测报告作为一个 d 维向量:

$$\mathbf{x} = [x_1, x_2, \dots, x_d]^T.$$

医生需要基于 \mathbf{x} 来初步判断该血检是否意味着癌症。如果医生的判断为 $y = 1$, 则要求

¹注意, 虽然“逻辑斯蒂回归”的名字有“回归”, 但其通常用于解决二分类问题, 而非回归问题

2.1 线性模型

病人做进一步检测；如果医生的判断为 $y = 0$ ，则意味着未患癌症。这就是一个典型的二元分类问题。是否可以让机器学习做这种二元分类呢？

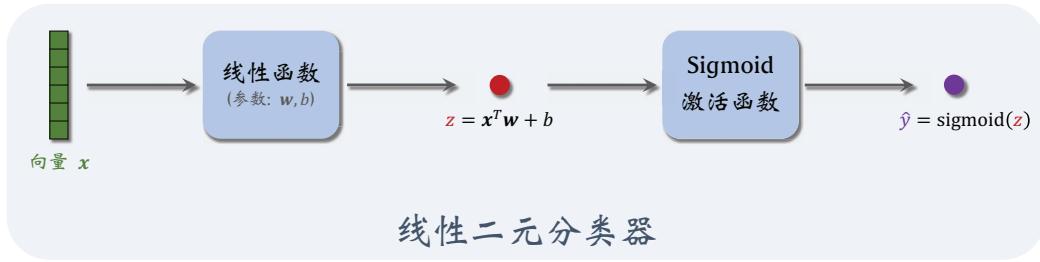


图 2.1：线性 Sigmoid 分类器的结构。输入是向量 $x \in \mathbb{R}^d$ ，输出是介于 0 和 1 之间的标量。

常用的是线性 Sigmoid 分类器，结构如图 2.1 所示。基于输入的向量 x ，线性分类器做出预测：

$$f(x; w, b) \triangleq \text{sigmoid}(x^T w + b).$$

此处的 Sigmoid 是个激活函数 (Activation Function)，定义为：

$$\text{sigmoid}(z) \triangleq \frac{1}{1 + \exp(-z)}.$$

如图 2.2 所示，Sigmoid 可以把任何实数映射到 0 到 1 之间。我们希望分类器的输出

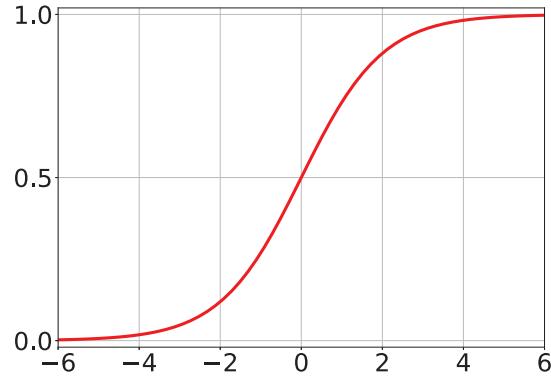


图 2.2：Sigmoid 函数的图像。

$\hat{y} = f(x; w, b)$ 有这样的性质：如果 x 是癌症患者的血检数据，那么 \hat{y} 接近 1；如果 x 是健康人的血检数据，那么 \hat{y} 接近 0。因此 \hat{y} 叫做“置信率”(Confidence)，即分类器有多大信心做出阳性的判断。比如 $\hat{y} = 0.9$ 表示分类器有 0.9 的信心判断血检为阳性； $\hat{y} = 0.05$ 表示分类器只有 0.05 的信心判断血检为阳性，即 0.95 的信心判断血检为阴性。

在介绍训练 Sigmoid 分类器的算法之前，先介绍交叉熵 (Cross Entropy)，它可以衡量两个概率分布的差别，因此常被用作分类问题的损失函数。用向量

$$\mathbf{p} = [p_1, \dots, p_m]^T \quad \text{和} \quad \mathbf{q} = [q_1, \dots, q_m]^T$$

表示两个离散概率分布。向量的元素都非负，而且 $\sum_{j=1}^m p_j = 1$ ， $\sum_{j=1}^m q_j = 1$ 。两个概率分布的交叉熵定义为：

$$H(\mathbf{p}, \mathbf{q}) = - \sum_{j=1}^m p_j \cdot \ln q_j.$$

两个概率分布越接近，则交叉熵越小。

我们做以下步骤，从数据中学习模型参数 w 和 b 。

- **第一，准备训练数据。** 收集 n 份血检报告和最终的诊断，作为训练数据集： $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ 。向量 $\mathbf{x}_i \in \mathbb{R}^d$ 表示第 i 份血检报告中的所有指标；二元标签 $y_i = 1$ 表示患有癌症（阳性）， $y_i = 0$ 表示健康（阴性）。
- **第二，把训练描述成优化问题。** 分类器对第 i 份血检报告的预测是 $f(\mathbf{x}_i; w, b)$ ，而真实患癌情况是 y_i 。想要用交叉熵衡量 y_i 与 $f(\mathbf{x}_i; w, b)$ 之间的差别，得把 y_i 与

$f(\mathbf{x}_i; \mathbf{w}, b)$ 表示成向量：

$$\begin{bmatrix} y_i \\ 1 - y_i \end{bmatrix} \quad \text{和} \quad \begin{bmatrix} f(\mathbf{x}_i; \mathbf{w}, b) \\ 1 - f(\mathbf{x}_i; \mathbf{w}, b) \end{bmatrix}.$$

两个向量的第一个元素都对应阳性的置信率，第二个元素都对应阴性的置信率。分类器预测越准确，则两个向量尽量越接近，它们的交叉熵越小。定义损失函数为平均交叉熵：

$$L(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n H \left(\begin{bmatrix} y_i \\ 1 - y_i \end{bmatrix}, \begin{bmatrix} f(\mathbf{x}_i; \mathbf{w}, b) \\ 1 - f(\mathbf{x}_i; \mathbf{w}, b) \end{bmatrix} \right).$$

我们希望找到 \mathbf{w} 和 b 使得损失函数尽量小，也就是让分类器的预测尽量准确。定义下面的优化问题：

$$\min_{\mathbf{w}, b} L(\mathbf{w}, b) + R(\mathbf{w}).$$

这个优化问题叫做正则化逻辑斯蒂回归。公式中的 $R(\mathbf{w})$ 是正则项。

- **第三，用数值优化算法求解。** 在建立优化模型之后，需要寻找最优解 (\mathbf{w}^*, b^*) 。通常随机初始化（或全零初始化）优化变量 \mathbf{w} 和 b ，然后用梯度下降、随机梯度下降、L-BFGS 等优化算法迭代更新优化变量。

2.1.3 Softmax 分类器

上一小节介绍了二元分类问题，数据只分为两个类别，比如患病和健康。本小节研究多分类问题，数据可以划分为 $k (> 2)$ 个类别。我们可以用线性 Softmax 分类器解决多分类问题。

本小节用 MNIST 手写数字识别为例讲解多分类问题。如图 2.3 所示，MNIST 数据集有 $n = 60,000$ 个样本，每个样本是 28×28 的图片。数据集有 $k = 10$ 个类别，每个样本有一个类别标签，它是介于 0 到 9 之间的整数，表示图片中的数字。为了训练 Softmax 分类器，我们要对标签做 One-Hot 编码，把每个标签（0 到 9 之间的整数）映射到 $k = 10$ 维的向量：



图 2.3: MNIST 数据集中的图片。

$$0 \implies [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],$$

$$1 \implies [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],$$

⋮

$$8 \implies [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],$$

$$9 \implies [0, 0, 0, 0, 0, 0, 0, 0, 0, 1].$$

2.1 线性模型

把得到的标签记作 $y_1, \dots, y_n \in \mathbb{R}^{10}$ 。把每张 28×28 的图片拉伸成 $d = 784$ 维的向量，记作 $x_1, \dots, x_n \in \mathbb{R}^{784}$ 。

在介绍 Softmax 分类器之前，先介绍 Softmax 激活函数。它的输入和输出都是 k 维向量。设 $z = [z_1, \dots, z_k]^T$ 是任意 k 维向量，它的元素可正可负。Softmax 函数的输出

$$\text{softmax}(z) \triangleq \frac{1}{\sum_{l=1}^k \exp(z_l)} [\exp(z_1), \exp(z_2), \dots, \exp(z_k)]^T$$

也是个 k 维向量，它的元素都是非负，而且相加等于 1。如图 2.4 所示，Softmax 函数让最大的元素相对变得更大，让小的元素接近 0。图 2.5 是 Max 函数，它把最大的元素映射到 1，其余所有元素映射到 0。对比一下图 2.4 和图 2.5，不难看出为什么 Softmax 没有让小的元素等于零，这就是为什么它的名字带有“Soft”。

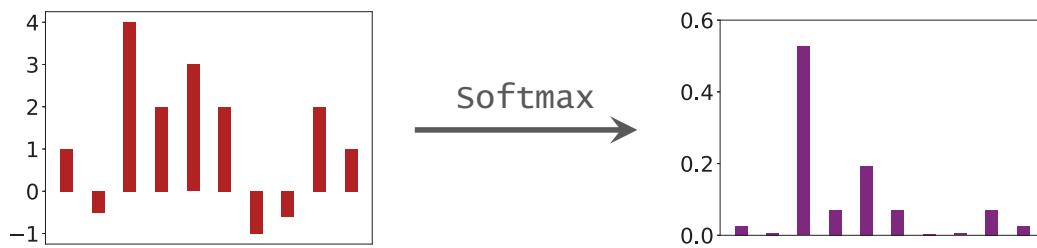


图 2.4: Softmax 函数把左边红色的 10 个数值映射到右边紫色的 10 个数值。

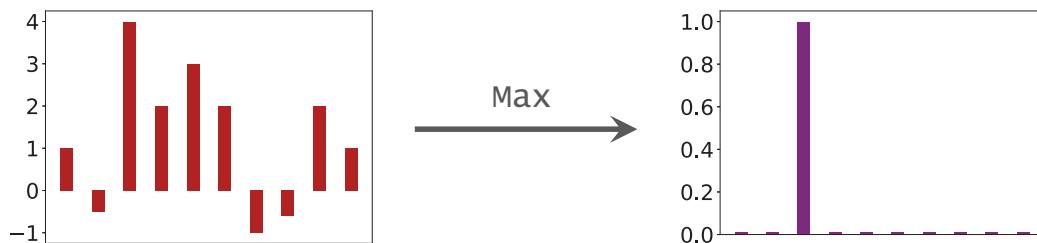


图 2.5: Max 函数把左边红色的 10 个数值映射到右边紫色的 10 个数值。

Softmax 分类器是常用的多元分类器。线性 Softmax 分类器其实是线性函数 + Softmax 激活函数；结构如图 2.6 所示。它的参数是矩阵 $\mathbf{W} \in \mathbb{R}^{k \times d}$ 和向量 $\mathbf{b} \in \mathbb{R}^k$ ，这里的 d 是输入向量的维度， k 是类别数量。基于输入的向量 $\mathbf{x} \in \mathbb{R}^d$ ，分类器做出预测：

$$f(\mathbf{x}; \mathbf{W}, \mathbf{b}) \triangleq \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}).$$

设 $\hat{\mathbf{y}} = f(\mathbf{x}; \mathbf{W}, \mathbf{b})$ 是 Softmax 分类器的输出，它的 $k = 10$ 个元素可以视为 $k = 10$ 个类别的置信率。举个例子，设

$$\hat{\mathbf{y}} = [0.1, 0.6, 0.02, 0.01, 0.01, 0.2, 0.01, 0.03, 0.01, 0.01]^T.$$

可以这样理解分类器的输出向量 $\hat{\mathbf{y}} = f(\mathbf{x}; \mathbf{W}, \mathbf{b})$:

- 第零个（从零计数）元素 0.1 表示分类器以 0.1 的信心判定图片 x 是数字“0”，
- 第一个元素 0.6 表示分类器以 0.6 的信心判定 x 是数字“1”，
- 第二个元素 0.02 表示分类器只有 0.02 的信心判定 x 是数字“2”，

以此类推。由于分类器的输出向量 $\hat{\mathbf{y}} = f(\mathbf{x}; \mathbf{W}, \mathbf{b})$ 的第 1 个元素 0.6 是最大的，分类器

认为图片 x 是数字“1”。

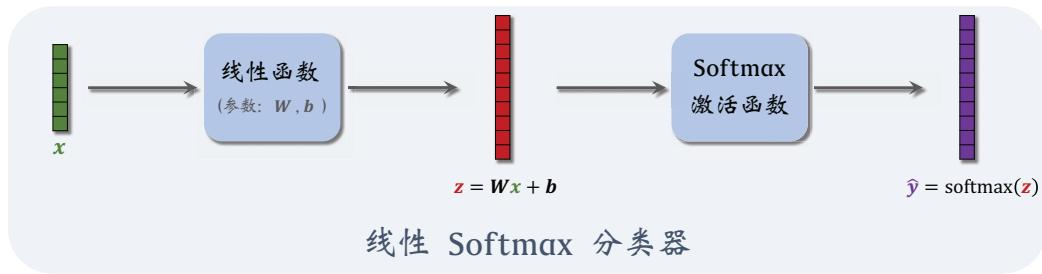


图 2.6: 线性 Softmax 分类器的结构。输入是向量 $x \in \mathbb{R}^d$, 输出是 $\hat{y} \in \mathbb{R}^k$ 。

我们做以下步骤, 从数据中学习模型参数 $\mathbf{W} \in \mathbb{R}^{k \times d}$ 和 $\mathbf{b} \in \mathbb{R}^k$ 。

- **第一, 准备训练数据。**一共有 $n = 60,000$ 张手写数字图片, 每张图片大小为 28×28 , 需要把图片变成 $d = 784$ 维的向量, 记作 $x_1, \dots, x_n \in \mathbb{R}^d$ 。每张图片有一个标签, 它是 0 到 9 之间的整数, 需要把它做 One-Hot 编码, 变成 $k = 10$ 维的 One-Hot 向量; 把 One-Hot 标签记作 y_1, \dots, y_n 。
- **第二, 把训练描述成优化问题。**分类器对第 i 张图片 x_i 的预测是 $\hat{y}_i = f(x_i; \mathbf{W}, \mathbf{b})$, 它是 $k = 10$ 维的向量, 可以反映出分类结果。我们希望 \hat{y}_i 尽量接近真实标签 y_i (10 维的 One-Hot 向量), 也就是希望交叉熵 $H(y_i, \hat{y}_i)$ 尽量小。定义损失函数为平均交叉熵:

$$L(\mathbf{W}, \mathbf{b}) = \frac{1}{n} \sum_{i=1}^n H(y_i, \hat{y}_i).$$

我们希望找到参数矩阵 \mathbf{W} 和向量 \mathbf{b} 使得损失函数尽量小, 也就是让分类器的预测尽量准确。定义下面的优化问题:

$$\min_{\mathbf{W}, \mathbf{b}} L(\mathbf{W}, \mathbf{b}) + R(\mathbf{W}).$$

- **第三, 用数值优化算法求解。**在建立优化模型之后, 需要寻找最优解 $(\mathbf{W}^*, \mathbf{b}^*)$ 。通常随机初始化 (或全零初始化) 优化变量 \mathbf{W} 和 \mathbf{b} , 然后用梯度下降、随机梯度下降等优化算法迭代更新优化变量。

2.2 非线性核方法

看是否有必要加一节??

2.3 神经网络

2.3.1 全连接神经网络（多层感知器）

接着上一节的内容，我们继续研究 MNIST 手写识别这个多元分类问题。人类做手写数字识别的准确率接近 100%，然而线性 Softmax 分类器在 MNIST 数据集只有 90% 的准确率，远低于人类的表现。线性分类器的表现差的原因在于模型太小，不能充分利用 $n = 60,000$ 个训练样本。我们可以把“线性函数 + 激活函数”这样的结构作为一个层，累起来，得到一个多层次的神经网络，获得更高的预测准确率。

全连接层：把输入记作向量 $\mathbf{x} \in \mathbb{R}^d$ ，神经网络的一个层把 \mathbf{x} 映射到 $\mathbf{x}' \in \mathbb{R}^{d'}$ 。全连接层是这样定义的：

$$\mathbf{x}' = \sigma(\mathbf{z}), \quad \mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b},$$

此处的矩阵 $\mathbf{W} \in \mathbb{R}^{d' \times d}$ 和向量 $\mathbf{b} \in \mathbb{R}^{d'}$ 是这一层的参数，需要从数据中学习； $\sigma(\cdot)$ 是激活函数，比如 Softmax 函数、Sigmoid 函数、ReLU 函数。最常用的激活函数是 ReLU，定义为：

$$\text{ReLU}(\mathbf{z}) = \left[\max\{0, z_1\}, \max\{0, z_2\}, \dots, \max\{0, z_{d'}\} \right]^T.$$

我们把这样的一个层叫做全连接层 (Dense Layer、Linear Layer、或 Fully Connected Layer)，结构如图 2.7 所示。只有线性函数、没有激活函数也算一层；但是单独的激活函数不算一层，因为激活函数中没有要学习的参数。

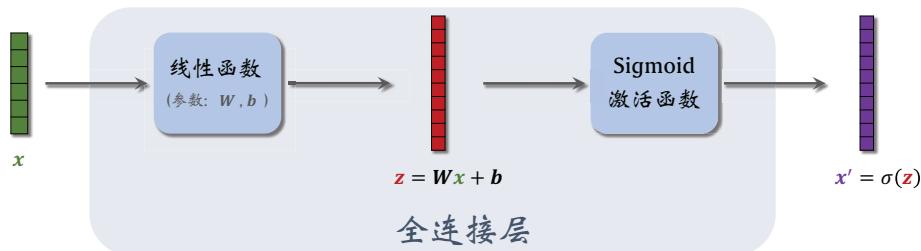


图 2.7：一个全连接层包括一个线性函数和一个激活函数。

全连接神经网络：我们可以把全连接层当做基本组件，然后像搭积木一样搭建一个全连接神经网络 (Fully-Connected Neural Network)，也叫多层次感知器 (Multi-Layer Perceptron, MLP)。图 2.8 中是一个三层的全连接神经网络，它把输入向量 $\mathbf{x}^{(0)}$ 映射到 $\mathbf{x}^{(3)}$ 。一个 l

层的全连接神经网络可以表示为：

$$\begin{aligned} \text{第 1 层: } \quad \boldsymbol{x}^{(1)} &= \sigma_1(\boldsymbol{W}^{(1)}\boldsymbol{x}^{(0)} + \boldsymbol{b}^{(1)}), \\ \text{第 2 层: } \quad \boldsymbol{x}^{(2)} &= \sigma_2(\boldsymbol{W}^{(2)}\boldsymbol{x}^{(1)} + \boldsymbol{b}^{(2)}), \\ &\vdots && \vdots \\ \text{第 } l \text{ 层: } \quad \boldsymbol{x}^{(l)} &= \sigma_l(\boldsymbol{W}^{(l)}\boldsymbol{x}^{(l-1)} + \boldsymbol{b}^{(l)}), \end{aligned}$$

其中的 $\boldsymbol{W}^{(1)}, \dots, \boldsymbol{W}^{(l)}, \boldsymbol{b}^{(1)}, \dots, \boldsymbol{b}^{(l)}$ 是神经网络的参数，需要从训练数据学习；不同层的参数是不同的。 $\sigma_1, \dots, \sigma_l$ 为激活函数；它们可以相同，也可以不同。

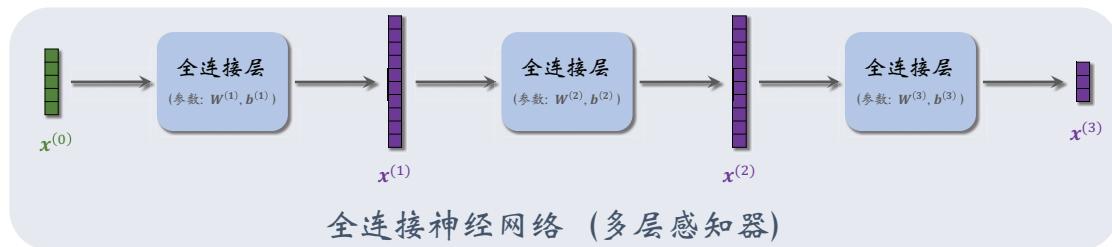


图 2.8：图中的神经网络由 3 个全连接层组成，每个层有自己的参数。

编程实现：可以用 TensorFlow、PyTorch、Keras 等深度学习标准库实现全连接神经网络，只需要一两行代码就能添加一个全连接层。添加一个全连接层需要手动指定两个超参数：

- **层的宽度。**如果一个层是隐层（即除了第 l 层之外的所有层），那么需要指定层的宽度（即输出向量的维度）。输出层（即第 l 层）的宽度由问题本身决定。比如 MNIST 数据集有 10 类，那么输出层的宽度必须是 10。而对于二元分类问题，输出层的宽度是 1。
- **激活函数。**用户需要决定每一层的激活函数。对于隐层，通常使用 ReLU 激活函数即可；使用其它激活函数或许能微弱地提高神经网络的表现。对于输出层，激活函数的选择要取决于具体问题。二元分类问题用 Sigmoid，多元分类问题用 Softmax，回归问题通常不用激活函数。

2.3.2 卷积神经网络

卷积神经网络 (Convolutional Neural Network)，缩写 CNN，是主要由卷积层组成的神经网络²。卷积神经网络的结构如图 2.9 所示。输入 $\boldsymbol{X}^{(0)}$ 是第三阶张量 (Tensor)³。卷积层的输入和输出都是第三阶张量，每个卷积层之后通常有一个 ReLU 激活函数（图 2.9 中没有画出）。可以把几个、甚至几十个卷积层累起来，得到深度卷积神经网络。把最后一个卷积层输出的张量做向量化 (Flatten)，得到一个向量。

²CNN 中也可以有池化层 (Pooling)，本书不做讨论

³第零阶张量为标量（实数），第一阶张量为向量，第二阶张量为矩阵，以此类推。

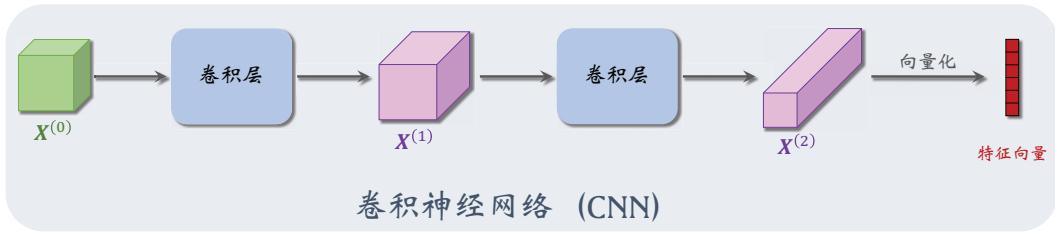


图 2.9: 图中的神经网络由 3 个全连接层组成，每个层有自己的参数。

本书不具体解释 CNN 的原理，本书也不会用到这些原理。读者仅需要记住这个知识点：卷积神经网络的输入是矩阵或三阶张量；卷积网络从张量提取特征，最终输出提取的特征向量。图片通常是矩阵（灰度图片）和三阶张量（彩色图片），可以用 CNN 从中提取特征，然后用一个或多个全连接层做分类或回归。

图 2.10 是一个由卷积、全连接等层组成的深度神经网络。其中的卷积网络从输入的矩阵（灰度图片）中提取特征，全连接网络把特征向量映射成 10 维的向量，最终的 Softmax 激活函数输出 10 维向量 \hat{y} 。输出向量 \hat{y} 的 10 个元素是 10 个类别的置信率，可以反映出分类结果。

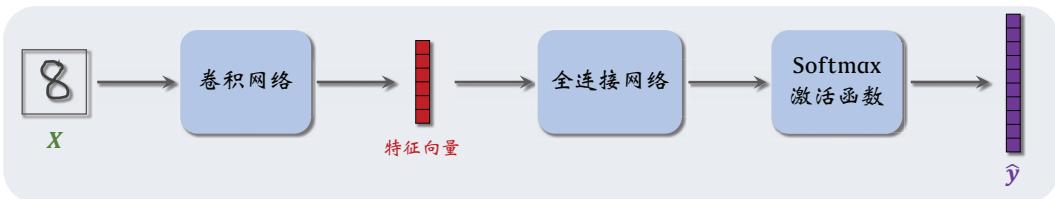


图 2.10: 用于分类 MNIST 手写数字的深度神经网络。

2.4 反向传播和梯度下降

线性模型和神经网络的训练都可以描述成一个优化问题。设 $\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(l)}$ 为优化变量（可以是向量、矩阵、张量）。我们希望求解这样一个优化问题：

$$\min_{\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(l)}} L(\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(l)}).$$

对于这样一个无约束的最小化问题，最常使用的算法是梯度下降 (Gradient Descent, 缩写 GD) 和随机梯度下降 (Stochastic Gradient Descent, 缩写 SGD)。本节的内容包括梯度、梯度算法、以及用反向传播计算梯度。

2.4.1 梯度下降

梯度：几乎所有常用的优化算法都需要计算梯度。目标函数 L 关于一个变量 $\mathbf{w}^{(i)}$ 的梯度记作：

$$\underbrace{\nabla_{\mathbf{w}^{(i)}} L(\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(l)})}_{\text{两种符号都表示 } L \text{ 关于 } \mathbf{w}^{(i)} \text{ 的梯度}} \triangleq \frac{\partial L(\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(l)})}{\partial \mathbf{w}^{(i)}}, \quad \forall i = 1, \dots, l.$$

目标函数的值是标量（实数），所以梯度 $\nabla_{\mathbf{w}^{(i)}} L$ 的形状与 $\mathbf{w}^{(i)}$ 完全相同。

- 如果 $\mathbf{w}^{(i)}$ 是 $d \times 1$ 的向量，那么 $\nabla_{\mathbf{w}^{(i)}} L$ 也是 $d \times 1$ 的向量；
- 如果 $\mathbf{w}^{(i)}$ 是 $d_1 \times d_2$ 的矩阵，那么 $\nabla_{\mathbf{w}^{(i)}} L$ 也是 $d_1 \times d_2$ 的矩阵；
- 如果 $\mathbf{w}^{(i)}$ 是 $d_1 \times d_2 \times d_3$ 的第三阶张量，那么 $\nabla_{\mathbf{w}^{(i)}} L$ 也是 $d_1 \times d_2 \times d_3$ 的张量。

不论是自己手动推导梯度，还是用程序自动求梯度，都需要检查梯度的形状与变量的形状是否相同；如果不同，梯度的计算肯定有错。

梯度下降 (GD)：梯度是上升方向，沿着梯度方向对优化变量 $\mathbf{w}^{(i)}$ 做一小步更新，可以让目标函数值增加。既然我们的目标是最小化目标函数，就应该沿着梯度的反方向更新优化变量。沿着梯度反方向走就叫做梯度下降 (GD)。设当前的优化变量为 $\mathbf{w}_{\text{now}}^{(1)}, \dots, \mathbf{w}_{\text{now}}^{(l)}$ ，计算目标函数 L 在当前的梯度，然后做 GD 更新优化变量：

$$\mathbf{w}_{\text{new}}^{(i)} \leftarrow \mathbf{w}_{\text{now}}^{(i)} - \alpha \cdot \nabla_{\mathbf{w}^{(i)}} L(\mathbf{w}_{\text{now}}^{(1)}, \dots, \mathbf{w}_{\text{now}}^{(l)}), \quad \forall i = 1, \dots, l.$$

此处的 $\alpha (> 0)$ 叫做学习率 (Learning Rate) 或者步长 (Step Size)，它的设置既影响 GD 收敛速度，也影响最终神经网络的测试准确率，所以 α 需要用户仔细调整。

随机梯度下降 (SGD)：如果目标函数可以写成连加或者期望的形式，那么可以用随机梯度求解最小化问题。假设目标函数可以写成 n 项连加形式：

$$L(\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(l)}) = \frac{1}{n} \sum_{j=1}^n F_j(\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(l)}).$$

函数 F_j 隐含第 j 个训练样本 $(\mathbf{x}_j, \mathbf{y}_j)$ 。每次随机从集合 $\{1, 2, \dots, n\}$ 中抽取一个整数，记作 j 。设当前的优化变量为 $\mathbf{w}_{\text{now}}^{(1)}, \dots, \mathbf{w}_{\text{now}}^{(l)}$ ，计算此处的随机梯度，并且做随机梯度下降：

$$\mathbf{w}_{\text{new}}^{(i)} \leftarrow \mathbf{w}_{\text{now}}^{(i)} - \alpha \cdot \underbrace{\nabla_{\mathbf{w}^{(i)}} F_j(\mathbf{w}_{\text{now}}^{(1)}, \dots, \mathbf{w}_{\text{now}}^{(l)})}_{\text{随机梯度}}, \quad \forall i = 1, \dots, l.$$

实际训练神经网络的时候，总是用 SGD（及其变体），而不用 GD。主要原因是 GD 用于非凸问题会卡在鞍点 (Saddle Point)，收敛不到局部最优，这会导致测试准确率很低；而 SGD 可以跳出鞍点，趋近局部最优。次要原因是 GD 每一步的计算量都很大，比 SGD 大 n 倍，所以 GD 通常很慢（除非用并行计算）。

SGD 的变体：理论分析和实践都表明 SGD 的一些变体比简单的 SGD 收敛更快。这些变体都基于随机梯度，只是会对随机梯度做一些变换。常见的变体有 Momentum、AdaGrad、Adam、RMSProp。能用 SGD 的地方就能用这些变体。因此，本书中只用 SGD 讲解强化学习算法，不去具体讨论 SGD 的变体。

2.4.2 反向传播

随机梯度下降需要用到损失关于优化变量（即模型参数）的梯度。对于一个深度神经网络，需要用反向传播 (Backpropagation) 求损失函数关于变量的梯度。如果用 TensorFlow 和 PyTorch 等深度学习平台，你不需要关心梯度是如何求出来的。只要你定义的函数对某个变量可微，TensorFlow 和 PyTorch 就可以自动求该函数关于该变量的梯度。

本节以全连接网络为例，简单介绍反向传播的原理。全连接神经网络（忽略掉偏移量 b ）是这样定义的：

$$\begin{aligned} \text{第 1 层: } \quad \mathbf{x}^{(1)} &= \sigma_1(\mathbf{W}^{(1)} \mathbf{x}^{(0)}), \\ \text{第 2 层: } \quad \mathbf{x}^{(2)} &= \sigma_2(\mathbf{W}^{(2)} \mathbf{x}^{(1)}), \\ &\vdots && \vdots \\ \text{第 } l \text{ 层: } \quad \mathbf{x}^{(l)} &= \sigma_l(\mathbf{W}^{(l)} \mathbf{x}^{(l-1)}). \end{aligned}$$

神经网络的输出 $\mathbf{x}^{(l)}$ 是神经网络做出的预测。设 h 为损失，比如

$$z = H(\mathbf{y}, \mathbf{x}^{(l)}),$$

其中函数 H 表示交叉熵，向量 \mathbf{y} 表示真实标签。为了做梯度下降更新参数 $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(l)}$ ，我们需要计算损失 z 关于每一个变量的梯度：

$$\frac{\partial z}{\partial \mathbf{W}^{(1)}}, \quad \frac{\partial z}{\partial \mathbf{W}^{(2)}}, \quad \dots, \quad \frac{\partial z}{\partial \mathbf{W}^{(l)}}.$$

损失 z 与参数 $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(l)}$ 、变量 $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(l)}$ 的关系如图 2.11 所示。

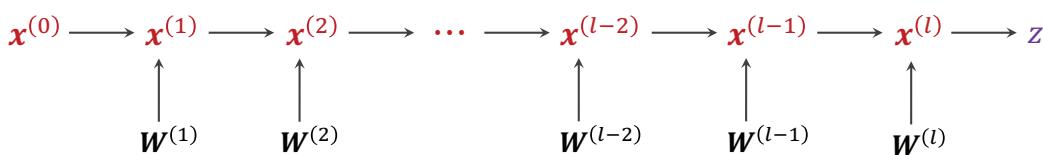


图 2.11：变量的函数关系。

反向传播的本质是求导的链式法则 (Chain Rule)。设变量有这样的关系： $x \rightarrow y \rightarrow z$ 。那么可以用链式法则求出 z 关于 x 的偏导：

$$\frac{\partial z}{\partial x} = \frac{\partial y}{\partial x} \cdot \frac{\partial z}{\partial y}.$$

同理，可以用链式法则做反向传播，得到损失关于神经网络参数的梯度。具体这样做。首先求出梯度 $\frac{\partial z}{\partial \mathbf{x}^{(l)}}$ 。然后做循环，从 $i = l, \dots, 1$ ，依次做如下操作：

- 根据链式法则可得损失 z 关于参数 $\mathbf{W}^{(i)}$ 的梯度：

$$\frac{\partial z}{\partial \mathbf{W}^{(i)}} = \frac{\partial \mathbf{x}^{(i)}}{\partial \mathbf{W}^{(i)}} \cdot \frac{\partial z}{\partial \mathbf{x}^{(i)}}.$$

这项梯度被用于更新参数 $\mathbf{W}^{(i)}$ 。

- 根据链式法则可得损失 z 关于参数 $\mathbf{x}^{(i-1)}$ 的梯度：

$$\frac{\partial z}{\partial \mathbf{x}^{(i-1)}} = \frac{\partial \mathbf{x}^{(i)}}{\partial \mathbf{x}^{(i-1)}} \cdot \frac{\partial z}{\partial \mathbf{x}^{(i)}}.$$

这项梯度被传播到下面一层（即第 $i - 1$ 层），继续循环。

反向传播的路径如图 2.12 所示。只要知道损失 z 关于 $\mathbf{x}^{(i)}$ 的梯度，就能求出 z 关于 $\mathbf{W}^{(i)}$ 和 $\mathbf{x}^{(i-1)}$ 的梯度。

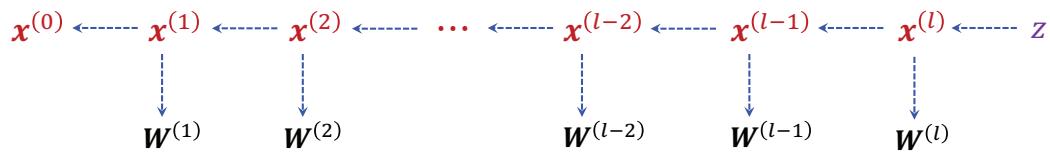


图 2.12：反向传播的路径。

第三章 马尔可夫决策过程 (MDP)

3.1 基本概念

强化学习的数学基础是马尔可夫决策过程 (Markov Decision Processes, 缩写 MDPs)。MDP 的专业术语非常多，这是强化学习较难入门的主要原因之一。想入门强化学习，没有什么捷径，只能理解和记住专业术语。为了方便读者理解和记忆，本节主要用超级玛丽的例子来解释强化学习的专业术语。



图 3.1：超级玛丽的例子中，玛丽奥是智能体，状态 s 是当前屏幕上的画面，动作空间是 $\mathcal{A} = \{\text{左, 右, 上}\}$ ，动作 a 是左、右、上三者中的一个。

状态 (State) 是对当前环境的一个概括。在超级玛丽的例子中，可以把屏幕当前的画面（或者最近几帧画面）看做状态。玩家只需要知道当前画面（或者最近几帧画面）就能够做出正确的决策，决定下一步是让超级玛丽向左、向右、或是向上。可以这样理解状态：状态是做决策的唯一依据。

再举一个例子，在中国象棋、五子棋游戏中，棋盘上所有棋子的位置就是状态，因为当前格局就足以供玩家做决策。假设你不是从头开始一局游戏，而是接手别人的残局。你只需要仔细观察棋盘上的格局，你就能够做出决策。知道这局游戏的历史记录（即每一步是怎么走的），并不会给你提供额外的信息。

举一个反例。星际争霸、红色警戒、英雄联盟这些游戏中，玩家屏幕上最近的 100 帧画面并不是状态，因为这些画面不是对当前环境完整的概括。在地图上某个你看不见的角落里可能正在发生些事件，这些事件足以改变游戏的结局。一个玩家屏幕上的画面只是对环境的部分观测 (Partial Observation)。最近的 100 帧画面并不足以供玩家做决策。

状态空间 (State Space) 是指所有可能存在状态的集合，记作花体字母 \mathcal{S} 。状态空间可能是有限集合，也可能是无限集合。在超级玛丽、星际争霸、无人驾驶这些例子中，状态空间是无限集合，存在无穷多种可能的状态。围棋、五子棋、中国象棋这些游戏中，状态空间是有限集合，可以枚举出所有可能存在的状态（也就是棋盘上的格局）。

动作 (Action) 是指做出的决策。在超级玛丽的例子中，假设玛丽奥只能向左走、向右走、向上跳。那么动作就是左、右、上三者中的一种。在围棋游戏中，棋盘上有 361 个

位置，于是有 361 种动作，第 i 种动作是指把棋子放到第 i 个位置上。

动作空间 (Action Space) 是指所有可能的动作的集合，记作花体字母 \mathcal{A} 。在超级玛丽的例子中，动作空间是 $\mathcal{A} = \{\text{左, 右, 上}\}$ 。在围棋的例子中，动作空间是 $\mathcal{A} = \{1, 2, 3, \dots, 361\}$ 。

智能体 (Agent) 是指做动作的主体：由谁做动作，谁就是智能体。在超级玛丽游戏中，玛丽奥就是智能体。在自动驾驶的应用中，无人车就是智能体。

策略函数 (Policy Function) 的意思是根据观测到的状态，做出决策，控制智能体运动。举个例子，假设你在玩超级玛丽游戏，当前屏幕上的画面是图 3.1。请问你该做什么决策？有很大概率你会决定向上跳，这样可以避开敌人，还能吃到金币。向上跳这个动作就是你大脑中的策略做出的决策。

有不同的方式定义策略函数。这里介绍一种最常用的定义。把状态记作 S 或 s ，动作记作 A 或 a 。策略函数 $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ 是一个条件概率密度函数：

$$\pi(a|s) = \mathbb{P}(A = a | S = s).$$

策略函数的输入是状态 s 和动作 a ，输出是一个 0 到 1 之间的概率值。举个例子，把图 3.1 中的屏幕画面作为状态 s 输入策略函数，策略函数输出动作的概率值：

$$\pi(\text{左} | s) = 0.2,$$

$$\pi(\text{右} | s) = 0.1,$$

$$\pi(\text{上} | s) = 0.7.$$

如果你让策略函数 π 来自动操作玛丽奥打游戏，它就会做一个随机抽样：以 0.2 的概率向左走，0.1 的概率向右走，0.7 的概率向上跳。三种动作都有可能发生，但是向上的概率最大，向左概率较小，向右概率最小。

强化学习学什么？就是学这个策略函数 π 。只要有了策略函数，就可以让它自动控制玛丽奥打赢游戏。

奖励 (Reward) 是在智能体执行一个动作之后，环境返回给智能体的一个数值。奖励往往由我们自己来定义；奖励定义得好坏非常影响强化学习的结果。比如可以这样定义，玛丽奥吃到一个金币，获得奖励 +1；如果玛丽奥通过一局关卡，奖励是 +1000；如果玛丽奥碰到敌人，游戏结束，奖励是 -1000；如果这一步什么都没发生，奖励就是 0。怎么定义奖励就见仁见智了。我们应该把打赢游戏的奖励定义得大一些，这样才能鼓励玛丽奥通过关卡，而不是一味地收集金币。

状态转移 (State Transition) 是指当前状态 s 变成新的状态 s' 。给定当前状态 s ，智能体执行动作 a ，环境 (Environment) 给出下一时刻的状态 s' 。请问 s' 是如何产生的呢？ s' 是由环境根据某个函数计算出来的，这个函数叫做状态转移函数，它把 (s, a) 映射到 s' ；稍后详细解释状态转移函数。

环境 (Environment) 又是什么呢？在超级玛丽的例子中，游戏程序就是环境。在围棋、象棋的例子中，游戏规则就是环境。在无人驾驶的应用中，真实的物理世界就是环境。谁能生成新的状态，谁就是环境。

状态转移函数 (State-Transition Function) 是环境用于生成新的状态 s' 时用到的函

3.1 基本概念

数。在超级玛丽的例子中，基于当前状态（屏幕上的画面），玛丽奥向上跳了一步，那么环境（即游戏程序）就会计算出新的状态（即下一帧画面）。在中国象棋的例子中，基于当前状态（棋盘上的格局），红方让“车”走到黑方“马”的位置上，那么环境（即游戏规则）就会将黑方的“马”移除，生成新的状态（棋盘上新的格局）。

状态转移函数可以是确定的。比如中国象棋的状态转移函数就是确定的：给定当前状态 s ，玩家执行动作 a ，那么新的状态 s' 是确定的，没有随机性。状态转移函数也可能是随机的；我们通常认为状态转移是随机的。状态转移的随机性是从环境来的。图 3.2 中的例子说明状态转移的随机性。



图 3.2：这个例子说明状态转移的随机性。如果玛丽奥向上跳，玛丽奥的位置就到上面来了；这个是确定的。但是标出的敌人 Goomba 有可能往左，也有可能往右。Goomba 移动的方向可以是随机的。即使当前状态 s 和智能体的动作 a 确定了，也无法确定下一个状态 s' 。

随机状态转移函数记作 $p(s'|s, a)$ ，它是一个条件概率密度函数：

$$p(s'|s, a) = \mathbb{P}(S' = s' | S = s, A = a).$$

意思是如果观测到当前状态 s 以及动作 a ，那么 p 函数输出状态变成 s' 的概率。本书中只考虑随机状态转移，因为确定状态转移是随机状态转移的一个特例：概率质量全部集中在一个状态 s' 上。

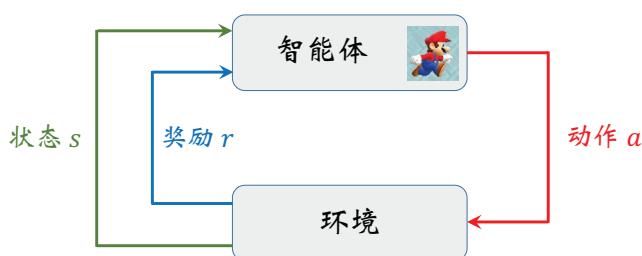


图 3.3：智能体与环境交互。

智能体与环境交互 (Agent Environment Interaction) 是指智能体观测到环境的状态 s ，做出动作 a ，动作会改变环境的状态，环境反馈给智能体奖励 r 以及新的状态 s' 。图 3.3 是智能体与环境交互的示意图。在超级玛丽的游戏中，智能体是玛丽奥，环境是游戏

程序。AI 以下面的方式控制玛丽奥跟游戏程序交互。观测到当前状态 s , AI 用策略函数 $\pi(a|s)$ 算出所有动作的概率, 比如算出

$$\pi(\text{左} | s) = 0.2, \quad \pi(\text{右} | s) = 0.1, \quad \pi(\text{上} | s) = 0.7.$$

按照概率做随机抽样, 得到其中一个动作 (比如向上), 记作 a , 然后玛丽奥执行这个动作。游戏程序会用状态转移函数 $p(s'|s, a)$ 随机生成新的状态 s' , 并反馈给玛丽奥一个奖励 r 。

3.2 随机性的来源

这一节的内容是强化学习中的随机性。随机性有两个来源：策略函数与状态转移函数。搞明白随机性的两个来源，对之后的学习很有帮助。

动作的随机性来自于策略函数。给定当前状态 s ，策略函数 $\pi(a|s)$ 会算出动作空间 \mathcal{A} 中每个动作 a 的概率值。智能体执行的动作是随机抽样的结果，所以带有随机性。见图 3.4 中的例子。

状态的随机性来自于状态转移函数。当状态 s 和动作 a 都被确定下来，下一个状态仍然有随机性。环境（比如游戏程序）用状态转移函数 $p(s'|s, a)$ 计算所有可能的状态的概率，然后做随机抽样，得到新的状态。见图 3.5 中的例子。

奖励可以看做状态和动作的函数。给定当前状态 s_t 和动作 a_t ，那么奖励 r_t 就是唯一确定的。假设给定当前状态 s_t ，但智能体尚未做决策，也就是说 t 时刻动作还未知，应当记作随机变量 A_t （而非 a_t ）；那么 t 时刻的奖励仍然未知，应当记作随机变量 R_t （而非 r_t ），它的随机性从未知的动作 $A_t \sim \pi(\cdot|s_t)$ 中来。

注在很多应用中，奖励 r_t 取决于 s_t, a_t, s_{t+1} 。在这种情况下，即使给定当前状态 s_t 和动作 a_t ，奖励 R_t 仍然是未知的变量，它的随机性从未知的新状态 $s_{t+1} \sim p(\cdot|s_t, a_t)$ 中来。

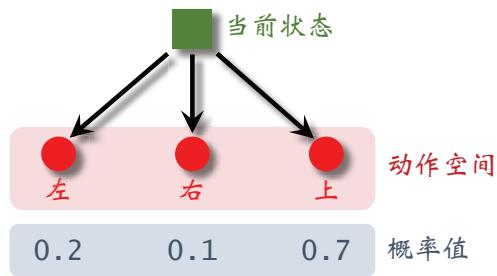


图 3.4：状态空间是 $\mathcal{A} = \{\text{左}, \text{中}, \text{右}\}$ 。把当前状态 s 输入策略函数，策略函数输出三个概率值：0.2, 0.1, 0.7。所以，对于确定的状态 s ，智能体执行的动作是不确定的，三个动作都可能被执行。

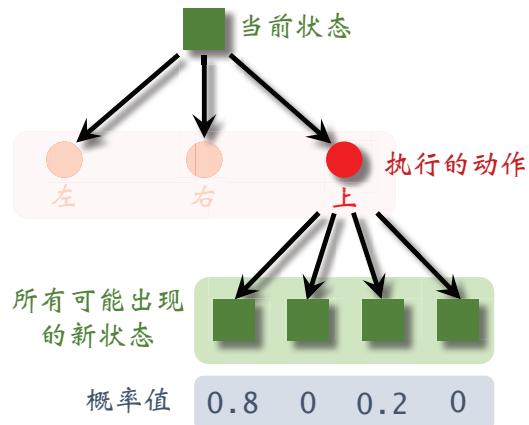


图 3.5：已知当前状态 s ，智能体已经做出决策——向上跳，那么环境会更新状态。环境把 s 和 a 输入状态转移函数，得到所有可能的状态的概率值。环境根据概率值做随机抽样，得到新的状态 s' 。

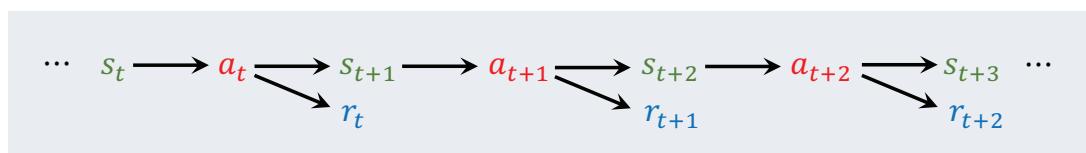


图 3.6：智能体的轨迹。

轨迹 (Trajectory) 是指一回合 (Episode) 游戏中，智能体观测到的所有的状态、动作、奖励：

$$s_1, a_1, r_1, s_2, a_2, r_2, s_3, a_3, r_3, \dots$$

图 3.6 描绘了轨迹中状态、动作、奖励的依赖关系。在 t 时刻，给定状态 $S_t = s_t$ ，下面

这些都是观测到的值：

$$s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_{t-1}, a_{t-1}, r_{t-1}, s_t,$$

而下面这些都是随机变量（尚未被观测到）：

$$A_t, R_t, S_{t+1}, A_{t+1}, R_{t+1}, S_{t+2}, A_{t+2}, R_{t+2}, \dots$$

3.3 回报与折扣回报

本节介绍回报 (Return) 和折扣回报 (Discounted Return) 这两个概念，并且讨论其随机性来源。由于回报是折扣率等于 1 的特殊折扣回报，后面的章节中用“回报”指代“折扣回报”，不再区分两者。

3.3.1 回报

回报 (Return) 是从当前时刻开始到一回合结束的所有奖励的总和，所以回报也叫做**累计奖励 (Cumulative Future Reward)**。把 t 时刻的回报记作随机变量 U_t ；如果一局游戏结束，已经观测到所有奖励，那么就把回报记作 u_t 。回报的定义是这样的：

$$U_t = R_t + R_{t+1} + R_{t+2} + R_{t+3} + \dots$$

回报有什么用呢？回报是未来获得的奖励总和，所以智能体的目标就是让回报尽量大，越大越好。强化学习的目标就是寻找一个策略，使得回报的期望最大化。

注 强化学习的目标是最大化**回报**，而不是最大化当前的**奖励**。打个比方，下棋的时候，你的目标是赢得一局比赛（回报），而非吃掉对方一个棋子（奖励）。

3.3.2 折扣回报

思考一个问题：在 t 时刻，请问奖励 r_t 和 r_{t+1} 同等重要吗？假如我给你两个选项：第一，现在我立刻给你 100 元钱；第二，等一年后我给你 100 元钱。你选哪个？理性人应该都会选现在得到 100 元钱。这是因为未来的不确定性很大，即使我现在答应明年给你 100 元，你也未必能拿到。大家都明白这个道理：明年得到 100 元不如现在立刻拿到 100 元。

要是换一个问题，现在我立刻给你 80 元钱，或者是明年我给你 100 元钱。你选哪一个？或许大家会做不同的选择，有的人愿意拿现在的 80，有的人愿意等一年拿 100。如果两种选择一样好，那么就意味着一年后的奖励的重要性只有今天的 $\gamma = 0.8$ 倍。这里的 $\gamma = 0.8$ 就是**折扣率 (Discount Factor)**。

同理，在强化学习中，通常使用**折扣回报 (Discounted Return)**，给未来的奖励做折扣。这是折扣回报的定义：

$$U_t = R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot R_{t+2} + \gamma^3 \cdot R_{t+3} + \dots$$

这里的 $\gamma \in [0, 1]$ 叫做折扣率。对待越久远的未来，给奖励打的折扣越大。折扣率是个超参数，需要手动调；折扣率的设置会影响强化学习的结果。

3.3.3 回报中的随机性

假设一回合游戏一共有 n 步。当完成这一回合之后，我们观测到所有 n 个奖励： r_1, r_2, \dots, r_n 。此时这些奖励不是随机变量，而是实际观测到的数值。此时我们可以实

际计算出折扣回报

$$u_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \cdots + \gamma^{n-t} \cdot r_n, \quad \forall t = 1, \dots, n.$$

此时的折扣回报 u_t 是实际观测到的数值，不具有随机性。

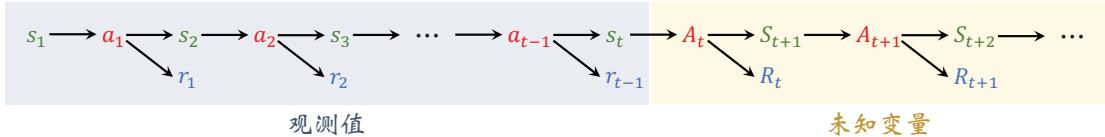


图 3.7: 智能体的轨迹中 s_t 及其之前的状态、动作、奖励都被观测到，而 A_t 及其之后的状态、动作、奖励都是未知变量。

假设我们此时在第 t 时刻，我们只观测到 s_t 及其之前的状态、动作、奖励

$$s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_{t-1}, a_{t-1}, r_{t-1}, s_t,$$

而下面这些都是随机变量（尚未被观测到）：

$$A_t, R_t, S_{t+1}, A_{t+1}, R_{t+1}, \dots, S_n, A_n, R_n.$$

见图 3.7。回报 U_t 依赖于奖励 R_t, R_{t+1}, \dots, R_n ，而这些奖励全都是未知的随机变量，所以 U_t 也是未知的随机变量。

请问回报 U_t 的随机性的来源是什么？奖励 R_t 依赖于状态 s_t （已观测到）与动作 A_t （未知变量），奖励 R_{t+1} 依赖于 S_{t+1} 和 A_{t+1} （未知变量），奖励 R_{t+2} 依赖于 S_{t+2} 和 A_{t+2} （未知变量），以此类推。所以 U_t 的随机性来自于这些动作和状态：

$$A_t, S_{t+1}, A_{t+1}, S_{t+2}, A_{t+2}, \dots, S_n, A_n.$$

动作的随机性来自于策略函数，状态的随机性来自于状态转移函数。

3.4 价值函数

本节介绍动作价值函数 $Q_\pi(s, a)$, 最优动作价值函数 $Q_*(s, a)$, 状态价值函数 $V_\pi(s)$ 。它们都是回报的期望。

3.4.1 动作价值函数

上一节介绍了（折扣）回报 U_t , 它是 t 时刻之后所有奖励的（加权）和。在 t 时刻，假如我们知道 U_t 的值，我们就知道游戏是快赢了还是快输了。然而在 t 时刻我们并不知道 U_t 的值，因为此时 U_t 仍然是个随机变量。在结束本回合游戏之前，我们都会不知道 U_t 的值。在 t 时刻，我们不知道 U_t 的值，而我们又想预判 U_t 的值从而知道局势的好坏。该怎么办呢？解决方案就是对 U_t 求期望，消除掉其中的随机性。

为什么求期望可以消除掉随机性呢？打个比方，抛硬币，正面记做 $X = 1$, 反面记做 $X = 0$ 。在抛硬币之前，并不知道随机变量 X 是 1 还是 0。如果对 X 求期望，可以消除掉随机性，得到一个具体的数值 $\mathbb{E}[X] = 0.5$ 。同理，对 U_t 求期望，就能得到一个具体的数值。

假设我们已经观测到状态 s_t , 而且做完决策，选中动作 a_t 。那么 U_t 中的随机性来自于 $t+1$ 时刻之后的状态和动作：

$$S_{t+1}, A_{t+1}, S_{t+2}, A_{t+2}, \dots, S_n, A_n.$$

对 U_t 关于变量 $S_{t+1}, A_{t+1}, \dots, S_n, A_n$ 求条件期望，得到

$$Q_\pi(s_t, a_t) = \mathbb{E}_{S_{t+1}, A_{t+1}, \dots, S_n, A_n} [U_t \mid S_t = s_t, A_t = a_t].$$

期望中的 $S_t = s_t$ 和 $A_t = a_t$ 是条件，意思是已经观测到 S_t 与 A_t 的值。条件期望的结果 $Q_\pi(s_t, a_t)$ 被称作**动作价值函数 (Action-Value Function)**。

动作价值函数 $Q_\pi(s_t, a_t)$ 依赖于 s_t 与 a_t ，而不依赖于 $t+1$ 时刻及其之后的状态和动作，这是因为随机变量 $S_{t+1}, A_{t+1}, \dots, S_n, A_n$ 都被期望消除了。从下面的公式中可以看出， $Q_\pi(s_t, a_t)$ 依赖于策略函数 $\pi(a|s)$ ：

$$\begin{aligned} Q_\pi(s_t, a_t) &= \mathbb{E}_{S_{t+1}, A_{t+1}, \dots, S_n, A_n} [U_t \mid S_t = s_t, A_t = a_t] \\ &= \int_S d s_{t+1} \int_A d a_{t+1} \cdots \int_S d s_n \int_A d a_n \underbrace{\left[\prod_{k=t+1}^n p(s_k \mid s_{k-1}, a_{k-1}) \cdot \pi(a_k \mid s_k) \right]}_{\text{概率密度函数}} \cdot U_t. \end{aligned}$$

公式中的 π 是动作的概率密度函数；用不同的 π ，连加结果就会不同。这就是为什么动作价值函数 Q_π 有下标 π 。综上所述， t 时刻的动作价值函数 $Q_\pi(s_t, a_t)$ 依赖于以下三个因素：

- 第一，当前状态 s_t 。当前状态越好，那么价值 $Q_\pi(s_t, a_t)$ 越大，也就是说回报的期望值越大。在超级玛丽的游戏中，如果玛丽奥当前已经接近终点，马上就能赢一局游戏，那么 $Q_\pi(s_t, a_t)$ 就非常大。
- 第二，当前动作 a_t 。智能体执行的动作越好，那么价值 $Q_\pi(s_t, a_t)$ 越大。举个例子，如果玛丽奥做正常的动作，那么 $Q_\pi(s_t, a_t)$ 就比较正常；如果玛丽奥的动作 a_t 是跳

下悬崖，那么 $Q_\pi(s_t, a_t)$ 就会非常小。

- 第三，策略函数 π 。策略决定未来的动作 $A_{t+1}, A_{t+2}, \dots, A_n$ 的好坏。策略越好，那么 $Q_\pi(s_t, a_t)$ 就越大。举个例子，顶级玩家相当于好的策略 π ；新手就相当于差的策略。让顶级玩家操作游戏，回报的期望非常高；换新手操作游戏，从相同的状态出发，回报的期望会很低。

3.4.2 最优动作价值函数

怎么样才能排除掉策略 π 的影响，只评价当前状态和动作的好坏呢？解决方案就是**最优动作价值函数 (Optimal Action-Value Function)**：

$$Q_*(s_t, a_t) = \max_{\pi} Q_{\pi}(s_t, a_t), \quad \forall s_t \in \mathcal{S}, \quad a_t \in \mathcal{A}.$$

公式的意思是有很多种策略函数 π 可供选择，而我们选择最好的策略函数：

$$\pi^* = \operatorname{argmax}_{\pi} Q_{\pi}(s_t, a_t), \quad \forall s_t \in \mathcal{S}, \quad a_t \in \mathcal{A}.$$

Q_* 和 Q_{π^*} 指的都是最优动作价值函数。 $Q_*(s_t, a_t)$ 只依赖于 s_t 和 a_t ，而与策略 π 无关。

最优动作价值函数 Q_* 非常有用：它就像是一个先知，能指引智能体做出正确决策。比如玩超级玛丽，给定当前状态 s_t ，智能体该执行动作空间 $\mathcal{A} = \{\text{左, 右, 上}\}$ 中的哪个动作呢？假设我们已知 Q_* 函数，那么我们就让 Q_* 给三个动作打分，比如：

$$Q_*(s_t, \text{左}) = 130, \quad Q_*(s_t, \text{右}) = -50, \quad Q_*(s_t, \text{上}) = 296.$$

这三个值是什么意思呢？ $Q_*(s_t, \text{左}) = 130$ 的意思是：如果现在智能体选择向左走，那么不管以后智能体用什么策略函数 π ，回报 U_t 的期望最多不会超过 130。同理，如果现在向右走，则回报的期望最多不超过 -50；如果现在向上跳，则回报的期望最多不超过 296。智能体应该执行哪个动作呢？毫无疑问，智能体当然应该向上跳，这样才能有希望获得尽量高的回报。

3.4.3 状态价值函数

假设 AI 用策略函数 π 下围棋。AI 想知道当前状态 s_t （即棋盘上的格局）是否对自己有利，以及自己和对手的胜算各有多大。该用什么来量化双方的胜算呢？答案是**状态价值函数 (State-Value Function)**：

$$\begin{aligned} V_{\pi}(s_t) &= \mathbb{E}_{A_t \sim \pi(\cdot|s_t)} [Q_{\pi}(s_t, A_t)] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s_t) \cdot Q_{\pi}(s_t, a). \end{aligned}$$

公式把动作 A_t 作为随机变量，关于 A_t 求期望，把 A_t 消掉。得到的状态价值函数 $V_{\pi}(s_t)$ 只依赖于策略 π 与当前状态 s_t ，不依赖于动作。状态价值函数 $V_{\pi}(s_t)$ 也是回报 U_t 的期望：

$$V_{\pi}(s_t) = \mathbb{E}_{A_t, S_{t+1}, A_{t+1}, \dots, S_n, A_n} [U_t \mid S_t = s_t].$$

期望消掉了 U_t 依赖的随机变量 $A_t, S_{t+1}, A_{t+1}, \dots, S_n, A_n$ 。状态价值 $V_{\pi}(s_t)$ 越大，就意味着回报 U_t 的期望越大。用状态价值可以衡量策略 π 与状态 s_t 的好坏。

3.5 策略学习和价值学习

假如我们想设计一种 AI，让它自动打超级玛丽游戏。AI 打游戏的目标是避开敌人、通过关卡、并收集尽量多的金币。我们需要自己来定义奖励，比如每个金币的奖励是 +1，通过一个关卡的奖励是 +1000，碰到敌人或落下悬崖的奖励是 -1000。AI 的目标是最大化（折扣）回报，也就是最大化奖励的（加权）总和。定义好了目标，就可以设计强化学习方法来实现目标。强化学习方法通常分为两类：**基于模型的方法 (Model-Based)** 和**无模型方法 (Model-Free)**，本书主要介绍后者。**无模型方法**又可以分为**价值学习**和**策略学习**。

价值学习 (Value-Based Learning) 通常是指学习最优价值函数 $Q_*(s, a)$ （或者动作价值函数、状态价值函数）。假如我们有了 Q_* ，智能体就可以根据 Q_* 来做决策，选出最好的动作。每次观测到一个状态 s_t ，把它输入 Q_* 函数，让 Q_* 对所有动作做评价，比如

$$Q_*(s_t, \text{左}) = 273, \quad Q_*(s_t, \text{右}) = -139, \quad Q_*(s_t, \text{上}) = 195.$$

这些 Q 值量化每个动作的好坏。智能体应该执行 Q 值最大的动作，也就是向左移动。这个动作预计能在未来获得最高不超过 273 的期望回报；而其他两个动作的期望回报不超过 -139 和 195。智能体的决策可以用这个公式表示：

$$a_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q_*(s_t, a).$$

如何去学习 Q_* 函数呢？我们需要用智能体收集到的状态、动作、奖励，用它们作为训练数据，学习一个表格或一个神经网络，用于近似 Q_* 。最有名的价值学习方法是深度 Q 网络 (DQN)，在后面章节中会详细介绍。

策略学习 (Policy-Based Learning) 指的是学习策略函数 $\pi(a|s)$ 。假如我们有了策略函数，我们就可以直接用它计算所有动作的概率值，然后随机抽样选出一个动作并执行。每次观测到一个状态 s_t ，把它输入 π 函数，让 π 对所有动作做评价，得到概率值：

$$\pi(\text{左} | s_t) = 0.6, \quad \pi(\text{右} | s_t) = 0.1, \quad \pi(\text{上} | s_t) = 0.3.$$

智能体做随机抽样，然后执行选中的动作。三个动作都有可能被选中。如何去学习策略 π 呢？本书后面的章节会介绍策略梯度等方法，用于学习 π 。

3.6 实验环境

如果你设计出一种新的强化学习方法，你应该将其与已有的标准方法做比较，看新的方法是否有优势。比较和评价强化学习算法最常用的是 OpenAI Gym，它相当于强化学习中的 ImageNet。Gym 有几大类控制问题，比如经典控制问题、Atari 游戏、机器人。

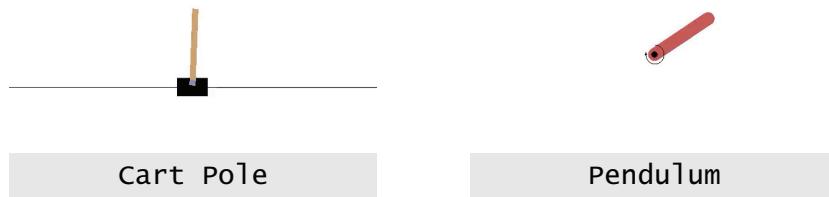


图 3.8：经典控制问题。

Gym 中第一类是经典控制问题，都是小规模的简单问题，比如 Cart Pole 和 Pendulum，见图 3.8。Cart Pole 要求给小车向左或向右的力，移动小车，让上面的杆子能竖起来。Pendulum 要求给钟摆一个力，让钟摆恰好能竖起来。

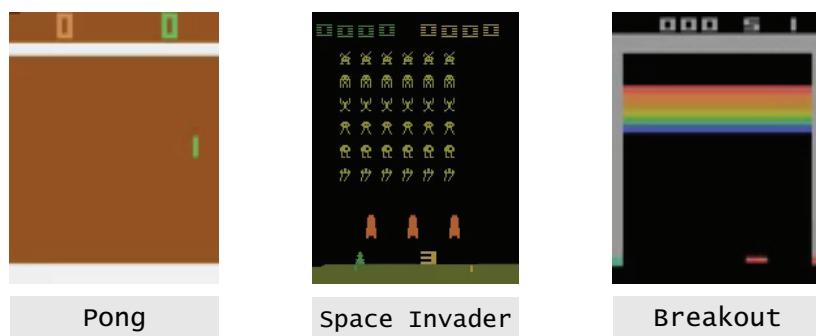


图 3.9：Atari 游戏。

第二类问题是 Atari 游戏，就是八、九十年代小霸王游戏机上拿手柄玩的那种游戏，见图 3.9。Pong 中的智能体是乒乓球拍，球拍可以上下运动，目标是接住对手的球，尽量让对手接不住球。Space Invader 中的智能体是小飞机，可以左右移动，可以发射炮弹。Breakout 中的智能体是下面的球拍，可以左右移动，目标是接住球，并且把上面的砖块都打掉。



图 3.10：机器人连续的控制问题，用到 MuJoCo 物理模拟器。

3.6 实验环境

第三类问题是机器人连续的控制问题，比如控制蚂蚁、人、猎豹等机器人走路，见图 3.10。这个模拟器叫做 MuJoCo，它可以模拟重力等物理量。机器人是智能体，AI 需要控制这些机器人站立和走路。MuJoCo 是付费软件，但是可以申请免费试用 license。

想要使用 Gym，应该先按照官方文档安装 <https://gym.openai.com/>。安装之后就可以在 Python 里面调用 Gym 库中的函数了。下面的程序以 Cart Pole 这个控制任务为例，说明怎么样使用 Gym 标准库。通过阅读这段程序，读者可以更好理解智能体与环境的交互。

```
import gym
env = gym.make('CartPole-v0')           | 生成环境。此处的环境是CartPole游戏程序。
state = env.reset()                     | 重置环境，让小车回到起点。并输出初始状态。
for t in range(100):                   | 弹出窗口，把游戏中发生的显示到屏幕上。
    env.render()
    print(state)
    action = env.action_space.sample()   | 方便起见，此处均匀抽样生成一个动作。在实际应用中，应当依据状态，用策略函数生成动作。
    state, reward, done, info = env.step(action) | 智能体真正执行动作。
                                                | done等于1意味着游戏结束；       | done等于0意味着游戏继续。
                                                | 然后环境更新状态，          |
                                                | 并反馈一个奖励。
    if done:
        print('Finished')
        break
env.close()
```

《深度强化学习》2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

第四章 DQN 与 Q 学习

本章的内容是价值学习的基础。第 4.1 节用神经网络近似最优动作价值函数 $Q^*(s, a)$ ，把这个神经网络称为深度 Q 网络 (DQN)。本章内容的难点在于训练 DQN 所用的时间差分算法 (TD)。第 4.2 节以“驾车时间估计”类比 DQN，讲解 TD 算法。第 4.3 节推导训练 DQN 用的 Q 学习算法；Q 学习属于 TD 算法的一种)。第 4.4 节介绍表格形式的 Q 学习算法。第 4.5 节解释同策略 (On-policy) 与异策略 (Off-policy) 的区别；本章介绍的 Q 学习算法属于异策略。

4.1 DQN

在学习 DQN 之前，首先复习一些基础知识。在一局 (Episode) 游戏中，把从起始到结束的所有奖励记作：

$$R_1, \dots, R_t, \dots, R_n.$$

定义折扣率 $\gamma \in [0, 1]$ 。折扣回报的定义是：

$$U_t = R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot R_{t+2} + \dots + \gamma^{n-t} \cdot R_n.$$

在游戏尚未结束的 t 时刻， U_t 是一个未知的随机变量，其随机性来自于 t 时刻之后的所有状态与动作。动作价值函数的定义是：

$$Q_\pi(s_t, a_t) = \mathbb{E}[U_t \mid S_t = s_t, A_t = a_t],$$

公式中的期望消除了 t 时刻之后的所有状态 S_{t+1}, \dots, S_n 与所有动作 A_{t+1}, \dots, A_n 。最优动作价值函数用最大化消除策略 π ：

$$Q_*(s_t, a_t) = \max_{\pi} Q_{\pi}(s_t, a_t), \quad \forall s_t \in \mathcal{S}, \quad a_t \in \mathcal{A}.$$

可以这样理解 Q_* ：已知 s_t 和 a_t ，不论未来采取什么样的策略 π ，回报 U_t 的期望不可能超过 Q_* 。

最优动作价值函数的用途：假如我们知道 Q_* ，我们就能用它做控制。举个例子，超级玛丽游戏中的动作空间是 $\mathcal{A} = \{\text{左}, \text{右}, \text{上}\}$ 。给定当前状态 s_t ，智能体该执行哪个动作呢？假设我们已知 Q_* 函数，那么我们就让 Q_* 给三个动作打分，比如：

$$Q_*(s_t, \text{左}) = 370, \quad Q_*(s_t, \text{右}) = -21, \quad Q_*(s_t, \text{上}) = 610.$$

这三个值是什么意思呢？ $Q_*(s_t, \text{左}) = 370$ 的意思是：如果现在智能体选择向左走，不论之后采取什么策略 π ，那么回报 U_t 的期望最多不会超过 370。同理，其他两个最优动作价值的也是回报的期望的上界。根据 Q_* 的评分，智能体应该选择向上跳，因为这样可以最大化回报 U_t 的期望。

我们希望知道 Q_* ，因为它就像是先知一般，可以预见未来，在 t 时刻就预见 t 到 n 时刻之间的累计奖励的期望。假如我们有 Q_* 这位先知，我们就遵照按照先知的指导，最大化未来的累计奖励。然而在实践中我们无法得到 Q_* 的函数表达式。是否有可能近似

出 Q_* 这位先知呢？对于超级玛丽这样的游戏，学出来一个“先知”并不难。假如让我重复玩超级玛丽一亿次，那我就像是先知一样：告诉我当前状态，我能准确判断出当前最优的动作是什么。这说明只要有足够多的“经验”，就能训练出超级玛丽中的“先知”。

最优动作价值函数的近似：在实践中，近似学习“先知” Q_* 最有效的方法是深度 Q 网络 (Deep Q Network)，缩写是 DQN，记作 $Q(s, a; \mathbf{w})$ ，其结构在图 4.1 中描述。其中的 \mathbf{w} 表示神经网络中的参数；一开始随机初始化 \mathbf{w} ，随后用“经验”去学习 \mathbf{w} 。学习的目标是：对于所有的 s 和 a ，DQN 的预测 $Q(s, a; \mathbf{w})$ 尽量接近 $Q_*(s, a)$ 。本章之后几节的内容都是如何学习 \mathbf{w} 。

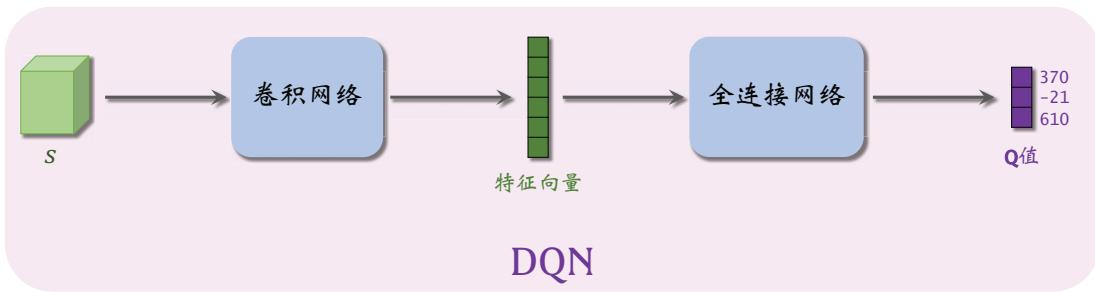


图 4.1: DQN 的神经网络结构。输入是状态 s ；输出是每个动作的 Q 值。。

可以这样理解 DQN 的表达式 $Q(s, a; \mathbf{w})$ 。DQN 的输出是离散动作空间 \mathcal{A} 上的每个动作的 Q 值，即给每个动作的评分，分数越高，动作越好。举个例子，动作空间是 $\mathcal{A} = \{\text{左, 右, 上}\}$ ，那么动作空间的大小等于 $|\mathcal{A}| = 3$ ，DQN 的输出是 3 维的向量 $\hat{\mathbf{q}}$ ，向量每个元素对应一个动作。在图 4.1 中，DQN 的输出是

$$\begin{aligned}\hat{q}_1 &= Q(s, \text{左}; \mathbf{w}) = 370, \\ \hat{q}_2 &= Q(s, \text{右}; \mathbf{w}) = -21, \\ \hat{q}_3 &= Q(s, \text{上}; \mathbf{w}) = 610.\end{aligned}$$

总结一下，DQN 的输出是 $|\mathcal{A}|$ 维的向量 $\hat{\mathbf{q}}$ ，包含所有动作的价值。而我们常用的符号 $Q(s, a; \mathbf{w})$ 是标量，是动作 a 对应的动作价值，是向量 $\hat{\mathbf{q}}$ 中的一个元素。

DQN 的梯度：在训练 DQN 的时候，需要对 DQN 关于神经网络参数 \mathbf{w} 求梯度。用

$$\nabla_{\mathbf{w}} Q(s, a; \mathbf{w}) \triangleq \frac{\partial Q(s, a; \mathbf{w})}{\partial \mathbf{w}}$$

表示函数值 $Q(s, a; \mathbf{w})$ 关于参数 \mathbf{w} 的梯度。因为函数值 $Q(s, a; \mathbf{w})$ 是一个实数，所以梯度的形状与 \mathbf{w} 完全相同：如果 \mathbf{w} 是 $d \times 1$ 的向量，那么梯度也是 $d \times 1$ 的向量；如果 \mathbf{w} 是 $d_1 \times d_2$ 的矩阵，那么梯度也是 $d_1 \times d_2$ 的矩阵；如果 \mathbf{w} 是 $d_1 \times d_2 \times d_3$ 的张量，那么梯度也是 $d_1 \times d_2 \times d_3$ 的张量。

给定观测值 s 和 a ，比如 $a = \text{“左”}$ ，可以用反向传播计算出梯度 $\nabla_{\mathbf{w}} Q(s, \text{左}; \mathbf{w})$ 。在编程实现的时候，TensorFlow 和 PyTorch 可以对 DQN 输出向量的一个元素，比如 $Q(s, \text{左}; \mathbf{w})$ ，关于变量 \mathbf{w} 自动求梯度，得到的梯度的形状与 \mathbf{w} 完全相同。

4.2 时间差分 (TD) 算法

训练 DQN 最常用的算法是时间差分 (Temporal Difference), 缩写 TD。TD 算法不太好理解, 所以本节举一个通俗易懂的例子讲解 TD 算法。

4.2.1 驾车时间预测的例子

假设我们有一个模型 $Q(s, d; \mathbf{w})$, 其中 s 是起点, d 是终点, \mathbf{w} 是参数。模型 Q 可以预测开车出行的时间开销。这个模型一开始不准确, 甚至是纯随机的。但是随着很多人用这个模型, 得到更多数据、更多训练, 这个模型就会越来越准, 会像谷歌地图一样准。

我们该如何训练这个模型呢? 在用户出发前, 用户告诉模型起点 s 和终点 d , 模型做一个预测 $\hat{q} = Q(s, d; \mathbf{w})$ 。当用户结束行程的时候, 把实际驾车时间 y 反馈给模型。两者之差 $\hat{q} - y$ 反映出模型是高估还是低估了驾驶时间, 以此来修正模型, 使得模型的估计更准确。

假设我是个用户, 我要从北京驾车去上海。从北京出发之前, 我让模型做预测, 模型告诉我总车程是 14 小时:

$$\hat{q} \triangleq Q(\text{“北京”}, \text{“上海”}; \mathbf{w}) = 14.$$

当我到达上海, 我知道自己花的实际时间是 16 小时, 并将结果反馈给模型; 见图 4.2。

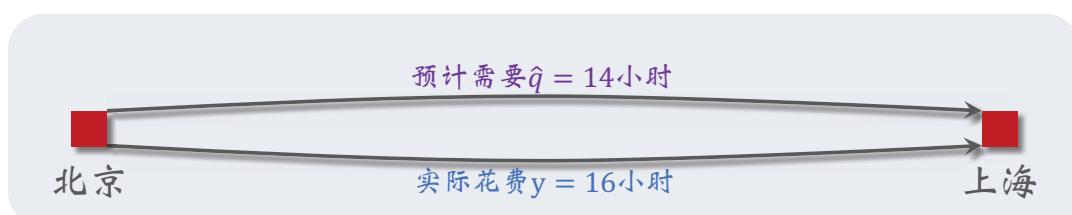


图 4.2: 模型估计驾驶时间是 $\hat{q} = 14$, 而实际花费时间 $y = 16$ 。

可以用梯度下降对模型做一次更新, 具体做法如下。把我的这次旅程作为一组训练数据:

$$s = \text{“北京”}, \quad d = \text{“上海”}, \quad \hat{q} = 14, \quad y = 16.$$

我们希望估计值 $\hat{q} = Q(s, d; \mathbf{w})$ 尽量接近真实观测到的 y , 所以用两者的平方差作为损失函数:

$$L(\mathbf{w}) = \frac{1}{2} [Q(s, d; \mathbf{w}) - y]^2.$$

用链式法则计算损失函数的梯度, 得到:

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = (\hat{q} - y) \cdot \nabla_{\mathbf{w}} Q(s, d; \mathbf{w}),$$

然后做一次梯度下降更新模型参数 \mathbf{w} :

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \nabla_{\mathbf{w}} L(\mathbf{w}),$$

此处的 α 是学习率, 需要手动调。在完成一次梯度下降之后, 如果再让模型做一次预测,

那么模型的预测值

$$Q(\text{“北京”}, \text{“上海”}; \mathbf{w})$$

会比原先更接近 $y = 16$.

4.2.2 TD 算法

接着上文驾车时间的例子。出发前模型估计全程时间为 $\hat{q} = 14$ 小时；模型建议的路线会途径济南。我从北京出发，过了 $r = 4.5$ 小时，我到达济南。此时我再让模型做一次预测，模型告诉我

$$\hat{q}' \triangleq Q(\text{“济南”}, \text{“上海”}; \mathbf{w}) = 11.$$

见图 4.3 的描述。假如此时我的车坏了，必须要在济南修理，我不得不取消此次行程。我没有完成旅途，那么我的这组数据是否能帮助训练模型呢？其实是可以的，用到的算法叫做时间差分 (Temporal Difference)，缩写为 TD。

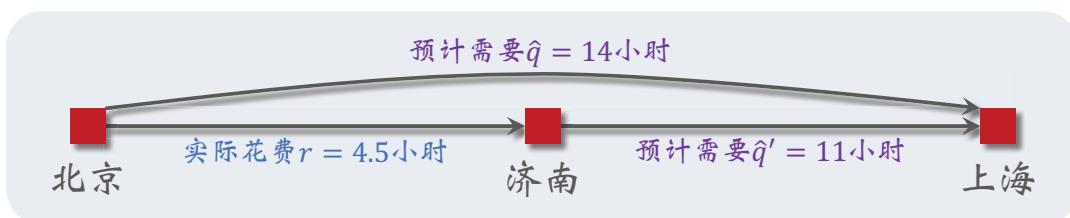


图 4.3: 紫色的数字 $\hat{q} = 14$ 和 $\hat{q}' = 11$ 是模型的估计值；蓝色的数字 $r = 4.5$ 是实际观测值。

下面解释 TD 算法的原理。回顾一下我们已有的数据：模型估计从北京到上海一共需要 $\hat{q} = 14$ 小时，我实际用了 $r = 4.5$ 小时到达济南，模型估计还需要 $\hat{q}' = 11$ 小时从济南到上海。到达济南时，根据模型最新估计，整个旅程的总时间为：

$$\hat{y} \triangleq r + \hat{q}' = 4.5 + 11 = 15.5.$$

TD 算法将 $\hat{y} = 15.5$ 称为 **TD 目标 (TD Target)**，它比最初的预测 $\hat{q} = 14$ 更可靠。最初的预测 $\hat{q} = 14$ 纯粹是估计的，没有任何事实的成分。TD 目标 $\hat{y} = 15.5$ 也是个估计，但其中有事实的成分：其中的 $r = 4.5$ 就是实际的观测。

基于以上讨论，我们认为 TD 目标 $\hat{y} = 15.5$ 比模型最初的估计值

$$\hat{q} = Q(\text{“北京”}, \text{“上海”}; \mathbf{w}) = 14$$

更可靠，所以可以用 \hat{y} 对模型做“修正”。我们希望估计值 \hat{q} 尽量接近 TD 目标 \hat{y} ，所以用两者的平方差作为损失函数：

$$L(\mathbf{w}) = \frac{1}{2} [Q(\text{“北京”}, \text{“上海”}; \mathbf{w}) - \hat{y}]^2.$$

4.2 时间差分 (TD) 算法

把 \hat{y} 看做常数，尽管它依赖于 w 。¹ 计算损失函数的梯度：

$$\nabla_w L(w) = \underbrace{(\hat{q} - \hat{y})}_{\text{记作 } \delta} \cdot \nabla_w Q(\text{“北京”, “上海”}; w),$$

此处的 $\delta = \hat{q} - \hat{y}$ 称作 **TD 误差 (TD Error)**。做一次梯度下降更新模型参数 w ：

$$w \leftarrow w - \alpha \cdot \delta \cdot \nabla_w Q(\text{“北京”, “上海”}; w).$$

TD 算法指的是用此公式更新模型参数 w 。

如果你仍然不理解 TD 算法，那么请换个角度来思考问题。模型估计从北京到上海全程需要 $\hat{q} = 14$ 小时，模型还估计从济南到上海需要 $\hat{q}' = 11$ 小时。这就相当于模型做了这样的估计：从北京到济南需要的时间为

$$\hat{q} - \hat{q}' = 14 - 11 = 3.$$

而我真实花费 $r = 4.5$ 小时从北京到济南。模型的估计与我的真实观测之差为

$$\delta = 3 - 4.5 = -1.5.$$

这就是 TD 误差！以上分析说明 TD 误差 δ 就是模型估计与真实观测之差。TD 算法的目的是通过更新参数 w 使得目标函数 $L(w) = \frac{1}{2}\delta^2$ 减小。

¹根据定义，TD 目标是 $\hat{y} = r + \hat{q}'$ ，其中 $\hat{q}' = Q(\text{“济南”, “上海”}; w)$ 依赖于 w 。因此， \hat{y} 其实是 w 的函数。然而 TD 算法忽视这一点，在求梯度的时候，将 \hat{y} 视为常数，而非 w 的函数。

4.3 用 TD 训练 DQN

上一节以驾车时间预测为例介绍了 TD 算法。本节用 TD 算法训练 DQN。第 4.3.1 小节推导算法，第 4.3.2 详细描述训练 DQN 的流程。注意，本节推导出的是最原始的 TD 算法，在实践中效果不佳。实际训练 DQN 的时候，应当使用第 6 章介绍的高级技巧。

4.3.1 算法推导

下面我们推导训练 DQN 的 TD 算法。² 回忆一下回报的定义： $U_t = \sum_{k=t}^n \gamma^{k-t} \cdot R_k$ 。由这个定义可得：

$$U_t = R_t + \gamma \cdot \underbrace{\sum_{k=t+1}^n \gamma^{k-t} \cdot R_k}_{= U_{t+1}}.$$

回忆一下，最优动作价值函数可以写成

$$Q_*(s_t, a_t) = \max_{\pi} \mathbb{E}[U_t \mid S_t = s_t, A_t = a_t].$$

从上面两个公式出发，经过一系列数学推导（见附录 A），可以得到下面的定理。这个定理是**最优贝尔曼方程 (Optimal Bellman Equations)**的一种形式。

定理 4.1. 最优贝尔曼方程

$$\underbrace{Q_*(s_t, a_t)}_{U_t \text{ 的期望}} = \mathbb{E}_{S_{t+1} \sim p(\cdot | s_t, a_t)} \left[R_t + \gamma \cdot \underbrace{\max_{A \in \mathcal{A}} Q_*(S_{t+1}, A)}_{U_{t+1} \text{ 的期望}} \mid S_t = s_t, A_t = a_t \right].$$



贝尔曼方程的右边是个期望，我们可以对期望做蒙特卡洛近似。当智能体执行动作 a_t 之后，环境通过状态转移函数 $p(s_{t+1}|s_t, a_t)$ 计算出新状态 s_{t+1} ，并将其反馈给智能体。奖励 R_t 最多只依赖于 S_t 、 A_t 、 S_{t+1} 。那么当我们观测到 s_t 、 a_t 、 s_{t+1} ，则奖励 R_t 也被观测到，记作 r_t 。有了四元组

$$(s_t, a_t, r_t, s_{t+1}),$$

我们就有了贝尔曼方程右边期望的一个蒙特卡洛近似，得到：

$$Q_*(s_t, a_t) \approx r_t + \gamma \cdot \max_{a \in \mathcal{A}} Q_*(s_{t+1}, a). \quad (4.1)$$

这是不是很像驾驶时间预测问题？左边的 $Q_*(s_t, a_t)$ 就像是模型预测“北京到上海”的总时间， r_t 像是实际观测的“北京到济南”的时间， $\gamma \cdot \max_{a \in \mathcal{A}} Q_*(s_{t+1}, a)$ 相当于模型预测剩余路程“济南到上海”的时间。见图 4.4 中的类比。

把公式 4.1 中的最优动作价值函数 $Q_*(s_t, a_t)$ 替换成神经网络 $Q(s_t, a_t; \mathbf{w})$ ，得到：

$$\underbrace{Q(s_t, a_t; \mathbf{w})}_{\text{预测 } \hat{q}_t} \approx \underbrace{r_t + \gamma \cdot \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \mathbf{w})}_{\text{TD 目标 } \hat{y}_t}.$$

²严格地讲，此处推导的是“Q 学习算法”，它属于 TD 算法的一种。本节就称其为 TD 算法；下一节再具体介绍 Q 学习算法。

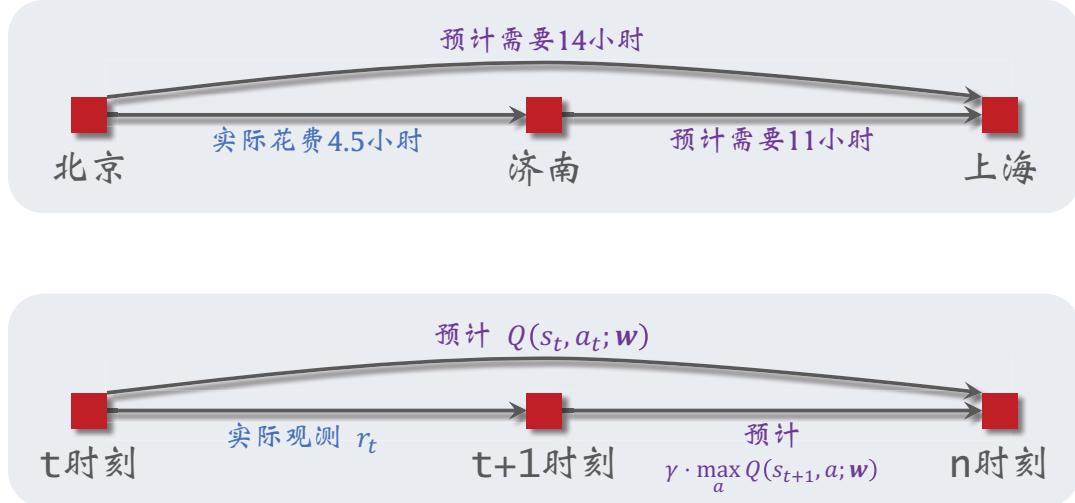


图 4.4: 用“驾车时间”类比 DQN。

左边的 $\hat{q}_t \triangleq Q(s_t, a_t; \mathbf{w})$ 是神经网络在 t 时刻做出的预测，其中没有任何事实成分。右边的 TD 目标 \hat{y}_t 是神经网络在 $t+1$ 时刻做出的预测，它部分基于真实观测到的奖励 r_t 。 \hat{q}_t 和 \hat{y}_t 两者都是对最优动作价值 $Q_*(s_t, a_t)$ 的估计，但是 \hat{y}_t 部分基于事实，因此比 \hat{q}_t 更可信。应当鼓励 $\hat{q}_t \triangleq Q(s_t, a_t; \mathbf{w})$ 接近 \hat{y}_t 。定义损失函数：

$$L(\mathbf{w}) = \frac{1}{2} [Q(s_t, a_t; \mathbf{w}) - \hat{y}_t]^2.$$

假装 \hat{y} 是常数³，计算 L 关于 \mathbf{w} 的梯度：

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \underbrace{(\hat{q}_t - \hat{y}_t)}_{\text{TD 误差 } \delta_t} \cdot \nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w}).$$

做一步梯度下降，可以让 \hat{q}_t 更接近 \hat{y}_t ：

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w}).$$

这个公式就是训练 DQN 的 TD 算法。

4.3.2 训练流程

首先总结上面的结论。给定一个四元组 (s_t, a_t, r_t, s_{t+1}) ，我们可以计算出 DQN 的预测值

$$\hat{q}_t = Q(s_t, a_t; \mathbf{w}),$$

以及 TD 目标和 TD 误差：

$$\hat{y}_t = r_t + \gamma \cdot \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \mathbf{w}) \quad \text{和} \quad \delta_t = \hat{q}_t - \hat{y}_t.$$

TD 算法用这个公式更新 DQN 的参数：

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w}).$$

³实际上 \hat{y}_t 依赖于 \mathbf{w} ，但是我们假装 \hat{y} 是常数。

注意，算法所需数据为 (s_t, a_t, r_t, s_{t+1}) 这个四元组，与控制智能体运动的策略 π 无关。这就意味着可以用任何策略控制智能体，同时记录下算法运动轨迹，作为 DQN 的训练数据。因此，DQN 的训练可以分割成两个独立的部分：收集训练数据、更新参数 w 。

收集训练数据：我们可以用任何策略函数 π 去控制智能体与环境交互，这个 π 就叫做**行为策略** (Behavior Policy)。比较常用的是 ϵ -greedy 策略：

$$a_t = \begin{cases} \operatorname{argmax}_a Q(s_t, a; w), & \text{以概率 } (1 - \epsilon); \\ \text{均匀抽取 } \mathcal{A} \text{ 中的一个动作,} & \text{以概率 } \epsilon. \end{cases}$$

把智能体在一局游戏中的轨迹记作：

$$s_1, a_1, r_1, s_2, a_2, r_2, \dots s_n, a_n, r_n.$$

把一条轨迹划分成 n 个 (s_t, a_t, r_t, s_{t+1}) 这种四元组，存入数组，这个数组叫做**经验回放数组** (Replay Buffer)。

更新 DQN 参数 w ：随机从经验回放数组中取出一个四元组，记作 (s_j, a_j, r_j, s_{j+1}) 。设 DQN 当前的参数为 w_{now} ，执行下面的步骤对参数做一次更新，得到新的参数 w_{new} 。

1. 对 DQN 做正向传播，得到 Q 值：

$$\hat{q}_j = Q(s_j, a_j; w_{\text{now}}) \quad \text{和} \quad \hat{q}_{j+1} = \max_{a \in \mathcal{A}} Q(s_{j+1}, a; w_{\text{now}}).$$

2. 计算 TD 目标和 TD 误差：

$$\hat{y}_j = r_j + \gamma \cdot \hat{q}_{j+1} \quad \text{和} \quad \delta_j = \hat{q}_j - \hat{y}_j.$$

3. 对 DQN 做反向传播，得到梯度：

$$\mathbf{g}_j = \nabla_w Q(s_j, a_j; w_{\text{now}}).$$

4. 做梯度下降更新 DQN 的参数：

$$w_{\text{new}} \leftarrow w_{\text{now}} - \alpha \cdot \delta_j \cdot \mathbf{g}_j.$$

智能体收集数据、更新 DQN 参数这两者可以同时进行。可以在智能体每执行一个动作之后，对 w 做几次更新。也可以在每完成一局游戏之后，对 w 做几次更新。

4.4 Q 学习算法

上一节用 TD 算法训练 DQN；准确地说，我们用的 TD 算法叫做 Q 学习算法 (Q-learning)。TD 算法是一大类算法，常见的有 Q 学习和 SARSA。Q 学习的目是学到最优动作价值函数 Q_* ；而 SARSA 的目的是学习动作价值函数 Q_π 。下一章会介绍 SARSA 算法。

Q 学习是在 1989 年提出的，而 DQN 则是 2013 年才提出。从 DQN 的名字（深度 Q 网络）就能看出 DQN 与 Q 学习的联系。最初的 Q 学习都是以表格形式出现的。表格形式的 Q 学习在实践中不常用，还是建议读者有所了解。

用表格表示 Q_* ：假设状态空间 \mathcal{S} 和动作空间 \mathcal{A} 都是有限集，即集合中元素数量有限。⁴ 比如， \mathcal{S} 中一共有 3 种状态， \mathcal{A} 中一共有 4 种动作。那么最优动作价值函数 $Q_*(s, a)$ 可以表示为一个 3×4 的表格，比如右边的表格。基于当前状态 s_t ，做决策时使用的公式

	第 1 种 动作	第 2 种 动作	第 3 种 动作	第 4 种 动作
第 1 种 状态	380	-95	20	173
第 2 种 状态	-7	64	-195	210
第 3 种 状态	152	72	413	-80

图 4.5：最优动作价值函数 Q_* 表示成表格形式。

$$a_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q_*(s_t, a)$$

的意思是找到 s_t 对应的行（3 行中的某一行），找到该行最大的价值，返回该元素对应的动作。举个例子，当前状态 s_t 是第 2 种状态，那么我们查看第 2 行，发现该行最大的价值是 210，对应第 4 种动作。那么应当执行的动作 a_t 就是第 4 种动作。

该如何通过智能体的轨迹来学习这样一个表格呢？用一个表格 \tilde{Q} 来近似 Q_* 。首先初始化 \tilde{Q} ，可以让它是全零的表格。然后用表格形式的 Q 学习算法更新 \tilde{Q} ，每次更新表格的一个元素。最终 \tilde{Q} 会收敛到 Q^* 。

算法推导：首先复习一下最优贝尔曼方程：

$$Q_*(s_t, a_t) = \mathbb{E}_{S_{t+1} \sim p(\cdot | s_t, a_t)} [R_t + \gamma \cdot \max_{A \in \mathcal{A}} Q_*(S_{t+1}, A) \mid S_t = s_t, A_t = a_t].$$

我们对方程左右两边做近似：

- 方程左边的 $Q_*(s_t, a_t)$ 可以近似成 $\tilde{Q}(s_t, a_t)$ 。 $\tilde{Q}(s_t, a_t)$ 是表格在 t 时刻对 $Q_*(s_t, a_t)$ 做出的估计。
- 方程右边的期望是关于下一时刻状态 S_{t+1} 求的。给定当前状态 s_t ，智能体执行动作 a_t ，环境会给出奖励 r_t 和新的状态 s_{t+1} 。用观测到的 r_t 和 s_{t+1} 对期望做蒙特卡洛近似，得到：

$$r_t + \gamma \cdot \max_{a \in \mathcal{A}} Q_*(s_{t+1}, a). \quad (4.2)$$

⁴如果 \mathcal{A} 是有限集，而 \mathcal{S} 是无限集，那么我们可以用神经网络形式的 Q 学习，即上一节的 DQN。如果 \mathcal{A} 是无限集，则问题属于连续控制，应当使用连续控制的方法，见第 10 章。

- 进一步把公式 (4.2) 中的 Q_* 近似成 \tilde{Q} , 得到

$$\hat{y}_t \triangleq r_t + \gamma \cdot \max_{a \in \mathcal{A}} \tilde{Q}(s_{t+1}, a).$$

把它称作 TD 目标。它是表格在 $t+1$ 时刻对 $Q_*(s_t, a_t)$ 做出的估计。

$\tilde{Q}(s_t, a_t)$ 和 \hat{y}_t 都是对最优动作价值 $Q_*(s_t, a_t)$ 的估计。由于 \hat{y}_t 部分基于真实观测到的奖励 r_t , 我们认为 \hat{y}_t 是更可靠的估计, 所以鼓励 $\tilde{Q}(s_t, a_t)$ 更接近 \hat{y}_t 。更新表格 \tilde{Q} 中 (s_t, a_t) 位置上的元素:

$$\tilde{Q}(s_t, a_t) \leftarrow (1 - \alpha) \cdot \tilde{Q}(s_t, a_t) + \alpha \cdot \hat{y}_t.$$

这样可以使得 $\tilde{Q}(s_t, a_t)$ 更接近 \hat{y}_t 。Q 学习的目的是让 \tilde{Q} 逐渐趋近于 Q_* 。

收集训练数据: Q 学习更新 \tilde{Q} 的公式不依赖于具体的策略。我们可以用任意策略控制智能体, 与环境交互, 把得到的轨迹划分成 (s_t, a_t, r_t, s_{t+1}) 这样的四元组, 存入经验回放数组。这个控制智能体的策略叫做行为策略 (Behavior Policy), 比较常用的行为策略是 ϵ -greedy:

$$a_t = \begin{cases} \operatorname{argmax}_a \tilde{Q}(s_t, a), & \text{以概率 } (1 - \epsilon); \\ \text{均匀抽取 } \mathcal{A} \text{ 中的一个动作,} & \text{以概率 } \epsilon. \end{cases}$$

事后用经验回放更新表格 \tilde{Q} , 可以重复利用收集到的四元组。

经验回放更新表格 \tilde{Q} : 随机从经验回放数组中抽取一个四元组, 记作 (s_j, a_j, r_j, s_{j+1}) 。设当前表格为 \tilde{Q}_{now} 。更新表格中 (s_j, a_j) 位置上的元素, 把更新之后的表格记作 \tilde{Q}_{new} 。

1. 把表格 \tilde{Q}_{now} 中第 (s_j, a_j) 位置上的元素记作:

$$\hat{q}_j = \tilde{Q}_{\text{now}}(s_j, a_j).$$

2. 查看表格 \tilde{Q}_{now} 的第 s_{j+1} 行, 把该行的最大值记作:

$$\hat{q}_{j+1} = \max_a \tilde{Q}_{\text{now}}(s_{j+1}, a).$$

3. 计算 TD 目标和 TD 误差:

$$\hat{y}_j = r_j + \gamma \cdot \hat{q}_{j+1}, \quad \delta_j = \hat{q}_j - \hat{y}_j.$$

4. 更新表格中 (s_j, a_j) 位置上的元素:

$$\tilde{Q}_{\text{new}}(s_j, a_j) \leftarrow \tilde{Q}_{\text{now}}(s_j, a_j) - \alpha \cdot \delta_j.$$

收集经验与更新表格 \tilde{Q} 可以同时进行。每当智能体执行一次动作, 我们可以用经验回放对 \tilde{Q} 做几次更新。也可以当完成一局游戏, 对 \tilde{Q} 做几次更新。

4.5 同策略 (On-policy) 与异策略 (Off-policy)

在强化学习中经常会遇到两个专业术语：同策略 (On-policy) 和异策略 (Off-policy)。为了解释同策略和异策略，我们要从行为策略 (Behavior Policy) 和目标策略 (Target Policy) 讲起。

在强化学习中，我们让智能体与环境交互，记录下观测到的状态、动作、奖励，用这些经验来学习一个策略函数。在这一过程中，控制智能体与环境交互的策略被称作行为策略。行为策略的作用是收集经验 (Experience)，即观测的环境、动作、奖励。

训练的目的是得到一个策略函数，在结束训练之后，用这个策略函数来控制智能体；这个策略函数就叫做目标策略。在本章中，目标策略是一个确定性的策略，即用 DQN 控制智能体：

$$a_t = \underset{a}{\operatorname{argmax}} Q(s_t, a; \mathbf{w}).$$

本章的 Q 学习算法用任意的行为策略收集 (s_t, a_t, r_t, s_{t+1}) 这样的四元组，然后拿它们训练目标策略，即 DQN。

行为策略和目标策略可以相同，也可以不同。同策略是指用相同的 行为策略 和 目标策略；我们暂时还没有学到同策略。异策略是指用不同的 行为策略 和 目标策略；本章的 DQN 是异策略。同策略和异策略如图 4.6、4.7 所示。

由于 DQN 是异策略，行为策略可以不同于目标策略，可以用任意的行为策略收集经验，比如最常用的 行为策略 是 ϵ -greedy：

$$a_t = \begin{cases} \underset{a}{\operatorname{argmax}} Q(s_t, a; \mathbf{w}), & \text{以概率 } (1 - \epsilon); \\ \text{均匀抽取 } \mathcal{A} \text{ 中的一个动作,} & \text{以概率 } \epsilon. \end{cases}$$

让行为策略带有随机性的好处在于能探索更多没见过的状态。

异策略的好处是可以用 行为策略 收集经验，把 (s_t, a_t, r_t, s_{t+1}) 这样的四元组记录到一个数组里，在事后反复利用这些经验去更新目标策略。这个数组被称作 经验回放数组 (Replay Buffer)，这种训练方式被称作 经验回放 (Experience Replay)。注意，经验回放只适用于异策略，不适用于同策略，其原因是收集经验时用的 行为策略 不同于想要训练出的目标策略。

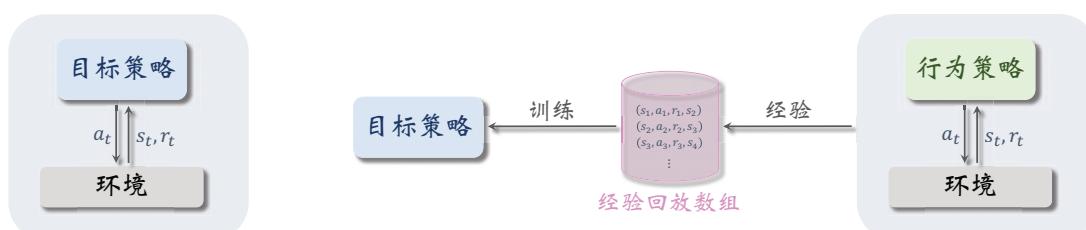


图 4.6: 同策略。

图 4.7: 异策略。

∽第四章 相关文献∽

DQN 首先由 Mnih 等人在 2013 年提出 [69]，其训练用的算法与本章介绍的基本一致，这种简单的训练算法实践中效果不佳。这篇论文用 Atari 游戏评价 DQN 的表现，虽然 DQN 的表现优于已有方法，但是它还是比人类的表现差一截。相同的作者在 2015 年发表了 DQN 的改进版本 [70]，其主要改进在于使用“目标网络”(Target Network)；这个版本的 DQN 在 Atari 游戏上的表现超越了人类玩家。

DQN 的本质是对最优动作价值函数 Q_* 的函数近似。早在 1995 年和 1997 年发表的论文 [8, 105] 就把函数近似用于价值学习中。本章使用的 TD 算法叫做 Q 学习算法，它是由 Watkins 在 1989 年在博士论文 [115] 提出的。Watkins 和 Dayan 发表在 1992 年的论文 [114] 分析了 Q 学习的收敛。1994 年的论文 [53, 104] 改进了 Q 学习算法的收敛分析。训练 DQN 用到的经验回放是由 Lin 在 1993 年的博士论文 [62] 中提出的。

第五章 SARSA 算法

上一章介绍了 Q 学习的表格形式和神经网络形式（即 DQN）。TD 算法是一大类算法的总称。上一章用的 Q 学习是一种 TD 算法，Q 学习的目的是学习最优动作价值函数 Q_* 。本章介绍 SARSA，它也是一种 TD 算法，SARSA 的目的是学习动作价值函数 $Q_\pi(s, a)$ 。

虽然传统的强化学习用 Q_π 作为确定性的策略控制智能体，但是现在 Q_π 通常被用于评价策略的好坏，而非用于控制智能体。 Q_π 常与策略函数 π 结合使用，被称作 Actor-Critic（演员—评委）方法。策略函数 π 控制智能体，因此被看做“演员”；而 Q_π 评价 π 的表现，帮助改进 π ，因此 Q_π 被看做“评委”。Actor-Critic 通常用 SARSA 训练“评委” Q_π 。在后面策略学习的章节会详细介绍 Actor-Critic 方法。

5.1 表格形式的 SARSA

假设状态空间 \mathcal{S} 和动作空间 \mathcal{A} 都是有限集，即集合中元素数量有限。比如， \mathcal{S} 中一共有 3 种状态， \mathcal{A} 中一共有 4 种动作。那么动作价值函数 $Q_\pi(s, a)$ 可以表示为一个 3×4 的表格，比如右边的表格。该表格与一个策略函数 $\pi(a|s)$ 相关联；如果 π 发生变化，表格 Q_π 也会发生变化。

	第 1 种 动作	第 2 种 动作	第 3 种 动作	第 4 种 动作
第 1 种 状态	380	-95	20	173
第 2 种 状态	-7	64	-195	210
第 3 种 状态	152	72	413	-80

图 5.1：动作价值函数 Q_π 表示成表格形式。

我们用表格 q 近似 Q_π 。该如何通过智能体与环境的交互来学习表格 q 呢？首先初始化 q ，可以让它是全零的表格。然后用表格形式的 SARSA 算法更新 q ，每次更新表格的一个元素。最终 q 收敛到 Q_π 。

推导表格形式的 SARSA 学习算法： SARSA 算法由下面的贝尔曼方程推导出：

$$Q_\pi(s_t, a_t) = \mathbb{E}_{S_{t+1}, A_{t+1}} [R_t + \gamma \cdot Q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s_t, A_t = a_t]$$

贝尔曼方程的证明见附录 A。我们对贝尔曼方程左右两边做近似：

- 方程左边的 $Q_\pi(s_t, a_t)$ 可以近似成 $q(s_t, a_t)$ 。 $q(s_t, a_t)$ 是表格在 t 时刻对 $Q_\pi(s_t, a_t)$ 做出的估计。
- 方程右边的期望是关于下一时刻状态 S_{t+1} 和动作 A_{t+1} 求的。给定当前状态 s_t ，智能体执行动作 a_t ，环境会给出奖励 r_t 和新的状态 s_{t+1} 。然后基于 s_{t+1} 做随机抽样，得到新的动作

$$\tilde{a}_{t+1} \sim \pi(\cdot \mid s_{t+1}).$$

用观测到的 r_t 、 s_{t+1} 和计算出的 \tilde{a}_{t+1} 对期望做蒙特卡洛近似，得到：

$$r_t + \gamma \cdot Q_\pi(s_{t+1}, \tilde{a}_{t+1}). \quad (5.1)$$

- 进一步把公式 (5.1) 中的 Q_π 近似成 q , 得到

$$\hat{y}_t \triangleq r_t + \gamma \cdot q(s_{t+1}, \tilde{a}_{t+1}).$$

把它称作 TD 目标。它是表格在 $t+1$ 时刻对 $Q_\pi(s_t, a_t)$ 做出的估计。

$q(s_t, a_t)$ 和 \hat{y}_t 都是对动作价值 $Q_\pi(s_t, a_t)$ 的估计。由于 \hat{y}_t 部分基于真实观测到的奖励 r_t , 我们认为 \hat{y}_t 是更可靠的估计, 所以鼓励 $q(s_t, a_t)$ 趋近 \hat{y}_t 。更新表格 (s_t, a_t) 位置上的元素:

$$q(s_t, a_t) \leftarrow (1 - \alpha) \cdot q(s_t, a_t) + \alpha \cdot \hat{y}_t.$$

这样可以使得 $q(s_t, a_t)$ 更接近 \hat{y}_t 。SARSA 是 State-Action-Reward-State-Action 的缩写, 原因是 SARSA 算法用到了这个五元组: $(s_t, a_t, r_t, s_{t+1}, \tilde{a}_{t+1})$ 。SARSA 算法学到的 q 依赖于策略 π , 这是因为五元组中的 \tilde{a}_{t+1} 是根据 $\pi(\cdot | s_{t+1})$ 抽样得到的。

训练流程: 设当前表格为 q_{now} , 当前策略为 π_{now} 。每一轮更新表格中的一个元素, 把更新之后的表格记作 q_{new} 。

1. 观测到当前状态 s_t , 根据当前策略做抽样: $a_t \sim \pi_{\text{now}}(\cdot | s_t)$ 。
2. 把表格 q_{now} 中第 (s_t, a_t) 位置上的元素记作:

$$\hat{q}_t = q_{\text{now}}(s_t, a_t).$$

3. 智能体执行动作 a_t 之后, 观测到奖励 r_t 和新的状态 s_{t+1} 。
4. 根据当前策略做抽样: $\tilde{a}_{t+1} \sim \pi_{\text{now}}(\cdot | s_{t+1})$ 。注意, \tilde{a}_{t+1} 只是假想的动作, 智能体不予执行。
5. 把表格 q_{now} 中第 $(s_{t+1}, \tilde{a}_{t+1})$ 位置上的元素记作:

$$\hat{q}_{t+1} = q_{\text{now}}(s_{t+1}, \tilde{a}_{t+1}).$$

6. 计算 TD 目标和 TD 误差:

$$\hat{y}_t = r_t + \gamma \cdot \hat{q}_{t+1}, \quad \delta_t = \hat{q}_t - \hat{y}_t.$$

7. 更新表格中 (s_t, a_t) 位置上的元素:

$$q_{\text{new}}(s_t, a_t) \leftarrow q_{\text{now}}(s_t, a_t) - \alpha \cdot \delta_t.$$

8. 用某种算法更新策略函数。该算法与 SARSA 算法无关。

Q 学习与 SARSA 的对比: Q 学习不依赖于 π , 因此 Q 学习属于异策略 (Off-policy), 可以用经验回放。而 SARSA 依赖于 π , 因此 SARSA 属于同策略 (On-policy), 不能用经验回放。两种算法的对比如图 5.2 所示。

Q 学习的目标是学到表格 \tilde{Q} , 作为最优动作价值函数 Q_* 的近似。因为 Q_* 与 π 无关, 所以在理想情况下, 不论收集经验用的**行为策略** π 是什么, 都不影响 Q 学习得到的 Q 。因此, Q 学习属于异策略 (Off-policy), 允许**行为策略**区别于**目标策略**。Q 学习允许使用经验回放, 可以重复利用过时的经验。

SARSA 算法的目标是学到表格 q , 作为动作价值函数 Q_π 的近似。 Q_π 与一个策略 π

5.1 表格形式的 SARSA

相对应；用不同的策略 π ，对应 Q_π 就会不同；策略 π 越好， Q_π 的值越大。经验回放数组里的经验 (s_j, a_j, r_j, s_{j+1}) 是过时的行为策略 π_{old} 收集到的，与当前策略 π_{now} 及其对应的价值 $Q_{\pi_{\text{now}}}$ 对应不上。想要学习 Q_π 的话，必须要用与当前策略 π_{now} 收集到的经验，而不能用过时的 π_{old} 收集到的经验。这就是为什么 SARSA 不能用经验回放。

Q 学习	近似 Q_\star	异策略	可以使用 经验回放
SARSA	近似 Q_π	同策略	不能使用 经验回放

图 5.2: Q 学习与 SARSA 的对比。

5.2 神经网络形式的 SARSA

价值网络：如果状态空间 \mathcal{S} 是无限集，那么我们无法用一张表格表示 Q_π ，否则表格的行数是无穷。一种可行的方案是用一个神经网络 $q(s, a; \mathbf{w})$ 来近似 $Q_\pi(s, a)$ ；理想情况下，

$$q(s, a; \mathbf{w}) = Q_\pi(s, a), \quad \forall s \in \mathcal{S}, a \in \mathcal{A}.$$

神经网络 $q(s, a; \mathbf{w})$ 被称为价值网络 (Value Network)，其中的 \mathbf{w} 表示神经网络中可训练的参数。神经网络的结构是人预先设定的（比如有多少层，每一层的宽度是多少），而参数 \mathbf{w} 需要通过智能体与环境的交互来学习。首先随机初始化 \mathbf{w} ，然后用 SARSA 算法更新 \mathbf{w} 。

神经网络的结构见图 5.3。价值网络的输入是状态 s ；如果 s 是矩阵或张量 (Tensor)，那么可以用卷积网络处理 s (如图 5.3)；如果 s 是向量，那么可以用全连接层处理 s 。价值网络的输出是每个动作的价值。动作空间 \mathcal{A} 中有多少种动作，则价值网络的输出就是多少维的向量，向量每个元素对应一个动作。举个例子，动作空间是 $\mathcal{A} = \{\text{左, 右, 上}\}$ ，价值网络的输出是

$$q(s, \text{左}; \mathbf{w}) = 219,$$

$$q(s, \text{右}; \mathbf{w}) = -73,$$

$$q(s, \text{上}; \mathbf{w}) = 580.$$

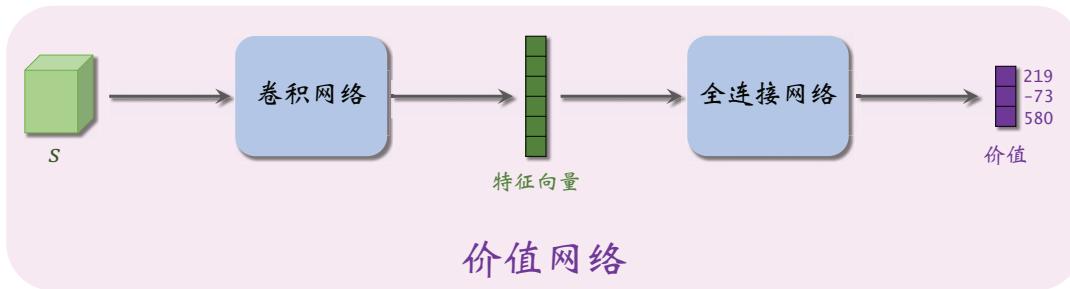


图 5.3: 价值网络 $q(s, a; \mathbf{w})$ 的结构。输入是状态 s ；输出是每个动作的价值。

算法推导：给定当前状态 s_t ，智能体执行动作 a_t ，环境会给出奖励 r_t 和新的状态 s_{t+1} 。然后基于 s_{t+1} 做随机抽样，得到新的动作 $\tilde{a}_{t+1} \sim \pi(\cdot | s_{t+1})$ 。定义 TD 目标：

$$\hat{y}_t \triangleq r_t + \gamma \cdot q(s_{t+1}, \tilde{a}_{t+1}; \mathbf{w}),$$

我们鼓励 $q(s_t, a_t; \mathbf{w})$ 接近 TD 目标，所以定义损失函数：

$$L(\mathbf{w}) \triangleq \frac{1}{2} [q(s_t, a_t; \mathbf{w}) - \hat{y}_t]^2.$$

损失函数的变量是 \mathbf{w} ，而 \hat{y}_t 被视为常数 (尽管 \hat{y}_t 也依赖于参数 \mathbf{w} ，但这一点被忽略掉)。设 $\hat{q}_t = q(s_t, a_t; \mathbf{w})$ 。损失函数关于 \mathbf{w} 的梯度是：

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \underbrace{(\hat{q}_t - \hat{y}_t)}_{\text{TD 误差 } \delta_t} \cdot \nabla_{\mathbf{w}} q(s_t, a_t; \mathbf{w}).$$

5.2 神经网络形式的 SARSA

做一次梯度下降更新 \mathbf{w} :

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} q(s_t, a_t; \mathbf{w}).$$

这样可以使得 $q(s_t, a_t; \mathbf{w})$ 更接近 \hat{y}_t 。此处的 α 是学习率，需要手动调。

训练流程：设当前价值网络的参数为 \mathbf{w}_{now} ，当前策略为 π_{now} 。每一轮训练用五元组 $(s_t, a_t, r_t, s_{t+1}, \tilde{a}_{t+1})$ 对价值网络参数做一次更新。

1. 观测到当前状态 s_t ，根据当前策略做抽样： $a_t \sim \pi_{\text{now}}(\cdot | s_t)$ 。
2. 用价值网络计算 (s_t, a_t) 的价值：

$$\hat{q}_t = q(s_t, a_t; \mathbf{w}_{\text{now}}).$$

3. 智能体执行动作 a_t 之后，观测到奖励 r_t 和新的状态 s_{t+1} 。
4. 根据当前策略做抽样： $\tilde{a}_{t+1} \sim \pi_{\text{now}}(\cdot | s_{t+1})$ 。注意， \tilde{a}_{t+1} 只是假想的动作，智能体不予执行。
5. 用价值网络计算 $(s_{t+1}, \tilde{a}_{t+1})$ 的价值：

$$\hat{q}_{t+1} = q(s_{t+1}, \tilde{a}_{t+1}; \mathbf{w}_{\text{now}}).$$

6. 计算 TD 目标和 TD 误差：

$$\hat{y}_t = r_t + \gamma \cdot \hat{q}_{t+1}, \quad \delta_t = \hat{q}_t - \hat{y}_t.$$

7. 对价值网络 q 做反向传播，计算 q 关于 \mathbf{w} 的梯度： $\nabla_{\mathbf{w}} q(s_t, a_t; \mathbf{w}_{\text{now}})$ 。
8. 更新价值网络参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} q(s_t, a_t; \mathbf{w}_{\text{now}}).$$

9. 用某种算法更新策略函数。该算法与 SARSA 算法无关。

5.3 多步 TD 目标

首先回顾一下 SARSA 算法。给定五元组 $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, SARSA 计算 TD 目标:

$$\hat{y}_t = r_t + \gamma \cdot q(s_{t+1}, a_{t+1}; \mathbf{w}).$$

公式中只用到一个奖励 r_t , 这样得到的 \hat{y}_t 叫做单步 TD 目标。多步 TD 目标用 m 个奖励, 可以视作单步 TD 目标的推广。下面我们推导多步 TD 目标。

数学推导: 设一局游戏的长度为 n 。根据定义, t 时刻的回报 U_t 是 t 时刻之后的所有奖励的加权和:

$$U_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots + \gamma^{n-t} R_n.$$

同理, $t+m$ 时刻的回报可以写成:

$$U_{t+m} = R_{t+m} + \gamma R_{t+m+1} + \gamma^2 R_{t+m+2} + \cdots + \gamma^{n-t-m} R_n.$$

下面我们推导两个回报的关系。把 U_t 写成:

$$\begin{aligned} U_t &= \left(R_t + \gamma R_{t+1} + \cdots + \gamma^{m-1} R_{t+m-1} \right) + \left(\gamma^m R_{t+m} + \cdots + \gamma^{n-t} R_n \right) \\ &= \left(\sum_{i=0}^{m-1} \gamma^i R_{t+i} \right) + \underbrace{\gamma^m \left(R_{t+m} + \gamma R_{t+m+1} + \cdots + \gamma^{n-t-m} R_n \right)}_{\text{等于 } U_{t+m}}. \end{aligned}$$

因此, 回报可以写成这种形式:

$$U_t = \left(\sum_{i=0}^{m-1} \gamma^i R_{t+i} \right) + \gamma^m U_{t+m}.$$

动作价值函数 $Q_\pi(s_t, a_t)$ 是回报 U_t 的期望, 而 $Q_\pi(s_{t+m}, a_{t+m})$ 是回报 U_{t+m} 的期望。利用上面的等式, 再按照贝尔曼方程的证明 (见附录 A), 不难得出下面的定理:

定理 5.1

设 R_k 是 S_k 、 A_k 、 S_{k+1} 的函数, $\forall k = 1, \dots, n$ 。那么

$$\underbrace{Q_\pi(s_t, a_t)}_{U_t \text{ 的期望}} = \mathbb{E} \left[\left(\sum_{i=0}^{m-1} \gamma^i R_{t+i} \right) + \gamma^m \cdot \underbrace{Q_\pi(S_{t+m}, A_{t+m})}_{U_{t+m} \text{ 的期望}} \mid S_t = s_t, A_t = a_t \right].$$

公式中的期望是关于随机变量 $S_{t+1}, A_{t+1}, \dots, S_{t+m}, A_{t+m}$ 求的。



注 回报 U_t 的随机性来自于 t 到 n 时刻的状态和动作:

$$S_t, A_t, S_{t+1}, A_{t+1}, \dots, S_{t+m}, A_{t+m}, S_{t+m+1}, A_{t+m+1}, \dots, S_n, A_n.$$

定理中把 $S_t = s_t$ 和 $A_t = a_t$ 看做是观测值, 用期望消掉 $S_{t+1}, A_{t+1}, \dots, S_{t+m}, A_{t+m}$, 而 $Q_\pi(S_{t+m}, A_{t+m})$ 则消掉了剩余的随机变量 $S_{t+m+1}, A_{t+m+1}, \dots, S_n, A_n$ 。

多步 TD 目标: 我们对定理 5.1 中的期望做蒙特卡洛近似, 然后再用价值网络 $q(s, a; \mathbf{w})$ 近似动作价值函数 $Q_\pi(s, a)$ 。具体做法如下:

- 在 t 时刻, 价值网络做出预测 $\hat{q}_t = q(s_t, a_t; \mathbf{w})$, 它是对 $Q_\pi(s_t, a_t)$ 的估计。

- 已知当前状态 s_t , 用策略 π 控制智能体与环境交互 m 次, 得到轨迹

$$r_t, s_{t+1}, a_{t+1}, r_{t+1}, \dots, s_{t+m-1}, a_{t+m-1}, r_{t+m-1}, s_{t+m}, a_{t+m}.$$

在 $t + m$ 时刻, 用观测到的轨迹对定理 5.1 中的期望做蒙特卡洛近似, 把近似的结果记作:

$$\left(\sum_{i=0}^{m-1} \gamma^i r_{t+i} \right) + \gamma^m \cdot Q_\pi(s_{t+m}, a_{t+m}).$$

- 进一步用 $q(s_{t+m}, a_{t+m}; \mathbf{w})$ 近似 $Q_\pi(s_{t+m}, a_{t+m})$, 得到:

$$\hat{y}_t = \left(\sum_{i=0}^{m-1} \gamma^i r_{t+i} \right) + \gamma^m \cdot q(s_{t+m}, a_{t+m}; \mathbf{w}).$$

把 \hat{y}_t 称作 m 步 TD 目标。

$\hat{q}_t = q(s_t, a_t; \mathbf{w})$ 和 \hat{y}_t 分别是价值网络在 t 时刻和 $t + m$ 时刻做出的预测, 两者都是对 $Q_\pi(s_t, a_t)$ 的估计值。 \hat{q}_t 是纯粹的预测, 而 \hat{y}_t 则基于 m 组实际观测, 因此 \hat{y}_t 比 \hat{q}_t 更可靠。我们鼓励 \hat{q}_t 接近 \hat{y}_t 。设损失函数为

$$L(\mathbf{w}) \triangleq \frac{1}{2} [q(s_t, a_t; \mathbf{w}) - \hat{y}_t]^2. \quad (5.2)$$

做单步梯度下降更新价值网络参数 \mathbf{w} :

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot (\hat{q}_t - \hat{y}_t) \cdot \nabla_{\mathbf{w}} q(s_t, a_t; \mathbf{w}).$$

训练流程: 设当前价值网络的参数为 \mathbf{w}_{now} , 当前策略为 π_{now} 。执行以下步骤更新价值网络和策略。

- 用策略网络 π_{now} 控制智能体与环境交互, 完成一个回合, 得到轨迹:

$$s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_n, a_n, r_n.$$

- 对于所有的 $t = 1, \dots, n - m$, 计算

$$\hat{q}_t = q(s_t, a_t; \mathbf{w}_{\text{now}}).$$

- 对于所有的 $t = 1, \dots, n - m$, 计算多步 TD 目标和 TD 误差:

$$\hat{y}_t = \sum_{i=0}^{m-1} \gamma^i r_{t+i} + \gamma^m \hat{q}_{t+m}, \quad \delta_t = \hat{q}_t - \hat{y}_t.$$

- 对于所有的 $t = 1, \dots, n - m$, 对价值网络 q 做反向传播, 计算 q 关于 \mathbf{w} 的梯度:

$$\nabla_{\mathbf{w}} q(s_t, a_t; \mathbf{w}_{\text{now}}).$$

- 更新价值网络参数:

$$\mathbf{w}_{\text{new}}(s_t, a_t) \leftarrow \mathbf{w}_{\text{now}}(s_t, a_t) - \alpha \cdot \sum_{t=1}^{n-m} \delta_t \cdot \nabla_{\mathbf{w}} q(s_t, a_t; \mathbf{w}_{\text{now}}).$$

- 用某种算法更新策略函数 π 。该算法与 SARSA 算法无关。

5.4 蒙特卡洛与自举

上一节介绍了多步 TD 目标。单步 TD 目标、回报是多步 TD 目标的两种特例。如下图所示，如果设 $m = 1$ ，那么多步 TD 目标变成单步 TD 目标；如果设 $m = n - t + 1$ ，那么多步 TD 目标变成实际观测的回报 u_t 。

$$\begin{array}{c} \text{单步TD目标:} \\ \hat{y}_t = r_t + \gamma \hat{q}_{t+1}. \\ (\text{自举}) \end{array} \xleftarrow{m=1} \quad \begin{array}{c} m\text{步TD目标:} \\ \hat{y}_t = \sum_{i=0}^{m-1} \gamma^i r_{t+i} + \gamma^m \hat{q}_{t+m}. \end{array} \xrightarrow{m=n-t+1} \quad \begin{array}{c} \text{观测到的回报:} \\ u_t = \sum_{i=0}^{n-t} \gamma^i r_{t+i}. \\ (\text{蒙特卡洛}) \end{array}$$

图 5.4: 单步 TD 目标、多步 TD 目标、回报的关系。

5.4.1 蒙特卡洛

训练价值网络 $q(s, a; \mathbf{w})$ 的时候，我们可以将一局游戏进行到底，观测到所有的奖励 r_1, \dots, r_n ，然后计算回报 $u_t = \sum_{i=0}^{n-t} \gamma^i r_{t+i}$ ，拿 u_t 作为目标，鼓励价值网络 $q(s_t, a_t; \mathbf{w})$ 接近 u_t 。定义损失函数：

$$L(\mathbf{w}) = \frac{1}{2} [q(s_t, a_t; \mathbf{w}) - u_t]^2.$$

然后做一次梯度下降更新 \mathbf{w} ：

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \nabla_{\mathbf{w}} L(\mathbf{w}),$$

这样可以让价值网络的预测 $q(s_t, a_t; \mathbf{w})$ 更接近 u_t 。这种训练价值网络的方法不是 TD。

在强化学习中，训练价值网络的时候以 u_t 作为目标，这种方式被称作“蒙特卡洛”。原因非常显然：动作价值函数可以写作 $Q_{\pi}(s_t, a_t) = \mathbb{E}[U_t | S_t = s_t, A_t = a_t]$ ，而我们用实际观测 u_t 去近似期望，这就是典型的蒙特卡洛近似。

蒙特卡洛的好处是无偏性： u_t 是 $Q_{\pi}(s_t, a_t)$ 的无偏估计。由于 u_t 的无偏性，拿 u_t 作为目标训练价值网络，得到的价值网络也是无偏的。

蒙特卡洛的坏处是方差大。随机变量 U_t 依赖于 $S_{t+1}, A_{t+1}, \dots, S_n, A_n$ 这些随机变量，其中不确定性很大。观测值 u_t 虽然是 U_t 的无偏估计，但可能实际上离 $\mathbb{E}[U_t]$ 很远。因此，拿 u_t 作为目标训练价值网络，收敛会很慢。

5.4.2 自举

在介绍价值学习的自举之前，先解释一下什么叫自举。大家可能经常在强化学习和统计学的文章里见到 Bootstrapping 这个词。它的字面意思是“拔自己的鞋带，把自己举起来”。所以 Bootstrapping 翻译成“自举”，即自己把自己举起来。自举听起来很荒谬。即使你“力拔山兮气盖世”，你也没办法拔自己的鞋带，把自己举起来。自举乍看起来很荒唐，但是在统计和机器学习是可以做到自举的；Bootstrapping 方法在统计和机器学习里面非常常用。

在强化学习中，“自举”的意思是“用一个估算去更新同类的估算”，类似于“自己

把自己给举起来”。SARSA 使用的单步 TD 目标定义为：

$$\hat{y}_t = r_t + \underbrace{\gamma \cdot q(s_{t+1}, a_{t+1}; \mathbf{w})}_{\text{价值网络做出的估计}}$$

SARSA 鼓励 $q(s_t, a_t; \mathbf{w})$ 接近 \hat{y}_t ，所以定义损失函数

$$L(\mathbf{w}) = \frac{1}{2} \left[\underbrace{q(s_t, a_t; \mathbf{w}) - \hat{y}_t}_{\text{让价值网络拟合 } \hat{y}_t} \right]^2$$

TD 目标 \hat{y}_t 的一部分是价值网络做出的估计 $\gamma \cdot q(s_{t+1}, a_{t+1}; \mathbf{w})$ ，然后 SARSA 让 $q(s_t, a_t; \mathbf{w})$ 去拟合 \hat{y}_t 。这就是用价值网络自己做出的估计去更新价值网络自己，这属于“自举”。¹

自举的好处是方差小。单步 TD 目标的随机性只来自于 S_{t+1} 和 A_{t+1} ，而回报 U_t 的随机性来自于 $S_{t+1}, A_{t+1}, \dots, S_n, A_n$ 。很显然，单步 TD 目标的随机性较小，因此方差较小。用自举的训练价值网络，收敛比较快。

自举的坏处是有偏差。价值网络 $q(s, a; \mathbf{w})$ 是对动作价值 $Q_\pi(s, a)$ 的近似；最理想的情况下， $q(s, a; \mathbf{w}) = Q_\pi(s, a)$ ， $\forall s, a$ 。假如碰巧 $q(s_{j+1}, a_{j+1}; \mathbf{w})$ 低估（或高估）真实价值 $Q_\pi(s_{j+1}, a_{j+1})$ ，则会发生下面的情况：

$$\begin{aligned} q(s_{j+1}, a_{j+1}; \mathbf{w}) &\quad \text{低估 (或高估)} & Q_\pi(s_{j+1}, a_{j+1}) \\ \Rightarrow \hat{y}_j &\quad \text{低估 (或高估)} & Q_\pi(s_j, a_j) \\ \Rightarrow q(s_j, a_j; \mathbf{w}) &\quad \text{低估 (或高估)} & Q_\pi(s_j, a_j). \end{aligned}$$

也就是说，自举会让偏差从 (s_{t+1}, a_{t+1}) 传播到 (s_t, a_t) 。第 6.2 节详细讨论自举造成的偏差以及解决方案。

5.4.3 蒙特卡洛和自举的对比

在价值学习中，用实际观测的回报 u_t 作为目标的方法被称为蒙特卡洛，即图 5.5 中的蓝色的箱型图。 u_t 是 $Q_\pi(s_t, a_t)$ 的无偏估计，即 U_t 的期望等于 $Q_\pi(s_t, a_t)$ 。但是它的方差很大，也就是说实际观测到的 u_t 可能离 $Q_\pi(s_t, a_t)$ 很远。

用单步 TD 目标 \hat{y}_t 作为目标的方法称为自举，即图 5.5 中的红色的箱型图。自举的好处在于方差小， \hat{y}_t 不会偏离期望太远。但是 \hat{y}_t 往往是有偏的，它的期望往往不等于 $Q_\pi(s_t, a_t)$ 。用自举训练出的价值网络往往有系统性的偏差（低估或者高估）。实践中，自举通常比蒙特卡洛收敛更快，这就是为什么训练 DQN 和价值网络通常用 TD 算法。

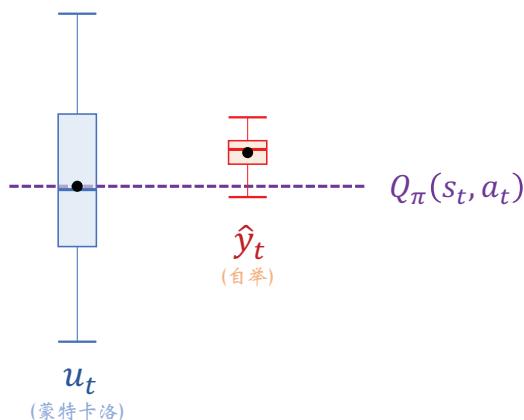


图 5.5： u_t 和 \hat{y}_t 的箱型图 (Boxplot) 示意。

¹严格地说，TD 目标 \hat{y}_t 中既有自举的成分，也有蒙特卡洛的成分。TD 目标中的 $\gamma \cdot q(s_{t+1}, a_{t+1}; \mathbf{w})$ 是自举，因为它拿价值网络自己的估计作为目标。TD 目标中的 r_t 是实际观测，它是对 $\mathbb{E}[R_t]$ 的蒙特卡洛。

如图 5.4 所示，多步 TD 目标 $\hat{y}_t = (\sum_{i=0}^{m-1} \gamma^i r_{t+i}) + \gamma^m \cdot q(s_{t+m}, a_{t+m}; \mathbf{w})$ 介于蒙特卡洛和自举之间。多步 TD 目标有很大的蒙特卡洛成分，其中的 $\sum_{i=0}^{m-1} \gamma^i r_{t+i}$ 基于 m 个实际观测到的奖励。多步 TD 目标也有自举的成分，其中的 $\gamma^m \cdot q(s_{t+m}, a_{t+m}; \mathbf{w})$ 是用价值网络自己算出来的。如果把 m 设置得比较好，可以在方差和偏差之间找到好的平衡，使得多步 TD 目标效果优于单步 TD 目标、也优于回报 u_t 。

∽第五章 相关文献∽

Q 学习算法首先由 Watkins 在他 1989 年的博士论文 [115] 中提出。Watkins 和 Dayan 发表在 1992 年的论文 [114] 分析了 Q 学习的收敛。1994 年的论文 [53, 104] 改进了 Q 学习算法的收敛分析。

SARSA 算法比 Q 学习提出得晚。SARSA 首先由 Rummery 和 Niranjan 于 1994 年提出 [81]，但名字不叫 SARSA。SARSA 的名字是 Sutton 在 1996 年起的 [95]。

多步 TD 目标也是 Watkins 1989 年的博士论文 [115] 提出的。Sutton 和 Barto 的书 [96] 对多步 TD 目标有详细介绍和分析。近年来有不少论文（比如 [68, 109, 46]）表明多步 TD 目标非常有用。

《深度强化学习》 2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

第六章 价值学习高级技巧

第 4 章介绍了 DQN，并且用 Q 学习算法（一种 TD 算法）训练 DQN。如果读者按照第 4 章最原始的方式实现 DQN，效果会很不理想。想要提升 DQN 的表现，需要用本章的高级技巧。文献中已经有充分实验结果表明这些高级技巧对 DQN 非常有效，而且这些技巧不冲突，可以一起使用。这些技巧并不局限于 DQN，而是可以用于多种价值学习和策略学习方法。

第 6.1、6.2 节介绍两种方法改进 TD 算法，让 DQN 训练得更好。第 6.1 节介绍经验回放 (Experience Replay) 和优先经验回放 (Prioritized Experience Replay)。第 6.2 节讨论 DQN 的高估问题以及解决方案——目标网络 (Target Network) 和双 Q 学习算法 (Double Q-learning)。

第 6.3、6.4 节介绍两种方法改进 DQN 神经网络结构（不是对 TD 算法的改进）。第 6.3 节介绍对决网络 (Dueling Network)，它把动作价值 (Action-Value) 分解成状态价值 (State-Value) 与优势 (Advantage)。第 6.4 节介绍噪声网络 (Noisy Net)，它往神经网络的参数中加入随机性，鼓励探索。

6.1 经验回放

经验回放 (Experience Replay) 是强化学习中一个重要的技巧，可以大幅提升强化学习的表现。经验回放的意思是把智能体与环境交互的记录（即经验）储存到一个数组里，事后反复利用这些经验训练智能体。这个数组被称为经验回放数组 (Replay Buffer)。

具体来说，把智能体的轨迹划分成 (s_t, a_t, r_t, s_{t+1}) 这样的四元组，存入一个数组。需要人为指定数组的大小（记作 b ）。数组中只保留最近 b 条数据；当数组存满之后，删除掉最旧的数据。数组的大小 b 是个需要调的超参数，会影响训练的结果；通常设置 b 为 $10^5 \sim 10^6$ 。

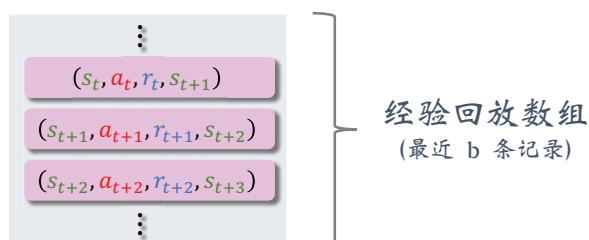


图 6.1：经验回放数组。

6.1.1 经验回放的优点

经验回放的一个好处在于打破序列的相关性。训练 DQN 的时候，每次我们用一个四元组对 DQN 的参数做一次更新。我们希望相邻两次使用的四元组是独立的。然而当智能体收集经验的时候，相邻两个四元组 (s_t, a_t, r_t, s_{t+1}) 和 $(s_{t+1}, a_{t+1}, r_{t+1}, s_{t+2})$ 有很强的相关性。依次使用这些强关联的四元组训练 DQN，效果往往会很差。经验回放每次从数组里随机抽取一个四元组，用来对 DQN 参数做一次更新。这样随机抽到的四元组都

是独立的，消除了相关性。

经验回放的另一个好处是重复利用收集到的经验，而不是用一次就丢弃，这样可以用更少的样本数量达到同样的表现。重复利用经验、不重复利用经验的收敛曲线通常如图 6.2 所示。图的横轴是样本数量，纵轴是平均回报。

注 在阅读文献的时候请注意“样本数量”(Sample Complexity) 与“更新次数”两者区别的区别。样本数量是指智能体从环境中获取的奖励 r 的数量。而一次更新的意思是从经验回放数组里取出一个或多个四元组，用它对参数 w 做一次更新。通常来说，样本数量更重要，因为在实际应用中收集经验比较困难；比如，在机器人的应用中，需要在现实世界里做一次实验才能收集到一条经验。做更新的次数不是那么重要，更新次数只会影响训练时的计算量而已。

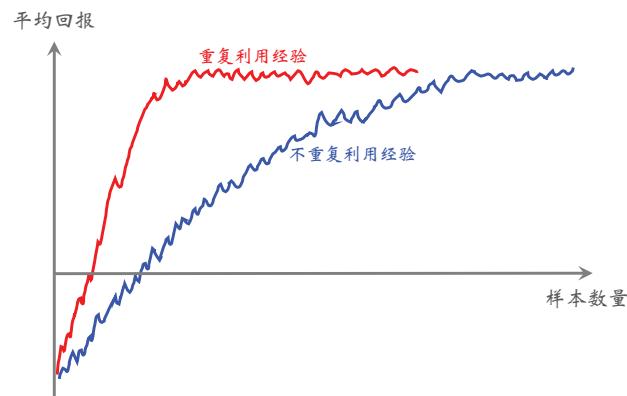


图 6.2: 收敛曲线示意图。

6.1.2 经验回放的局限性

需要注意，并非所有的强化学习方法都允许重复使用过去的经验。经验回放数组里的数据全都是用**行为策略**(Behavior Policy)控制智能体收集到的。在收集经验同时，我们也在不断地改进策略。策略的变化导致收集经验时用的**行为策略**是过时的策略，不同于当前我们想要更新的策略——即**目标策略**(Target Policy)。也就是说，经验回放数组中的经验通常是过时的**行为策略**收集的，而我们真正想要学的**目标策略**不同于过时的**行为策略**。

有些强化学习方法允许**行为策略**不同于**目标策略**。这样的强化学习方法叫做**异策略**(Off-policy)。比如 Q 学习、确定策略梯度(DPG)都属于异策略。由于它们允许**行为策略**不同于**目标策略**，因此过时**行为策略**收集到的经验可以被重复利用。**经验回放适用于异策略**。

有些强化学习方法要求**行为策略**与**目标策略**必须相同。这样的强化学习方法叫做**同策略**(On-policy)。比如 SARSA、REINFORCE、A2C 都属于同策略。它们要求经验必须是当前的**目标策略**收集到的，而不能使用过时的经验。**经验回放不适用于同策略**。

6.1.3 优先经验回放

优先经验回放(Prioritized Experience Replay)是一种特殊的经验回放方法，它比普通的经验回放效果更好：既能让收敛更快，也能让收敛时的平均回报更高。经验回放数组里有 b 个四元组，普通经验回放每次均匀抽样得到一个样本——即四元组 (s_j, a_j, r_j, s_{j+1}) ，用它来更新 DQN 的参数。优先经验回放给每个四元组一个权重，然后根据权重做非均匀

6.1 经验回放

随机抽样。如果 DQN 对 (s_j, a_j) 的价值判断不准确，即 $Q(s_j, a_j; \mathbf{w})$ 离 $Q_*(s_j, a_j)$ 较远，则四元组 (s_j, a_j, r_j, s_{j+1}) 应当有较高的权重。

为什么样本的重要性会有所不同呢？设想你用强化学习训练一辆无人车。经验回放数组中的样本绝大多数都是车辆正常行驶的情形，只有极少数样本是意外情况，比如旁边车辆强行变道、行人横穿马路、警察封路要求绕行。数组中的样本的重要性显然是不同的。车辆正常行驶的样本要多少有多少，而且正常行驶的情形很容易处理，出错的可能性非常小。意外情况的样本非常少，但是又极其重要，处理不好就会车毁人亡。所以意外情况的样本应当有更高的权重，受到更多关注。不应该同等对待正常行驶、意外情况的样本。

如何自动判断哪些样本更重要呢？举个例子，自动驾驶中的意外情况数量少、而且难以处理，导致 DQN 的预测 $Q(s_j, a_j; \mathbf{w})$ 严重偏离真实价值 $Q_*(s_j, a_j)$ 。因此，要是 $|Q(s_j, a_j; \mathbf{w}) - Q_*(s_j, a_j)|$ 较大，则应该给样本 (s_j, a_j, r_j, s_{j+1}) 较高的权重。然而实际上我们无从得知 $|Q(s_j, a_j; \mathbf{w}) - Q_*(s_j, a_j)|$ ；不妨把它替换成 TD 误差。回忆一下，TD 误差的定义是：

$$\delta_j \triangleq Q(s_j, a_j; \mathbf{w}_{\text{now}}) - \underbrace{\left[r_t + \gamma \cdot \max_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w}_{\text{now}}) \right]}_{\text{即 TD 目标}}.$$

如果 TD 误差的绝对值 $|\delta_j|$ 大，说明当前的 DQN（参数是 \mathbf{w}_{now} ）对 (s_j, a_j) 的真实价值的评估不准确，那么应该给 (s_j, a_j, r_j, s_{j+1}) 设置较高的权重。

优先经验回放对数组里的样本做非均匀抽样。四元组 (s_j, a_j, r_j, s_{j+1}) 的权重是 TD 误差的绝对值 $|\delta_j|$ ，它的抽样概率取决于 TD 误差。有两种方法设置抽样概率。一种抽样概率是：

$$p_j \propto |\delta_j| + \epsilon.$$

此处的 ϵ 是个很小的数，防止抽样概率接近零，用于保证所有样本都以非零的概率被抽到。另一种抽样方式先对 $|\delta_j|$ 做降序排列，然后计算

$$p_j \propto \frac{1}{\text{rank}(j)}.$$

此处的 $\text{rank}(j)$ 是 $|\delta_j|$ 的序号；大的 $|\delta_j|$ 的序号小，小的 $|\delta_j|$ 的序号大。两种方式的原理是一样的：TD 误差大的样本被抽样到的概率大。

优先经验回放做非均匀抽样，四元组 (s_j, a_j, r_j, s_{j+1}) 被抽到的概率是 p_j 。抽样是非均匀的，不同的样本有不同的抽样概率，这样会导致 DQN 的预测有偏差。应该相应调整学习率，抵消掉不同抽样概率造成的偏差。TD 算法用“随机梯度下降”来更新参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \mathbf{g},$$

此处的 α 是学习率， \mathbf{g} 是损失函数关于 \mathbf{w} 的梯度。如果用均匀抽样，那么所有样本有相同的学习率 α 。如果做非均匀抽样的话，应该根据抽样概率来调整学习率 α ；如果一条样本被抽样的概率大，那么它的学习率就应该比较小。可以这样设置学习率：

$$\alpha_j = \frac{\alpha}{(b \cdot p_j)^\beta},$$

此处的 b 是经验回放数组中样本的总数， $\beta \in (0, 1)$ 是个需要调的超参数¹。

注 均匀抽样是一种特例，即所有抽样概率都相等： $p_1 = \dots = p_b = \frac{1}{b}$ 。在这种情况下，有 $(b \cdot p_j)^\beta = 1$ ，因此学习率都相同： $\alpha_1 = \dots = \alpha_b = \alpha$ 。

注 读者可能会问下面的问题。如果样本 (s_j, a_j, r_j, s_{j+1}) 很重要，它被抽到的概率 p_j 很大，可是它的学习率却很小。当 $\beta = 1$ 时，如果抽样概率 p_j 变大 10 倍，则学习率 α_j 减小 10 倍。抽样概率、学习率两者岂不是抵消了吗？优先经验回放有什么意义呢？两者其实并没有抵消，因为下面两种方式并不等价：

- 设置学习率为 α ，使用样本 (s_j, a_j, r_j, s_{j+1}) 计算一次梯度，更新一次参数 \mathbf{w} ；
- 设置学习率为 $\frac{\alpha}{10}$ ，使用样本 (s_j, a_j, r_j, s_{j+1}) 计算十次梯度，更新十次参数 \mathbf{w} 。

乍看起来两种方式区别不大，但其实第二种方式是对样本更有效的利用。第二种方式的缺点在于计算量大了十倍；所以第二种方式只被用于重要的样本。

序号	四元组	TD 误差	抽样概率	学习率
\vdots	\vdots	\vdots	\vdots	\vdots
$j - 1$	$(s_{j-1}, a_{j-1}, r_{j-1}, s_j)$	δ_{j-1}	$p_{j-1} \propto \delta_{j-1} + \epsilon$	$\alpha \cdot (b \cdot p_{j-1})^{-\beta}$
j	(s_j, a_j, r_j, s_{j+1})	δ_j	$p_j \propto \delta_j + \epsilon$	$\alpha \cdot (b \cdot p_j)^{-\beta}$
$j + 1$	$(s_{j+1}, a_{j+1}, r_{j+1}, s_{j+2})$	δ_{j+1}	$p_{j+1} \propto \delta_{j+1} + \epsilon$	$\alpha \cdot (b \cdot p_{j+1})^{-\beta}$
\vdots	\vdots	\vdots	\vdots	\vdots

图 6.3：优先经验回放数组。

优先经验回放数组如图 6.3 所示。设 b 为数组大小，需要手动调整。如果样本（即四元组）的数量超过了 b ，那么要删除最旧的样本。数组里记录了四元组、TD 误差、抽样概率、以及学习率。注意，数组里存的 TD 误差 δ_j 是用过时 DQN 参数计算出来的：

$$\delta_j = Q(s_j, a_j; \mathbf{w}_{\text{old}}) - [r_t + \gamma \cdot \max_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w}_{\text{old}})].$$

做经验回放的时候，每次取出一个（或多个）四元组，用它计算出新的 TD 误差：

$$\delta'_j = Q(s_j, a_j; \mathbf{w}_{\text{now}}) - [r_t + \gamma \cdot \max_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w}_{\text{now}})]$$

然后用它更新 DQN 的参数。用这个新的 δ'_j 取代数组中旧的 δ_j 。

¹论文里建议一开始让 β 比较小，最终增长到 1。

6.2 高估问题及解决方法

Q 学习算法有一个缺陷：用 Q 学习训练出的 DQN 会高估真实的价值，而且高估通常是非均匀的。这个缺陷导致 DQN 的表现很差。高估问题并不是 DQN 本身的缺陷，而是训练 DQN 用的 Q 学习算法的缺陷。 Q 学习产生高估的原因有两个：第一，自举导致偏差的传播；第二，最大化导致 TD 目标高估真实价值。为了缓解高估，需要从导致高估的两个原因下手，改进 Q 学习算法。双 Q 学习算法是一种有效的改进，可以大幅缓解高估及其危害。

6.2.1 自举导致偏差的传播

在强化学习中，自举意思是“用一个估算去更新同类的估算”，类似于“自己把自己给举起来”。我们在第 5.4 节讨论过 SARSA 算法中的自举。下面回顾训练 DQN 用的 Q 学习算法，研究其中存在的自举。算法每次从经验回放数组 (Replay Buffer) 中抽取一个四元组 (s_j, a_j, r_j, s_{j+1}) 。然后执行以下步骤，对 DQN 的参数做一轮更新：

1. 计算 TD 目标：

$$\hat{y}_j = r_j + \gamma \cdot \underbrace{\max_{a_{j+1} \in \mathcal{A}} Q(s_{j+1}, a_{j+1}; \mathbf{w}_{\text{now}})}_{\text{DQN 自己做出的估计}}.$$

2. 定义损失函数

$$L(\mathbf{w}) = \frac{1}{2} \left[\underbrace{Q(s_j, a_j; \mathbf{w}) - \hat{y}_j}_{\text{让 DQN 拟合 } \hat{y}_j} \right]^2.$$

3. 把 \hat{y}_j 看做常数，做一次梯度下降更新参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \nabla_{\mathbf{w}} L(\mathbf{w}_{\text{now}}).$$

第一步中的 TD 目标 \hat{y}_j 部分基于 DQN 自己做出的估计；第二步让 DQN 去拟合 \hat{y}_j 。这就意味着我们用了 DQN 自己做出的估计去更新 DQN 自己，这属于自举。

自举对 DQN 的训练有什么影响呢？ $Q(s, a; \mathbf{w})$ 是对价值 $Q_*(s, a)$ 的近似；最理想的情况下， $Q(s, a; \mathbf{w}) = Q_*(s, a)$ ， $\forall s, a$ 。假如碰巧 $Q(s_{j+1}, a_{j+1}; \mathbf{w})$ 低估（或高估）真实价值 $Q_*(s_{j+1}, a_{j+1})$ ，则会发生下面的情况：

$$\begin{aligned} & Q(s_{j+1}, a_{j+1}; \mathbf{w}) \quad \text{低估 (或高估)} \quad Q_*(s_{j+1}, a_{j+1}) \\ \Rightarrow & \hat{y}_j \quad \text{低估 (或高估)} \quad Q_*(s_j, a_j) \\ \Rightarrow & Q(s_j, a_j; \mathbf{w}) \quad \text{低估 (或高估)} \quad Q_*(s_j, a_j). \end{aligned}$$

结论 6.1. 自举导致偏差的传播

如果 $Q(s_{j+1}, a_{j+1}; \mathbf{w})$ 是对真实价值 $Q_*(s_{j+1}, a_{j+1})$ 的低估（或高估），就会导致 $Q(s_j, a_j; \mathbf{w})$ 低估（或高估）价值 $Q_*(s_j, a_j)$ 。也就是说低估（或高估）从 (s_{j+1}, a_{j+1}) 传播到 (s_j, a_j) ，让更多的价值被低估（或高估）。



6.2.2 最大化导致高估

首先用数学解释为什么最大化会导致高估。设 x_1, \dots, x_d 为任意 d 个实数。往 x_1, \dots, x_d 中加入任意均值为零的随机噪声，得到 Z_1, \dots, Z_d ，它们是随机变量，随机性来源于随机噪声。很容易证明均值为零的随机噪声不会影响均值：

$$\mathbb{E}[\text{mean}(Z_1, \dots, Z_d)] = \text{mean}(x_1, \dots, x_d).$$

用稍微复杂一点的证明，可以得到：

$$\mathbb{E}[\max(Z_1, \dots, Z_d)] \geq \max(x_1, \dots, x_d).$$

公式中的期望是关于噪声求的。这个不等式意味着先加入均值为零的噪声，然后求最大值，会产生高估。

假设对于所有的动作 $a \in \mathcal{A}$ 和状态 $s \in \mathcal{S}$ ，DQN 的输出是真实价值 $Q_*(s, a)$ 加上均值为零的随机噪声 ϵ ：

$$Q(s, a; \mathbf{w}) = Q_*(s, a) + \epsilon.$$

显然 $Q(s, a; \mathbf{w})$ 是对真实价值 $Q_*(s, a)$ 的无偏估计。然而有这个不等式：

$$\mathbb{E}_\epsilon[\max_{a \in \mathcal{A}} Q(s, a; \mathbf{w})] \geq \max_{a \in \mathcal{A}} Q_*(s, a).$$

公式说明尽管 DQN 是对真实价值的无偏估计，但如果求最大化，DQN 则会高估真实价值。复习一下，TD 目标是这样算出来的：

$$\hat{y}_j = r_j + \gamma \cdot \underbrace{\max_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w})}_{\text{高估 } \max_{a \in \mathcal{A}} Q_*(s_{j+1}, a)}.$$

这说明 TD 目标 \hat{y}_j 通常是对真实价值 $Q_*(s_j, a_j)$ 的高估。TD 算法鼓励 $Q(s_j, a_j; \mathbf{w})$ 接近 TD 目标 \hat{y}_j ，这会导致 $Q(s_j, a_j; \mathbf{w})$ 高估真实价值 $Q_*(s_j, a_j)$ 。

结论 6.2. 最大化导致高估

即使 DQN 是真实价值 Q_* 的无偏估计，只要 DQN 不恒等于 Q_* ，TD 目标就会高估真实价值。TD 目标是高估，而 Q 学习算法鼓励 DQN 预测接近 TD 目标，因此 DQN 就会出现高估。



6.2.3 高估的危害

我们为什么要避免高估？高估真的有害吗？高估本身是无害的，除非高估是非均匀的。举个例子，动作空间是 $\mathcal{A} = \{\text{左}, \text{右}, \text{上}\}$ 。给定当前状态 s ，每个动作有一个真实价值：

$$Q_*(s, \text{左}) = 200, \quad Q_*(s, \text{右}) = 100, \quad Q_*(s, \text{上}) = 230.$$

智能体应当选择动作“上”，因为“上”的价值最高。假如高估是均匀的，所有的价值都被高估了 100：

$$Q(s, \text{左}; \mathbf{w}) = 300, \quad Q(s, \text{右}; \mathbf{w}) = 200, \quad Q(s, \text{上}; \mathbf{w}) = 330.$$

6.2 高估问题及解决方法

那么动作“上”仍然有最大的价值，智能体会选择“上”。这个例子说明高估本身不是问题，只要所有动作价值被同等高估。

但实践中，所有的动作价值会被同等高估吗？每当取出一个四元组 (s, a, r, s') 用来更新一次 DQN，就很有可能加重 DQN 对 $Q_*(s, a)$ 的高估。对于同一个状态 s ，三种组合 $(s, \text{左})$ 、 $(s, \text{右})$ 、 $(s, \text{上})$ 出现在经验回放数组中的频率是不同的，所以三种动作被高估的程度是不同的。假如动作价值被高估的程度不同，比如

$$Q(s, \text{左}; \mathbf{w}) = 280, \quad Q(s, \text{右}; \mathbf{w}) = 300, \quad Q(s, \text{上}; \mathbf{w}) = 260,$$

那么智能体做出的决策就是向右走，因为“右”的价值貌似最高。但实际上“右”是最差的动作，它的实际价值低于其余两个动作。

综上所述，用 Q 学习算法训练 DQN 总会导致 DQN 高估真实价值。对于多数的 $s \in \mathcal{S}$ 和 $a \in \mathcal{A}$ ，有这样的不等式：

$$Q(s, a; \mathbf{w}) > Q_*(s, a).$$

高估本身不是问题，真正的麻烦在于 DQN 的高估往往是非均匀的。如果 DQN 有非均匀的高估，那么用 DQN 做出的决策是不可靠的。我们已经分析过导致高估的原因：

- TD 算法属于“自举”，即用 DQN 的估计值去更新 DQN 自己。自举会导致偏差的传播。如果 $Q(s_{j+1}, a_{j+1}; \mathbf{w})$ 是对 $Q_*(s_{j+1}, a_{j+1})$ 的高估，那么高估会传播到 (s_j, a_j) ，让 $Q(s_j, a_j; \mathbf{w})$ 高估 $Q_*(s_j, a_j)$ 。自举导致 DQN 的高估从一个二元组 (s, a) 传播到更多的二元组。
- TD 目标 \hat{y} 中包含一项最大化，这会导致 TD 目标高估真实价值 Q_* 。Q 学习算法鼓励 DQN 的预测接近 TD 目标，因此 DQN 会高估 Q_* 。

找到了产生高估的原因，就可以想办法解决问题。想要避免 DQN 的高估，要么切断“自举”，要么避免最大化造成高估。注意，高估并不是 DQN 自身的属性；高估纯粹是算法造成的。想要避免高估，就要用更好的算法替代原始的 Q 学习算法。

6.2.4 使用目标网络

上文已经讨论过，切断“自举”可以避免偏差的传播，从而缓解 DQN 的高估。回顾一下，Q 学习算法这样计算 TD 目标：

$$\hat{y}_j = r_t + \underbrace{\gamma \cdot \max_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w})}_{\text{DQN 做出的估计}}.$$

然后做梯度下降更新 \mathbf{w} ，使得 $Q(s_j, a_j; \mathbf{w})$ 更接近 \hat{y}_j 。想要切断自举，可以用另一个神经网络计算 TD 目标，而不是用 DQN 自己计算 TD 目标。另一个神经网络就被称作**目标网络** (Target Network)。把目标网络记作：

$$Q(s, a; \mathbf{w}^-).$$

它的神经网络结构与 DQN 完全相同，但是参数 \mathbf{w}^- 不同于 \mathbf{w} 。

使用目标网络的话，Q 学习算法的用下面的方式实现。每次随机从经验回放数组中取一个四元组，记作 (s_j, a_j, r_j, s_{j+1}) 。设 DQN 和目标网络当前的参数分别为 \mathbf{w}_{now} 和

$\mathbf{w}_{\text{now}}^-$, 执行下面的步骤对参数做一次更新:

1. 对 DQN 做正向传播, 得到:

$$\hat{q}_j = Q(s_j, a_j; \mathbf{w}_{\text{now}}).$$

2. 对目标网络做正向传播, 得到

$$\widehat{q}_{j+1} = \max_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w}_{\text{now}}^-).$$

3. 计算 TD 目标和 TD 误差:

$$\widehat{y}_j = r_j + \gamma \cdot \widehat{q}_{j+1} \quad \text{和} \quad \delta_j = \hat{q}_j - \widehat{y}_j.$$

4. 对 DQN 做反向传播, 得到梯度 $\nabla_{\mathbf{w}} Q(s_j, a_j; \mathbf{w}_{\text{now}})$ 。

5. 做梯度下降更新 DQN 的参数:

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_j \cdot \nabla_{\mathbf{w}} Q(s_j, a_j; \mathbf{w}_{\text{now}}).$$

6. 设 $\tau \in (0, 1)$ 是需要手动调的超参数。做加权平均更新目标网络的参数:

$$\mathbf{w}_{\text{new}}^- \leftarrow \tau \cdot \mathbf{w}_{\text{new}} + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^-.$$



图 6.4

如图 6.4(左)所示, 原始的 Q 学习算法用 DQN 计算 \widehat{y} , 然后拿 \widehat{y} 更新 DQN 自己, 造成自举。如图 6.4(右)所示, 可以改用目标网络计算 \widehat{y} , 这样就避免了用 DQN 的估计更新 DQN 自己, 降低自举造成的危害。然而这种方法并不可能完全避免自举, 原因是目标网络的参数仍然与 DQN 相关。

6.2.5 双 Q 学习算法

造成 DQN 高估的原因不是 DQN 模型本身的缺陷, 而是训练 DQN 所用的算法有不足之处: 第一, 自举造成偏差的传播; 第二, 最大化造成 TD 目标的高估。在 Q 学习算法中使用目标网络, 可以缓解自举造成的偏差, 但是无助于缓解最大化造成的高估。本小节介绍**双 Q 学习 (Double Q Learning)** 算法, 它在目标网络的基础上做改进, 缓解最大化造成的高估。

注 本小节介绍的双 Q 学习算法在文献中被称作 Double DQN, 缩写 DDQN。本书不采用 DDQN 这名字, 因为这个名字比较误导。双 Q 学习 (即所谓的 DDQN) 只是一种 **TD 算法**而已, 它可以把 DQN 训练得更好。双 Q 学习并没有用区别于 DQN 的**模型**。本节中的模型只有一个, 就是 DQN。而我们所讨论的只是训练 DQN 的三种 TD 算法: 原始的 Q 学习、用目标网络的 Q 学习、以及双 Q 学习。

为了解释原始 Q 学习、用目标网络的 Q 学习、以及双 Q 学习三者的区别, 我们再回

6.2 高估问题及解决方法

回顾一下 Q 学习算法中的 TD 目标：

$$\hat{y}_j = r_j + \gamma \cdot \max_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w}).$$

不妨把最大化拆成两步：

1. 选择——即基于状态 s_{j+1} , 选出一个动作使得 DQN 的输出最大化：

$$a^* = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w}).$$

2. 求值——即计算 (s_{j+1}, a^*) 的价值, 从而算出 TD 目标：

$$\hat{y}_j = r_j + Q(s_{j+1}, a^*; \mathbf{w}).$$

以上是原始的 Q 学习算法, 选择和求值都用 DQN。上一小节改进了 Q 学习, 选择和求值都用目标网络：

$$\text{选择: } a^- = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w}^-),$$

$$\text{求值: } \tilde{y}_t = r_t + Q(s_{j+1}, a^-; \mathbf{w}^-).$$

本小节介绍双 Q 学习, 第一步的选择用 DQN, 第二步的求值用目标网络：

$$\text{选择: } a^* = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w}),$$

$$\text{求值: } \tilde{y}_t = r_t + Q(s_{j+1}, a^*; \mathbf{w}^-).$$

为什么双 Q 学习可以缓解最大化造成的高估呢? 不难证明出这个不等式:

$$\underbrace{Q(s_{j+1}, a^*; \mathbf{w}^-)}_{\text{双 Q 学习}} \leq \underbrace{\max_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w}^-)}_{\text{用目标网络的 Q 学习}}.$$

因此,

$$\underbrace{\tilde{y}_t}_{\text{双 Q 学习}} \leq \underbrace{\hat{y}_t}_{\text{用目标网络的 Q 学习}}.$$

这个公式说明双 Q 学习得到的 TD 目标更小; 也就是说, 与用目标网络的 Q 学习相比, 双 Q 学习缓解了高估。

双 Q 学习算法的流程如下。每次随机从经验回放数组中取出一个四元组, 记作 (s_j, a_j, r_j, s_{j+1}) 。设 DQN 和目标网络当前的参数分别为 \mathbf{w}_{now} 和 $\mathbf{w}_{\text{now}}^-$, 执行下面的步骤对参数做一次更新:

1. 对 DQN 做正向传播, 得到:

$$\hat{q}_j = Q(s_j, a_j; \mathbf{w}_{\text{now}}).$$

2. 选择:

$$a^* = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w}_{\text{now}}).$$

3. 求值:

$$\hat{q}_{j+1} = Q(s_{j+1}, a^*; \mathbf{w}_{\text{now}}^-).$$

4. 计算 TD 目标和 TD 误差:

$$\tilde{y}_j = r_j + \gamma \cdot \hat{q}_{j+1} \quad \text{和} \quad \delta_j = \hat{q}_j - \tilde{y}_j.$$

5. 对 DQN 做反向传播，得到梯度 $\nabla_{\mathbf{w}} Q(s_j, a_j; \mathbf{w}_{\text{now}})$ 。

6. 做梯度下降更新 DQN 的参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_j \cdot \nabla_{\mathbf{w}} Q(s_j, a_j; \mathbf{w}_{\text{now}}).$$

7. 设 $\tau \in (0, 1)$ 是需要手动调的超参数。做加权平均更新目标网络的参数：

$$\mathbf{w}_{\text{new}}^- \leftarrow \tau \cdot \mathbf{w}_{\text{new}} + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^-.$$

6.2.6 总结

本节研究了 DQN 的高估问题以及解决方案。DQN 的高估不是 DQN 模型造成的，不是 DQN 的本质属性；高估只是因为原始 Q 学习算法不好。Q 学习算法产生高估的原因有两个：第一，自举导致偏差从一个 (s, a) 二元组传播到更多的二元组；第二，最大化造成 TD 目标高估真实价值。

想要解决高估问题，就要从自举、最大化这两方面下手。本节介绍了两种缓解高估的思路：使用目标网络、双 Q 学习。Q 学习算法与目标网络的结合可以缓解自举造成的偏差。双 Q 学习基于目标网络的想法，进一步将 TD 目标的计算分解成选择和求值两步，缓解了最大化造成的高估。图 6.5 总结了本节研究的三种算法。

	选择	求值	自举造成偏差	最大化造成高估
原始 Q 学习	DQN	DQN	严重	严重
Q 学习 + 目标网络	目标网络	目标网络	不严重	严重
双 Q 学习	DQN	目标网络	不严重	不严重

图 6.5：三种 TD 算法的对比。

注 如果使用原始 Q 学习算法，自举和最大化的麻烦都会出现。在实践中，应当尽量使用双 Q 学习，它是三种算法中最好的。

注 如果使用 SARSA 算法（比如在 Actor-Critic 方法中），自举的问题依然存在，但是不存在最大化造成高估这一问题。对于 SARSA，只需要解决自举问题，所以应当将目标网络应用到 SARSA。

6.3 对决网络 (Dueling Network)

本节介绍对决网络 (Dueling Network)，它是对 DQN 的神经网络的结构的改进。它的基本想法是将最优动作价值 Q_* 分解成最优状态价值 V_* 与最优势 D_* 。对决网络的训练与 DQN 完全相同，可以用 Q 学习算法或者双 Q 学习算法。

6.3.1 最优优势函数

在介绍对决网络 (Dueling Network) 之前，先复习一些基础知识。动作价值函数 $Q_\pi(s, a)$ 是回报的期望：

$$Q_\pi(s, a) = \mathbb{E}[U_t \mid S_t = s, A_t = a].$$

最优动作价值 Q_* 的定义是：

$$Q_*(s, a) = \max_{\pi} Q_\pi(s, a), \quad \forall s \in \mathcal{S}, a \in \mathcal{A}.$$

状态价值函数 $V_\pi(s)$ 是 $Q_\pi(s, a)$ 关于 a 的期望：

$$V_\pi(s) = \mathbb{E}_{A \sim \pi}[Q_\pi(s, A)].$$

最优状态价值函数 V_* 的定义是：

$$V_*(s) = \max_{\pi} V_\pi(s), \quad \forall s \in \mathcal{S}.$$

最优势函数 (Optimal Advantage Function) 的定义是：

$$D_*(s, a) \triangleq Q_*(s, a) - V_*(s).$$

通过数学推导，可以证明下面的定理：

定理 6.1

$$Q_*(s, a) = V_*(s) + D_*(s, a) - \underbrace{\max_{a \in \mathcal{A}} D_*(s, a)}_{\text{恒等于零}}, \quad \forall s \in \mathcal{S}, a \in \mathcal{A}.$$



6.3.2 对决网络

与 DQN 一样，对决网络 (Dueling Network) 也是对最优动作价值函数 Q_* 的近似。对决网络与 DQN 的区别在于神经网络结构不同。由于对决网络与 DQN 都是对 Q_* 的近似，可以用完全相同的算法训练两种神经网络。

对决网络由两个神经网络组成。一个神经网络记作 $D(s, a; \mathbf{w}^D)$ ，它是对最优势函数 $D_*(s, a)$ 的近似。另一个神经网络记作 $V(s; \mathbf{w}^V)$ ，它是对最优状态价值函数 $V_*(s)$ 的近似。把定理 6.1 中的 D_* 和 V_* 替换成相应的神经网络，那么最优动作价值函数 Q_* 就被近似成下面的神经网络：

$$Q(s, a; \mathbf{w}) \triangleq V(s; \mathbf{w}^V) + D(s, a; \mathbf{w}^D) - \max_{a \in \mathcal{A}} D(s, a; \mathbf{w}^D). \quad (6.1)$$

公式左边的 $Q(s, a; \mathbf{w})$ 就是对决网络，它是对最优动作价值函数 Q_* 的近似。它的参数记作 $\mathbf{w} \triangleq (\mathbf{w}^V; \mathbf{w}^D)$ 。

对决网络的结构如图 6.6 所示。可以让两个神经网络 $D(s, a; \mathbf{w}^D)$ 与 $V(s; \mathbf{w}^V)$ 共享部分卷积层；这些卷积层把输入的状态 s 映射成特征向量，特征向量是“优势头”与“状态价值头”的输入。优势头输出一个向量，向量的维度是动作空间的大小 $|\mathcal{A}|$ ，向量每个元素对应一个动作。举个例子，动作空间是 $\mathcal{A} = \{\text{左, 右, 上}\}$ ，优势头的输出是三个值：

$$D(s, \text{左}; \mathbf{w}^D) = -90, \quad D(s, \text{右}; \mathbf{w}^D) = -420, \quad D(s, \text{上}; \mathbf{w}^D) = 30.$$

状态价值头输出的是一个实数，比如

$$V(s; \mathbf{w}^V) = 300.$$

首先计算

$$\max_a D(s, a; \mathbf{w}^D) = \max \{-90, -420, 30\} = 30.$$

然后用公式 (6.1) 计算出：

$$Q(s, \text{左}; \mathbf{w}) = 180, \quad Q(s, \text{右}; \mathbf{w}) = -150, \quad Q(s, \text{上}; \mathbf{w}) = 300.$$

这样就得到了对决网络的最终输出。

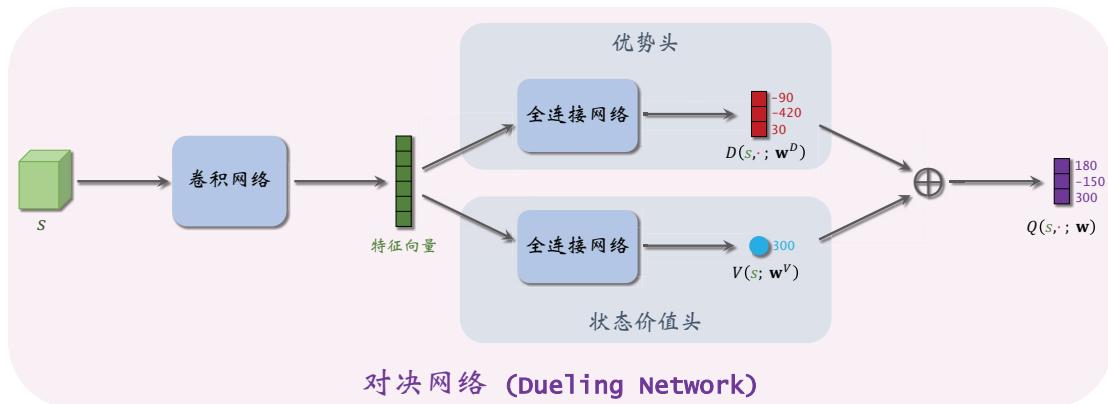


图 6.6: 对决网络的结构。输入是状态 s ；红色的向量是每个动作的优势值；蓝色的标量是状态价值；最终输出的紫色向量是每个动作的动作价值。

6.3.3 解决不唯一性

读者可能会有下面的疑问。对决网络是由定理 6.1 推导出的，而定理中最右的一项恒等于零：

$$\max_{a \in \mathcal{A}} D_*(s, a) = 0, \quad \forall s \in \mathcal{S}.$$

也就是说，可以把最优动作价值写成两种等价形式：

$$Q_*(s, a) = V_*(s) + D_*(s, a) \quad (\text{第一种形式})$$

$$= V_*(s) + D_*(s, a) - \max_{a \in \mathcal{A}} D_*(s, a). \quad (\text{第二种形式})$$

6.3 对决网络 (Dueling Network)

之前我们根据第二种形式实现对决网络。我们可否根据第一种形式，把对决网络按照下面的方式实现呢：

$$Q(s, a; \mathbf{w}) = V(s; \mathbf{w}^V) + D(s, a; \mathbf{w}^D)?$$

答案是不可以这样实现对决网络，因为这样会导致不唯一性。假如这样实现对决网络，那么 V 和 D 可以随意上下波动，比如一个增大 100，另一个减小 100：

$$\begin{aligned} V(s; \tilde{\mathbf{w}}^V) &\triangleq V(s; \mathbf{w}^V) + 100, \\ D(s, a; \tilde{\mathbf{w}}^D) &\triangleq D(s, a; \mathbf{w}^D) - 100. \end{aligned}$$

这样的上下波动不影响最终的输出：

$$V(s; \mathbf{w}^V) + D(s, a; \mathbf{w}^D) = V(s; \tilde{\mathbf{w}}^V) + D(s, a; \tilde{\mathbf{w}}^D).$$

这就意味着 V 和 D 的参数可以很随意地变化，却不会影响输出的 Q 。我们不希望这种情况出现，因为这会导致训练的过程中参数不稳定。

因此很有必要在对决网络中加入 $\max_{a \in \mathcal{A}} D(s, a; \mathbf{w}^D)$ 这一项。它使得 V 和 D 不能随意上下波动。假如让 V 变大 100，让 D 变小 100，则对决网络的输出会增大 100，而非不变：

$$\begin{aligned} &V(s; \tilde{\mathbf{w}}^V) + D(s, a; \tilde{\mathbf{w}}^D) - \max_a D(s, a; \tilde{\mathbf{w}}^D) \\ &= V(s; \mathbf{w}^V) + D(s, a; \mathbf{w}^D) - \max_a D(s, a; \mathbf{w}^D) + 100. \end{aligned}$$

以上讨论说明了为什么 $\max_{a \in \mathcal{A}} D(s, a; \mathbf{w}^D)$ 这一项不能省略。

6.3.4 对决网络的实际实现

按照定理 6.1，对决网络应该定义成：

$$Q(s, a; \mathbf{w}) \triangleq V(s; \mathbf{w}^V) + D(s, a; \mathbf{w}^D) - \max_{a \in \mathcal{A}} D(s, a; \mathbf{w}^D).$$

最右边的 \max 项的目的是解决不唯一性。实际实现的时候，用 mean 代替 \max 会有更好的效果。所以实际上会这样定义对决网络：

$$Q(s, a; \mathbf{w}) \triangleq V(s; \mathbf{w}^V) + D(s, a; \mathbf{w}^D) - \text{mean}_{a \in \mathcal{A}} D(s, a; \mathbf{w}^D).$$

对决网络与 DQN 都是对最优动作价值函数 Q_* 的近似，所以对决网络与 DQN 的训练和决策是完全一样的。比如可以这样训练对决网络：

- 用 ϵ -greedy 算法控制智能体，收集经验，把 (s_j, a_j, r_j, s_{j+1}) 这样的四元组存入经验回放数组。
- 从数组里随机抽取四元组，用双 Q 学习算法更新对决网络参数 $\mathbf{w} = (\mathbf{w}^D, \mathbf{w}^V)$ 。

完成训练之后，基于当前状态 s_t ，让对决网络给所有动作打分，然后选择分数最高的动作：

$$a_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q(s_t, a; \mathbf{w}).$$

简而言之，怎么样训练 DQN，就怎么样训练对决网络；怎么样用 DQN 做控制，就怎么

样用对决网络做控制。如果一个技巧能改进 DQN 的训练，这个技巧也能改进对决网络。同样的道理，Q 学习算法导致 DQN 出现高估，同样也会导致对决网络出现高估。

6.4 噪声网络

本节介绍噪声网络 (Noisy Net)，这是一种非常简单的方法，可以显著提高 DQN 的表现。噪声网络的应用不局限于 DQN，它可用于几乎所有的强化学习方法。

6.4.1 噪声网络的原理

把神经网络中的参数 w 替换成 $\mu + \sigma \circ \xi$ 。此处的 μ 、 σ 、 ξ 的形状与 w 完全相同。 μ 、 σ 分别表示均值和标准差，它们是神经网络的参数，需要从经验中学习。 ξ 是随机噪声，它的每个元素独立从标准正态分布 $\mathcal{N}(0, 1)$ 中随机抽取。符号 “ \circ ” 表示逐项乘积。如果 w 是向量，那么有

$$w_i = \mu_i + \sigma_i \cdot \xi_i.$$

如果 w 是矩阵，那么有

$$w_{ij} = \mu_{ij} + \sigma_{ij} \cdot \xi_{ij}.$$

噪声网络的意思是参数 w 的每个元素 w_i 从均值为 μ_i 、标准差为 σ_i 的正态分布中抽取。

举个例子，某一个全连接层记作：

$$z = \text{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b}).$$

公式中的向量 \mathbf{x} 是输入，矩阵 \mathbf{W} 和向量 \mathbf{b} 是参数，ReLU 是激活函数， z 是这一层的输出。噪声网络把这个全连接层替换成：

$$z = \text{ReLU}\left((\mathbf{W}^\mu + \mathbf{W}^\sigma \circ \mathbf{W}^\xi)\mathbf{x} + (\mathbf{b}^\mu + \mathbf{b}^\sigma \circ \mathbf{b}^\xi)\right).$$

公式中的 \mathbf{W}^μ 、 \mathbf{W}^σ 、 \mathbf{b}^μ 、 \mathbf{b}^σ 是参数，需要从经验中学习。矩阵 \mathbf{W}^ξ 和向量 \mathbf{b}^ξ 的每个元素都是独立从 $\mathcal{N}(0, 1)$ 中随机抽取的，表示噪声。

训练噪声网络的方法与训练标准的神经网络完全相同，都是做反向传播计算梯度，然后用梯度更新神经参数。把损失函数记作 L 。已知梯度 $\frac{\partial L}{\partial z}$ ，可以用链式法则算出损失关于参数的梯度：

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{W}^\mu} &= \frac{\partial z}{\partial \mathbf{W}^\mu} \cdot \frac{\partial L}{\partial z}, & \frac{\partial L}{\partial \mathbf{b}^\mu} &= \frac{\partial z}{\partial \mathbf{b}^\mu} \cdot \frac{\partial L}{\partial z}, \\ \frac{\partial L}{\partial \mathbf{W}^\sigma} &= \frac{\partial z}{\partial \mathbf{W}^\sigma} \cdot \frac{\partial L}{\partial z}, & \frac{\partial L}{\partial \mathbf{b}^\sigma} &= \frac{\partial z}{\partial \mathbf{b}^\sigma} \cdot \frac{\partial L}{\partial z}. \end{aligned}$$

然后可以做梯度下降更新参数 \mathbf{W}^μ 、 \mathbf{W}^σ 、 \mathbf{b}^μ 、 \mathbf{b}^σ 。

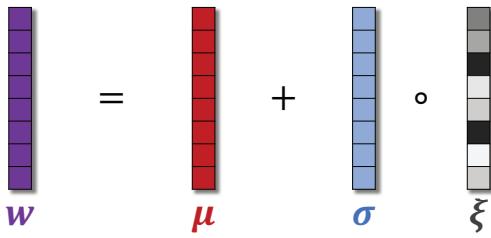


图 6.7：这个例子中， w 、 μ 、 σ 、 ξ 是形状相同的向量。

6.4.2 噪声 DQN

噪声网络可以用于 DQN。标准的 DQN 记作 $Q(s, a; \mathbf{w})$, 其中的 \mathbf{w} 表示参数。把 \mathbf{w} 替换成 $\mu + \sigma \circ \xi$, 得到噪声 DQN, 记作:

$$\tilde{Q}(s, a, \xi; \mu, \sigma) \triangleq Q(s, a; \mu + \sigma \circ \xi).$$

其中的 μ 和 σ 是参数, 一开始随机初始化, 然后从经验中学习; 而 ξ 则是随机生成, 每个元素都从 $\mathcal{N}(0, 1)$ 中抽取。噪声 DQN 的参数数量比标准 DQN 多一倍。

收集经验: DQN 属于异策略 (Off-policy)。我们用任意的行为策略 (Behavior Policy) 控制智能体, 收集经验, 事后做经验回放更新参数。在之前章节中, 我们用 ϵ -Greedy 作为行为策略:

$$a_t = \begin{cases} \operatorname{argmax}_{a \in \mathcal{A}} Q(s_t, a; \mathbf{w}), & \text{以概率 } (1 - \epsilon); \\ \text{均匀抽取 } \mathcal{A} \text{ 中的一个动作,} & \text{以概率 } \epsilon. \end{cases}$$

ϵ -Greedy 策略带有一定的随机性, 可以让智能体尝试更多动作, 探索更多状态。

噪声 DQN 本身就带有随机性, 可以鼓励探索, 起到 ϵ -Greedy 策略相同的作用。我们直接用

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} \tilde{Q}(s, a, \xi; \mu, \sigma)$$

作为行为策略, 效果比 ϵ -Greedy 更好。每做一个决策, 要重新随机生成一个 ξ 。

Q 学习算法: 训练的时候, 每一轮从经验回放数组中随机抽样出一个四元组, 记作 (s_j, a_j, r_j, s_{j+1}) 。从标准正态分布中做抽样, 得到 ξ' 的每一个元素。计算 TD 目标:

$$\hat{y}_j = r_j + \gamma \cdot \max_{a \in \mathcal{A}} \tilde{Q}(s_{j+1}, a, \xi'; \mu, \sigma).$$

把损失函数记作:

$$L(\mu, \sigma) = \frac{1}{2} [\tilde{Q}(s_j, a_j, \xi; \mu, \sigma) - \hat{y}_j]^2,$$

其中的 ξ 也是随机生成的噪声, 但是它与 ξ' 不同。然后做梯度下降更新参数:

$$\mu \leftarrow \mu - \alpha_\mu \cdot \nabla_\mu L(\mu, \sigma), \quad \sigma \leftarrow \sigma - \alpha_\sigma \cdot \nabla_\sigma L(\mu, \sigma).$$

公式中的 α_μ 和 α_σ 是学习率。这样做梯度下降更新参数, 可以让损失函数减小, 让噪声 DQN 的预测更接近 TD 目标。

做决策: 做完训练之后, 可以用噪声 DQN 做决策。做决策的时候不再需要噪声, 因此可以把参数 σ 设置成全零, 只保留参数 μ 。这样一来, 噪声 DQN 就变成标准的 DQN:

$$\underbrace{\tilde{Q}(s, a, \xi'; \mu, 0)}_{\text{噪声 DQN}} = \underbrace{Q(s, a; \mu)}_{\text{标准 DQN}}.$$

在训练的时候往 DQN 的参数中加入噪声, 不仅有利于探索, 还能增强鲁棒性。鲁棒性的意思是即使参数被扰动, DQN 也能对动作价值 Q_* 做出可靠的估计。为什么噪声可以让 DQN 有更强的鲁棒性呢?

假设在训练的过程中不加入噪声。把学出的参数记作 μ 。当参数严格等于 μ 的时候, DQN 可以对最优动作价值做出较为准确的估计。但是对 μ 做较小的扰动, 就可能会让

DQN 的输出偏离很远。所谓“失之毫厘，谬以千里”。

噪声 DQN 训练的过程中，参数带有噪声： $w = \mu + \sigma \circ \xi$ 。训练迫使 DQN 在参数带有噪声的情况下最小化 TD 误差，也就是迫使 DQN 容忍对参数的扰动。训练出的 DQN 具有鲁棒性：参数不严格等于 μ 也没关系，只要参数在 μ 的邻域内，DQN 做出的预测都应该比较合理。用噪声 DQN，不会出现“失之毫厘，谬以千里”。

6.4.3 训练流程

实际编程实现 DQN 的时候，应该将本章的四种技巧——优先经验回放、双 Q 学习、对决网络、噪声 DQN——全部用到。应该用**对决网络**的神经网络结构，而不是简单的 DQN 结构。往对决网络中的参数 w 中加入噪声，得到噪声 DQN，记作 $\tilde{Q}(s, a, \xi; \mu, \sigma)$ 。训练要用**双 Q 学习、优先经验回放**，而不是原始的 Q 学习。双 Q 学习需要目标网络 $\tilde{Q}(s, a, \xi; \mu^-, \sigma^-)$ 计算 TD 目标。它跟噪声 DQN 的结构相同，但是参数不同。

初始的时候，随机初始化 μ, σ ，并且把它们赋值给目标网络参数： $\mu^- \leftarrow \mu, \sigma^- \leftarrow \sigma$ 。然后重复下面的步骤更新参数。把当前的参数记作 $\mu_{\text{now}}, \sigma_{\text{now}}, \mu_{\text{now}}^-, \sigma_{\text{now}}^-$ 。

1. 用优先经验回放，从数组中抽取一个四元组，记作 (s_j, a_j, r_j, s_{j+1}) 。
2. 用标准正态分布生成 ξ 。对噪声 DQN 做正向传播，得到：

$$\hat{q}_j = \tilde{Q}(s_j, a_j, \xi; \mu_{\text{now}}, \sigma_{\text{now}}).$$

3. 根据噪声 DQN 选出最优动作：

$$\tilde{a}_{j+1} = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \tilde{Q}(s_{j+1}, a, \xi; \mu_{\text{now}}, \sigma_{\text{now}}).$$

4. 用标准正态分布生成 ξ' 。根据目标网络计算价值：

$$\widehat{q}_{j+1} = \tilde{Q}(s_{j+1}, \tilde{a}_{j+1}, \xi'; \mu_{\text{now}}^-, \sigma_{\text{now}}^-).$$

5. 计算 TD 目标和 TD 误差：

$$\widehat{y}_j = r_j + \gamma \cdot \widehat{q}_{j+1} \quad \text{和} \quad \delta_j = \widehat{q}_j - \widehat{y}_j.$$

6. 设 α_μ 和 α_σ 为学习率。做梯度下降更新噪声 DQN 的参数：

$$\begin{aligned} \mu_{\text{new}} &\leftarrow \mu_{\text{now}} - \alpha_\mu \cdot \delta_j \cdot \nabla_\mu \tilde{Q}(s_j, a_j, \xi; \mu_{\text{now}}, \sigma_{\text{now}}), \\ \sigma_{\text{new}} &\leftarrow \sigma_{\text{now}} - \alpha_\sigma \cdot \delta_j \cdot \nabla_\sigma \tilde{Q}(s_j, a_j, \xi; \mu_{\text{now}}, \sigma_{\text{now}}). \end{aligned}$$

7. 设 $\tau \in (0, 1)$ 是需要手动调的超参数。做加权平均更新目标网络的参数：

$$\begin{aligned} \mu_{\text{new}}^- &\leftarrow \tau \cdot \mu_{\text{new}} + (1 - \tau) \cdot \mu_{\text{now}}^-, \\ \sigma_{\text{new}}^- &\leftarrow \tau \cdot \sigma_{\text{new}} + (1 - \tau) \cdot \sigma_{\text{now}}^-. \end{aligned}$$

∽第六章 相关文献∽

训练 DQN 用到的经验回放是由 Lin 在 1993 年的博士论文 [62] 中提出的。优先经验回放是由 Schaul 等人 2015 年的论文 [86] 提出。目标网络由 Mnih 等人 2015 年的论文 [70] 提出。双 Q 学习由 van Hasselt 2010 年的论文 [107] 提出。双 Q 学习与 DQN 的结合被称为 Double DQN，由 van Hasselt 等人 2010 年的论文提出 [108]。对决网络在 Wang 等人 2016 年的论文中提出 [113]。噪声网络在 Fortunato 等人 2018 年的论文中提出 [39]。

Hessel 等人在 2018 年发表的论文 [46] 将优先经验回放、双 Q 学习、对决网络、多步 TD 目标等方法结合，改进 DQN，把组成称为 Rainbow，用实验证明高级技巧的有效性。此外，Rainbow 还用到了 Distributional Learning [12]，这种技巧也非常有用。

第七章 策略梯度方法

本章的内容是策略学习 (Policy-Based Reinforcement Learning) 以及策略梯度 (Policy Gradient)。策略学习的意思是通过求解一个优化问题，学出最优策略函数或它的近似（比如策略网络）。第 7.1 节描述策略网络。第 7.2 节把策略学习描述成一个最大化的问题。第 7.3 节推导策略梯度。第 7.4 和 7.5 节用不同的方法近似策略梯度，得到两种训练策略网络的方法——REINFORCE 和 Actor-Critic。本章介绍的 REINFORCE 和 Actor-Critic 只是帮助大家理解算法而已，实际效果并不好。在实践中不建议用本章原始的方法，而应该用下一章的方法。

7.1 策略网络

本章考虑离散动作空间，比如 $\mathcal{A} = \{\text{左}, \text{右}, \text{上}\}$ 。策略函数 π 是个条件概率质量函数：

$$\pi(a | s) \triangleq \mathbb{P}(A = a | S = s).$$

策略函数 π 的输入是状态 s 和动作 a ，输出是一个 0 到 1 之间的概率值。举个例子，把超级玛丽游戏当前屏幕上的画面作为 s ，策略函数会输出每个动作的概率值：

$$\pi(\text{左} | s) = 0.5,$$

$$\pi(\text{右} | s) = 0.2,$$

$$\pi(\text{上} | s) = 0.3.$$

如果我们有这样一个策略函数，我们就可以拿它控制智能体。每当观测到一个状态 s ，就用策略函数计算出每个动作的概率值，然后做随机抽样，得到一个动作 a ，让智能体执行 a 。

怎么样才能得到这样一个策略函数呢？当前最有效的方法是用神经网络 $\pi(a|s; \theta)$ 近似策略函数 $\pi(a|s)$ 。神经网络 $\pi(a|s; \theta)$ 被称为策略网络。 θ 表示神经网络的参数；一开始随机初始化 θ ，随后利用收集的状态、动作、奖励去更新 θ 。



图 7.1：策略网络 $\pi(a|s; \theta)$ 的神经网络结构。输入是状态 s ，输出是动作空间 \mathcal{A} 中每个动作的概率值。

策略网络的结构如图 7.1 所示。策略网络的输入是状态 s 。在 Atari 游戏、围棋等应用中，状态是张量（比如图片），那么应该如图 7.1 所示用卷积网络处理输入。在机器人控制等应用中，状态 s 是向量，它的元素是多个传感器的数值，那么应该把卷积网络换成

全连接网络。策略网络输出层的激活函数是 Softmax，因此输出的向量（记作 f ）所有元素都是正数，而且相加等于 1。动作空间 \mathcal{A} 的大小是多少，向量 f 的维度就是多少。在超级玛丽的例子中， $\mathcal{A} = \{\text{左, 右, 上}\}$ ，那么 f 就是 3 维的向量，比如 $f = [0.2, 0.1, 0.7]$ 。 f 描述了动作空间 \mathcal{A} 上的离散概率分布， f 每个元素对应一个动作：

$$\begin{aligned}f_1 &= \pi(\text{左} | s) = 0.2, \\f_2 &= \pi(\text{右} | s) = 0.1, \\f_3 &= \pi(\text{上} | s) = 0.7.\end{aligned}$$

7.2 策略学习的目标函数

为了推导策略学习的目标函数，我们需要先复习回报和价值函数。回报 U_t 是从 t 时刻开始的所有奖励之和。 U_t 依赖于 t 时刻开始的所有状态和动作：

$$S_t, A_t, S_{t+1}, A_{t+1}, S_{t+2}, A_{t+2}, \dots$$

在 t 时刻， U_t 是随机变量，它的不确定性来自于未来未知的状态和动作。动作价值函数的定义是：

$$Q_\pi(s_t, a_t) = \mathbb{E}[U_t | S_t = s_t, A_t = a_t].$$

条件期望把 t 时刻状态 s_t 和动作 a_t 看做已知观测值，把 $t + 1$ 时刻后的状态和动作看做未知变量，并消除这些变量。状态价值函数的定义是

$$V_\pi(s_t) = \mathbb{E}_{A_t \sim \pi(\cdot | s_t; \theta)}[Q_\pi(s_t, A_t)].$$

状态价值既依赖于当前状态 s_t ，也依赖于策略网络 π 的参数 θ 。

- 当前状态 s_t 越好，则 $V_\pi(s_t)$ 越大，也就是回报 U_t 的期望越大。例如，在超级玛丽游戏中，如果玛丽奥已经接近终点（也就是说当前状态 s_t 很好），那么回报的期望就会很大。
- 策略 π 越好（即参数 θ 越好），那么 $V_\pi(s_t)$ 也会越大。例如，从同一起点出发打游戏，高手（好的策略）的期望回报远高于初学者（差的策略）。

如果一个策略很好，那么对于所有的状态 S ，状态价值 $V_\pi(S)$ 的均值应当很大。因此我们定义目标函数：

$$J(\theta) = \mathbb{E}_S[V_\pi(S)].$$

这个目标函数排除掉了状态 S 的因素，只依赖于策略网络 π 的参数 θ ；策略越好，则 $J(\theta)$ 越大。所以策略学习可以描述为这样一个优化问题：

$$\max_{\theta} J(\theta).$$

我们希望通过策略网络参数 θ 的更新，使得目标函数 $J(\theta)$ 越来越大，也就意味着策略网络越来越强。想要求解最大化问题，显然可以用梯度上升更新 θ ，使得 $J(\theta)$ 增大。设当前策略网络的参数为 θ_{now} 。做梯度上升更新参数，得到新的参数 θ_{new} ：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \nabla_{\theta} J(\theta_{\text{now}}).$$

此处的 β 是学习率，需要手动调。上面的公式就是训练策略网络的基本想法，其中的梯度

$$\nabla_{\theta} J(\theta_{\text{now}}) \triangleq \frac{\partial J(\theta)}{\partial \theta} \Big|_{\theta=\theta_{\text{now}}}$$

被称作策略梯度。策略梯度可以写成下面定理中的期望形式。之后的算法推导都要基于这个定理，并对其中的期望做近似。

定理 7.1. 策略梯度定理（不严谨的表述）

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_S \left[\mathbb{E}_{A \sim \pi(\cdot|S;\theta)} \left[\frac{\partial \ln \pi(A|S;\theta)}{\partial \theta} \cdot Q_\pi(S, A) \right] \right].$$



注 上面是策略梯度定理是不严谨的表述，尽管大多数论文和书籍使用这种表述。严格地讲，这个定理只有在“状态 S 服从马尔科夫链的稳态分布 $d(\cdot)$ ”这个假设下才成立。定理中的等号其实是不对的，期望前面应该有一项系数 $1 + \gamma + \dots + \gamma^{n-1} = \frac{1-\gamma^n}{1-\gamma}$ ，其中 γ 是折扣率， n 是一局游戏的长度。策略梯度定理应该是：

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1-\gamma^n}{1-\gamma} \cdot \mathbb{E}_{S \sim d(\cdot)} \left[\mathbb{E}_{A \sim \pi(\cdot|S;\theta)} \left[\frac{\partial \ln \pi(A|S;\theta)}{\partial \theta} \cdot Q_\pi(S, A) \right] \right].$$

在实际应用中，系数 $\frac{1-\gamma^n}{1-\gamma}$ 无关紧要，可以忽略掉。其原因是做梯度上升的时候，系数 $\frac{1-\gamma^n}{1-\gamma}$ 会被学习率 β 吸收。

7.3 策略梯度定理的证明

策略梯度定理是策略学习的关键所在。本节的内容是证明策略梯度定理。尽管本节数学较多，但还是建议读者认真读完第 7.3.1 小节，理解策略梯度简化的推导。第 7.3.2 小节是策略梯度定理完整的证明。由于完整证明较为复杂，大多数教材中不涉及这部分内容，本书也不建议读者理解、掌握完整证明，除非读者从事强化学习科研。

7.3.1 简化的证明

把策略网络 $\pi(a | s; \theta)$ 看做动作的概率质量函数（或概率密度函数）。状态价值函数 $V_\pi(s)$ 可以写成：

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_{A \sim \pi(\cdot | s; \theta)} [Q_\pi(s, A)] \\ &= \sum_{a \in \mathcal{A}} \pi(a | s; \theta) \cdot Q_\pi(s, a). \end{aligned}$$

状态价值 $V_\pi(s)$ 关于 θ 的梯度可以写作：

$$\begin{aligned} \frac{\partial V_\pi(s)}{\partial \theta} &= \frac{\partial}{\partial \theta} \sum_{a \in \mathcal{A}} \pi(a | s; \theta) \cdot Q_\pi(s, a) \\ &= \sum_{a \in \mathcal{A}} \frac{\partial \pi(a | s; \theta) \cdot Q_\pi(s, a)}{\partial \theta}. \end{aligned} \quad (7.1)$$

上面第二个等式把求导放入连加里面；等式成立的原因是求导的对象 θ 与连加的对象 a 不同。回忆一下链式法则：设 $z = f(x) \cdot g(x)$ ，那么

$$\frac{\partial z}{\partial x} = \frac{\partial f(x)}{\partial x} \cdot g(x) + f(x) \cdot \frac{\partial g(x)}{\partial x}.$$

应用链式法则，公式 (7.1) 中的梯度可以写作：

$$\begin{aligned} \frac{\partial V_\pi(s)}{\partial \theta} &= \sum_{a \in \mathcal{A}} \frac{\partial \pi(a | s; \theta)}{\partial \theta} \cdot Q_\pi(s, a) + \sum_{a \in \mathcal{A}} \pi(a | s; \theta) \cdot \frac{\partial Q_\pi(s, a)}{\partial \theta} \\ &= \sum_{a \in \mathcal{A}} \frac{\partial \pi(a | s; \theta)}{\partial \theta} \cdot Q_\pi(s, a) + \underbrace{\mathbb{E}_{A \sim \pi(\cdot | s; \theta)} \left[\frac{\partial Q_\pi(s, A)}{\partial \theta} \right]}_{\text{设为 } x}. \end{aligned}$$

上面公式最右边一项 x 的分析非常复杂，此处不具体分析了。由上面的公式可得：

$$\begin{aligned} \frac{\partial V_\pi(s)}{\partial \theta} &= \sum_{A \in \mathcal{A}} \frac{\partial \pi(A | S; \theta)}{\partial \theta} \cdot Q_\pi(S, A) + x \\ &= \sum_{A \in \mathcal{A}} \pi(A | S; \theta) \cdot \underbrace{\frac{1}{\pi(A | S; \theta)} \cdot \frac{\partial \pi(A | S; \theta)}{\partial \theta}}_{\text{等于 } \partial \ln \pi(A | S; \theta) / \partial \theta} \cdot Q_\pi(S, A) + x. \end{aligned}$$

上面第二个等式成立的原因是添加的两个红色项相乘等于一。公式中用下花括号标出的项等于 $\frac{\partial \ln \pi(A | S; \theta)}{\partial \theta}$ 。由此可得

$$\begin{aligned} \frac{\partial V_\pi(s)}{\partial \theta} &= \sum_{A \in \mathcal{A}} \pi(A | S; \theta) \cdot \frac{\partial \ln \pi(A | S; \theta)}{\partial \theta} \cdot Q_\pi(S, A) + x \\ &= \mathbb{E}_{A \sim \pi(\cdot | S; \theta)} \left[\frac{\partial \ln \pi(A | S; \theta)}{\partial \theta} \cdot Q_\pi(S, A) \right] + x. \end{aligned} \quad (7.2)$$

公式中红色标出的 $\pi(A|S; \theta)$ 被看做概率质量函数，因此连加可以写成期望的形式。根据目标函数的定义 $J(\theta) = \mathbb{E}_S[V_\pi(S)]$ 可得

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta} &= \mathbb{E}_S\left[\frac{\partial V_\pi(S)}{\partial \theta}\right] \\ &= \mathbb{E}_S\left[\mathbb{E}_{A \sim \pi(\cdot|S; \theta)}\left[\frac{\partial \ln \pi(A|S; \theta)}{\partial \theta} \cdot Q_\pi(S, A)\right]\right] + \mathbb{E}_S[x].\end{aligned}$$

不严谨的证明通常忽略掉 x ，于是得到定理 7.1。在下一小节中，我们给出严格的证明。除非读者对强化学习的数学很感兴趣，否则没必要阅读下一小节。

7.3.2 完整的证明

本小节给出策略梯度定理的严格数学证明。首先证明几个引理，最后用引理证明策略梯度定理。引理 7.2 分析梯度 $\frac{\partial V_\pi(s)}{\partial \theta}$ ，并把它递归地表示为 $\frac{\partial V_\pi(S')}{\partial \theta}$ 的期望，其中 S' 是下一时刻的状态。

引理 7.2. 递归公式

$$\frac{\partial V_\pi(s)}{\partial \theta} = \mathbb{E}_{A \sim \pi(\cdot|s; \theta)}\left[\frac{\partial \ln \pi(A|s; \theta)}{\partial \theta} \cdot Q_\pi(s, A) + \gamma \cdot \mathbb{E}_{S' \sim p(\cdot|s, A)}\left[\frac{\partial V_\pi(S')}{\partial \theta}\right]\right].$$



证明 设奖励 R 和新状态 S' 是在智能体执行动作 A 之后由环境给出的。新状态 S' 的概率密度函数是状态转移函数 $p(S'|S, A)$ 。设奖励 R 是 S, A, S' 三者的函数，因此可以将其记为 $R(S, A, S')$ 。由贝尔曼方程可得：

$$\begin{aligned}Q_\pi(s, a) &= \mathbb{E}_{S' \sim p(\cdot|s, a)}[R(s, a, S') + \gamma \cdot V_\pi(s')] \\ &= \sum_{s' \in \mathcal{S}} p(s'|s, a) \cdot [R(s, a, s') + \gamma \cdot V_\pi(s')] \\ &= \sum_{s' \in \mathcal{S}} p(s'|s, a) \cdot R(s, a, s') + \gamma \cdot \sum_{s' \in \mathcal{S}} p(s'|s, a) \cdot V_\pi(s').\end{aligned}\quad (7.3)$$

在观测到 s, a, s' 之后， $p(s'|s, a)$ 和 $R(s, a, s')$ 都与策略网络 π 无关，因此

$$\frac{\partial}{\partial \theta}[p(s'|s, a) \cdot R(s, a, s')] = 0.\quad (7.4)$$

由公式 (7.3) 与 (7.4) 可得：

$$\begin{aligned}\frac{\partial Q_\pi(s, a)}{\partial \theta} &= \sum_{s' \in \mathcal{S}} \underbrace{\frac{\partial}{\partial \theta}[p(s'|s, a) \cdot R(s, a, s')]}_{\text{等于零}} + \gamma \cdot \sum_{s' \in \mathcal{S}} \frac{\partial}{\partial \theta}[p(s'|s, a) \cdot V_\pi(s')] \\ &= \gamma \cdot \sum_{s' \in \mathcal{S}} p(s'|s, a) \cdot \frac{\partial V_\pi(s')}{\partial \theta} \\ &= \gamma \cdot \mathbb{E}_{S' \sim p(\cdot|s, a)}\left[\frac{\partial V_\pi(S')}{\partial \theta}\right].\end{aligned}\quad (7.5)$$

由上一小节的公式 (7.2) 可得：

$$\begin{aligned}\frac{\partial V_\pi(s)}{\partial \theta} &= \mathbb{E}_{A \sim \pi(\cdot|S; \theta)}\left[\frac{\partial \ln \pi(A|S; \theta)}{\partial \theta} \cdot Q_\pi(S, A)\right] + \mathbb{E}_{A \sim \pi(\cdot|S; \theta)}\left[\frac{\partial Q_\pi(s, a)}{\partial \theta}\right].\end{aligned}\quad (7.6)$$

结合公式(7.5)、(7.6)可得引理7.2 □

引理7.3. 策略梯度的连加形式

设 $\mathbf{g}(s, a; \boldsymbol{\theta}) \triangleq Q_\pi(s, a) \cdot \frac{\partial \ln \pi(a|s; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$ 。设一局游戏在第 n 步之后结束。那么

$$\begin{aligned} \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} &= \mathbb{E}_{S_1, A_1} [\mathbf{g}(S_1, A_1; \boldsymbol{\theta})] \\ &\quad + \gamma \cdot \mathbb{E}_{S_1, A_1, S_2, A_2} [\mathbf{g}(S_2, A_2; \boldsymbol{\theta})] \\ &\quad + \gamma^2 \cdot \mathbb{E}_{S_1, A_1, S_2, A_2, S_3, A_3} [\mathbf{g}(S_3, A_3; \boldsymbol{\theta})] \\ &\quad + \dots \\ &\quad + \gamma^{n-1} \cdot \mathbb{E}_{S_1, A_1, S_2, A_2, S_3, A_3, \dots, S_n, A_n} [\mathbf{g}(S_n, A_n; \boldsymbol{\theta})]. \end{aligned}$$



证明 设 S 、 A 为当前状态和动作， S' 为下一个状态。引理7.2 证明了下面的结论：

$$\frac{\partial V_\pi(S)}{\partial \boldsymbol{\theta}} = \mathbb{E}_A \left[\underbrace{\frac{\partial \ln \pi(A|S; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \cdot Q_\pi(S, A)}_{\text{定义为 } \mathbf{g}(S, A; \boldsymbol{\theta})} + \gamma \cdot \mathbb{E}_{S'} \left[\frac{\partial V_\pi(S')}{\partial \boldsymbol{\theta}} \right] \right].$$

这样我们可以把 $\frac{\partial V_\pi(S_1)}{\partial \boldsymbol{\theta}}$ 写成递归的形式：

$$\frac{\partial V_\pi(S_1)}{\partial \boldsymbol{\theta}} = \mathbb{E}_{A_1} [\mathbf{g}(S_1, A_1; \boldsymbol{\theta})] + \gamma \cdot \mathbb{E}_{A_1, S_2} \left[\frac{\partial V_\pi(S_2)}{\partial \boldsymbol{\theta}} \right]. \quad (7.7)$$

同理， $\frac{\partial V_\pi(S_2)}{\partial \boldsymbol{\theta}}$ 可以写成

$$\frac{\partial V_\pi(S_2)}{\partial \boldsymbol{\theta}} = \mathbb{E}_{A_2} [\mathbf{g}(S_2, A_2; \boldsymbol{\theta})] + \gamma \cdot \mathbb{E}_{A_2, S_3} \left[\frac{\partial V_\pi(S_3)}{\partial \boldsymbol{\theta}} \right]. \quad (7.8)$$

把等式(7.8)插入等式(7.7)，得到

$$\begin{aligned} \frac{\partial V_\pi(S_1)}{\partial \boldsymbol{\theta}} &= \mathbb{E}_{A_1} [\mathbf{g}(S_1, A_1; \boldsymbol{\theta})] \\ &\quad + \gamma \cdot \mathbb{E}_{A_1, S_2, A_2} [\mathbf{g}(S_2, A_2; \boldsymbol{\theta})] \\ &\quad + \gamma^2 \cdot \mathbb{E}_{A_1, S_2, A_2, S_3} \left[\frac{\partial V_\pi(S_3)}{\partial \boldsymbol{\theta}} \right]. \end{aligned}$$

按照这种规律递归下去，可得：

$$\begin{aligned} \frac{\partial V_\pi(S_1)}{\partial \boldsymbol{\theta}} &= \mathbb{E}_{A_1} [\mathbf{g}(S_1, A_1; \boldsymbol{\theta})] \\ &\quad + \gamma \cdot \mathbb{E}_{A_1, S_2, A_2} [\mathbf{g}(S_2, A_2; \boldsymbol{\theta})] \\ &\quad + \gamma^2 \cdot \mathbb{E}_{A_1, S_2, A_2, S_3, A_3} [\mathbf{g}(S_3, A_3; \boldsymbol{\theta})] \\ &\quad + \dots \\ &\quad + \gamma^{n-1} \cdot \mathbb{E}_{A_1, S_2, A_2, S_3, A_3, \dots, S_n, A_n} [\mathbf{g}(S_n, A_n; \boldsymbol{\theta})] \\ &\quad + \gamma^n \cdot \mathbb{E}_{A_1, S_2, A_2, S_3, A_3, \dots, S_n, A_n, S_{n+1}} \left[\underbrace{\frac{\partial V_\pi(S_{n+1})}{\partial \boldsymbol{\theta}}}_{\text{等于零}} \right]. \end{aligned}$$

上式中最后一项等于零，原因是游戏在 n 时刻后结束，而 $n+1$ 时刻之后没有奖励，所以 $n+1$ 时刻的回报和价值都是零。最后，由上面的公式和

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \mathbb{E}_{S_1} \left[\frac{\partial V_\pi(S_1)}{\partial \boldsymbol{\theta}} \right]$$

可得引理 7.3. □

稳态分布: 想要严格证明策略梯度定理, 需要用到马尔科夫链 (Markov Chain) 的稳态分布 (Stationary Distribution)。设状态 s' 是这样得到的: $s \rightarrow a \rightarrow s'$ 。回忆一下, 状态转移函数 $p(s'|s, a)$ 是一个概率密度函数。设 $d(s)$ 是状态 s 的概率密度函数。那么状态 s' 的边缘分布是

$$\tilde{d}(s') = \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} p(s'|s, a) \cdot \pi(a|s; \boldsymbol{\theta}) \cdot d(s).$$

如果 $\tilde{d}(\cdot)$ 与 $d(\cdot)$ 是相同的概率密度函数, 即 $d'(s) = d(s), \forall s \in \mathcal{S}$, 则意味着马尔科夫链达到稳态, 而 $d(\cdot)$ 就是稳态时的概率密度函数。

引理 7.4

设 $d(\cdot)$ 是马尔科夫链稳态时的概率密度函数。那么对于任意函数 $f(S')$,

$$\mathbb{E}_{S \sim d(\cdot)} [\mathbb{E}_{A \sim \pi(\cdot|S; \boldsymbol{\theta})} [\mathbb{E}_{S' \sim p(\cdot|s, A)} [f(S')]]] = \mathbb{E}_{S' \sim d(\cdot)} [f(S')].$$
♡

证明 把引理中的期望写成连加的形式:

$$\begin{aligned} & \mathbb{E}_{S \sim d(\cdot)} [\mathbb{E}_{A \sim \pi(\cdot|S; \boldsymbol{\theta})} [\mathbb{E}_{S' \sim p(\cdot|s, A)} [f(S')]]] \\ &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi(a|s; \boldsymbol{\theta}) \sum_{s' \in \mathcal{S}} p(s'|s, a) \cdot f(s') \\ &= \sum_{s' \in \mathcal{S}} f(s') \underbrace{\sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} p(s'|s, a) \cdot \pi(a|s; \boldsymbol{\theta}) \cdot d(s)}_{\text{等于 } d(s')} . \end{aligned}$$

上面等式最右边标出的项等于 $d(s')$, 这是根据稳态分布的定义得到的。于是有

$$\begin{aligned} \mathbb{E}_{S \sim d(\cdot)} [\mathbb{E}_{A \sim \pi(\cdot|S; \boldsymbol{\theta})} [\mathbb{E}_{S' \sim p(\cdot|s, A)} [f(S')]]] &= \sum_{s' \in \mathcal{S}} f(s') \cdot d(s') \\ &= \mathbb{E}_{S' \sim d(\cdot)} [f(S')]. \end{aligned}$$

由此可得引理 7.4 □

定理 7.5. 策略梯度定理 (严谨的表达)

设目标函数为 $J(\boldsymbol{\theta}) = \mathbb{E}_{S \sim d(\cdot)} [V_\pi(S)]$, 设 $d(s)$ 为马尔科夫链稳态分布的概率密度函数。那么

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \left(1 + \gamma + \gamma^2 + \dots + \gamma^{n-1}\right) \cdot \mathbb{E}_{S \sim d(\cdot)} \left[\mathbb{E}_{A \sim \pi(\cdot|S; \boldsymbol{\theta})} \left[\frac{\partial \ln \pi(A|S; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \cdot Q_\pi(S, A) \right] \right].$$
♡

证明 设初始状态 S_1 服从马尔科夫链的稳态分布, 它的概率密度函数是 $d(S_1)$ 。对于所有的 $t = 1, \dots, n$, 动作 A_t 根据策略网络抽样得到:

$$A_t \sim \pi(\cdot | S_t; \boldsymbol{\theta}),$$

新的状态 S_{t+1} 根据状态转移函数抽样得到:

$$S_{t+1} \sim p(\cdot | S_t, A_t).$$

7.3 策略梯度定理的证明

对于任意函数 f , 反复应用引理 7.4 可得:

$$\begin{aligned}
 & \mathbb{E}_{S_1 \sim d} \left\{ \mathbb{E}_{A_1 \sim \pi, S_2 \sim p} \left\{ \mathbb{E}_{A_2, S_3, A_3, S_4, \dots, A_{t-1}, S_t} [f(S_t)] \right\} \right\} \\
 &= \mathbb{E}_{S_2 \sim d} \left\{ \mathbb{E}_{A_2, S_3, A_3, S_4, \dots, A_{t-1}, S_t} [f(S_t)] \right\} \quad (\text{由引理 7.4 得出}) \\
 &= \mathbb{E}_{S_2 \sim d} \left\{ \mathbb{E}_{A_2 \sim \pi, S_3 \sim p} \left\{ \mathbb{E}_{A_3, S_4, A_4, S_5, \dots, A_{t-1}, S_t} [f(S_t)] \right\} \right\} \\
 &= \mathbb{E}_{S_3 \sim d} \left\{ \mathbb{E}_{A_3, S_4, A_4, S_5, \dots, A_{t-1}, S_t} [f(S_t)] \right\} \quad (\text{由引理 7.4 得出}) \\
 &\vdots \\
 &= \mathbb{E}_{S_{t-1} \sim d} \left\{ \mathbb{E}_{A_{t-1} \sim \pi, S_t \sim p} \left\{ f(S_t) \right\} \right\} \\
 &= \mathbb{E}_{S_t \sim d} \left\{ f(S_t) \right\}. \quad (\text{由引理 7.4 得出})
 \end{aligned}$$

设 $\mathbf{g}(s, a; \theta) \triangleq Q_\pi(s, a) \cdot \frac{\partial \ln \pi(a|s; \theta)}{\partial \theta}$ 。设一局游戏在第 n 步之后结束。由引理 7.3 与上面的公式可得:

$$\begin{aligned}
 \frac{\partial J(\theta)}{\partial \theta} &= \mathbb{E}_{S_1, A_1} [\mathbf{g}(S_1, A_1; \theta)] \\
 &\quad + \gamma \cdot \mathbb{E}_{S_1, A_1, S_2, A_2} [\mathbf{g}(S_2, A_2; \theta)] \\
 &\quad + \gamma^2 \cdot \mathbb{E}_{S_1, A_1, S_2, A_2, S_3, A_3} [\mathbf{g}(S_3, A_3; \theta)] \\
 &\quad + \dots \\
 &\quad + \gamma^{n-1} \cdot \mathbb{E}_{S_1, A_1, S_2, A_2, S_3, A_3, \dots, S_n, A_n} [\mathbf{g}(S_n, A_n; \theta)] \\
 &= \mathbb{E}_{S_1 \sim d(\cdot)} \left\{ \mathbb{E}_{A_1 \sim \pi(\cdot|S_1; \theta)} [\mathbf{g}(S_1, A_1; \theta)] \right\} \\
 &\quad + \gamma \cdot \mathbb{E}_{S_2 \sim d(\cdot)} \left\{ \mathbb{E}_{A_2 \sim \pi(\cdot|S_2; \theta)} [\mathbf{g}(S_2, A_2; \theta)] \right\} \\
 &\quad + \gamma^2 \cdot \mathbb{E}_{S_3 \sim d(\cdot)} \left\{ \mathbb{E}_{A_3 \sim \pi(\cdot|S_3; \theta)} [\mathbf{g}(S_3, A_3; \theta)] \right\} \\
 &\quad + \dots \\
 &\quad + \gamma^{n-1} \cdot \mathbb{E}_{S_n \sim d(\cdot)} \left\{ \mathbb{E}_{A_n \sim \pi(\cdot|S_n; \theta)} [\mathbf{g}(S_n, A_n; \theta)] \right\} \\
 &= (1 + \gamma + \gamma^2 + \dots + \gamma^{n-1}) \cdot \mathbb{E}_{S \sim d(\cdot)} \left\{ \mathbb{E}_{A \sim \pi(\cdot|S; \theta)} [\mathbf{g}(S, A; \theta)] \right\}.
 \end{aligned}$$

由此可得定理 7.5。 □

7.3.3 近似策略梯度

先复习一下前两小节的内容。策略学习可以表述为这样一个优化问题：

$$\max_{\theta} \left\{ J(\theta) \triangleq \mathbb{E}_S [V_\pi(S)] \right\}.$$

求解这个最大化问题最简单的算法就是梯度上升：

$$\theta \leftarrow \theta + \beta \cdot \nabla_{\theta} J(\theta).$$

其中的 $\nabla_{\theta} J(\theta)$ 是策略梯度。策略梯度定理证明：

$$\nabla_{\theta} J(\theta) = \mathbb{E}_S \left[\mathbb{E}_{A \sim \pi(\cdot | S; \theta)} \left[Q_\pi(S, A) \cdot \nabla_{\theta} \ln \pi(A | S; \theta) \right] \right].$$

解析求出这个期望是不可能的，因为我们并不知道状态 S 概率密度函数；即使我们知道 S 的概率密度函数，能够通过连加或者定积分求出期望，我们也不愿意这样做，因为连加或者定积分的计算量非常大。

回忆一下，第 1 章介绍了期望的蒙特卡洛近似，可以将这种方法用来近似策略梯度中的期望。每次从环境中观测到一个状态 s ，它相当于随机变量 S 的观测值。然后再根据当前的策略网络（策略网络的参数必须是最新的）随机抽样得出一个动作：

$$a \sim \pi(\cdot | s; \theta).$$

计算随机梯度：

$$g(s, a; \theta) \triangleq Q_\pi(s, a) \cdot \nabla_{\theta} \ln \pi(a | s; \theta).$$

很显然， $g(s, a; \theta)$ 是策略梯度 $\nabla_{\theta} J(\theta)$ 的无偏估计：

$$\nabla_{\theta} J(\theta) = \mathbb{E}_S \left[\mathbb{E}_{A \sim \pi(\cdot | S; \theta)} \left[g(S, A; \theta) \right] \right].$$

于是我们得到下面的结论：

结论 7.1

随机梯度 $g(s, a; \theta) \triangleq Q_\pi(s, a) \cdot \nabla_{\theta} \ln \pi(a | s; \theta)$ 是策略梯度 $\nabla_{\theta} J(\theta)$ 的无偏估计。 

应用上述结论，我们可以做随机梯度上升来更新 θ ，使得目标函数 $J(\theta)$ 逐渐增长：

$$\theta \leftarrow \theta + \beta \cdot g(s, a; \theta).$$

此处的 β 是学习率，需要手动调。但是这种方法仍然不可行，我们计算不出 $g(s, a; \theta)$ ，原因在于我们不知道动作价值函数 $Q_\pi(s, a)$ 。在后面两节中，我们用两种方法对 $Q_\pi(s, a)$ 做近似：一种方法是 REINFORCE，用实际观测的回报 u 近似 $Q_\pi(s, a)$ ；另一种方法是 Actor-Critic，用神经网络 $q(s, a; w)$ 近似 $Q_\pi(s, a)$ 。

7.4 REINFORCE

策略梯度方法用策略梯度 $\nabla_{\theta} J(\theta)$ 更新策略网络参数 θ , 从而增大目标函数。上一节中, 我们推导出策略梯度 $\nabla_{\theta} J(\theta)$ 的无偏估计, 即下面的随机梯度:

$$\mathbf{g}(s, a; \theta) \triangleq Q_{\pi}(s, a) \cdot \nabla_{\theta} \ln \pi(a | s; \theta).$$

但是其中的动作价值函数 Q_{π} 是未知的, 导致无法直接计算 $\mathbf{g}(s, a; \theta)$ 。REINFORCE 进一步对 Q_{π} 做蒙特卡洛近似, 把它替换成回报 u 。REINFORCE 属于策略梯度方法。

7.4.1 REINFORCE 的简化推导

设一局游戏有 n 步, 一局中的奖励记作 R_1, \dots, R_n 。回忆一下, t 时刻的折扣回报定义为:

$$U_t = \sum_{k=t}^n \gamma^{k-t} \cdot R_k.$$

而动作价值定义为 U_t 的条件期望:

$$Q_{\pi}(s_t, a_t) = \mathbb{E}[U_t | S_t = s_t, A_t = a_t].$$

我们可以用蒙特卡洛近似上面的条件期望。从时刻 t 开始, 智能体完成一局游戏, 观测到全部奖励 r_t, \dots, r_n , 然后可以计算出 $u_t = \sum_{k=t}^n \gamma^{k-t} \cdot r_k$ 。因为 u_t 是随机变量 U_t 的观测值, 所以 u_t 是上面公式中期望的蒙特卡洛近似。在实践中, 可以用 u_t 代替 $Q_{\pi}(s_t, a_t)$, 那么随机梯度 $\mathbf{g}(s_t, a_t; \theta)$ 可以近似成

$$\tilde{\mathbf{g}}(s_t, a_t; \theta) = u_t \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta).$$

$\tilde{\mathbf{g}}$ 是 \mathbf{g} 的无偏估计, 所以也是策略梯度 $\nabla_{\theta} J(\theta)$ 的无偏估计; $\tilde{\mathbf{g}}$ 也是一种随机梯度。

我们可以用反向传播计算出 $\ln \pi$ 关于 θ 的梯度, 而且可以实际观测到 u_t , 于是我们可以实际计算出随机梯度 $\tilde{\mathbf{g}}$ 的值。有了随机梯度的值, 我们可以做随机梯度上升更新策略网络参数 θ :

$$\theta \leftarrow \theta + \beta \cdot \tilde{\mathbf{g}}(s_t, a_t; \theta). \quad (7.9)$$

根据上述推导, 我们得到了训练策略网络的方法, 这种方法叫做 REINFORCE。

7.4.2 训练流程

当前策略网络的参数是 θ_{now} 。REINFORCE 执行下面的步骤对策略网络的参数做一次更新:

1. 用策略网络 θ_{now} 控制智能体从头开始玩一局游戏, 得到一条轨迹 (Trajectory):

$$s_1, a_1, r_1, \quad s_2, a_2, r_2, \quad \dots, \quad s_n, a_n, r_n.$$

2. 计算所有的回报:

$$u_t = \sum_{k=t}^n \gamma^{k-t} \cdot r_k, \quad \forall t = 1, \dots, n.$$

3. 用 $\{(s_t, a_t)\}_{t=1}^n$ 作为数据，做反向传播计算：

$$\nabla_{\theta} \ln \pi(a_t | s_t; \theta_{\text{now}}), \quad \forall t = 1, \dots, n.$$

4. 做随机梯度上升更新策略网络参数：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \sum_{t=1}^n \gamma^{t-1} \cdot \underbrace{u_t \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta_{\text{now}})}_{\text{即随机梯度 } \tilde{g}(s_t, a_t; \theta_{\text{now}})}.$$

注 在算法最后一步中，随机梯度前面乘以系数 γ^{t-1} 。读者可能会好奇，为什么需要这个系数呢？原因是这样的：前面 REINFORCE 的推导是简化的，而非严谨的数学推导；按照我们简化的推导，不应该乘以系数 γ^{t-1} 。下一小节做严格的数学推导，得出的 REINFORCE 算法需要系数 γ^{t-1} 。读者只要知道这个事实就行了，不必读懂下一小节的数学推导。

注 REINFORCE 是一种同策略 (On-Policy) 方法，要求行为策略 (Behavior Policy) 与目标策略 (Target Policy) 相同，两者都必须是策略网络 $\pi(a|s; \theta_{\text{now}})$ ，其中 θ_{now} 是策略网络当前的参数。所以经验回放不适用于 REINFORCE。

7.4.3 REINFORCE 严格的推导

第 7.4.1 小节对策略梯度做近似，推导出 REINFORCE 方法。那种推导是简化过的，帮助读者理解 REINFORCE 算法，但实际上那种推导并不够严谨。本小节做严格的数学推导，对策略梯度做近似，得出真正的 REINFORCE 方法。建议对数学证明不感兴趣的读者跳过本小节。

根据定义， $\mathbf{g}(s, a; \theta) \triangleq Q_{\pi}(s, a) \cdot \nabla_{\theta} \ln \pi(a | s; \theta)$ 。引理 7.3 把策略梯度 $\nabla_{\theta} J(\theta)$ 表示成期望的连加：

$$\begin{aligned} \nabla_{\theta} J(\theta) &= \mathbb{E}_{S_1, A_1} [\mathbf{g}(S_1, A_1; \theta)] \\ &\quad + \gamma \cdot \mathbb{E}_{S_1, A_1, S_2, A_2} [\mathbf{g}(S_2, A_2; \theta)] \\ &\quad + \gamma^2 \cdot \mathbb{E}_{S_1, A_1, S_2, A_2, S_3, A_3} [\mathbf{g}(S_3, A_3; \theta)] \\ &\quad + \dots \\ &\quad + \gamma^{n-1} \cdot \mathbb{E}_{S_1, A_1, S_2, A_2, S_3, A_3, \dots, S_n, A_n} [\mathbf{g}(S_n, A_n; \theta)]. \end{aligned} \quad (7.10)$$

我们可以对期望做蒙特卡洛近似。首先观测到第一个状态 $S_1 = s_1$ 。然后用最新的策略网络 $\pi(a|s; \theta_{\text{now}})$ 控制智能体与环境交互，观测到到轨迹

$$s_1, a_1, r_1, \quad s_2, a_2, r_2, \quad \dots, \quad s_n, a_n, r_n.$$

对公式 (7.10) 中的期望做蒙特卡洛近似，得到：

$$\nabla_{\theta} J(\theta_{\text{now}}) \approx \mathbf{g}(s_1, a_1; \theta_{\text{now}}) + \gamma \cdot \mathbf{g}(s_2, a_2; \theta_{\text{now}}) + \dots + \gamma^{n-1} \cdot \mathbf{g}(s_n, a_n; \theta_{\text{now}}).$$

进一步把 $\mathbf{g}(s_t, a_t; \theta_{\text{now}}) \triangleq Q_{\pi}(s_t, a_t) \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta_{\text{now}})$ 中的 $Q_{\pi}(s_t, a_t)$ 替换成 u_t ，那么 $\mathbf{g}(s_t, a_t; \theta_{\text{now}})$ 就被近似成为

$$\mathbf{g}(s_t, a_t; \theta_{\text{now}}) \approx u_t \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta_{\text{now}}).$$

7.4 REINFORCE

经过上述两次近似，策略梯度被近似成为下面的随机梯度

$$\nabla_{\theta} J(\theta_{\text{now}}) \approx \sum_{t=1}^n \gamma^{t-1} \cdot u_t \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta_{\text{now}}).$$

这样就得到了 REINFORCE 算法的随机梯度上升公式：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \sum_{t=1}^n \gamma^{t-1} \cdot u_t \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta_{\text{now}}).$$

7.5 Actor-Critic

策略梯度方法用策略梯度 $\nabla_{\theta} J(\theta)$ 更新策略网络参数 θ ，从而增大目标函数。第 7.2 节推导出策略梯度 $\nabla_{\theta} J(\theta)$ 的无偏估计，即下面的随机梯度：

$$g(s, a; \theta) \triangleq Q_{\pi}(s, a) \cdot \nabla_{\theta} \ln \pi(a | s; \theta).$$

但是其中的动作价值函数 Q_{π} 是未知的，导致无法直接计算 $g(s, a; \theta)$ 。上一节的 REINFORCE 用实际观测的回报近似 Q_{π} ，本节的 Actor-Critic 方法用神经网络近似 Q_{π} 。

7.5.1 价值网络

Actor-Critic 方法中用一个神经网络近似动作价值函数 $Q_{\pi}(s, a)$ ，这个神经网络叫做“价值网络”，记为 $q(s, a; w)$ ，其中的 w 表示神经网络中可训练的参数。价值网络的输入是状态 s ，输出是每个动作的价值。动作空间 \mathcal{A} 中有多少种动作，那么价值网络的输出就是多少维的向量，向量每个元素对应一个动作。举个例子，动作空间是 $\mathcal{A} = \{\text{左, 右, 上}\}$ ，价值网络的输出是

$$\begin{aligned} q(s, \text{左}; w) &= 219, \\ q(s, \text{右}; w) &= -73, \\ q(s, \text{上}; w) &= 580. \end{aligned}$$

神经网络的结构见图 7.2。

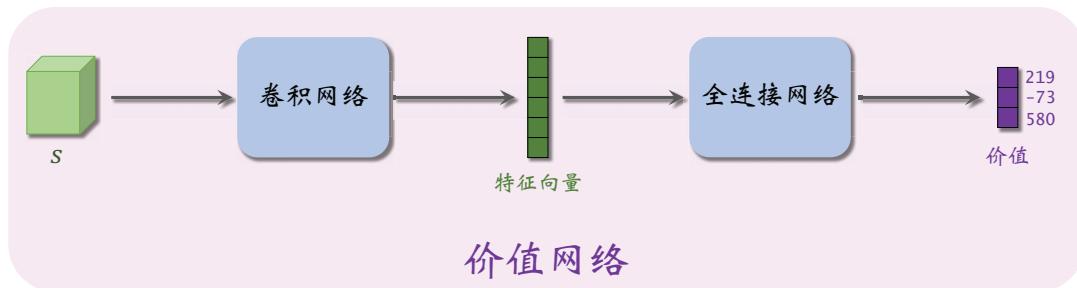


图 7.2：价值网络 $q(s, a; w)$ 的结构。输入是状态 s ；输出是每个动作的价值。

虽然价值网络 $q(s, a; w)$ 与之前学的 DQN 有相同的结构，但是两者的意义不同，训练算法也不同。

- 价值网络是对动作价值函数 $Q_{\pi}(s, a)$ 的近似。而 DQN 则是对最优动作价值函数 $Q_{\star}(s, a)$ 的近似。
- 对价值网络的训练使用的是 SARSA 算法，它属于同策略，不能用经验回放。对 DQN 的训练使用的是 Q 学习算法，它属于异策略，可以用经验回放。

7.5.2 算法的推导

Actor-Critic 翻译成“演员—评委”方法。策略网络 $\pi(a|s; \theta)$ 相当于演员，它基于状态 s 做出动作 a 。价值网络 $q(s, a; \theta)$ 相当于评委，它给演员的表现打分，量化在状态 s 的情况下做出动作 a 的好坏程度。策略网络（演员）和价值网络（评委）的关系如图 7.3 所示。

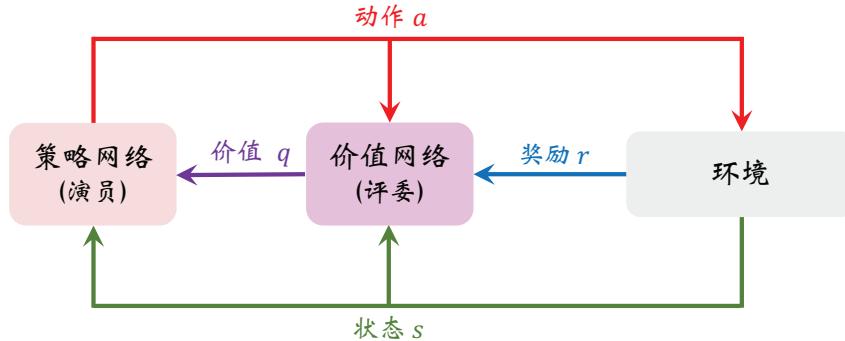


图 7.3: Actor-Critic 方法中策略网络（演员）和价值网络（评委）的关系图。

读者可能会对图 7.3 感到不解：为什么不直接把奖励 R 反馈给策略网络（演员），而要用价值网络（评委）这样一个中介呢？原因是这样的。策略学习的目标函数 $J(\theta)$ 是回报 U 的期望，而不是奖励 R 的期望；注意回报 U 和奖励 R 的区别。虽然能观测到当前的奖励 R ，但是它对价值网络是毫无意义的；训练策略网络（演员）需要的是回报 U ，也就是未来所有奖励的加权和。价值网络（评委）能够估算出回报 U 的期望，因此能帮助训练策略网络（演员）。

训练策略网络（演员）：策略网络（演员）想要改进自己的演技，但是演员自己不知道什么样的表演才算更好，所以需要价值网络（评委）的帮助。在演员做出动作 a 之后，评委打一个分数 $\hat{q} \triangleq q(s, a; \mathbf{w})$ ，并把分数反馈给演员，帮助演员做出改进。演员利用当前状态 s ，自己的动作 a ，以及评委的打分 \hat{q} ，计算近似策略梯度，然后更新自己的参数 θ （相当于改变自己的技术）。通过这种方式，演员的表现越来越受评委的好评，于是演员获得的评分 \hat{q} 越来越高。

训练策略网络的基本想法是用策略梯度 $\nabla_{\theta} J(\theta)$ 的近似来更新参数 θ 。之前我们推导过策略梯度的无偏估计：

$$g(s, a; \theta) \triangleq Q_{\pi}(s, a) \cdot \nabla_{\theta} \ln \pi(a | s; \theta).$$

价值网络 $q(s, a; \mathbf{w})$ 是对动作价值函数 $Q_{\pi}(s, a)$ 的近似，所以把上面公式中的 Q_{π} 替换成价值网络，得到近似策略梯度：

$$\hat{g}(s, a; \theta) \triangleq \underbrace{q(s, a; \mathbf{w})}_{\text{评委的打分}} \cdot \nabla_{\theta} \ln \pi(a | s; \theta). \quad (7.11)$$

最后做梯度上升更新策略网络的参数：

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \beta \cdot \hat{\mathbf{g}}(s, a; \boldsymbol{\theta}). \quad (7.12)$$

注 用上述方式更新参数之后，会让评委打出的分数越来越高，原因是这样的。状态价值函数 $V_\pi(s)$ 可以近似成为：

$$v(s; \boldsymbol{\theta}) = \mathbb{E}_{A \sim \pi(\cdot|s; \boldsymbol{\theta})} [q(s, A; \mathbf{w})].$$

因此可以将 $v(s; \boldsymbol{\theta})$ 看做评委打分的均值。不难证明，公式 (7.11) 中定义的近似策略梯度 $\hat{\mathbf{g}}(s, a; \boldsymbol{\theta})$ 的期望等于 $v(s; \boldsymbol{\theta})$ 关于 $\boldsymbol{\theta}$ 的梯度；

$$\nabla_{\boldsymbol{\theta}} v(s; \boldsymbol{\theta}) = \mathbb{E}_{A \sim \pi(\cdot|s; \boldsymbol{\theta})} [\hat{\mathbf{g}}(s, A; \boldsymbol{\theta})].$$

因此，用公式 7.12 中的梯度上升更新 $\boldsymbol{\theta}$ ，会让 $v(s; \boldsymbol{\theta})$ 变大，也就是让评委打分的均值更高。

训练价值网络（评委）： 通过以上分析，我们不难发现上述训练策略网络（演员）的方法不是真正让演员表现更好，只是让演员更迎合评委的喜好而已。因此，评委的水平也很重要，只有当评委的打分 \hat{q} 真正反映出动作价值 Q_π ，演员的水平才能真正提高。初始的时候，价值网络的参数 \mathbf{w} 是随机的，也就是说评委的打分是瞎猜。可以用 SARSA 算法更新 \mathbf{w} ，提高评委的水平。每次从环境中观测到一个奖励 r ，把 r 看做是真相，用 r 来校准评委的打分。

第 5.1 节已经推导过 SARSA 算法，现在我们再回顾一下。在 t 时刻，价值网络输出

$$\hat{q}_t = q(s_t, a_t; \mathbf{w}),$$

它是对动作价值函数 $Q_\pi(s_t, a_t)$ 的估计。在 $t+1$ 时刻，实际观测到 r_t, s_{t+1}, a_{t+1} ，于是可以计算 TD 目标

$$\hat{y}_t \triangleq r_t + \gamma \cdot q(s_{t+1}, a_{t+1}; \mathbf{w}),$$

它也是对动作价值函数 $Q_\pi(s_t, a_t)$ 的估计。由于 \hat{y}_t 部分基于实际观测到的奖励 r_t ，我们认为 \hat{y}_t 比 $q(s_t, a_t; \mathbf{w})$ 更接近事实真相。所以把 \hat{y}_t 固定住，鼓励 $q(s_t, a_t; \mathbf{w})$ 去接近 \hat{y}_t 。SARSA 算法具体这样更新价值网络参数 \mathbf{w} 。定义损失函数：

$$L(\mathbf{w}) \triangleq \frac{1}{2} [q(s_t, a_t; \mathbf{w}) - \hat{y}_t]^2,$$

设 $\hat{q}_t \triangleq q(s_t, a_t; \mathbf{w})$ 。损失函数的梯度是：

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \underbrace{(\hat{q}_t - \hat{y}_t)}_{\text{TD 误差 } \delta_t} \cdot \nabla_{\mathbf{w}} q(s_t, a_t; \mathbf{w}).$$

做一轮梯度下降更新 \mathbf{w} ：

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \nabla_{\mathbf{w}} L(\mathbf{w}).$$

这样更新 \mathbf{w} 可以让 $q(s_t, a_t; \mathbf{w})$ 更接近 \hat{y}_t 。可以这样理解 SARSA：用观测到的奖励 r_t 来“校准”评委的打分 $q(s_t, a_t; \mathbf{w})$ 。

7.5.3 训练流程

最后概括 Actor-Critic 训练流程。设当前策略网络参数是 θ_{now} ，价值网络参数是 w_{now} 。执行下面的步骤，将参数更新成 θ_{new} 和 w_{new} ：

1. 观测到当前状态 s_t ，根据策略网络做决策： $a_t \sim \pi(\cdot | s_t; \theta_{\text{now}})$ ，并让智能体执行动作 a_t 。
2. 从环境中观测到奖励 r_t 和新的状态 s_{t+1} 。
3. 根据策略网络做决策： $\tilde{a}_{t+1} \sim \pi(\cdot | s_{t+1}; \theta_{\text{now}})$ ，但不让智能体执行动作 \tilde{a}_{t+1} 。
4. 让价值网络打分：

$$\hat{q}_t = q(s_t, a_t; w_{\text{now}}) \quad \text{和} \quad \hat{q}_{t+1} = q(s_{t+1}, \tilde{a}_{t+1}; w_{\text{now}})$$

5. 计算 TD 目标和 TD 误差：

$$\hat{y}_t = r_t + \gamma \cdot \hat{q}_{t+1} \quad \text{和} \quad \delta_t = \hat{q}_t - \hat{y}_t.$$

6. 更新价值网络：

$$w_{\text{new}} \leftarrow w_{\text{now}} - \alpha \cdot \delta_t \cdot \nabla_w q(s_t, a_t; w_{\text{now}}).$$

7. 更新策略网络：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \hat{q}_t \cdot \nabla_\theta \ln \pi(a_t | s_t; \theta_{\text{now}}).$$

7.5.4 用目标网络改进训练

第 6.2 节讨论了 Q 学习中的自举及其危害，以及用目标网络 (Target Network) 缓解自举造成的偏差。SARSA 算法中也存在自举——即用价值网络自己的估值 \hat{q}_{t+1} 去更新价值网络自己；我们同样可以用目标网络计算 TD 目标，从而缓解偏差。把目标网络记作 $q(s, a; w^-)$ ，它的结构与价值网络的结构相同，但是参数不同。使用目标网络计算 TD 目标，那么 Actor-Critic 的训练就变成了：

1. 观测到当前状态 s_t ，根据策略网络做决策： $a_t \sim \pi(\cdot | s_t; \theta_{\text{now}})$ ，并让智能体执行动作 a_t 。
2. 从环境中观测到奖励 r_t 和新的状态 s_{t+1} 。
3. 根据策略网络做决策： $\tilde{a}_{t+1} \sim \pi(\cdot | s_{t+1}; \theta_{\text{now}})$ ，但是不让智能体执行动作 \tilde{a}_{t+1} 。
4. 让价值网络给 (s_t, a_t) 打分：

$$\hat{q}_t = q(s_t, a_t; w_{\text{now}}).$$

5. 让目标网络给 $(s_{t+1}, \tilde{a}_{t+1})$ 打分：

$$\widehat{q}_{t+1} = q(s_{t+1}, \tilde{a}_{t+1}; w_{\text{now}}^-).$$

6. 计算 TD 目标和 TD 误差：

$$\widehat{y}_t = r_t + \gamma \cdot \widehat{q}_{t+1} \quad \text{和} \quad \delta_t = \hat{q}_t - \widehat{y}_t.$$

7. 更新价值网络：

$$w_{\text{new}} \leftarrow w_{\text{now}} - \alpha \cdot \delta_t \cdot \nabla_w q(s_t, a_t; w_{\text{now}}).$$

8. 更新策略网络：

$$\boldsymbol{\theta}_{\text{new}} \leftarrow \boldsymbol{\theta}_{\text{now}} + \beta \cdot \hat{q}_t \cdot \nabla_{\boldsymbol{\theta}} \ln \pi(a_t | s_t; \boldsymbol{\theta}_{\text{now}}).$$

9. 设 $\tau \in (0, 1)$ 是需要手动调的超参数。做加权平均更新目标网络的参数：

$$\mathbf{w}_{\text{new}}^- \leftarrow \tau \cdot \mathbf{w}_{\text{new}} + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^-.$$

∽第七章 相关文献∽

REINFORCE 方法由 Williams 在 1987 年提出 [117-118]。Actor-Critic 方法在 Barto 等人 1983 年的论文 [10] 中提出。很多论文分析过 Actor-Critic 方法的收敛，比如 [55, 14, 2, 15, 121]。策略梯度定理由 Marbach 和 Tsitsiklis 1999 年的论文 [66] 和 Sutton 等人 2000 年的论文 [97] 独立提出。

《深度强化学习》 2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

第八章 带基线的策略梯度方法

上一章推导出策略梯度，并介绍了两种策略梯度方法——REINFORCE 和 Actor-Critic。虽然上一章的方法在理论上是正确的，但是在实践中效果并不理想。本章介绍的带基线的策略梯度 (Policy Gradient with Baseline) 可以大幅提升策略梯度方法的表现。使用基线 (Baseline) 之后，REINFORCE 变成 REINFORCE with Baseline，Actor-Critic 变成 Advantage Actor-Critic (A2C)。

8.1 策略梯度中的基线

首先回顾上一章的内容。策略学习通过最大化目标函数 $J(\theta) = \mathbb{E}_S[V_\pi(S)]$ ，训练出策略网络 $\pi(a|s; \theta)$ 。可以用策略梯度 $\nabla_\theta J(\theta)$ 来更新参数 θ ：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \nabla_\theta J(\theta_{\text{now}}).$$

策略梯度定理证明：

$$\nabla_\theta J(\theta) = \mathbb{E}_S \left[\mathbb{E}_{A \sim \pi(\cdot|S; \theta)} \left[Q_\pi(S, A) \cdot \nabla_\theta \ln \pi(A|S; \theta) \right] \right]. \quad (8.1)$$

REINFORCE 和 Actor-Critic 都是通过对策略梯度 $\nabla_\theta J(\theta)$ 做近似推导出的；两种方法区别在于具体如何做近似。

8.1.1 基线 (Baseline)

基于策略梯度公式 (8.1) 得出的 REINFORCE 和 Actor-Critic 方法效果通常不好。但是只需对策略梯度公式 (8.1) 做一个微小的改动，就能大幅提升表现：把 b 作为动作价值函数 $Q_\pi(S, A)$ 的基线 (Baseline)，用 $Q_\pi(S, A) - b$ 替换掉 Q_π 。设 b 是任意的函数，只要不依赖于动作 A 就可以；例如， b 可以是状态价值函数 $V_\pi(S)$ 。

定理 8.1. 带基线的策略梯度定理

设 b 是任意的函数，但是 b 不能依赖于 A 。把 b 作为动作价值函数 $Q_\pi(S, A)$ 的基线，对策略梯度没有影响：

$$\nabla_\theta J(\theta) = \mathbb{E}_S \left[\mathbb{E}_{A \sim \pi(\cdot|S; \theta)} \left[(Q_\pi(S, A) - b) \cdot \nabla_\theta \ln \pi(A|S; \theta) \right] \right].$$

定理 8.1 说明 b 的取值不影响策略梯度的正确性。不论是让 $b = 0$ 还是让 $b = V_\pi(S)$ ，对期望的结果毫无影响，期望的结果都会等于 $\nabla_\theta J(\theta)$ 。其原因在于

$$\mathbb{E}_S \left[\mathbb{E}_{A \sim \pi(\cdot|S; \theta)} \left[b \cdot \nabla_\theta \ln \pi(A|S; \theta) \right] \right] = 0.$$

定理的证明放到第 8.4 节，对数学感兴趣的读者可以阅读。

定理中的策略梯度表示成了期望的形式，我们对期望做蒙特卡洛近似。从环境中观

测到一个状态 s , 然后根据策略网络抽样得到 $a \sim \pi(\cdot|s; \theta)$ 。那么策略梯度 $\nabla_{\theta} J(\theta)$ 可以近似为下面的随机梯度:

$$\mathbf{g}_b(s, a; \theta) = [Q_{\pi}(S, A) - b] \cdot \nabla_{\theta} \ln \pi(A | S; \theta).$$

不论 b 的取值是 0 还是 $V_{\pi}(s)$, 得到的随机梯度 $\mathbf{g}_b(s, a; \theta)$ 都是 $\nabla_{\theta} J(\theta)$ 的无偏估计:

$$\text{Bias} = \mathbb{E}_{S,A}[\mathbf{g}_b(S, A; \theta)] - \nabla_{\theta} J(\theta) = 0.$$

虽然 b 的取值对 $\mathbb{E}_{S,A}[\mathbf{g}_b(S, A; \theta)]$ 毫无影响, 但是 b 对随机梯度 $\mathbf{g}_b(s, a; \theta)$ 是有影响的。用不同的 b , 得到的方差

$$\text{Var} = \mathbb{E}_{S,A}[\|\mathbf{g}_b(S, A; \theta) - \nabla_{\theta} J(\theta)\|^2]$$

会有所不同。如果 b 很接近 $Q_{\pi}(s, a)$ 关于 a 的均值, 那么方差会比较小。所以 $b = V_{\pi}(s)$ 是很好的基线。

8.1.2 基线的直观解释

策略梯度公式 (8.1) 期望中的 $Q_{\pi}(S, A) \cdot \nabla_{\theta} \ln \pi(A | S; \theta)$ 的意义是什么呢? 以图 8.1 中的左图为例。给定状态 s_t , 动作空间是 $\mathcal{A} = \{\text{左}, \text{右}, \text{上}\}$, 动作价值函数给每个动作打分:

$$Q_{\pi}(s_t, \text{左}) = 80, \quad Q_{\pi}(s_t, \text{右}) = -20, \quad Q_{\pi}(s_t, \text{上}) = 180,$$

这些分值会乘到梯度 $\nabla_{\theta} \ln \pi(A | S; \theta)$ 上。在做完梯度上升之后, 新的策略会倾向于分值高的动作。

- 动作价值 $Q_{\pi}(s_t, \text{上}) = 180$ 很大, 说明基于状态 s_t 选择动作 “上” 是很好的决策。让梯度 $\nabla_{\theta} \ln \pi(\text{上} | s_t; \theta)$ 乘以大的系数 $Q_{\pi}(s_t, \text{上}) = 180$, 那么做梯度上升更新 θ 之后, 会让 $\pi(\text{上} | s_t; \theta)$ 变大, 在状态 s_t 的情况下更倾向于动作 “上”。
- 相反, $Q_{\pi}(s_t, \text{右}) = -20$ 说明基于状态 s_t 选择动作 “右” 是糟糕的决策。让梯度 $\nabla_{\theta} \ln \pi(\text{右} | s_t; \theta)$ 乘以负的系数 $Q_{\pi}(s_t, \text{右}) = -20$, 那么做梯度上升更新 θ 之后, 会让 $\pi(\text{右} | s_t; \theta)$ 变小, 在状态 s_t 的情况下选择动作 “右”的概率更小。

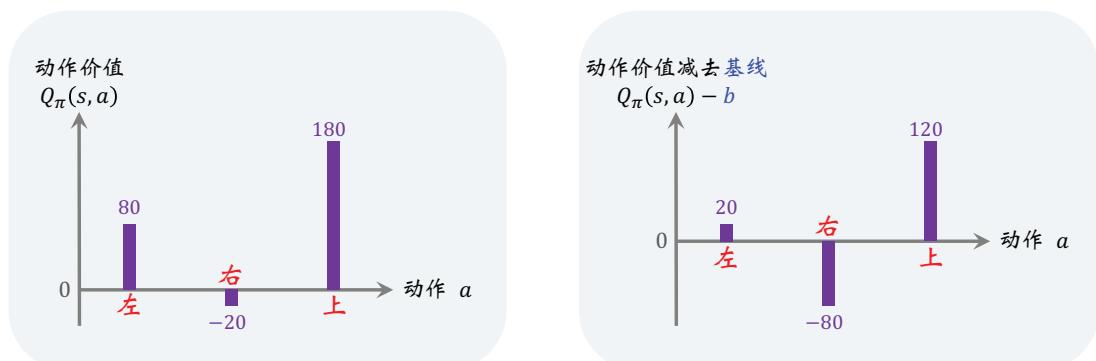


图 8.1: 动作空间是 $\mathcal{A} = \{\text{左}, \text{右}, \text{上}\}$ 。给定状态 s 。左图纵轴表示动作价值 $Q_{\pi}(s, a)$ 。右图纵轴表示动作价值减去基线 $Q_{\pi}(s, a) - b$, 其中基线 $b = 60$ 。

8.1 策略梯度中的基线

根据上述分析，我们在乎的是动作价值 $Q_\pi(s_t, \text{左})$ 、 $Q_\pi(s_t, \text{右})$ 、 $Q_\pi(s_t, \text{上})$ 三者的相对大小，而非绝对大小。如果给三者都减去 $b = 60$ ，那么三者的相对大小是不变的；动作“上”仍然是最好的，动作“右”仍然是最差的。见图 8.1 中的右图。因此

$$[Q_\pi(s_t, a_t) - b] \cdot \nabla_{\theta} \ln \pi(A | S; \theta)$$

依然能指导 θ 做调整，使得 $\pi(\text{上} | s_t; \theta)$ 变大，而 $\pi(\text{右} | s_t; \theta)$ 变小。

8.2 带基线的 REINFORCE 算法

上一节推导出了带基线的策略梯度，并且对策略梯度做了蒙特卡洛近似。本节中，我们使用状态价值 $V_\pi(s)$ 作基线，得到策略梯度的一个无偏估计：

$$g(s, a; \theta) = [Q_\pi(s, a) - V_\pi(s)] \cdot \nabla_\theta \ln \pi(a | s; \theta).$$

我们在第 7.4 节中学过 REINFORCE，它使用实际观测的回报 u 来代替动作价值 $Q_\pi(s, a)$ 。此处我们同样用 u 代替 $Q_\pi(s, a)$ 。此外，我们还用一个神经网络 $v(s; w)$ 近似状态价值函数 $V_\pi(s)$ 。这样一来， $g(s, a; \theta)$ 就被近似成了：

$$\tilde{g}(s, a; \theta) = [u - v(s; w)] \cdot \nabla_\theta \ln \pi(a | s; \theta).$$

可以用 $\tilde{g}(s, a; \theta)$ 作为策略梯度 $\nabla_\theta J(\theta)$ 的近似，更新策略网络参数：

$$\theta \leftarrow \theta + \beta \cdot \tilde{g}(s, a; \theta)$$

8.2.1 策略网络和价值网络

带基线的 REINFORCE 需要两个神经网络：策略网络 $\pi(a|s; \theta)$ 和价值网络 $v(s; w)$ ；神经网络结构如图 8.2 和 8.3 所示。策略网络与之前章节一样：输入是状态 s ，输出是一个向量，每个元素表示一个动作的概率。

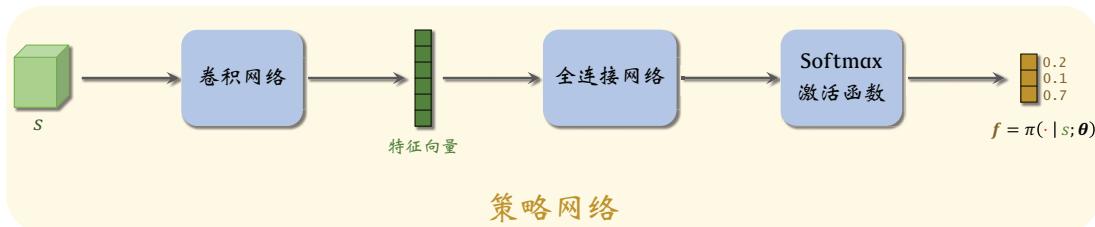


图 8.2: 策略网络 $\pi(a|s; \theta)$ 的神经网络结构。输入是状态 s ，输出是动作空间中每个动作的概率值。举个例子，动作空间是 $\mathcal{A} = \{\text{左, 右, 上}\}$ ，策略网络的输出是三个概率值： $\pi(\text{左}|s; \theta) = 0.2$, $\pi(\text{右}|s; \theta) = 0.1$, $\pi(\text{上}|s; \theta) = 0.7$ 。

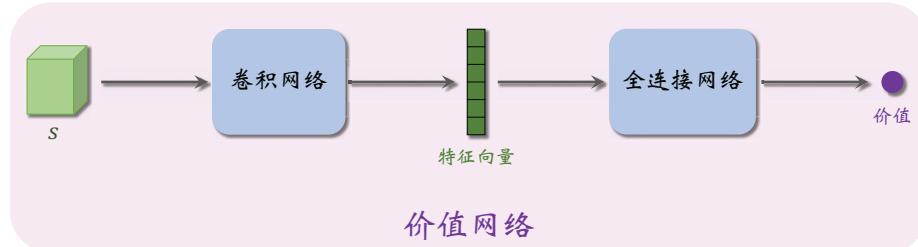


图 8.3: 价值网络 $v(s; w)$ 的结构。输入是状态 s ；输出是状态的价值。

此处的价值网络 $v(s; w)$ 与之前使用的价值网络 $q(s, a; w)$ 区别较大。此处的 $v(s; w)$ 是对状态价值 V_π 的近似，而非对动作价值 Q_π 的近似。 $v(s; w)$ 的输入是状态 s ，输出是

8.2 带基线的 REINFORCE 算法

一个实数，作为基线。策略网络和价值网络的输入都是状态 s ，因此可以让两个神经网络共享卷积网络的参数，这是编程实现中常用的技巧。

虽然带基线的 REINFORCE 有一个策略网络和一个价值网络，但是这种方法不是 Actor-Critic。价值网络没有起到“评委”的作用，只是作为基线而已，目的在于降低方差，加速收敛。真正帮助策略网络（演员）改进参数 θ （演员的演技）的不是价值网络，而是实际观测到的回报 u 。

8.2.2 算法的推导

训练策略网络的方法是近似的策略梯度上升。从 t 时刻开始，智能体完成一局游戏，观测到全部奖励 r_t, r_{t+1}, \dots, r_n ，然后计算回报 $u_t = \sum_{k=t}^n \gamma^{k-t} \cdot r_k$ 。让价值网络做出预测 $\hat{v}_t = v(s_t; \mathbf{w})$ ，作为基线。这样就得到了带基线的策略梯度：

$$\tilde{\mathbf{g}}(s_t, a_t; \theta) = (u_t - \hat{v}_t) \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta).$$

它是策略梯度 $\nabla_{\theta} J(\theta)$ 的近似。最后做梯度上升更新 θ ：

$$\theta \leftarrow \theta + \beta \cdot \tilde{\mathbf{g}}(s_t, a_t; \theta).$$

这样可以让目标函数 $J(\theta)$ 逐渐增大。

训练价值网络的方法是回归 (Regression)。回忆一下，状态价值是回报的期望：

$$V_{\pi}(s_t) = \mathbb{E}[U_t | S_t = s_t],$$

期望消掉了动作 A_t, A_{t+1}, \dots, A_n 和状态 S_{t+1}, \dots, S_n 。训练价值网络的目的是让 $v(s_t; \mathbf{w})$ 拟合 $V_{\pi}(s_t)$ ，即拟合 u_t 的期望。定义损失函数：

$$L(\mathbf{w}) = \frac{1}{2n} \sum_{t=1}^n \frac{1}{2} [v(s_t; \mathbf{w}) - u_t]^2.$$

设 $\hat{v}_t = v(s_t; \mathbf{w})$ 。损失函数的梯度是：

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \frac{1}{n} \sum_{t=1}^n (\hat{v}_t - u_t) \cdot \nabla_{\mathbf{w}} v(s_t; \mathbf{w}).$$

做一次梯度下降更新 \mathbf{w} ：

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \nabla_{\mathbf{w}} L(\mathbf{w}).$$

8.2.3 训练流程

当前策略网络的参数是 θ_{now} ，价值网络的参数是 \mathbf{w}_{now} 。执行下面的步骤，对参数做一轮更新。

1. 用策略网络 θ_{now} 控制智能体从头开始玩一局游戏，得到一条轨迹 (Trajectory)：

$$s_1, a_1, r_1, \quad s_2, a_2, r_2, \quad \dots, \quad s_n, a_n, r_n.$$

2. 计算所有的回报：

$$u_t = \sum_{k=t}^n \gamma^{k-t} \cdot r_k, \quad \forall t = 1, \dots, n.$$

3. 让价值网络做预测：

$$\hat{v}_t = v(s_t; \mathbf{w}_{\text{now}}), \quad \forall t = 1, \dots, n.$$

4. 计算误差 $\delta_t = \hat{v}_t - u_t, \forall t = 1, \dots, n.$

5. 用 $\{s_t\}_{t=1}^n$ 作为价值网络输入，做反向传播计算：

$$\nabla_{\mathbf{w}} v(s_t; \mathbf{w}_{\text{now}}), \quad \forall t = 1, \dots, n.$$

6. 更新价值网络参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \sum_{t=1}^n \delta_t \cdot \nabla_{\mathbf{w}} v(s_t; \mathbf{w}_{\text{now}}).$$

7. 用 $\{(s_t, a_t)\}_{t=1}^n$ 作为数据，做反向传播计算：

$$\nabla_{\theta} \ln \pi(a_t | s_t; \boldsymbol{\theta}_{\text{now}}), \quad \forall t = 1, \dots, n.$$

8. 做随机梯度上升更新策略网络参数：

$$\boldsymbol{\theta}_{\text{new}} \leftarrow \boldsymbol{\theta}_{\text{now}} + \beta \cdot \sum_{t=1}^n \gamma^{t-1} \cdot \underbrace{\delta_t \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \boldsymbol{\theta}_{\text{now}})}_{\text{负的近似梯度 } -\tilde{g}(s_t, a_t; \boldsymbol{\theta}_{\text{now}})}.$$

8.3 Advantage Actor-Critic (A2C)

之前我们推导出了带基线的策略梯度，并且对策略梯度做了蒙特卡洛近似，得到得到策略梯度的一个无偏估计：

$$\mathbf{g}(s, a; \boldsymbol{\theta}) = \underbrace{[Q_\pi(s, a) - V_\pi(s)]}_{\text{优势函数}} \cdot \nabla_{\boldsymbol{\theta}} \ln \pi(a | s; \boldsymbol{\theta}). \quad (8.2)$$

公式中的 $Q_\pi - V_\pi$ 被称作优势函数 (Advantage Function)。因此，基于上面公式得到的 Actor-Critic 方法被称为 Advantage Actor-Critic，缩写 A2C。

A2C 属于 Actor-Critic 方法。有一个策略网络 $\pi(a|s; \boldsymbol{\theta})$ ，相当于演员，用于控制智能体运动。还有一个价值网络 $v(s; \mathbf{w})$ ，相当于评委，他的评分可以帮助策略网络（演员）改进技术。两个神经网络的结构与上一节中的完全相同，但是本节和上一节用不同的方法训练两个神经网络。

8.3.1 算法推导

训练价值网络：训练价值网络 $v(s; \mathbf{w})$ 的算法是从贝尔曼公式来的：

$$V_\pi(s_t) = \mathbb{E}_{A_t \sim \pi(\cdot | s_t; \boldsymbol{\theta})} \left[\mathbb{E}_{S_{t+1} \sim p(\cdot | s_t, A_t)} \left[R_t + \gamma \cdot V_\pi(S_{t+1}) \right] \right].$$

我们对贝尔曼方程左右两边做近似：

- 方程左边的 $V_\pi(s_t)$ 可以近似成 $v(s_t; \mathbf{w})$ 。 $v(s_t; \mathbf{w})$ 是价值网络在 t 时刻对 $V_\pi(s_t)$ 做出的估计。
- 方程右边的期望是关于当前时刻动作 A_t 与下一时刻状态 S_{t+1} 求的。给定当前状态 s_t ，智能体执行动作 a_t ，环境会给出奖励 r_t 和新的状态 s_{t+1} 。用观测到的 r_t 、 s_{t+1} 对期望做蒙特卡洛近似，得到：

$$r_t + \gamma \cdot V_\pi(s_{t+1}). \quad (8.3)$$

- 进一步把公式 (8.3) 中的 $V_\pi(s_{t+1})$ 近似成 $v(s_{t+1}; \mathbf{w})$ ，得到

$$\hat{y}_t \triangleq r_t + \gamma \cdot v(s_{t+1}; \mathbf{w}).$$

把它称作 TD 目标。它是价值网络在 $t+1$ 时刻对 $V_\pi(s_t)$ 做出的估计。

$v(s_t; \mathbf{w})$ 和 \hat{y}_t 都是对动作价值 $V_\pi(s_t)$ 的估计。由于 \hat{y}_t 部分基于真实观测到的奖励 r_t ，我们认为 \hat{y}_t 比 $v(s_t; \mathbf{w})$ 更可靠。所以把 \hat{y}_t 固定住，更新 \mathbf{w} ，使得 $v(s_t; \mathbf{w})$ 更接近 \hat{y}_t 。

具体这样更新价值网络参数 \mathbf{w} 。定义损失函数

$$L(\mathbf{w}) \triangleq \frac{1}{2} [v(s_t; \mathbf{w}) - \hat{y}_t]^2.$$

设 $\hat{v}_t \triangleq v(s_t; \mathbf{w})$ 。损失函数的梯度是：

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \underbrace{(\hat{v}_t - \hat{y}_t)}_{\text{TD 误差 } \delta_t} \cdot \nabla_{\mathbf{w}} v(s_t; \mathbf{w}).$$

定义 TD 误差为 $\delta_t \triangleq \hat{v}_t - \hat{y}_t$ 。做一轮梯度下降更新 \mathbf{w} :

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} v(s_t; \mathbf{w}).$$

这样可以让价值网络的预测 $v(s_t; \mathbf{w})$ 更接近 \hat{y}_t 。

训练策略网络: A2C 从公式 (8.2) 出发, 对 $\mathbf{g}(s, a; \theta)$ 做近似, 记作 $\tilde{\mathbf{g}}$, 然后用 $\tilde{\mathbf{g}}$ 更新策略网络参数 θ 。下面我们做数学推导。回忆一下贝尔曼公式:

$$Q_{\pi}(s_t, a_t) = \mathbb{E}_{S_{t+1} \sim p(\cdot | s_t, a_t)} [R_t + \gamma \cdot V_{\pi}(S_{t+1})].$$

把近似策略梯度 $\mathbf{g}(s_t, a_t; \theta)$ 中的 $Q_{\pi}(s_t, a_t)$ 替换成上面的期望, 得到:

$$\begin{aligned} \mathbf{g}(s_t, a_t; \theta) &= [Q_{\pi}(s_t, a_t) - V_{\pi}(s_t)] \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta) \\ &= [\mathbb{E}_{S_{t+1}} [R_t + \gamma \cdot V_{\pi}(S_{t+1})] - V_{\pi}(s_t)] \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta). \end{aligned}$$

当智能体执行动作 a_t 之后, 环境给出新的状态 s_{t+1} 和奖励 r_t ; 利用 s_{t+1} 和 r_t 对上面的期望做蒙特卡洛近似, 得到:

$$\mathbf{g}(s_t, a_t; \theta) \approx [r_t + \gamma \cdot V_{\pi}(s_{t+1}) - V_{\pi}(s_t)] \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta).$$

进一步把状态价值函数 $V_{\pi}(s)$ 替换成价值网络 $v(s; \mathbf{w})$, 得到:

$$\tilde{\mathbf{g}}(s_t, a_t; \theta) \triangleq \underbrace{[r_t + \gamma \cdot v(s_{t+1}; \mathbf{w}) - v(s_t; \mathbf{w})]}_{\text{TD 目标 } \hat{y}_t} \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta).$$

前面定义了 TD 目标和 TD 误差:

$$\hat{y}_t \triangleq r_t + \gamma \cdot v(s_{t+1}; \mathbf{w}) \quad \text{和} \quad \delta_t \triangleq v(s_t; \mathbf{w}) - \hat{y}_t.$$

因此, 可以把 $\tilde{\mathbf{g}}$ 写成:

$$\tilde{\mathbf{g}}(s_t, a_t; \theta) \triangleq -\delta_t \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta).$$

$\tilde{\mathbf{g}}$ 是 \mathbf{g} 的近似, 所以也是策略梯度 $\nabla_{\theta} J(\theta)$ 的近似。用 $\tilde{\mathbf{g}}$ 更新策略网络参数 θ :

$$\theta \leftarrow \theta + \beta \cdot \tilde{\mathbf{g}}(s_t, a_t; \theta).$$

这样可以让目标函数 $J(\theta)$ 变大。

策略网络与价值网络的关系: A2C 中策略网络 (演员) 和价值网络 (评委) 的关系如图 8.4 所示。智能体由策略网络 π 控制, 与环境交互, 并收集状态、动作、奖励。策略网络 (演员) 基于状态 s_t 做出动作 a_t 。价值网络 (评委) 基于 s_t, s_{t+1}, r_t 算出 TD 误差 δ_t 。策略网络 (演员) 依靠 δ_t 来判断自己动作的好坏, 从而改进自己的演技 (即参数 θ)。

读者可能会有疑问: 价值网络 v 只知道两个状态 s_t, s_{t+1} , 而并不知道动作 a_t , 那么价值网络为什么能评价 a_t 的好坏呢? 价值网络 v 告诉策略网络 π 的唯一信息是 δ_t 。回顾一下 δ_t 的定义:

$$-\delta_t = \underbrace{r_t + \gamma \cdot v(s_{t+1}; \mathbf{w})}_{\text{TD 误差 } \hat{y}_t} - \underbrace{v(s_t; \mathbf{w})}_{\text{基线}}.$$

基线 $v(s_t; \mathbf{w})$ 是价值网络在 t 时刻对 $\mathbb{E}[U_t]$ 的估计; 此时智能体尚未执行动作 a_t 。而 TD 目标 \hat{y}_t 是价值网络在 $t+1$ 时刻对 $\mathbb{E}[U_t]$ 的估计; 此时智能体已经执行动作 a_t 。

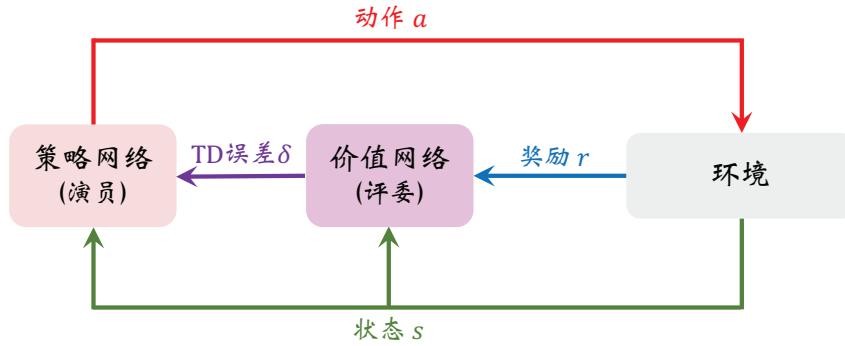


图 8.4: A2C 中策略网络 (演员) 和价值网络 (评委) 的关系图。

- 如果 $\hat{y}_t > v(s_t; \mathbf{w})$, 说明动作 a_t 很好, 使得奖励 r_t 超出预期, 或者新的状态 s_{t+1} 比预期好; 这种情况下应该更新 θ , 使得 $\pi(a_t | s_t; \theta)$ 变大。
- 如果 $\hat{y}_t < v(s_t; \mathbf{w})$, 说明动作 a_t 不好, 导致奖励 r_t 不及预期, 或者新的状态 s_{t+1} 比预期差; 这种情况下应该更新 θ , 使得 $\pi(a_t | s_t; \theta)$ 减小。

综上所述, δ_t 中虽然不包含动作 a_t , 但是 δ_t 可以间接反映出动作 a_t 的好坏, 可以帮助策略网络 (演员) 改进演技。

8.3.2 训练流程

下面概括 A2C 训练流程。设当前策略网络参数是 θ_{now} , 价值网络参数是 \mathbf{w}_{now} 。执行下面的步骤, 将参数更新成 θ_{new} 和 \mathbf{w}_{new} :

1. 观测到当前状态 s_t , 根据策略网络做决策: $a_t \sim \pi(\cdot | s_t; \theta_{\text{now}})$, 并让智能体执行动作 a_t 。
2. 从环境中观测到奖励 r_t 和新的状态 s_{t+1} 。
3. 让价值网络打分:

$$\hat{v}_t = v(s_t; \mathbf{w}_{\text{now}}) \quad \text{和} \quad \hat{v}_{t+1} = v(s_{t+1}; \mathbf{w}_{\text{now}})$$

4. 计算 TD 目标和 TD 误差:

$$\hat{y}_t = r_t + \gamma \cdot \hat{v}_{t+1} \quad \text{和} \quad \delta_t = \hat{v}_t - \hat{y}_t.$$

5. 更新价值网络:

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} v(s_t; \mathbf{w}_{\text{now}}).$$

6. 更新策略网络:

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \beta \cdot \delta_t \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta_{\text{now}}).$$

注 此处训练策略网络和价值网络的方法属于同策略 (On-policy), 要求行为策略 (Behavior Policy) 与目标策略 (Target Policy) 相同, 都是最新的策略网络 $\pi(a|s; \theta_{\text{now}})$ 。不能使用经验回放, 因为经验回放数组中的数据是用旧的策略网络 $\pi(a|s; \theta_{\text{old}})$ 获取的, 不能在当前重复利用。

8.3.3 用目标网络改进训练

上述训练价值网络的算法存在自举——即用价值网络自己的估值 \hat{v}_{t+1} 去更新价值网络自己。为了缓解自举造成的偏差，可以使用目标网络 (Target Network) 计算 TD 目标。把目标网络记作 $v(s; \mathbf{w}^-)$ ，它的结构与价值网络的结构相同，但是参数不同。使用目标网络计算 TD 目标，那么 A2C 的训练就变成了：

1. 观测到当前状态 s_t ，根据策略网络做决策： $a_t \sim \pi(\cdot | s_t; \boldsymbol{\theta}_{\text{now}})$ ，并让智能体执行动作 a_t 。
2. 从环境中观测到奖励 r_t 和新的状态 s_{t+1} 。
3. 让价值网络给 s_t 打分：

$$\hat{v}_t = v(s_t; \mathbf{w}_{\text{now}}).$$

4. 让目标网络给 s_{t+1} 打分：

$$\bar{v}_{t+1} = v(s_{t+1}; \mathbf{w}_{\text{now}}^-).$$

5. 计算 TD 目标和 TD 误差：

$$\bar{y}_t = r_t + \gamma \cdot \bar{v}_{t+1} \quad \text{和} \quad \delta_t = \hat{v}_t - \bar{y}_t.$$

6. 更新价值网络：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} v(s_t; \mathbf{w}_{\text{now}}).$$

7. 更新策略网络：

$$\boldsymbol{\theta}_{\text{new}} \leftarrow \boldsymbol{\theta}_{\text{now}} - \beta \cdot \delta_t \cdot \nabla_{\boldsymbol{\theta}} \ln \pi(a_t | s_t; \boldsymbol{\theta}_{\text{now}}).$$

8. 设 $\tau \in (0, 1)$ 是需要手动调的超参数。做加权平均更新目标网络的参数：

$$\mathbf{w}_{\text{new}}^- \leftarrow \tau \cdot \mathbf{w}_{\text{new}} + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^-.$$

8.4 证明带基线的策略梯度定理

本节证明带基线的策略梯度定理 8.1。将定理 7.1 与引理 8.2 相结合，即可证得定理 8.1。

引理 8.2

设 b 是任意函数， b 不依赖于 A 。那么对于任意的 s ，

$$\mathbb{E}_{A \sim \pi(\cdot|s; \theta)} \left[b \cdot \frac{\partial \ln \pi(A|s; \theta)}{\partial \theta} \right] = 0.$$



证明 由于基线 b 不依赖于动作 A ，可以把 b 提取到期望外面：

$$\begin{aligned} \mathbb{E}_{A \sim \pi(\cdot|s; \theta)} \left[b \cdot \frac{\partial \ln \pi(A|s; \theta)}{\partial \theta} \right] &= b \cdot \mathbb{E}_{A \sim \pi(\cdot|s; \theta)} \left[\frac{\partial \ln \pi(A|s; \theta)}{\partial \theta} \right] \\ &= b \cdot \sum_{a \in \mathcal{A}} \pi(a|s; \theta) \cdot \frac{\partial \ln \pi(a|s; \theta)}{\partial \theta} \\ &= b \cdot \sum_{a \in \mathcal{A}} \pi(a|s; \theta) \cdot \frac{1}{\pi(a|s; \theta)} \cdot \frac{\partial \pi(a|s; \theta)}{\partial \theta} \\ &= b \cdot \sum_{a \in \mathcal{A}} \frac{\partial \pi(a|s; \theta)}{\partial \theta}. \end{aligned}$$

上式最右边的连加是关于 a 求的，而偏导是关于 θ 求的，因此可以把连加放入偏导内部：

$$\mathbb{E}_{A \sim \pi(\cdot|s; \theta)} \left[b \cdot \frac{\partial \ln \pi(A|s; \theta)}{\partial \theta} \right] = b \cdot \frac{\partial}{\partial \theta} \underbrace{\sum_{a \in \mathcal{A}} \pi(a|s; \theta)}_{\text{恒等于 } 1}.$$

因此

$$\mathbb{E}_{A \sim \pi(\cdot|s; \theta)} \left[b \cdot \frac{\partial \ln \pi(A|s; \theta)}{\partial \theta} \right] = b \cdot \frac{\partial 1}{\partial \theta} = 0.$$



《深度强化学习》2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

第九章 策略学习高级技巧

本章介绍策略学习的高级技巧。第 9.1 节介绍置信域策略优化 (TRPO)，它是一种策略学习方法，可以代替策略梯度方法。第 9.2 节介绍熵正则，可以用在所有的策略学习方法中。

9.1 Trust Region Policy Optimization (TRPO)

置信域策略优化 (Trust Region Policy Optimization, TRPO) 是一种策略学习方法，跟以前学的策略梯度有很多相似之处。跟策略梯度方法相比，TRPO 有两个优势：第一，TRPO 表现更稳定，收敛曲线不会剧烈波动，而且对学习率不敏感；第二，TRPO 用更少的经验（即智能体收集到的状态、动作、奖励）就能达到与策略梯度方法相同的表现。

学习 TRPO 的关键在于理解置信域方法 (Trust Region Methods)。置信域方法不是 TRPO 的论文提出的，而是数值最优化领域中一类经典的算法，历史至少可以追溯到 1970 年。TRPO 论文的贡献在于巧妙地把置信域方法应用到强化学习中，取得非常好的效果。

本节分以下 4 小节讲解 TRPO：第 9.1.1 小节介绍置信域方法，第 9.1.2 节回顾策略学习，第 9.1.3 节推导 TRPO，第 9.1.4 节讲解 TRPO 的算法流程。

9.1.1 置信域方法

有这样一个优化问题： $\max_{\theta} J(\theta)$ 。这里的 $J(\theta)$ 是目标函数， θ 是优化变量。求解这个优化问题的目的是找到一个变量 θ 使得目标函数 $J(\theta)$ 取得最大值。有各种各样的优化算法用于解决这个问题。几乎所有的数值优化算法都是做这样的迭代：

$$\theta_{\text{new}} \leftarrow \text{Update}(\text{Data}; \theta_{\text{now}}).$$

此处的 θ_{now} 和 θ_{new} 分别是优化变量当前的值和新的值。不同算法的区别在于具体怎么样利用数据更新优化变量。

置信域方法用到一个概念——置信域。下面介绍置信域。给定变量当前的值 θ_{now} ，用 $\mathcal{N}(\theta_{\text{now}})$ 表示 θ_{now} 的一个邻域。举个例子：

$$\mathcal{N}(\theta_{\text{now}}) = \left\{ \theta \mid \|\theta - \theta_{\text{now}}\|_2 \leq \Delta \right\}. \quad (9.1)$$

这个例子中，集合 $\mathcal{N}(\theta_{\text{now}})$ 是以 θ_{now} 为球心、以 Δ 为半径的球；见右图。球中的点都足够接近 θ_{now} 。

置信域方法需要构造一个函数 $L(\theta | \theta_{\text{now}})$ ，这个函数要满足这个条件：

$$L(\theta | \theta_{\text{now}}) \text{ 很接近 } J(\theta), \quad \forall \theta \in \mathcal{N}(\theta_{\text{now}}),$$

那么集合 $\mathcal{N}(\theta_{\text{now}})$ 就被称作置信域。顾名思义，在 θ_{now} 的邻域上，我们可以信任 $L(\theta | \theta_{\text{now}})$ ，可以拿 $L(\theta | \theta_{\text{now}})$ 来替代目标函数 $J(\theta)$ 。

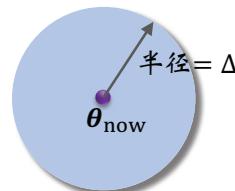


图 9.1：公式(9.1)中的邻域 $\mathcal{N}(\theta_{\text{now}})$ 。

图 9.2 用一个一元函数的例子解释 $J(\theta)$ 和 $L(\theta | \theta_{\text{now}})$ 的关系。图中横轴是优化变量 θ , 纵轴是函数值。如图 9.2(a) 所示, 函数 $L(\theta | \theta_{\text{now}})$ 未必在整个定义域上都接近 $J(\theta)$, 而只是在 θ_{now} 的领域里接近 $J(\theta)$ 。 θ_{now} 的邻域就叫做置信域。

通常来说, J 是个很复杂的函数, 我们甚至可能不知道 J 的解析表达式 (比如 J 是某个函数的期望)。而我们人为构造出的函数 L 相对较为简单, 比如 L 是 J 的蒙特卡洛近似, 或者是 J 在 θ_{now} 这个点的二阶泰勒展开。既然可以信任 L , 那么不妨用 L 代替复杂的函数 J , 然后对 L 做最大化。这样比直接优化 J 要容易得多。这就是**置信域方法**的思想。具体来说, 置信域方法做下面这两个步骤, 一直重复下去, 当无法让 J 的值增大的时候终止算法。

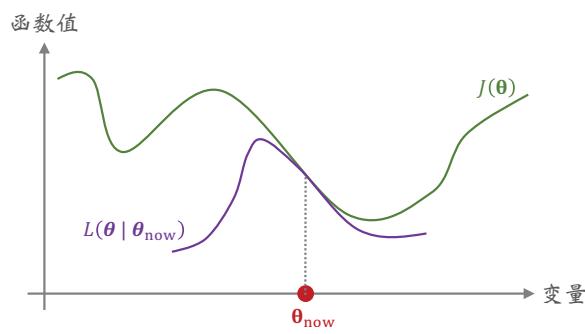
第一步——做近似: 给定 θ_{now} , 构造函数 $L(\theta | \theta_{\text{now}})$, 使得对于所有的 $\theta \in \mathcal{N}(\theta_{\text{now}})$, 函数值 $L(\theta | \theta_{\text{now}})$ 与 $J(\theta)$ 足够接近。图 9.2(b) 解释了做近似这一步。

第二步——最大化: 在置信域 $\mathcal{N}(\theta_{\text{now}})$ 中寻找变量 θ 的值, 使得函数 L 的值最大化。把找到的值记作

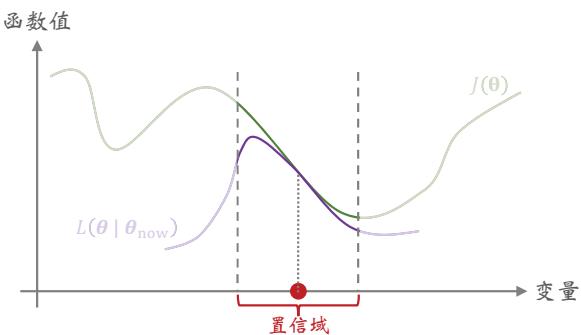
$$\theta_{\text{new}} = \underset{\theta \in \mathcal{N}(\theta_{\text{now}})}{\operatorname{argmax}} L(\theta | \theta_{\text{now}}).$$

图 9.2(c) 解释了最大化这一步。

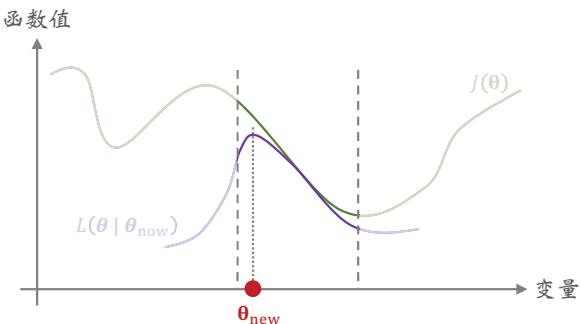
置信域方法其实是一类算法框架, 而非一个具体的算法。有很多种方式实现实现置信域方法。第一步需要做近似, 而做近似的方法有多种多样, 比如蒙特卡洛、二阶泰勒展开。第二步需要解一个带约束的最大化问题; 求解这个问题又需要单独的数值优化算法, 比如梯度投影算法、拉格朗日法。除此之外, 置信域 $\mathcal{N}(\theta_{\text{now}})$ 也有多种多样的选择, 既可以是球, 也可以是两个概率分布的 KL 散度 (KL Divergence), 稍后会介绍。



(a) 构造 $L(\theta | \theta_{\text{now}})$ 作为 $J(\theta)$ 在点 θ_{now} 附近的近似。



(b) L 在点 θ_{now} 的邻域内接近 J ; 这个领域就叫置信域。



(c) 在置信域内寻找最大化 L 的解, 记作 θ_{new} 。

图 9.2: 一元函数的例子解释置信域和置信域算法。

9.1.2 策略学习

首先复习策略学习的基础知识。策略网络记作 $\pi(a|s; \theta)$ ，它是个概率质量函数。动作价值函数记作 $Q_\pi(s, a)$ ，它是回报的期望。状态价值函数记作

$$V_\pi(s) = \mathbb{E}_{A \sim \pi(\cdot|s; \theta)}[Q_\pi(s, A)] = \sum_{a \in \mathcal{A}} \pi(a|s; \theta) \cdot Q_\pi(s, a). \quad (9.2)$$

注意， $V_\pi(s)$ 依赖于策略网络 π ，所以依赖于 π 的参数 θ 。策略学习的目标函数是

$$J(\theta) = \mathbb{E}_S[V_\pi(S)]. \quad (9.3)$$

$J(\theta)$ 只依赖于 θ ，不依赖于状态 S 和动作 A 。第 7 章介绍的策略梯度方法（包括 REINFORCE 和 Actor-Critic）用蒙特卡洛近似梯度 $\nabla_\theta J(\theta)$ ，得到随机梯度，然后做随机梯度上升更新 θ ，使得目标函数 $J(\theta)$ 增大。

下面我们要把目标函数 $J(\theta)$ 变换成一种等价形式。从等式(9.2)出发，把状态价值写成

$$\begin{aligned} V_\pi(s) &= \sum_{a \in \mathcal{A}} \pi(a|s; \theta_{\text{now}}) \cdot \frac{\pi(a|s; \theta)}{\pi(a|s; \theta_{\text{now}})} \cdot Q_\pi(s, a) \\ &= \mathbb{E}_{A \sim \pi(\cdot|s; \theta_{\text{now}})} \left[\frac{\pi(A|s; \theta)}{\pi(A|s; \theta_{\text{now}})} \cdot Q_\pi(s, A) \right]. \end{aligned} \quad (9.4)$$

第一个等式很显然，因为连加中的第一项可以消掉第二项的分母。第二个等式把策略网络 $\pi(A|s; \theta_{\text{now}})$ 看做动作 A 的概率质量函数，所以可以把连加写成期望。由公式 (9.3) 与 (9.4) 可得定理 9.1。定理 9.1 是 TRPO 的关键所在，甚至可以说 TRPO 就是从这个公式推出的。

定理 9.1. 目标函数的等价形式

目标函数 $J(\theta)$ 可以等价写成：

$$J(\theta) = \mathbb{E}_S \left[\mathbb{E}_{A \sim \pi(\cdot|S; \theta_{\text{now}})} \left[\frac{\pi(A|S; \theta)}{\pi(A|S; \theta_{\text{now}})} \cdot Q_\pi(S, A) \right] \right].$$



公式中的期望是关于状态 S 和动作 A 求的。状态 S 的概率密度函数只有环境知道，而我们并不知道，但是我们可以从环境中获取 S 的观测值。动作 A 的概率质量函数是策略网络 $\pi(A|S; \theta_{\text{now}})$ ；注意，策略网络的参数是旧的值 θ_{now} 。

9.1.3 TRPO 数学推导

前面介绍了数值优化的基础和价值学习的基础，终于可以开始推导 TRPO。TRPO 是置信域方法在策略学习中的应用，所以 TRPO 也遵循置信域方法的框架，重复做近似和最大化这两个步骤，直到算法收敛。收敛指的是无法增大目标函数 $J(\theta)$ ，即无法增大期望回报。

第一步——做近似： 我们从定理 9.1 出发。定理把目标函数 $J(\theta)$ 写成了期望的形式。我们无法直接算出期望，无法得到 $J(\theta)$ 的解析表达式；原因在于只有环境知道状态 S 的概率密度函数，而我们不知道。我们可以对期望做蒙特卡洛近似，从而把函数 J 近

似成函数 L 。用策略网络 $\pi(A|S; \theta_{\text{now}})$ 控制智能体跟环境交互，从头到尾玩完一局游戏，观测到一条轨迹：

$$s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_n, a_n, r_n.$$

其中的状态 $\{s_t\}_{t=1}^n$ 都是从环境中观测到的，其中的动作 $\{a_t\}_{t=1}^n$ 都是根据策略网络 $\pi(\cdot|s_t; \theta_{\text{now}})$ 抽取的样本。所以，

$$\frac{\pi(a_t|s_t; \theta)}{\pi(a_t|s_t; \theta_{\text{now}})} \cdot Q_\pi(s_t, a_t) \quad (9.5)$$

是对定理 9.1 中期望的无偏估计。我们观测到了 n 组状态和动作，于是应该对公式 (9.5) 求平均，把得到均值记作：

$$L(\theta|\theta_{\text{now}}) = \frac{1}{n} \sum_{t=1}^n \underbrace{\frac{\pi(a_t|s_t; \theta)}{\pi(a_t|s_t; \theta_{\text{now}})} \cdot Q_\pi(s_t, a_t)}_{\text{定理 9.1 中期望的无偏估计}}. \quad (9.6)$$

既然连加里每一项都是期望的无偏估计，那么 n 项的均值 L 也是无偏估计。所以可以拿 L 作为目标函数 J 的蒙特卡洛近似。

公式 (9.6) 中的 $L(\theta|\theta_{\text{now}})$ 是对目标函数 $J(\theta)$ 的近似。可惜我们还无法直接对 L 求最大化，原因是我们知道动作价值 $Q_\pi(s_t, a_t)$ 。解决方法是把 $Q_\pi(s_t, a_t)$ 近似成观测到的折扣回报：

$$u_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \dots + \gamma^{n-t} \cdot r_n.$$

拿 u_t 替代 $Q_\pi(s_t, a_t)$ ，那么公式 (9.6) 中的 $L(\theta|\theta_{\text{now}})$ 变成了

$$\tilde{L}(\theta|\theta_{\text{now}}) = \sum_{t=1}^n \frac{\pi(a_t|s_t; \theta)}{\pi(a_t|s_t; \theta_{\text{now}})} \cdot u_t. \quad (9.7)$$

总结一下，我们把目标函数 J 近似成 L ，然后又把 L 近似成 \tilde{L} 。

第二步——最大化： TRPO 把公式(9.7)中的 $\tilde{L}(\theta|\theta_{\text{now}})$ 作为对目标函数 $J(\theta)$ 的近似，然后求解这个带约束的最大化问题：

$$\max_{\theta} \tilde{L}(\theta|\theta_{\text{now}}); \quad \text{s.t. } \theta \in \mathcal{N}(\theta_{\text{now}}). \quad (9.8)$$

公式中的 $\mathcal{N}(\theta_{\text{now}})$ 是置信域，即 θ_{now} 的一个邻域。该用什么样的置信域呢？

- 一种方法是用以 θ_{now} 为球心、以 Δ 为半径的球作为置信域。这样的话，公式(9.8)就变成

$$\max_{\theta} \tilde{L}(\theta|\theta_{\text{now}}); \quad \text{s.t. } \|\theta - \theta_{\text{now}}\|_2 \leq \Delta. \quad (9.9)$$

- 另一种方法是用 KL 散度衡量两个概率质量函数—— $\pi(\cdot|s_i; \theta_{\text{now}})$ 和 $\pi(\cdot|s_i; \theta)$ ——的距离。两个概率质量函数区别越大，它们的 KL 散度就越大。反之，如果 θ 很接近 θ_{now} ，那么两个概率质量函数就越接近。用 KL 散度的话，公式(9.8)就变成

$$\max_{\theta} \tilde{L}(\theta|\theta_{\text{now}}); \quad \text{s.t. } \frac{1}{t} \sum_{i=1}^t \text{KL}\left[\pi(\cdot|s_i; \theta_{\text{now}}) \parallel \pi(\cdot|s_i; \theta)\right] \leq \Delta. \quad (9.10)$$

用球作为置信域的好处是置信域是简单的形状，求解最大化问题比较容易，但是用球做

置信域的实际效果不如用 KL 散度。

TRPO 的第二步——最大化——需要求解带约束的最大化问题 (9.9) 或者 (9.10)。注意，这种问题的求解不容易；简单的梯度上升算法并不能解带约束的最大化问题。数值优化教材通常有介绍带约束问题的求解，有兴趣的话自己去阅读数值优化教材，这里就不详细解释如何求解问题 (9.9) 或者 (9.10)。读者可以这样看待优化问题：只要你能把一个优化问题的目标函数和约束条件解析地写出来，通常会有数值算法能解决这个问题。

9.1.4 训练流程

在本节的最后，我们总结一下用 TRPO 训练策略网络的流程。TRPO 需要重复做近似和最大化这两个步骤：

1. 做近似——构造函数 \tilde{L} 近似目标函数 $J(\theta)$ ：

- 设当前策略网络参数是 θ_{now} 。用策略网络 $\pi(a | s; \theta_{\text{now}})$ 控制智能体与环境交互，玩完一局游戏，记录下轨迹：

$$s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_n, a_n, r_n.$$

- 对于所有的 $t = 1, \dots, n$ ，计算折扣回报 $u_t = \sum_{k=t}^n \gamma^{k-t} \cdot r_k$ 。

- 得出近似函数：

$$\tilde{L}(\theta | \theta_{\text{now}}) = \sum_{t=1}^n \frac{\pi(a_t | s_t; \theta)}{\pi(a_t | s_t; \theta_{\text{now}})} \cdot u_t.$$

2. 最化——用某种数值算法求解带约束的最大化问题：

$$\theta_{\text{new}} = \underset{\theta}{\operatorname{argmax}} \tilde{L}(\theta | \theta_{\text{now}}); \quad \text{s.t. } \|\theta - \theta_{\text{now}}\|_2 \leq \Delta.$$

此处的约束条件是二范数距离。可以把它替换成 KL 散度，即公式 (9.10)。

TRPO 中有两个需要调的超参数：一个是置信域的半径 Δ ，另一个是求解最大化问题的数值算法的学习率。通常来说， Δ 在算法的运行过程中要逐渐缩小。虽然 TRPO 需要调参，但是 TRPO 对超参数的设置并不敏感。即使超参数设置不够好，TRPO 的表现也不会太差。相比之下，策略梯度算法对超参数更敏感。

TRPO 算法真正实现起来不容易，主要难点在于第二步——最大化。不建议读者自己去实现 TRPO。

9.2 熵正则 (Entropy Regularization)

策略学习的目的是学出一个策略网络 $\pi(a|s; \theta)$ 用于控制智能体。每当智能体观测到当前状态 s , 策略网络输出一个概率分布, 智能体依据概率分布抽样一个动作, 并执行这个动作。举个例子, 在超级玛丽游戏中, 动作空间是 $\mathcal{A} = \{\text{左, 右, 上}\}$ 。基于当前状态 s , 策略网络的输出是

$$p_1 = \pi(\text{左} | s; \theta) = 0.03,$$

$$p_2 = \pi(\text{右} | s; \theta) = 0.96,$$

$$p_3 = \pi(\text{上} | s; \theta) = 0.01.$$

那么超级玛丽做的动作可能是左、右、上三者中的任何一个, 概率分别是 0.03, 0.96, 0.01。概率都集中在“向右”的动作上, 接近确定性的决策。确定性大的好处在于不容易选中很差的动作, 比较安全。但是确定性大也有缺点。假如策略网络的输出总是这样确定性很大的概率分布, 那么智能体就会安于现状, 不去尝试没做过的动作, 不去探索更多的状态, 无法找到更好的策略。

我们希望策略网络的输出的概率不要集中在一个动作上, 至少要给其他动作一些非零的概率, 让这些动作能被探索到。可以用熵 (Entropy) 来衡量概率分布的不确定性。对于上述离散概率分布 $\mathbf{p} = [p_1, p_2, p_3]$, 熵等于

$$\text{Entropy}(\mathbf{p}) = - \sum_{i=1}^3 p_i \cdot \ln p_i.$$

熵小说明概率质量很集中, 熵大说明随机性很大; 见图 9.3 的解释。

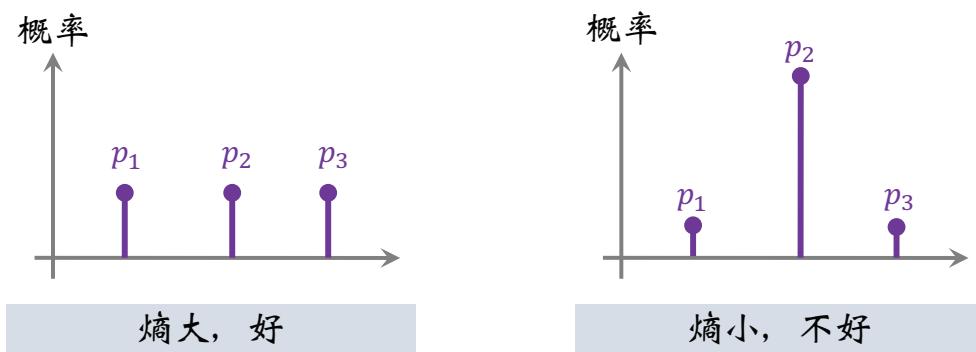


图 9.3: 两张图中分别描述两个离散概率分布。左边的概率比较均匀, 这种情况熵很大。右边的概率集中在 p_2 上, 这种情况的熵较小。

策略学习中的熵正则: 我们希望策略网络输出的概率分布的熵不要太小。我们不妨把熵作为正则项, 放到策略学习的目标函数中。策略网络的输出是维度等于 $|\mathcal{A}|$ 的向量, 它表示定义在动作空间上的离散概率分布。这个概率分布的熵定义为:

$$H(s; \theta) \triangleq \text{Entropy} [\pi(\cdot | s; \theta)] = - \sum_{a \in \mathcal{A}} \pi(a | s; \theta) \cdot \ln \pi(a | s; \theta). \quad (9.11)$$

熵 $H(s; \theta)$ 只依赖于状态 s 与策略网络参数 θ 。我们希望对于大多数的状态 s , 熵都会比

9.2 熵正则 (Entropy Regularization)

较大，也就是让 $\mathbb{E}_S[H(S; \theta)]$ 比较大。

回忆一下， $V_\pi(s)$ 是状态价值函数，衡量在状态 s 的情况下，策略网络 π 表现的好坏程度。策略学习的目标函数是 $J(\theta) = \mathbb{E}_S[V_\pi(S)]$ 。策略学习的目的是寻找参数 θ 使得 $J(\theta)$ 最大化。同时，我们还希望让熵比较大，所以把熵作为正则项，放到目标函数里。使用熵正则的策略学习可以写作这样的最大化问题：

$$\max_{\theta} J(\theta) + \lambda \cdot \mathbb{E}_S[H(S; \theta)]. \quad (9.12)$$

此处的 λ 是个超参数，需要手动调。

优化：带熵正则的最大化问题 (9.12) 可以用各种方法求解，比如策略梯度方法（包括 REINFORCE 和 Actor-Critic）、TRPO 等。此处只讲解策略梯度方法。公式 (9.12) 中目标函数关于 θ 的梯度是：

$$g(\theta) \triangleq \nabla_{\theta} [J(\theta) + \lambda \cdot \mathbb{E}_S[H(S; \theta)]].$$

观测到状态 s ，按照策略网络做随机抽样，得到动作 $a \sim \pi(\cdot | s; \theta)$ 。那么

$$\tilde{g}(s, a; \theta) \triangleq [Q_\pi(s, a) - \lambda \cdot \ln \pi(a | s; \theta) - \lambda] \cdot \nabla_{\theta} \ln \pi(a | s; \theta)$$

是梯度 $g(\theta)$ 的无偏估计（见定理 9.2）。因此可以用 $\tilde{g}(s, a; \theta)$ 更新策略网络的参数：

$$\theta \leftarrow \theta + \beta \cdot \tilde{g}(s, a; \theta).$$

此处的 β 是学习率。

定理 9.2. 带熵正则的策略梯度

$$\nabla_{\theta} [J(\theta) + \lambda \cdot \mathbb{E}_S[H(S; \theta)]] = \mathbb{E}_S [\mathbb{E}_{A \sim \pi(\cdot | s; \theta)} [\tilde{g}(S, A; \theta)]].$$

证明 首先推导熵 $H(S; \theta)$ 关于 θ 的梯度。由公式 (9.11) 中 $H(S; \theta)$ 的定义可得

$$\begin{aligned} \frac{\partial H(s; \theta)}{\partial \theta} &= - \sum_{a \in \mathcal{A}} \frac{\partial \pi(a | s; \theta) \cdot \ln \pi(a | s; \theta)}{\partial \theta} \\ &= - \sum_{a \in \mathcal{A}} \left[\ln \pi(a | s; \theta) \cdot \frac{\partial \pi(a | s; \theta)}{\partial \theta} + \pi(a | s; \theta) \cdot \frac{\partial \ln \pi(a | s; \theta)}{\partial \theta} \right]. \end{aligned}$$

第二个等式由链式法则得到。由于 $\frac{\partial \pi(a | s; \theta)}{\partial \theta} = \pi(a | s; \theta) \cdot \frac{\partial \ln \pi(a | s; \theta)}{\partial \theta}$ ，上面的公式可以写成：

$$\begin{aligned} \frac{\partial H(s; \theta)}{\partial \theta} &= - \sum_{a \in \mathcal{A}} \left[\ln \pi(a | s; \theta) \cdot \pi(a | s; \theta) \cdot \frac{\partial \ln \pi(a | s; \theta)}{\partial \theta} + \pi(a | s; \theta) \cdot \frac{\partial \ln \pi(a | s; \theta)}{\partial \theta} \right] \\ &= - \sum_{a \in \mathcal{A}} \pi(a | s; \theta) \cdot \left[\ln \pi(a | s; \theta) + 1 \right] \cdot \frac{\partial \ln \pi(a | s; \theta)}{\partial \theta} \\ &= - \mathbb{E}_{A \sim \pi(\cdot | s; \theta)} \left[\left[\ln \pi(A | s; \theta) + 1 \right] \cdot \frac{\partial \ln \pi(A | s; \theta)}{\partial \theta} \right]. \end{aligned} \quad (9.13)$$

应用第 7 章推导的策略梯度定理，可以把 $J(\theta)$ 关于 θ 的梯度写作

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_S \left\{ \mathbb{E}_{A \sim \pi(\cdot|S;\theta)} \left[Q_\pi(S, A) \cdot \frac{\partial \ln \pi(A|S;\theta)}{\partial \theta} \right] \right\}. \quad (9.14)$$

由公式 (9.13) 与 (9.14) 可得：

$$\begin{aligned} & \frac{\partial}{\partial \theta} \left[J(\theta) + \lambda \cdot \mathbb{E}_S [H(S; \theta)] \right] \\ &= \mathbb{E}_S \left\{ \mathbb{E}_{A \sim \pi(\cdot|S;\theta)} \left[\left(Q_\pi(S, A) - \lambda \cdot \ln \pi(A|S;\theta) - \lambda \right) \cdot \frac{\partial \ln \pi(A|S;\theta)}{\partial \theta} \right] \right\} \\ &= \mathbb{E}_S \left\{ \mathbb{E}_{A \sim \pi(\cdot|S;\theta)} [\tilde{g}(S, A; \theta)] \right\}. \end{aligned}$$

上面第二个等式由 \tilde{g} 的定义得到。 □

∽第九章 相关文献∽

TRPO 由 Schulman 等人在 2015 年提出 [87]。TRPO 是置信域方法在强化学习中的成功应用。置信域是经典的数值优化算法，对此感兴趣的读者可以阅读这些教材：[75, 30]。TRPO 每一轮循环都要求解带约束的最大化问题；这类问题的求解可以参考这些教材：[13, 19]。

熵正则是策略学习中常见的方法，在很多论文中有使用，比如 [119, 68, 76, 3, 43, 88]。虽然熵正则能鼓励探索，但是增大决策的不确定性是有风险的：很差的动作可能也有非零的概率。一个好的办法是用 Tsallis Entropy [103] 做正则，让离散概率具有稀疏性，每次决策只给少部分动作非零的概率，“过滤掉”很差的动作。有兴趣的读者可以阅读这些论文：[29, 58, 120]。

《深度强化学习》 2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

第十章 连续控制

本书前面章节的内容全部都是离散控制，即动作空间是一个离散的集合，比如超级玛丽游戏中的动作空间 $\mathcal{A} = \{\text{左, 右, 上}\}$ 就是个离散集合。本章的内容是连续控制，即动作空间是个连续集合，比如汽车的转向 $\mathcal{A} = [-40^\circ, 40^\circ]$ 就是连续集合。如果把连续动作空间做离散化，那么离散控制的方法就能直接解决连续控制问题；我们在第10.1节讨论连续集合的离散化。然而更好的办法是直接用连续控制方法，而非离散化之后借用离散控制方法。本章介绍两种连续控制方法：第10.2节介绍确定策略网络，第10.4节介绍随机策略网络。

10.1 离散控制与连续控制的区别

考虑这样一个问题：我们需要控制一只机械手臂，完成某些任务，获取奖励。机械手臂有两个关节，分别可以在 $[0^\circ, 360^\circ]$ 与 $[0^\circ, 180^\circ]$ 的范围内转动。这个问题的自由度是 $d = 2$ ，动作是二维向量，动作空间是连续集合 $\mathcal{A} = [0, 360] \times [0, 180]$ 。

此前我们学过的强化学习方法全部都是针对离散动作空间，不能直接解决上述连续控制问题。想把此前学过的离散控制方法应用到连续控制上，必须要对连续动作空间做离散化（网格化）。比如把连续集合 $\mathcal{A} = [0, 360] \times [0, 180]$ 变成离散集合 $\mathcal{A}' = \{0, 20, 40, \dots, 360\} \times \{0, 20, 40, \dots, 180\}$ ；见图 10.1。

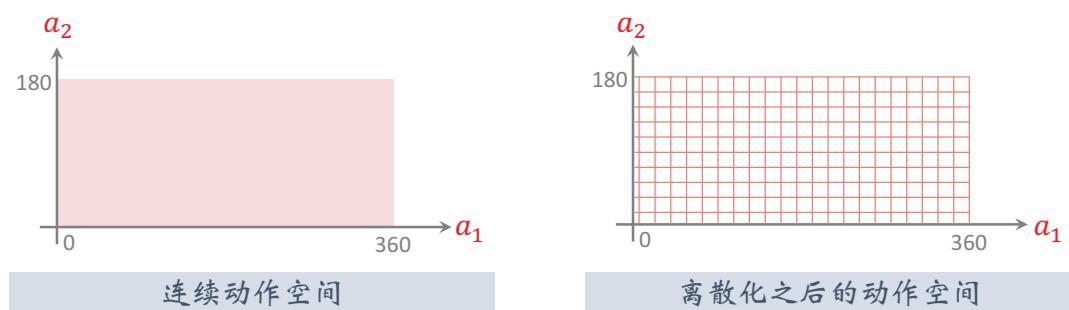


图 10.1：对连续动作空间 $\mathcal{A} = [0, 360] \times [0, 180]$ 做离散化（网格化）。

对动作空间做离散化之后，就可以应用之前学过的方法训练 DQN 或者策略网络，用于控制机械手臂。可是用离散化解决连续控制问题有个缺点。把自由度记作 d 。自由度 d 越大，网格上的点就越多，而且数量随着 d 指数增长，会造成维度灾难。动作空间的大小即网格上点的数量。如果动作空间太大，DQN 和策略网络的训练都变得很困难，强化学习的结果会不好。上述离散化方法只适用于自由度 d 很小的情况；如果 d 不是很小，就应该使用连续控制方法。后面两节介绍两种连续控制的方法。

10.2 确定策略梯度 (DPG)

确定策略梯度 (Deterministic Policy Gradient, DPG) 是最常用的连续控制方法。DPG 是一种 Actor-Critic 方法，它有一个策略网络（演员），一个价值网络（评委）。策略网络控制智能体做运动，它基于状态 s 做出动作 a 。¹ 价值网络不控制智能体，只是基于状态 s 给动作 a 打分，从而指导策略网络做出改进。图 10.2 是两个神经网络的关系。

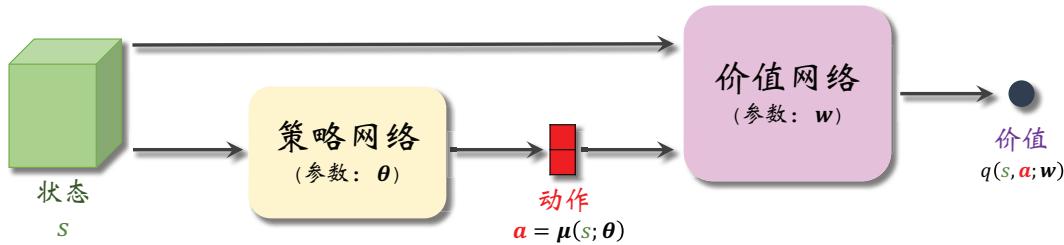


图 10.2: 确定策略梯度 (DPG) 方法的示意图。策略网络 $\mu(s; \theta)$ 的输入是状态 s ，输出是动作 a (d 维向量)。价值网络 $q(s, a; w)$ 的输入是状态 s 和动作 a ，输出是价值 (实数)。

10.2.1 策略网络和价值网络

本节的策略网络完全不同于前面章节的策略网络。在之前章节里，策略网络 $\pi(a|s; \theta)$ 是一个概率质量函数，它输出的是概率值。本节的确定策略网络 $\mu(s; \theta)$ 的输出是 d 维的向量 a ，作为动作。两种策略网络一个是随机的，一个是确定性的：

- 之前章节中的策略网络 $\pi(a|s; \theta)$ 带有随机性：给定状态 s ，策略网络输出的是离散动作空间 \mathcal{A} 上的概率分布； \mathcal{A} 中的每个元素（动作）都有一个概率值。智能体依据概率分布，随机从 \mathcal{A} 中抽取一个动作，并执行动作。
- 本节的确定策略网络没有随机性：对于确定的状态 s ，策略网络 μ 输出的动作 a 是确定的。动作 a 直接是 μ 的输出，而非随机抽样得到的。

确定策略网络 μ 的结构如图 10.3 所示。如果输入的状态 s 是个矩阵或者张量（例如图片、视频），那么 μ 就由若干卷积层、全连接层等组成。

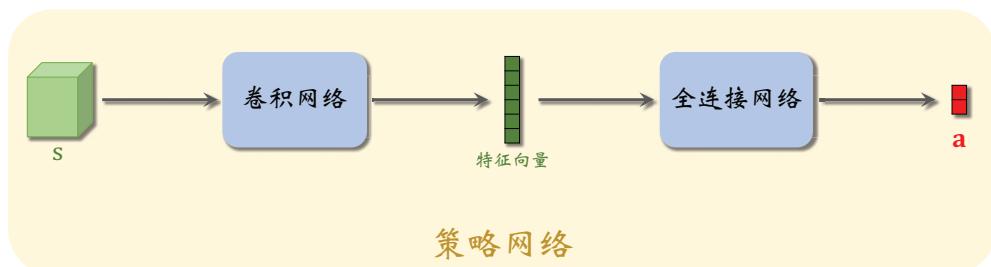


图 10.3: 确定策略网络 $\mu(s; \theta)$ 的结构。输入是状态 s ，输出是动作 a 。

¹本节中，动作 a 是一个 d 维向量，而不是离散集合中的一个元素。因此本节用粗体 a 表示动作的观测值。

本节的**价值网络** $q(s, a; w)$ 是对动作价值函数 $Q_\pi(s, a)$ 的近似。价值网络的结构如图 10.4 所示。价值网络的输入是状态 s 和动作 a ，输出的价值 $\hat{q} = q(s, a; w)$ 是个实数，可以反映动作的好坏；动作 a 越好，则价值 \hat{q} 就越大。所以价值网络可以评价策略网络的表现。在训练的过程中，价值网络帮助训练策略网络；在训练结束之后，价值网络就被丢弃，由策略网络控制智能体。

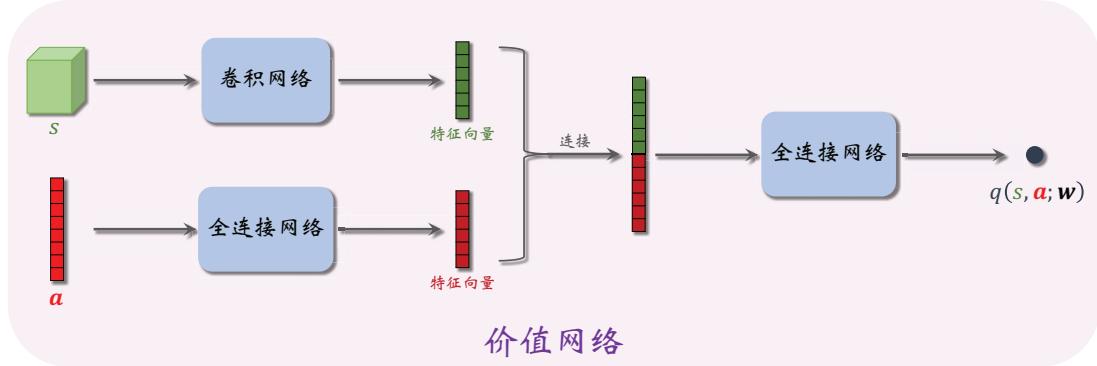


图 10.4：价值网络 $q(s, a; w)$ 的结构。输入是状态 s 和动作 a ，输出是实数。

10.2.2 算法推导

用行为策略收集经验： 本节的确定策略网络属于异策略 (Off-policy) 方法，即行为策略 (Behavior Policy) 可以不同于目标策略 (Target Policy)。目标策略即确定策略网络 $\mu(s; \theta_{\text{now}})$ ，其中 θ_{now} 是策略网络最新的参数。行为策略可以是任意的，比如

$$a = \mu(s; \theta_{\text{old}}) + \epsilon.$$

公式的意思是行为策略可以用过时的策略网络参数，而且可以往动作中加入噪声 $\epsilon \in \mathbb{R}^d$ 。异策略的好处在于可以把**收集经验与训练神经网络分割开**；把收集到的经验存入经验回放数组 (Replay Buffer)，在做训练的时候重复利用收集到的经验。见图 10.5。

用行为策略控制智能体与环境交互，把智能体的轨迹 (Trajectory) 整理成 (s_t, a_t, r_t, s_{t+1}) 这样的四元组，存入经验回放数组。在训练的时候，随机从数组中抽取一个四元组，记作 (s_j, a_j, r_j, s_{j+1}) 。在训练策略网络 $\mu(s; \theta)$ 的时候，只用到状态 s_j 。在训练价值网络 $q(s, a; w)$ 的时候，要用到四元组中全部四个元素： s_j, a_j, r_j, s_{j+1} 。

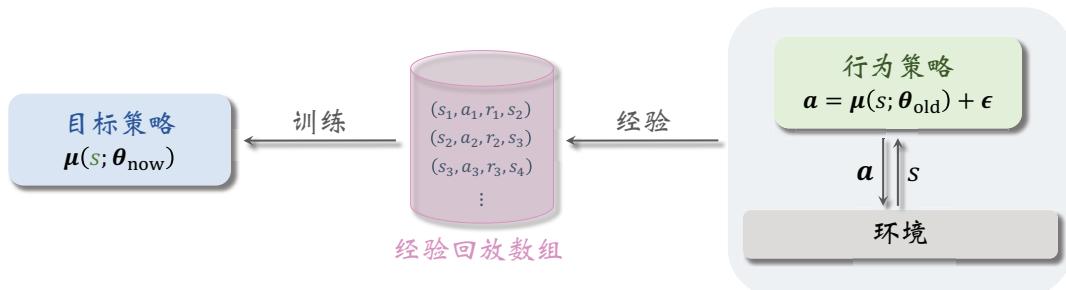


图 10.5：DPG 属于异策略，收集经验与更新策略分开做。

训练策略网络：首先通俗解释训练策略网络的原理。如图 10.6 所示，给定状态 s ，策略网络输出一个动作 $a = \mu(s; \theta)$ ，然后价值网络会给 a 打一个分数： $\hat{q} = q(s, a; w)$ 。参数 θ 影响 a ，从而影响 \hat{q} 。分数 \hat{q} 可以反映出 θ 的好坏程度。训练策略网络的目标就是改进参数 θ ，使 \hat{q} 变得更大。把策略网络看做演员，价值网络看做评委。训练演员（策略网络）的目的就是让他迎合评委（价值网络）的喜好，改变自己的表演技巧（即参数 θ ），使得评委打分 \hat{q} 的均值更高。

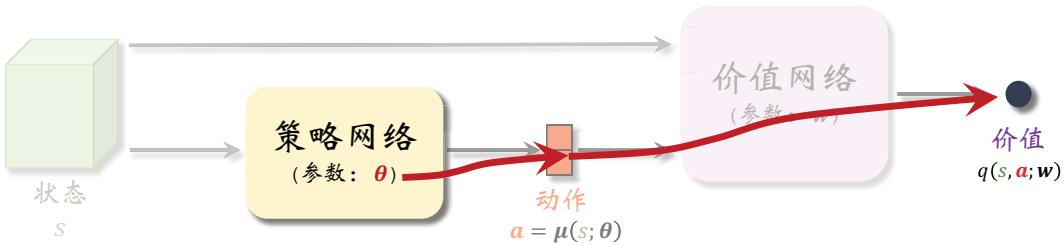


图 10.6：给定状态 s ，策略网络的参数 θ 会影响 a ，从而影响 $\hat{q} = q(s, a; w)$ 。

根据以上解释，我们来推导目标函数。如果当前状态是 s ，那么价值网络的打分就是：

$$q(s, \mu(s; \theta); w).$$

我们希望打分的期望尽量高，所以把目标函数定义为打分的期望：

$$J(\theta) = \mathbb{E}_S [q(S, \mu(S; \theta); w)].$$

关于状态 S 求期望消除了 S 的影响；不管面对什么样的状态 S ，策略网络（演员）都应该做出很好的动作，使得平均分 $J(\theta)$ 尽量高。策略网络的学习可以建模成这样一个最大化问题：

$$\max_{\theta} J(\theta).$$

注意，这里我们只训练策略网络，所以最大化问题中的优化变量是策略网络的参数 θ ，而价值网络的参数 w 被固定住。

可以用梯度上升来增大 $J(\theta)$ 。每次用随机变量 S 的一个观测值（记作 s_j ）来计算梯度：

$$\mathbf{g}_j \triangleq \nabla_{\theta} q(s_j, \mu(s_j; \theta); w).$$

它是 $\nabla_{\theta} J(\theta)$ 的无偏估计。 \mathbf{g}_j 叫做**确定策略梯度** (Deterministic Policy Gradient)，缩写 DPG。

可以用链式法则求出梯度 \mathbf{g}_j 。复习一下链式法则。如果有这样的函数关系： $\theta \rightarrow a \rightarrow q$ ，那么 q 关于 θ 的导数可以写成

$$\frac{\partial q}{\partial \theta} = \frac{\partial a}{\partial \theta} \cdot \frac{\partial q}{\partial a}$$

价值网络的输出与 θ 的函数关系如图 10.6 所示。应用链式法则，我们得到下面的定理。

定理 10.1. 确定策略梯度

$$\nabla_{\theta} q(s_j, \mu(s_j; \theta); w) = \nabla_{\theta} \mu(s_j; \theta) \cdot \nabla_a q(s_j, \hat{a}_j; w), \quad \text{其中 } \hat{a}_j = \mu(s_j; \theta).$$



由此我们得到更新 θ 的算法。每次从经验回放数组里随机抽取一个状态，记作 s_j 。计算 $\hat{a}_j = \mu(s_j; \theta)$ 。用梯度上升更新一次 θ :

$$\theta \leftarrow \theta + \beta \cdot \nabla_{\theta} \mu(s_j; \theta) \cdot \nabla_a q(s_j, \hat{a}_j; w).$$

此处的 β 是学习率，需要手动调。这样做梯度上升，可以逐渐让目标函数 $J(\theta)$ 增大，也就是让评委给演员的平均打分更高。

训练价值网络：首先通俗解释训练价值网络的原理。训练价值网络的目标是让价值网络 $q(s, a; w)$ 的预测越来越接近真实价值函数 $Q_{\pi}(s, a)$ 。如果把价值网络看做评委，那么训练评委的目标就是让他的打分越来越准确。每一轮训练都要用到一个实际观测的奖励 r ，可以把 r 看做“真理”，用它来校准评委的打分。

训练价值网络要用 TD 算法。这里的 TD 算法与之前学过的标准 Actor-Critic 类似，都是让价值网络去拟合 TD 目标。每次从经验回放数组中取出一个四元组 (s_j, a_j, r_j, s_{j+1}) ，用它更新一次参数 w 。首先让价值网络做预测：

$$\hat{q}_j = q(s_j, a_j; w) \quad \text{和} \quad \hat{q}_{j+1} = q(s_{j+1}, \mu(s_{j+1}; \theta); w).$$

计算 TD 目标 $\hat{y}_j = r_j + \gamma \cdot \hat{q}_{j+1}$ 。定义损失函数

$$L(w) = \frac{1}{2} \left[q(s_j, a_j; w) - \hat{y}_j \right]^2,$$

计算梯度

$$\nabla_w L(w) = \underbrace{(\hat{q}_j - \hat{y}_j)}_{\text{TD 误差 } \delta_j} \cdot \nabla_w q(s_j, a_j; w),$$

做一轮梯度下降更新参数 w :

$$w \leftarrow w - \alpha \cdot \nabla_w L(w).$$

这样可以让损失函数 $L(w)$ 减小，也就是让价值网络的预测 $\hat{q}_j = q(s, a; w)$ 更接近 TD 目标 \hat{y}_j 。公式中的 α 是学习率，需要手动调。

训练流程：做训练的时候，可以同时对价值网络和策略网络做训练。每次从经验回放数组中抽取一个四元组，记作 (s_j, a_j, r_j, s_{j+1}) 。把神经网络当前参数记作 w_{now} 和 θ_{now} 。执行以下步骤更新策略网络和价值网络：

1. 让策略网络做预测：

$$\hat{a}_j = \mu(s_j; \theta_{\text{now}}) \quad \text{和} \quad \hat{a}_{j+1} = \mu(s_{j+1}; \theta_{\text{now}}).$$

注 计算动作 \hat{a}_j 用的是当前的策略网络 $\mu(s_j; \theta_{\text{now}})$ ，用 \hat{a}_j 来更新 θ_{now} ；而从经验回放数组中抽取的 a_j 则是用过时的策略网络 $\mu(s_j; \theta_{\text{old}})$ 算出的，用 a_j 来更新 w_{now} 。请注意 \hat{a}_j 与 a_j 的区别。

2. 让价值网络做预测：

$$\hat{q}_j = q(s_j, \mathbf{a}_j; \mathbf{w}_{\text{now}}) \quad \text{和} \quad \hat{q}_{j+1} = q(s_{j+1}, \hat{\mathbf{a}}_{j+1}; \mathbf{w}_{\text{now}}).$$

3. 计算 TD 目标和 TD 误差：

$$\hat{y}_j = r_j + \gamma \cdot \hat{q}_{j+1} \quad \text{和} \quad \delta_j = \hat{q}_j - \hat{y}_j.$$

4. 更新价值网络：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_j \cdot \nabla_{\mathbf{w}} q(s_j, \mathbf{a}_j; \mathbf{w}_{\text{now}}).$$

5. 更新策略网络：

$$\boldsymbol{\theta}_{\text{new}} \leftarrow \boldsymbol{\theta}_{\text{now}} + \beta \cdot \nabla_{\boldsymbol{\theta}} \mu(s_j; \boldsymbol{\theta}_{\text{now}}) \cdot \nabla_{\mathbf{a}} q(s_j, \hat{\mathbf{a}}_j; \mathbf{w}_{\text{now}}).$$

在实践中，上述算法的表现并不好；读者应当采用下一节介绍的技巧训练策略网络和价值网络。

10.3 双延时确定策略梯度 (TD3)

上一节介绍的算法实际效果并不好。本节介绍的 Twin Delayed Deep Deterministic Policy Gradient (TD3) 可以大幅提升算法的表现，把策略网络和价值网络训练得更好。注意，本节只是改进训练用的算法，并不改变神经网络的结构。

10.3.1 高估问题及其原因

第 6.2 节讨论过用 Q 学习算法训练 DQN 会导致高估。上一节介绍的 DPG 也会出现相同的高估问题，下面我们分析造成高估的原因。训练策略网络的时候，我们希望策略网络计算出的动作 $\hat{\mathbf{a}} = \mu(s; \boldsymbol{\theta})$ 能得到价值网络尽量高的评价，也就是让 $q(s, \hat{\mathbf{a}}; \mathbf{w})$ 尽量大。我们通过求解下面的优化模型来学习策略网络：

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \mathbb{E}_S [q(S, \hat{A}; \mathbf{w})], \quad \text{s.t. } \hat{A} = \mu(S; \boldsymbol{\theta}).$$

这个公式的意思是 $\mu(s; \boldsymbol{\theta}^*)$ 是最优的确定策略。上面的公式与下面的公式意义相同（虽然不严格等价）：

$$\mu(s; \boldsymbol{\theta}^*) = \underset{\mathbf{a} \in \mathcal{A}}{\operatorname{argmax}} q(s, \mathbf{a}; \mathbf{w}), \quad \forall s \in \mathcal{S}.$$

这个公式的意思也是 $\mu(s; \boldsymbol{\theta}^*)$ 是最优的确定策略。

最大化造成高估： 回忆一下，DQN 的决策方式是 $a_j = \operatorname{argmax}_a Q(s_j, a; \mathbf{w})$ 。上面的分析说明确定策略 μ 做决策的方式是 $a_j \approx \operatorname{argmax}_a q(s_j, a; \mathbf{w})$ 。不难看出，DQN 与 DPG 的决策方式都是选择最大化 Q 或 q 的动作。

第 6.2 节讨论过造成 DQN 的高估问题的原因：最大化会造成 TD 目标 \hat{y}_j 高估真实最优动作价值 $Q_\star(s_j, a_j)$ 。同理，训练价值网络 q 用的 TD 目标

$$\begin{aligned} \hat{y}_j &= r_j + \gamma \cdot q(s_{j+1}, \mu(s_{j+1}; \boldsymbol{\theta}); \mathbf{w}) \\ &\approx r_j + \gamma \cdot \max_{\mathbf{a}_{j+1}} q(s_{j+1}, \mathbf{a}_{j+1}; \mathbf{w}) \end{aligned}$$

是对真实动作价值的高估。TD 算法把 \hat{y}_j 作为目标，鼓励价值网络 $q(s_j, a_j; \mathbf{w})$ 接近 \hat{y}_j 。这会导致 $q(s_j, a_j; \mathbf{w})$ 高估真实动作价值。

自举造成偏差传播： 我们在第 6.2 节讨论过自举 (Bootstrapping) 造成偏差的传播。TD 目标

$$\hat{y}_j = r_j + \gamma \cdot q(s_{j+1}, \mu(s_{j+1}; \boldsymbol{\theta}); \mathbf{w})$$

是用价值网络算出来的，而它又被用于更新价值网络 q 本身，这属于自举。假如价值网络 $q(s_{j+1}, \mathbf{a}_{j+1}; \boldsymbol{\theta})$ 高估了真实动作价值 $Q_\pi(s_{j+1}, \mathbf{a}_{j+1})$ ，那么 TD 目标 \hat{y}_j 则是对 $Q_\pi(s_j, \mathbf{a}_j)$ 的高估，这会导致 $q(s_j, a_j; \mathbf{w})$ 高估 $Q_\pi(s_j, \mathbf{a}_j)$ 。自举让高估从 $(s_{j+1}, \mathbf{a}_{j+1})$ 传播到 (s_j, \mathbf{a}_j) 。

10.3.2 高估问题的解决方案

解决方案——目标网络：为了解决自举和最大化造成的高估，我们需要使用目标网络 (Target Networks) 计算 TD 目标 \hat{y}_j 。训练中需要两个目标网络：

$$q(s, \mathbf{a}; \mathbf{w}^-) \text{ 和 } \mu(s; \boldsymbol{\theta}^-).$$

它们与价值网络、策略网络的结构完全相同，但是参数不同。TD 目标是用目标网络算的：

$$\hat{y}_j = r_j + \gamma \cdot q(s_{j+1}, \hat{\mathbf{a}}_{j+1}; \mathbf{w}^-), \quad \text{其中 } \hat{\mathbf{a}}_{j+1} = \mu(s_{j+1}; \boldsymbol{\theta}^-).$$

把 \hat{y}_j 作为目标，更新 \mathbf{w} ，鼓励 $q(s_j, a_j; \mathbf{w})$ 接近 \hat{y}_j 。四个神经网络之间的关系如图 10.7 所示。这种方法可以在一定程度上缓解高估，但是实验表明高估仍然很严重。

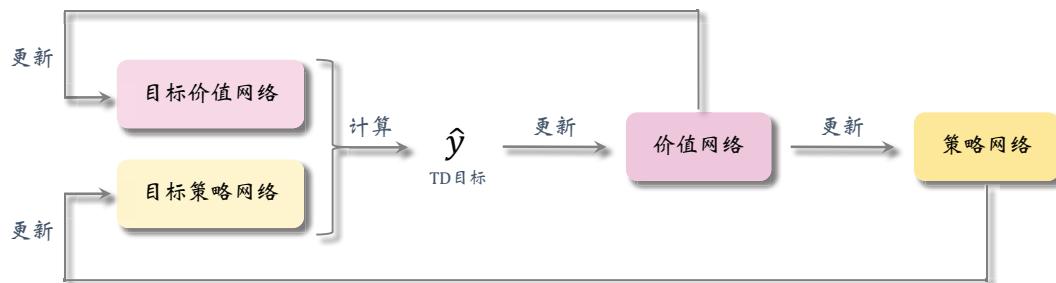


图 10.7：四个神经网络之间的关系。

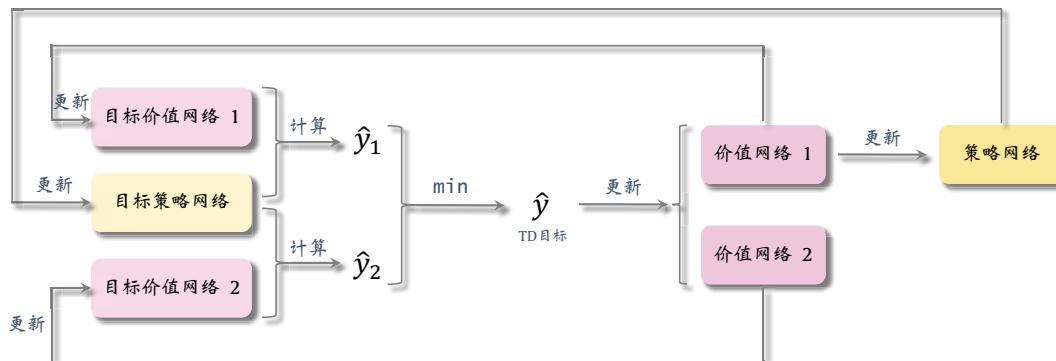


图 10.8：截断双 Q 学习算法中六个神经网络之间的关系。

更好的解决方案——截断双 Q 学习 (Clipped Double Q-Learning)：这种方法使用两个价值网络和一个策略网络：

$$q(s, \mathbf{a}; \mathbf{w}_1), \quad q(s, \mathbf{a}; \mathbf{w}_2), \quad \mu(s; \boldsymbol{\theta}).$$

三个神经网络各对应一个目标网络：

$$q(s, \mathbf{a}; \mathbf{w}_1^-), \quad q(s, \mathbf{a}; \mathbf{w}_2^-), \quad \mu(s; \boldsymbol{\theta}^-).$$

用目标策略网络计算动作：

$$\hat{\mathbf{a}}_{j+1}^- = \mu(s_{j+1}; \boldsymbol{\theta}^-),$$

然后用两个目标价值网络计算：

$$\begin{aligned}\hat{y}_{j,1} &= r_j + \gamma \cdot q(s_{j+1}, \hat{\mathbf{a}}_{j+1}^-; \mathbf{w}_1^-), \\ \hat{y}_{j,2} &= r_j + \gamma \cdot q(s_{j+1}, \hat{\mathbf{a}}_{j+1}^-; \mathbf{w}_2^-).\end{aligned}$$

取两者较小者为 TD 目标：

$$\hat{y}_j = \min \left\{ \hat{y}_{j,1}, \hat{y}_{j,2} \right\}.$$

截断双 Q 学习中的六个神经网络的关系如图 10.8 所示。

10.3.3 其他改进方法

可以在截断双 Q 学习算法的基础上做两处小的改进，进一步提升算法的表现。两种改进分别是往动作中加噪声、减小更新策略网络和目标网络的频率。

往动作中加噪声：上一小节中截断双 Q 学习用目标策略网络计算动作： $\hat{\mathbf{a}}_{j+1}^- = \mu(s_{j+1}; \theta^-)$ 。把这一步改成：

$$\hat{\mathbf{a}}_{j+1}^- = \mu(s_{j+1}; \theta^-) + \xi.$$

公式中的 ξ 是个随机向量，表示噪声，它的每一个元素独立随机从截断正态分布 (Clipped Normal Distribution) 中抽取。把截断正态分布记作 $\mathcal{CN}(0, \sigma^2, -c, c)$ ，意思是均值为零，标准差为 σ 的正态分布，但是变量落在区间 $[-c, c]$ 之外的概率为零。正态分布与截断正态分布的对比如图 10.9 所示。使用截断正态分布，而非正态分布，是为了防止噪声 ξ 过大。使用截断，保证噪声大小不会超过 $-c$ 和 c 。

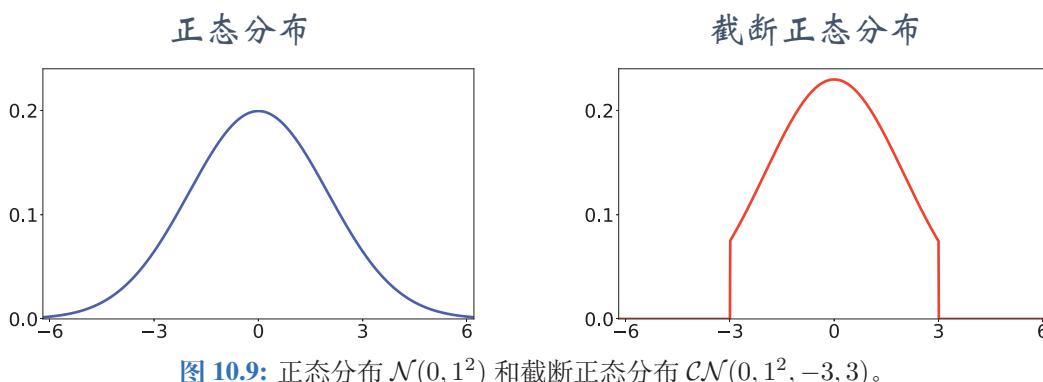


图 10.9：正态分布 $\mathcal{N}(0, 1^2)$ 和截断正态分布 $\mathcal{CN}(0, 1^2, -3, 3)$ 。

减小更新策略网络和目标网络的频率：Actor-Critic 用价值网络来指导策略网络的更新。如果价值网络 q 本身不可靠，那么用价值网络 q 给动作打的分数是不准确的，无助于改进策略网络 μ 。在价值网络 q 还很差的时候就急于更新 μ ，非但不能改进 μ ，反而会由于 μ 的变化导致 q 的训练不稳定。

实验表明，应当让策略网络 μ 以及三个目标网络的更新慢于价值网络 q 。传统的 Actor-Critic 的每一轮训练都对策略网络、价值网络、以及目标网络做一次更新。更好的方法是每一轮更新一次价值网络，但是每隔 k 轮更新一次策略网络和三个目标网络。 k 是超参数，需要调。

10.3.4 训练流程

本节介绍了三种技巧，改进 DPG 的训练。第一，用截断双 Q 学习，缓解价值网络的高估。第二，往目标策略网络中加噪声，起到平滑作用。第三，降低策略网络和三个目标网络的更新频率。使用这三种技巧的算法被称作双延时确定策略梯度 (Twin Delayed Deep Deterministic Policy Gradient)，缩写是 TD3。

TD3 与 DPG 都属于异策略 (Off-policy)，可以用任意的行为策略收集经验，事后做经验回放训练策略网络和价值网络。收集经验的方式与原始的训练算法相同，用 $a_t = \mu(s_t; \theta) + \epsilon$ 与环境交互，把观测到的四元组 (s_t, a_t, r_t, s_{t+1}) 存入经验回放数组。

初始的时候，策略网络和价值网络的参数都是随机的。这样初始化目标网络的参数：

$$\mathbf{w}_1^- \leftarrow \mathbf{w}_1, \quad \mathbf{w}_2^- \leftarrow \mathbf{w}_2, \quad \boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}.$$

训练策略网络和价值网络的时候，每次从数组中随机抽取一个四元组，记作 (s_j, a_j, r_j, s_{j+1}) 。用下标 now 表示神经网络当前的参数，用下标 new 表示更新后的参数。然后执行下面的步骤，更新价值网络、策略网络、目标网络。

1. 让目标策略网络做预测： $\hat{a}_{j+1}^- = \mu(s_{j+1}; \boldsymbol{\theta}_{\text{now}}^-) + \xi$ 。其中向量 ξ 的每个元素都独立从截断正态分布 $\mathcal{CN}(0, \sigma^2, -c, c)$ 中抽取。
2. 让两个目标价值网络做预测：

$$\widehat{q}_{1,j+1} = q(s_{j+1}, \hat{a}_{j+1}^-; \mathbf{w}_{1,\text{now}}^-) \quad \text{和} \quad \widehat{q}_{2,j+1} = q(s_{j+1}, \hat{a}_{j+1}^-; \mathbf{w}_{2,\text{now}}^-).$$

3. 计算 TD 目标：

$$\widehat{y}_j = r_j + \gamma \cdot \min \left\{ \widehat{q}_{1,j+1}, \widehat{q}_{2,j+1} \right\}.$$

4. 让两个价值网络做预测：

$$\widehat{q}_{1,j} = q(s_j, a_j; \mathbf{w}_{1,\text{now}}) \quad \text{和} \quad \widehat{q}_{2,j} = q(s_j, a_j; \mathbf{w}_{2,\text{now}}).$$

5. 计算 TD 误差：

$$\delta_{1,j} = \widehat{q}_{1,j} - \widehat{y}_j \quad \text{和} \quad \delta_{2,j} = \widehat{q}_{2,j} - \widehat{y}_j.$$

6. 更新价值网络：

$$\begin{aligned} \mathbf{w}_{1,\text{new}} &\leftarrow \mathbf{w}_{1,\text{now}} - \alpha \cdot \delta_{1,j} \cdot \nabla_{\mathbf{w}} q(s_j, a_j; \mathbf{w}_{1,\text{now}}), \\ \mathbf{w}_{2,\text{new}} &\leftarrow \mathbf{w}_{2,\text{now}} - \alpha \cdot \delta_{2,j} \cdot \nabla_{\mathbf{w}} q(s_j, a_j; \mathbf{w}_{2,\text{now}}). \end{aligned}$$

7. 每隔 k 轮更新一次策略网络和三个目标网络：

- 让策略网络做预测： $\hat{a}_j = \mu(s_j; \boldsymbol{\theta})$ 。然后更新策略网络：

$$\boldsymbol{\theta}_{\text{new}} \leftarrow \boldsymbol{\theta}_{\text{now}} + \beta \cdot \nabla_{\boldsymbol{\theta}} \mu(s_j; \boldsymbol{\theta}_{\text{now}}) \cdot \nabla_{\mathbf{a}} q(s_j, \hat{a}_j; \mathbf{w}_{1,\text{now}}).$$

- 更新目标网络的参数：

$$\begin{aligned} \boldsymbol{\theta}_{\text{new}}^- &\leftarrow \tau \boldsymbol{\theta}_{\text{new}} + (1 - \tau) \boldsymbol{\theta}_{\text{now}}^-, \\ \mathbf{w}_{1,\text{new}}^- &\leftarrow \tau \mathbf{w}_{1,\text{new}} + (1 - \tau) \mathbf{w}_{1,\text{now}}^-, \\ \mathbf{w}_{2,\text{new}}^- &\leftarrow \tau \mathbf{w}_{2,\text{new}} + (1 - \tau) \mathbf{w}_{2,\text{now}}^-. \end{aligned}$$

10.4 随机高斯策略

上一节用确定策略网络解决连续控制问题。本节用不同的方法做连续控制，本节的策略网络是随机的，它是随机正态分布（也叫高斯分布）。

10.4.1 基本思路

我们先研究最简单的情形：自由度等于 1，也就是说动作 a 是实数，动作空间 $\mathcal{A} \subset \mathbb{R}$ 。把动作的均值记作 $\mu(s)$ ，标准差记作 $\sigma(s)$ ，它们都是状态 s 的函数。用正态分布的概率密度函数作为策略函数：

$$\pi(a | s) = \frac{1}{\sqrt{6.28 \cdot \sigma(s)}} \cdot \exp\left(-\frac{[a - \mu(s)]^2}{2 \cdot \sigma^2(s)}\right). \quad (10.1)$$

假如我们知道函数 $\mu(s)$ 和 $\sigma(s)$ 的解析表达式，可以这样做控制：

1. 观测到当前状态 s ，预测均值 $\hat{\mu} = \mu(s)$ 和标准差 $\hat{\sigma} = \sigma(s)$ 。
2. 从正态分布中做随机抽样： $a \sim \mathcal{N}(\hat{\mu}, \hat{\sigma}^2)$ ；智能体执行动作 a 。

然而我们并不知道 $\mu(s)$ 和 $\sigma(s)$ 是怎么样的函数。一个很自然的想法是用神经网络来近似这两个函数。把神经网络记作 $\mu(s; \theta)$ 和 $\sigma(s; \theta)$ ，其中 θ 表示神经网络中的可训练参数。但实践中最好不要直接近似标准差 σ ，而是近似方差对数 $\ln \sigma^2$ ²。定义两个神经网络：

$$\mu(s; \theta) \text{ 和 } \rho(s; \theta),$$

分别用于预测均值和方差对数。可以按照图10.11来搭建神经网络。神经网络的输入是状态 s ，通常是向量、矩阵、或者张量。神经网络有两个输出头，分别记作 $\mu(s; \theta)$ 和 $\rho(s; \theta)$ 。可以这样用神经网络做控制：

1. 观测到当前状态 s ，计算均值 $\hat{\mu} = \mu(s; \theta)$ ，方差对数 $\hat{\rho} = \rho(s; \theta)$ ，以及方差 $\hat{\sigma}^2 = \exp(\hat{\rho})$ 。
2. 从正态分布中做随机抽样： $a \sim \mathcal{N}(\hat{\mu}, \hat{\sigma}^2)$ ；智能体执行动作 a 。

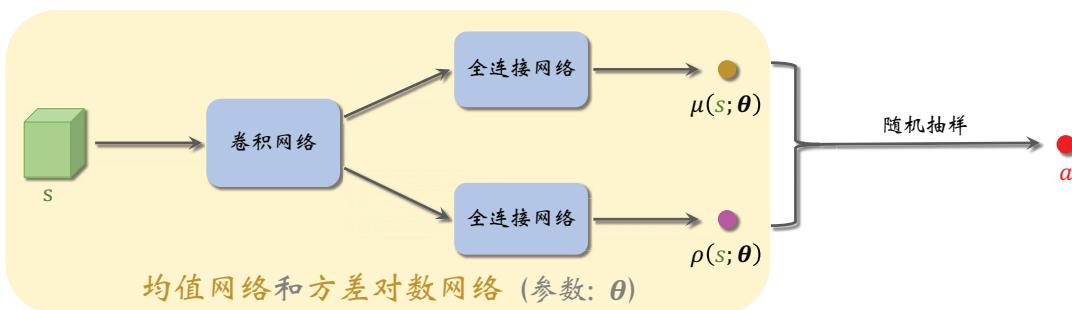


图 10.10：高斯策略网络有两个头，一个输出均值 $\hat{\mu}$ ，另一个输出方差对数 $\hat{\rho}$ 。

用神经网络近似均值和标准差之后，公式 (10.1) 中的策略函数 $\pi(a|s)$ 变成了下面的

²标准差 σ 必须非负，如果把 σ 作为优化变量，那么优化模型有约束条件，给求解造成困难。方差对数 ρ 的取值范围是所有实数，因此不需要约束条件。

策略网络：

$$\pi(a|s; \theta) = \frac{1}{\sqrt{6.28 \cdot \exp[\rho(s; \theta)]}} \cdot \exp\left(-\frac{[a - \mu(s; \theta)]^2}{2 \cdot \exp[\rho(s; \theta)]}\right).$$

实际做控制的时候，我们只需要神经网络 $\mu(s; \theta)$ 和 $\rho(s; \theta)$ ，用不到真正的策略网络 $\pi(a|s; \theta)$ 。

10.4.2 随机高斯策略网络

上一小节假设控制问题的自由度是 $d = 1$ ，也就是说动作 a 是标量。实际问题中的自由度 d 往往大于 1，那么动作 a 是 d 维向量。对于这样的问题，我们修改一下神经网络结构，让两个输出 $\mu(s; \theta)$ 和 $\rho(s; \theta)$ 都 d 维向量；见图10.11。

用标量 a_i 表示动作向量 a 的第 i 个元素。用函数 $\mu_i(s; \theta)$ 和 $\rho_i(s; \theta)$ 分别表示 $\mu(s; \theta)$ 和 $\rho(s; \theta)$ 的第 i 个元素。我们用下面这个特殊的多元正态分布的概率密度函数作为策略网络：

$$\pi(a|s; \theta) = \prod_{i=1}^d \frac{1}{\sqrt{6.28 \cdot \exp[\rho_i(s; \theta)]}} \cdot \exp\left(-\frac{[a_i - \mu_i(s; \theta)]^2}{2 \cdot \exp[\rho_i(s; \theta)]}\right).$$

做控制的时候只需要均值网络 $\mu(s; \theta)$ 和方差对数网络 $\rho(s; \theta)$ ，不需要策略网络 $\pi(a|s; \theta)$ 。做训练的时候也不需要 $\pi(a|s; \theta)$ ，而是要用辅助网络 $f(s, a; \theta)$ 。总而言之，策略网络 π 只是帮助你理解本节的方法而已，实际算法中不会出现 π 。

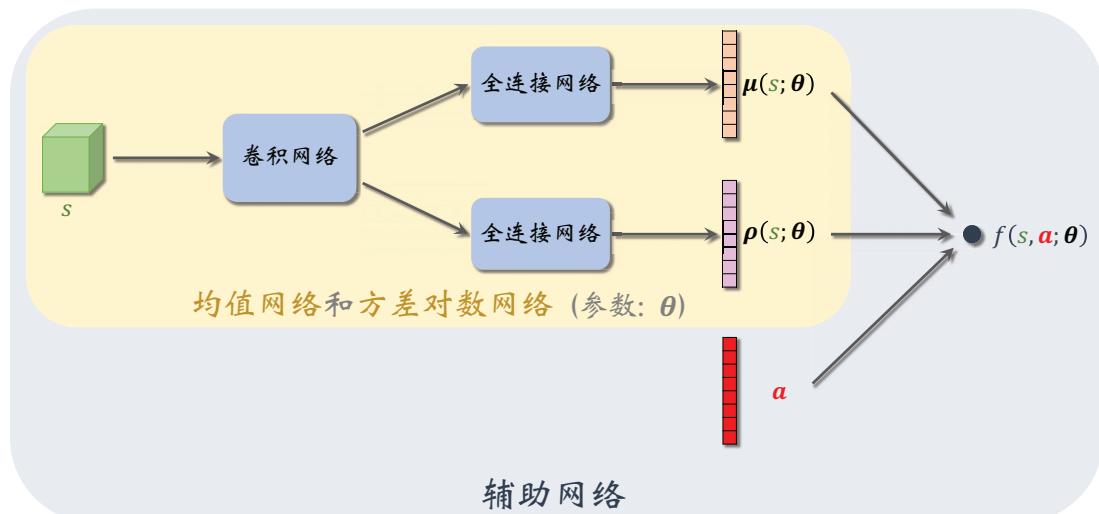


图 10.11：辅助网络的结构示意图。辅助神经网络的输入是状态 s 与动作 a ，输出是实数 $f(s, a; \theta)$ 。

图10.11描述了辅助网络 $f(s, a; \theta)$ 与 μ 、 ρ 、 a 的关系。辅助网络具体是这样定义的：

$$f(s, a; \theta) = -\frac{1}{2} \sum_{i=1}^d \left(\rho_i(s; \theta) + \frac{[a_i - \mu_i(s; \theta)]^2}{\exp[\rho_i(s; \theta)]} \right).$$

它的可训练参数 θ 都是从 $\mu(s; \theta)$ 和 $\rho(s; \theta)$ 中来的。不难发现，辅助网络与策略网络有

这样的关系：

$$f(s, \mathbf{a}; \boldsymbol{\theta}) = \ln \pi(\mathbf{a}|s; \boldsymbol{\theta}) + \text{Constant.} \quad (10.2)$$

10.4.3 策略梯度

回忆一下之前学过的内容。在 t 时刻的折扣回报记作随机变量

$$U_t = R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot R_{t+2} + \cdots + \gamma^{n-t} \cdot R_n.$$

动作价值函数 $Q_\pi(s_t, \mathbf{a}_t)$ 是对折扣回报 U_t 的条件期望。前面章节推导过策略梯度的蒙特卡洛近似：

$$\mathbf{g} = Q_\pi(s, \mathbf{a}) \cdot \nabla_{\boldsymbol{\theta}} \ln \pi(\mathbf{a}|s; \boldsymbol{\theta}).$$

由公式 (10.2) 可得：

$$\mathbf{g} = Q_\pi(s, \mathbf{a}) \cdot \nabla_{\boldsymbol{\theta}} f(s, \mathbf{a}; \boldsymbol{\theta}). \quad (10.3)$$

有了策略梯度，就可以学习参数 $\boldsymbol{\theta}$ 。训练的过程大致如下：

1. 搭建均值网络 $\mu(s; \boldsymbol{\theta})$ 、方差对数网络 $\rho(s; \boldsymbol{\theta})$ 、辅助网络 $f(s, \mathbf{a}; \boldsymbol{\theta})$ 。
2. 让智能体与环境交互，记录每一步的状态、动作、奖励，并对参数 $\boldsymbol{\theta}$ 做更新：
 - (a). 观测到当前状态 s ，计算均值、方差对数、方差：

$$\hat{\mu} = \mu(s; \boldsymbol{\theta}), \quad \hat{\rho} = \rho(s; \boldsymbol{\theta}), \quad \hat{\sigma}^2 = \exp(\hat{\rho}).$$

此处的指数函数 $\exp(\cdot)$ 应用到向量的每一个元素上。

- (b). 设 $\hat{\mu}_i$ 和 $\hat{\sigma}_i$ 分别是 d 维向量 $\hat{\mu}$ 和 $\hat{\sigma}$ 的第 i 个元素。从正态分布中做抽样：

$$a_i \sim \mathcal{N}(\hat{\mu}_i, \hat{\sigma}_i^2), \quad \forall i = 1, \dots, d.$$

把得到的动作记作 $\mathbf{a} = [a_1, \dots, a_d]$ 。

- (c). 近似计算动作价值： $\hat{q} \approx Q_\pi(s, \mathbf{a})$ 。
- (d). 用反向传播计算出辅助网络关于参数 $\boldsymbol{\theta}$ 的梯度： $\nabla_{\boldsymbol{\theta}} f(s, \mathbf{a}; \boldsymbol{\theta})$ 。
- (e). 用策略梯度上升更新参数：

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \beta \cdot \hat{q} \cdot \nabla_{\boldsymbol{\theta}} f(s, \mathbf{a}; \boldsymbol{\theta}).$$

此处的 β 是学习率。

但是算法中有一个没解决的问题：我们并不知道动作价值 $Q_\pi(s, \mathbf{a})$ 。有两种办法近似 $Q_\pi(s, \mathbf{a})$ ：REINFORCE 用实际观测的折扣回报代替 $Q_\pi(s, \mathbf{a})$ ，Actor-Critic 用价值网络近似 Q_π 。后面两小节具体讲解这两种算法。

10.4.4 用 REINFORCE 学习参数 $\boldsymbol{\theta}$

REINFORCE 用实际观测的折扣回报 $u_t = \sum_{k=t}^n \gamma^{k-t} \cdot r_k$ 代替动作价值 $Q_\pi(s_t, \mathbf{a}_t)$ 。道理是这样的。动作价值是回报的期望：

$$Q_\pi(s_t, \mathbf{a}_t) = \mathbb{E}[U_t | S_t = s_t, A_t = \mathbf{a}_t].$$

随机变量 U_t 的一个实际观测值 u_t 是期望的蒙特卡洛近似。这样一来，公式 (10.3) 中的

策略梯度就能近似成

$$\mathbf{g} \approx u_t \cdot \nabla_{\theta} f(s, a; \theta).$$

在搭建好均值网络 $\mu(s; \theta)$ 、方差对数网络 $\rho(s; \theta)$ 、辅助网络 $f(s, a; \theta)$ 之后，我们用 REINFORCE 更新参数 θ 。设当前参数为 θ_{now} 。REINFORCE 重复以下步骤，直到收敛：

1. 用 $\mu(s; \theta_{\text{now}})$ 和 $\rho(s; \theta_{\text{now}})$ 控制智能体与环境交互，完成一局游戏，得到一条轨迹：

$$s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_n, a_n, r_n.$$

2. 计算所有的回报：

$$u_t = \sum_{k=t}^T \gamma^{k-t} \cdot r_k, \quad \forall t = 1, \dots, n.$$

3. 对辅助网络做反向传播，得到所有的梯度：

$$\nabla_{\theta} f(s_t, a_t; \theta_{\text{now}}), \quad \forall t = 1, \dots, n.$$

4. 用策略梯度上升更新参数：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \sum_{t=1}^n \gamma^{t-1} \cdot u_t \cdot \nabla_{\theta} f(s_t, a_t; \theta_{\text{now}})$$

上述算法标准的 REINFORCE，效果不如使用基线的 REINFORCE。读者可以参考第 8.2 节的内容，把状态价值作为基线，改进上面描述的算法。REINFORCE 算法属于同策略(On-policy)，不能使用经验回放。

10.4.5 用 Actor-Critic 学习参数 θ

Actor-Critic 需要搭建一个价值网络 $q(s, a; w)$ ，用于近似动作价值函数 $Q_{\pi}(s, a)$ 。价值网络的结构如图 10.12 所示。此外，还需要一个目标价值网络 $q(s, a; w^-)$ ，网络结构相同，但是参数不同。

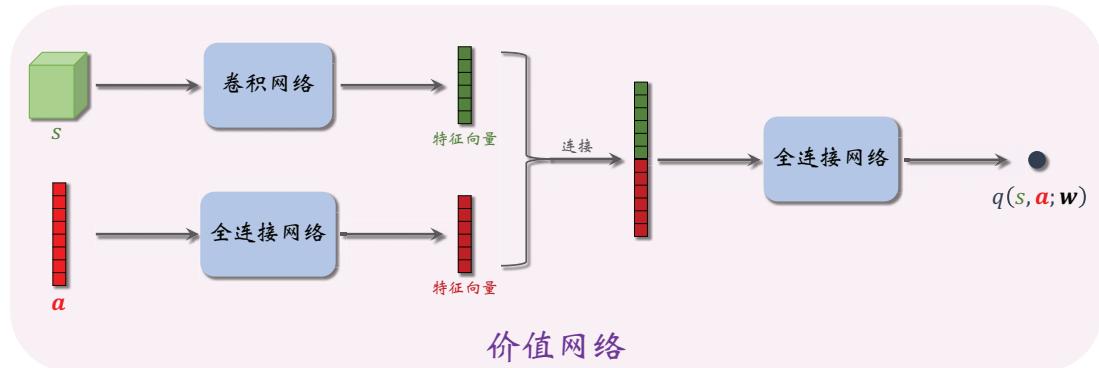


图 10.12：价值网络 $q(s, a; w)$ 的结构。输入是状态 s 和动作 a ，输出是实数。

在搭建好均值网络 μ 、方差对数网络 ρ 、辅助网络 f 、价值网络 q 之后，我们用 SARSA 算法更新价值网络参数 w ，用近似策略梯度更新控制器参数 θ 。设当前参数为 w_{now} 和 θ_{now} 。重复以下步骤更新价值网络参数、控制器参数，直到收敛：

1. 实际观测到当前状态 s_t ，用控制器算出均值 $\mu(s_t; \theta_{\text{now}})$ 和方差对数 $\rho(s_t; \theta_{\text{now}})$ ，然

10.4 随机高斯策略

后随机抽样得到动作 \mathbf{a}_t 。智能体执行动作 \mathbf{a}_t ，观测到奖励 r_t 与新的状态 s_{t+1} 。

2. 计算均值 $\mu(s_{t+1}; \theta_{\text{now}})$ 和方差对数 $\rho(s_{t+1}; \theta_{\text{now}})$ ，然后随机抽样得到动作 $\tilde{\mathbf{a}}_{t+1}$ 。这个动作只是假想动作，智能体不予执行。
3. 用价值网络计算出：

$$\hat{q}_t = q(s_t, \mathbf{a}_t; \mathbf{w}_{\text{now}}).$$

4. 用目标网络计算出：

$$\hat{q}_{t+1} = q(s_{t+1}, \tilde{\mathbf{a}}_{t+1}; \mathbf{w}_{\text{now}}^-).$$

5. 计算 TD 目标和 TD 误差：

$$\hat{y}_t = r_t + \gamma \cdot \hat{q}_{t+1}, \quad \delta_t = \hat{q}_t - \hat{y}_t.$$

6. 更新价值网络的参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} q(s_t, \mathbf{a}_t; \mathbf{w}_{\text{now}}).$$

7. 更新策略网络参数：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \hat{q}_t \cdot \nabla_{\theta} f(s_t, \mathbf{a}_t; \theta_{\text{now}})$$

8. 更新目标网络参数：

$$\mathbf{w}_{\text{new}}^- \leftarrow \tau \cdot \mathbf{w}_{\text{new}} + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^-.$$

算法中的 α 、 β 、 τ 都是超参数，需要手动调整。上述算法标准的 Actor-Critic，效果不如 Advantage Actor-Critic (A2C)。读者可以参考第 8.3 节的内容，用 A2C 改进上面描述的算法。

∽第十章 相关文献∽

确定策略梯度 (Deterministic Policy Gradient, DPG) 方法由 David Silver 等人在 2014 年提出 [91]。随后同一批作者把相似的想法与深度学习结合起来，提出深度确定策略梯度 (Deep Deterministic Policy Gradient, 缩写 DDPG)，文章在 2016 年发表 [61]。这两篇论文使得 DPG 方法流行起来。但值得注意的是，相似的想法在更早的论文中有提出：[44, 78]。

2018 年的论文 [40] 提出三种对 DPG 的改进方法，并将改进的算法命名为 TD3。2017 年的论文 [43] 提出了 Soft Actor-Critic (SAC)，也可以解决连续控制问题。

Degris 等人在 2012 年发表的论文 [34] 使用正态分布的概率密度函数作为策略函数，并且用线性函数近似均值和方差对数。类似的连续控制方法最早由 Williams 在 1987 和 1992 年提出 [117-118]。

第十一章 对状态的不完全观测

11.1 不完全观测问题

之前章节中的 DQN $Q(s, a; \mathbf{w})$, 策略网络 $\pi(a|s; \boldsymbol{\theta})$ 、 $\mu(s; \boldsymbol{\theta})$, 价值网络 $q(s, a; \mathbf{w})$ 、 $v(s; \mathbf{w})$ 都需要把当前状态 s 作为输入。之前我们一直假设可以完全观测到状态 s ; 在围棋、象棋、五子棋等简单的游戏中, 棋盘上当前的格局就是完整的状态, 符合完全观测的假设。但是在很多实际应用中, 完全观测假设往往不符合实际。比如在星际争霸、英雄联盟等电子游戏中, 屏幕上当前的画面并不能完整反映出游戏的状态, 因为观测只是地图的一小部分; 甚至最近的 100 帧也无法反映出游戏真实的状态。

把 t 时刻的状态记作 s_t , 把观测记作 o_t 。观测 o_t 可以是当前游戏屏幕上的画面, 也可以是最近 100 帧画面。我们无法用 $\pi(a_t|s_t; \boldsymbol{\theta})$ 做决策, 因为我们不知道 s_t 。最简单的解决办法就是用当前观测 o_t 代替状态 s_t , 用 $\pi(a_t|o_t; \boldsymbol{\theta})$ 做决策。同理, 对于 DQN 和价值网络, 也用 o_t 代替 s_t 。虽然这种简单的方法可行, 但是效果恐怕不好。



图 11.1: 在迷宫问题中, 智能体可能知道迷宫的整体格局, 也可能只知道自己附近的格局。

图 11.1 的例子是让智能体走迷宫。图 11.1(a) 中智能体可以完整观测到迷宫 s ; 这种问题最容易解决。图 11.1(b) 中智能体只能观测到自身附近一小块区域 o_t , 这属于不完全观测问题, 这种问题较难解决。如果仅仅靠当前观测 o_t 做决策, 智能体做出的决策是非常盲目的, 很难走出迷宫。一种更合理的办法是让智能体记住过去的观测, 这样就能对状态的观测越来越完整, 做出越来越理性的决策; 如图 11.1(c) 所示。

对于不完全观测的强化学习问题, 应当记忆过去的观测, 用所有已知的信息做决策。这正是人类解决不完全观测问题的方式。对于星际争霸、扑克牌、麻将等不完全观测的游戏, 人类玩家也需要记忆; 人类玩家的决策不止依赖于当前时刻的观测 o_t , 而是依赖于过去所有的观测 o_1, \dots, o_t 。把从初始到 t 时刻为止的所有观测记作:

$$\mathbf{o}_{1:t} = [o_1, o_2, \dots, o_t]$$

可以用 $\mathbf{o}_{1:t}$ 代替状态 s , 作为策略网络的输入, 那么策略网络就记作:

$$\pi(a_t | \mathbf{o}_{1:t}; \boldsymbol{\theta}).$$

该如何实现这样一个策略网络呢？请注意， $\mathbf{o}_{1:t}$ 的大小是变化的。如果 $\mathbf{o}_1, \dots, \mathbf{o}_t$ 都是 $d \times 1$ 的向量，那么 $\mathbf{o}_{1:t}$ 是 $d \times t$ 的矩阵或 $dt \times 1$ 的向量，它的大小随 t 增长。卷积层和全连接层都要求输入大小固定，因此不能简单地用卷积层和全连接层实现策略网络。一种可行的办法是将卷积层、全连接层与循环层结合，这样就能处理不固定长度的输入。

11.2 循环神经网络 (RNN)

循环神经网络 (Recurrent Neural Network), 缩写 RNN, 是一类神经网络的总称, 由循环层 (Recurrent Layers) 和其他种类的层组成。循环层的作用是把一个序列 (比如时间序列、文本、语音) 映射到一个特征向量。设向量 x_1, \dots, x_n 是一个序列。对于所有的 $t = 1, \dots, n$, 循环层把 (x_1, \dots, x_t) 映射到特征向量 h_t 。依次把 x_1, \dots, x_n 输入循环层, 会得到:

$$\begin{aligned} (x_1) &\Rightarrow h_1, \\ (x_1, x_2) &\Rightarrow h_2, \\ (x_1, x_2, x_3) &\Rightarrow h_3, \\ &\vdots \\ (x_1, x_2, x_3, \dots, x_{n-1}) &\Rightarrow h_{n-1}, \\ (x_1, x_2, x_3, \dots, x_{n-1}, x_n) &\Rightarrow h_n. \end{aligned}$$

RNN 的好处在于不论输入序列的长度 t 是多少, 从序列中提取出的特征向量 h_t 的大小是固定的。请特别注意, h_t 并非只依赖于 x_t 这一个向量, 而是依赖于 $[x_1, \dots, x_t]$; 理想情况下, h_t 记住了 $[x_1, \dots, x_t]$ 中的主要信息。比如 h_3 是对 $[x_1, x_2, x_3]$ 的概要, 而非是对 x_3 这一个向量的概要。

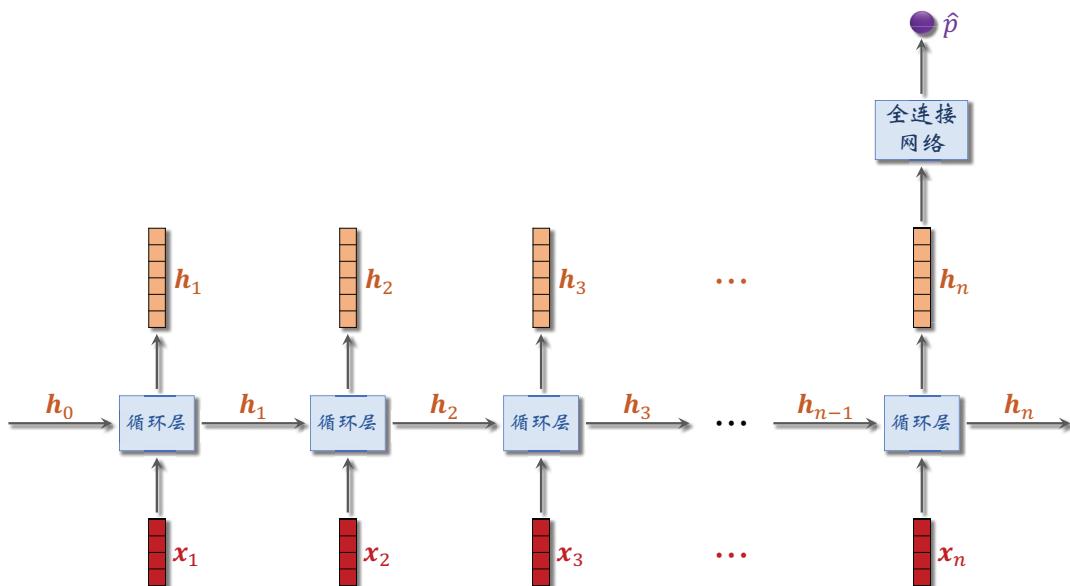


图 11.2: 输入是序列 x_1, \dots, x_t 。向量 h_t 是从所有 t 个输入中提取的特征, 可以把它看做输入序列的一个概要。把 h_t 输入全连接层 (带 Sigmoid 激活函数), 得到分类结果 \hat{p} 。

举个例子, 用户给商品写的评论由 n 个字组成 (不同的评论有不同的 n) , 我们想要判断评论是正面的还是负面的, 这是个二元分类问题。用词嵌入 (Word Embedding) 把每个字映射到一个向量, 得到 x_1, \dots, x_n , 把它们依次输入循环层。循环层依次输出 h_1, \dots, h_n 。我们只需要用 h_n , 因为它是从全部输入 x_1, \dots, x_n 中提取的特征; 可以忽略掉 h_1, \dots, h_{n-1} 。最后, 二元分类器把 h_n 作为输入, 输出一个介于 0 到 1 之间的数 \hat{p} ,

0 代表负面，1 代表正面。图 11.2 描述了神经网络的结构。

循环层的种类有很多，常见的包括简单循环层、LSTM、GRU。本书只介绍简单循环层。LSTM、GRU 是对简单循环层的改进，结构更复杂，效果更好；但是它们的原理与简单循环层基本相同。读者只需要理解简单循环层就足够了。用 TensorFlow、PyTorch、Keras 编程实现的话，几种循环层的使用方法完全相同（唯一区别是函数名）。

简单循环层的输入记作 $x_1, \dots, x_n \in \mathbb{R}^{d_{\text{in}}}$ ，输出记作 $h_1, \dots, h_n \in \mathbb{R}^{d_{\text{out}}}$ 。循环层的参数是矩阵 $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ 和向量 $b \in \mathbb{R}^{d_{\text{in}}}$ 。循环层的输出是这样计算出来的：从 $t = 1, \dots, n$ ，依次计算

$$h_t = \tanh(W[h_{t-1}; x_t] + b).$$

图 11.3 解释上面的公式。注意，不论输入序列长度 n 是多少，简单循环层的参数只有唯一的 W 和 b 。公式中的 \tanh 是双曲正切函数，见图 11.4。 \tanh 是标量函数；如果输入是向量，那么 \tanh 应用到向量的每一个元素上。对于 $d \times 1$ 的向量 z ，有

$$\tanh(z) = [\tanh(z_1), \tanh(z_2), \dots, \tanh(z_d)]^T.$$

$$h_t = \tanh \left[\begin{array}{c} W \\ \cdot \\ x_t \\ + \\ b \end{array} \right]$$

图 11.3: 简单循环层。

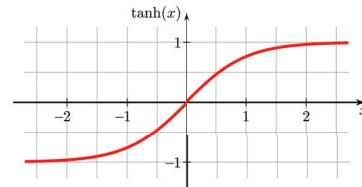


图 11.4: 双曲正切函数。

11.3 RNN 作为策略网络

在不完全观测的设定下，我们希望策略网络能利用所有已经收集的观测 $\mathbf{o}_{1:t} = [o_1, \dots, o_t]$ 做决策。定义策略网络为

$$\mathbf{f}_t = \pi(a_t | \mathbf{o}_{1:t}; \boldsymbol{\theta})$$

结构如图 11.5 所示。在第 t 时刻，观测到 o_t ，用卷积网络提取特征，得到向量 \mathbf{x}_t 。循环层把 \mathbf{x}_t 作为输入，然后输出 \mathbf{h}_t 。 \mathbf{h}_t 是从 $\mathbf{x}_1, \dots, \mathbf{x}_t$ 中提取出的特征，是对所有观测 $\mathbf{o}_{1:t} = [o_1, \dots, o_t]$ 的一个概要。全连接网络（输出层激活函数是 Softmax）把 \mathbf{h}_t 作为输入，然后输出向量 \mathbf{f}_t ，作为 t 时刻决策的依据。 \mathbf{f}_t 的维度是动作空间的大小 $|\mathcal{A}|$ ，它的每个元素对应一个动作，表示选择该动作的概率。

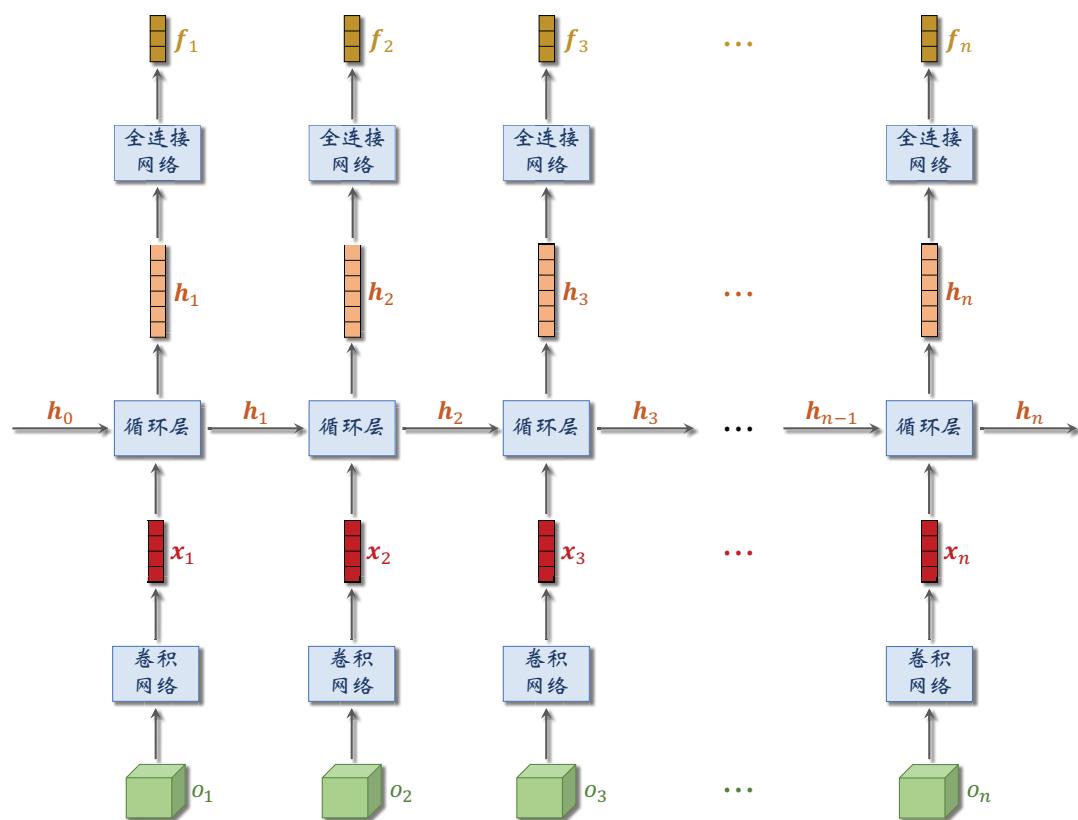


图 11.5：基于 RNN 的策略网络。图中所有的全连接网络都有相同的参数；所有的循环层都有相同的参数；所有的卷积层都有相同的参数。

对于不完全观测问题，我们可以类似地搭建 DQN 和价值网络。DQN 可以定义为：

$$Q(\mathbf{o}_{1:t}, a_t; \mathbf{w}).$$

价值网络可以定义为：

$$q(\mathbf{o}_{1:t}, a_t; \mathbf{w}) \quad \text{或} \quad v(\mathbf{o}_{1:t}; \mathbf{w}).$$

这些神经网络与图 11.5 中策略网络的区别仅在于全连接网络的结构而已；它们使用的卷积网络、循环层与图 11.5 相同。

∽ 第十一章 相关文献 ∽

RNN 是一类很重要的神经网络。学术界认为最早的 RNN 是 Hopfield network [50]，尽管它跟我们今天用的 RNN 很不一样。现在最常用的 RNN 包括 LSTM [49] 和 GRU [28]。注意力机制 (Attention) 由 2015 年的论文 [6] 提出，将注意力机制与 RNN 结合，可以大幅提升 RNN 在机器翻译任务上的表现。注意力机制显然可以用于本章介绍的 RNN 策略网络，但是这样会大幅增加计算量。

2015 年的论文 [45] 首先将 RNN 应用于深度强化学习，把 RNN 与 DQN 相结合，把得到的方法叫做 DRQN。在此之后，RNN 成为解决不完全观测问题的一种标准技巧，比如论文 [67, 37, 79]。

第十二章 并行计算

机器学习的实践中普遍使用并行计算，利用大量的计算资源（比如很多块 GPU）缩短训练所需的时间，用几个小时就能完成原本需要很多天才能完成的训练。深度强化学习自然也不例外；可以用很多处理器同时收集经验、计算梯度，让原本需要很长时间的训练在较短的时间内完成。第 12.1 以并行梯度下降为例讲解并行计算基础知识。第 12.2 介绍异步并行梯度下降算法。第 12.3 介绍两种异步强化学习算法。

12.1 并行计算基础

本节以并行梯度下降 (Parallel Gradient Descent) 为例讲解并行计算的基础知识，用 MapReduce 架构实现并行梯度下降，并且分析并行计算中的时间开销。

12.1.1 并行梯度下降

本节用最小二乘回归 (Least Squares Regression) 为例讲解并行梯度下降的基本原理。把训练数据记作 $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$ 。最小二乘回归定义为：

$$\min_{\mathbf{w}} \left\{ L(\mathbf{w}) \triangleq \frac{1}{2n} \sum_{j=1}^n (\mathbf{x}_j^T \mathbf{w} - y_j)^2 \right\}.$$

这个优化问题的目标是寻找向量 $\mathbf{w}^* \in \mathbb{R}^d$ ，使得对于所有的 j ， $\mathbf{x}_j^T \mathbf{w}^*$ 都很接近 y_j 。我们可以用梯度下降算法求解这个优化问题。梯度下降重复这个步骤，直到收敛：

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}} L(\mathbf{w}).$$

公式中的 η 是学习率。如果 η 的取值比较合理，那么梯度下降可以保证 \mathbf{w} 收敛到最优解 \mathbf{w}^* 。目标函数 $L(\mathbf{w})$ 的梯度可以写作：

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \frac{1}{n} \sum_{j=1}^n \mathbf{g}(\mathbf{x}_j, y_j; \mathbf{w}), \quad \text{其中 } \mathbf{g}(\mathbf{x}_j, y_j; \mathbf{w}) \triangleq (\mathbf{x}_j^T \mathbf{w} - y_j) \mathbf{x}_j \in \mathbb{R}^d.$$

由于 \mathbf{x}_j 和 \mathbf{w} 都是 d 维向量，因此计算一个 $\mathbf{g}(\mathbf{x}_j, y_j; \mathbf{w})$ 的时间复杂度是 $\mathcal{O}(d)$ 。计算梯度 $\nabla_{\mathbf{w}} L(\mathbf{w})$ 需要计算 \mathbf{g} 函数 n 次，所以计算 $\nabla_{\mathbf{w}} L(\mathbf{w})$ 的时间复杂度是 $\mathcal{O}(nd)$ 。如果用 m 块处理器做并行计算，那么理想情况下每块处理器的计算量是 $\mathcal{O}(\frac{nd}{m})$ 。

下面举一个简单的例子讲解并行梯度下降。假设我们有两块处理器。把梯度 $\nabla_{\mathbf{w}} L(\mathbf{w})$ 展开，得到：

$$\begin{aligned} & \nabla_{\mathbf{w}} L(\mathbf{w}) \\ &= \frac{1}{n} \left[\underbrace{\mathbf{g}(\mathbf{x}_1, y_1; \mathbf{w}) + \dots + \mathbf{g}(\mathbf{x}_{\frac{n}{2}}, y_{\frac{n}{2}}; \mathbf{w})}_{\text{用一号处理器计算, 把结果记作 } \tilde{\mathbf{g}}^1} + \underbrace{\mathbf{g}(\mathbf{x}_{\frac{n}{2}+1}, y_{\frac{n}{2}+1}; \mathbf{w}) + \dots + \mathbf{g}(\mathbf{x}_n, y_n; \mathbf{w})}_{\text{用二号处理器计算, 把结果记作 } \tilde{\mathbf{g}}^2} \right]. \end{aligned}$$

两块处理器各承担一半的计算量，分别输出 d 维向量 $\tilde{\mathbf{g}}^1$ 和 $\tilde{\mathbf{g}}^2$ 。将两块处理器的结果汇总，得到梯度：

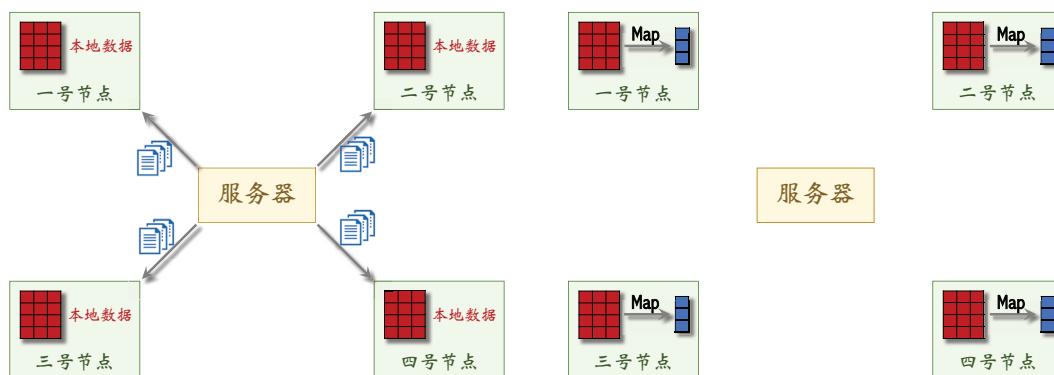
$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \frac{1}{n} (\tilde{\mathbf{g}}^1 + \tilde{\mathbf{g}}^2).$$

并行梯度下降中的“计算”非常简单；而并行计算的复杂之处在于通信。在一轮梯度下降开始之前，需要把最新的模型参数 w 发送给两块处理器，否则处理器无法计算梯度。在两块处理器完成计算之后，需要做通信，把结果 \tilde{g}^1 和 \tilde{g}^2 汇总到一块处理器上。下一小节以 MapReduce 架构为例，讲解并行梯度下降的实现。

12.1.2 MapReduce

并行计算需要在计算机集群上完成。一个集群有很多处理器和内存条，它们被划分到多个节点 (Compute Node) 上。一个节点上可以有多个处理器，处理器可以共享内存。节点之间不能共享内存，即一个节点不能访问另一个节点的内存。如果两个节点相连接，它们可以通过计算机网络通信（比如 TCP/IP 协议）。

为了协调节点的计算和通信，需要有相应的软件系统。MapReduce 是由 Google 开发的一种软件系统，用于大规模的数据分析和机器学习。MapReduce 原本是软件系统的名字，但是后来人们把类似的系统架构都称作 MapReduce。除了 Google 自己的 MapReduce，比较有名的系统还有 Hadoop¹ 和 Spark²。MapReduce 属于 Server-Client 架构，有一个节点作为中央服务器，其余节点作为 Worker，受服务器控制。服务器用于协调整个系统，而计算主要由 Worker 节点并行完成。



服务器可以与 Worker 节点做通信传输数据（但是 Worker 节点之间不能相互通信）。一种通信方式是广播 (Broadcast)，即服务器将同一条信息同时发送给所有 Worker 节点；如图 12.1 所示。比如做并行梯度下降的时候，服务器需要把更新过的参数 $w \in \mathbb{R}^d$ 广播到所有 Worker 节点。MapReduce 架构不允许服务器将一条信息只发送给一号节点，而将一条不同的信息只发送给二号节点。服务器只能把相同信息广播到所有节点。

每个节点都可以做计算。映射 (Map) 操作让所有 Worker 节点同时并行做计算；如图 12.2 所示。如果我们要编程实现一个算法，需要自己定义一个函数，它可以让每个 Worker 节点把它的本地数据映射到一些输出值。比如做并行梯度下降的时候，定义函数 g 把三

¹<https://hadoop.apache.org/>

²<https://spark.apache.org/>

元组 $(\mathbf{x}_j, y_j, \mathbf{w})$ 映射到向量

$$z_j = (\mathbf{x}_j^T \mathbf{w} - y_j) \mathbf{x}_j.$$

映射操作要求所有节点都要同时执行同一个函数，比如 $g(\mathbf{x}_j, y_j, \mathbf{w})$ 。节点不能各自执行不同的函数。

Worker 节点可以向服务器发送信息，最常用的通信操作是 **规约 (Reduce)**。这种操作可以把 Worker 节点上的数据做归并，并且传输到服务器上。如图 12.3 所示，系统对 Worker 节点输出的蓝色向量做规约。如果执行 `sum` 规约函数，那么结果是四个蓝色向量的加和。如果执行 `mean` 规约函数，那么结果是四个蓝色向量的均值。如果执行 `count` 规约函数，那么结果是整数 4，即蓝色向量的数量。

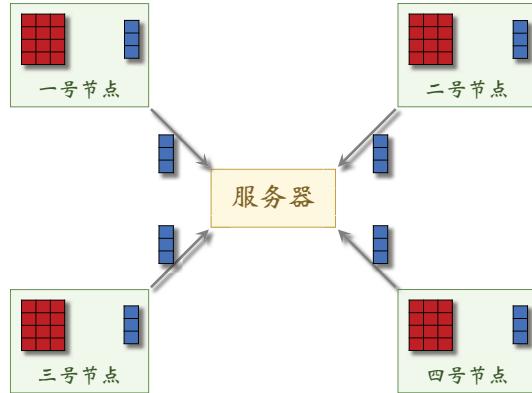


图 12.3: MapReduce 中的规约 (Reduce) 操作。

12.1.3 用 MapReduce 实现并行梯度下降

数据并发 (Data Parallelism): 为了使用 MapReduce 实现并行梯度下降，我们需要把数据集 $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ 划分到 m 个 Worker 节点上，每个节点上存一部分数据；见图 12.4。这种划分方式叫做数据并发。与数据并发相对的是模型并发 (Model Parallelism)，即将模型参数 \mathbf{w} 划分到 m 个 Worker 节点上；每个节点有全部数据，但是只有一部分模型参数。本书只介绍数据并发，不讨论模型并发。

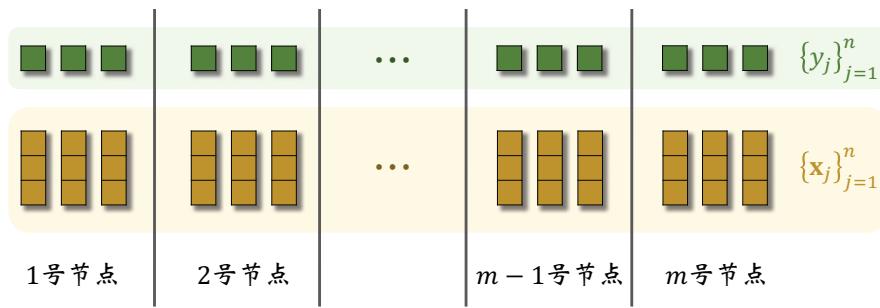


图 12.4: 将数据集 $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ 划分到 m 个 Worker 节点上。

并行梯度下降的流程: 用数据并发，设集合 $\mathcal{I}_1, \dots, \mathcal{I}_m$ 是集合 $\{1, 2, \dots, n\}$ 的划分；集合 \mathcal{I}_k 包含第 k 个 Worker 节点上所有样本的序号。并行梯度下降需要重复——广播、映射、规约、更新参数——这四个步骤，直到算法收敛；见示意图 12.5。

1. 广播 (Broadcast): 服务器将当前的模型参数 \mathbf{w}_{now} 广播到 m 个 Worker 节点。这样一来，所有节点都知道 \mathbf{w}_{now} 。
2. 映射 (Map): 这一步让 m 个 Worker 节点做并行计算，用本地数据计算梯度。需要

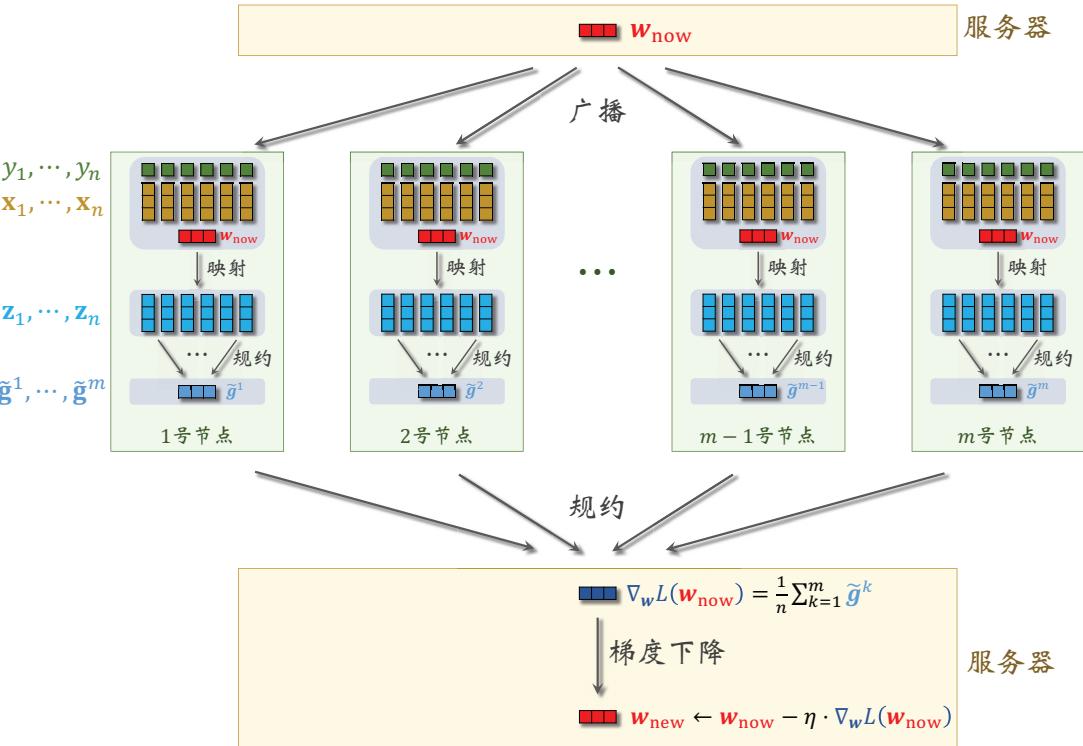


图 12.5: 并行梯度下降的流程。

在编程的时候定义这样一个映射函数:

$$g(x, y, \mathbf{w}) = (\mathbf{x}^T \mathbf{w} - y) \mathbf{x}.$$

第 k 号 Worker 节点做如下映射:

$$g : (x_j, y_j, \mathbf{w}_{\text{now}}) \mapsto z_j = (\mathbf{x}_j^T \mathbf{w}_{\text{now}} - y_j) \mathbf{x}_j, \quad \forall j \in \mathcal{I}_k.$$

这样一来, 第 k 号 Worker 节点得到向量的集合 $\{z_j\}_{j \in \mathcal{I}_k}$ 。

3. 规约 (Reduce): 在做完映射之后, 向量 $z_1, \dots, z_n \in \mathbb{R}^d$ 分布式存储在 m 个 Worker 节点上, 每个节点有一个子集。不难看出, 目标函数 $L(\mathbf{w}) = \frac{1}{2n} \sum_{j=1}^n (\mathbf{x}_j^T \mathbf{w} - y_j)^2$ 在 \mathbf{w}_{now} 处的梯度等于:

$$\nabla_{\mathbf{w}} L(\mathbf{w}_{\text{now}}) = \frac{1}{n} \sum_{j=1}^n z_j.$$

因此, 我们应该使用 `sum` 规约函数。每个 Worker 节点首先会规约自己本地的 $\{z_j\}_{j \in \mathcal{I}_k}$, 得到

$$\tilde{g}^k \triangleq \sum_{j \in \mathcal{I}_k} z_j, \quad \forall k = 1, \dots, m.$$

然后将 $\tilde{g}_k \in \mathbb{R}^d$ 发送给服务器, 服务器对 $\tilde{g}^1, \dots, \tilde{g}^m$ 求和, 再除以 n , 得到梯度:

$$\nabla_{\mathbf{w}} L(\mathbf{w}_{\text{now}}) \leftarrow \frac{1}{n} \sum_{k=1}^m \tilde{g}^k.$$

先在本地做规约, 再做通信, 只需要传输 md 个浮点数; 如果不先在本地归约, 直接把所有的 $\{z_j\}_{j=1}^n$ 都发送给服务器, 那么需要传输 nd 个浮点数, 通信代价大得

多。

- 更新参数：最后，服务器在本地做梯度下降，更新模型参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \eta \cdot \nabla_{\mathbf{w}} L(\mathbf{w}_{\text{now}}).$$

这样就完成了一轮梯度下降，对参数做了一次更新。

12.1.4 并行计算的代价

通常用算法实际运行所需的时间来衡量并行计算的表现。时间有两种定义，请读者注意区分。

- 钟表时间 (Wall-clock Time)**，也叫 Elapsed Real Time，意思是程序实际运行的时间。可以这样理解钟表时间：在程序开始运行的时候，记录下墙上钟表的时刻；在程序结束的时候，再记录钟表的时刻；两者之差就是钟表时间。
- 处理器时间 (CPU Time 或 GPU Time)** 是所有处理器运行时间的总和。比如使用 4 块 CPU 做并行计算，程序运行的钟表时间是 1 分钟，期间 CPU 没有空闲，那么系统的 CPU 时间等于 4 分钟。

处理器数量越多，每块处理器承担的计算量就越小，那么程序运行速度就会越快。所以并行计算可以让钟表时间更短。用多个处理器做并行计算，然而总计算量没有减少，因此并行计算不会让处理器时间更短。

通常用**加速比 (Speedup Ratio)** 衡量并行计算带来的速度提升。加速比是这样计算的：

$$\text{加速比} = \frac{\text{使用一个节点所需的钟表时间}}{\text{使用 } m \text{ 个节点所需的钟表时间}}.$$

通常来说，节点数量越多，算力越强，加速比就越大。在实验报告中，通常需要把加速比绘制成一条曲线。把节点数量设置为不同的值，比如 $m = 1, 2, 4, 8, 16, 32$ ，得到相应的加速比。把 m 作为横轴，把加速比作为纵轴，绘制出加速比曲线；见图 12.6。

在最理想的情况下，使用 m 个节点，每个节点承担 $\frac{1}{m}$ 的计算量，那么钟表时间会减小到原来的 $\frac{1}{m}$ ，即加速比等于 m 。图 12.6 中的蓝色直线是理想情况下的加速比。但实际的加速比往往是图中的红色曲线，即加速比小于 m 。其原因在于计算所需时间只占总的钟表时间的一部分。通信等操作也要花费时间，导致加速比达不到 m 。下面分析并行计算中常见的时间开销。

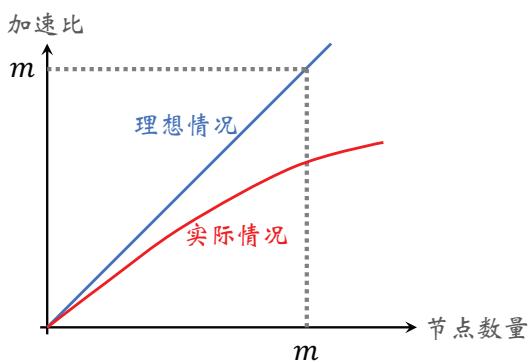


图 12.6：加速比曲线。

通信量 (Communication Complexity) 的意思是又有多少个比特或者浮点数在服务器与 Worker 节点之间传输。在并行梯度下降的例子中，每一轮梯度下降需要做两次通信：服务器将模型参数 $\mathbf{w} \in \mathbb{R}^d$ 广播给 m 个 Worker 节点，Worker 节点将计算出的梯度 $\tilde{\mathbf{g}}^1, \dots, \tilde{\mathbf{g}}^m$ 发送给服务器。因此每一轮梯度下降的通信量都是 $\mathcal{O}(md)$ 。很显然，通信量越大，通信

花的时间越长。

延迟 (Latency) 是由计算机网络的硬件和软件系统决定的。做通信的时候，需要把大的矩阵、向量拆分成小数据包，通过计算机网络逐个传输数据包。即使数据包再小，从发送到接收之间也需要花费一定时间，这个时间就是延迟。通常来说，延迟与通信次数成正比，而跟通信量关系不大。

通信时间 主要由通信量和延迟造成。我们无法准确预估通信时间（指的是钟表时间），除非实际做实验测量。但我们不妨用下面的公式粗略估计通信时间：

$$\text{通信时间} \approx \frac{\text{通信量}}{\text{带宽}} + \text{延迟}.$$

在并行计算中，通信时间是不容忽视的，通信时间甚至有可能超过计算时间。降低通信量和通信次数是设计并行算法的关键。只有当通信时间远低于计算时间，才能取得较高的加速比。

12.2 同步与异步

本节讨论同步算法、异步算法的区别，重点介绍异步并行梯度下降。用在机器学习中，异步算法的表现通常优于同步算法。

12.2.1 同步算法

上一节介绍的并行梯度下降算法属于**同步算法 (Synchronous Algorithm)**。如图 12.7 所示，在所有 Worker 节点都完成映射 (Map) 的计算之后，系统才能执行规约 (Reduce) 通信。这意味着即使有些节点先完成计算，也必须等待最慢节点；在等待期间，节点处于空闲状态。图 12.7 中黑色的竖线表示同步屏障，即所有节点都完成计算之后才能开始通信，当通信完成之后才能开始下一轮计算。

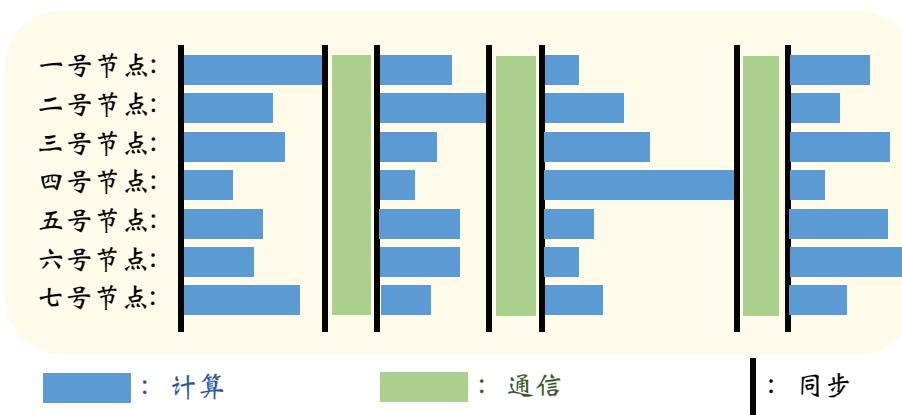


图 12.7: 同步梯度下降中的计算、通信、同步。图中横向表示时间。

同步的代价： 实际软硬件系统中存在负载不平衡、软硬件不稳定、I/O 速度不稳定等因素。因此 Worker 节点会有先后、快慢之分，不会恰好在同一时刻完成任务。同步要求每一轮都必须等待所有节点完成计算，这势必导致“短板效应”，即任务所需时间取决于最慢的节点。同步会造成很多节点处于空闲状态，无法有效利用集群的算力。

Straggler Effect 意思是一个节点的速度远慢于其余节点，导致整个系统长时间处于空闲状态，等待最慢的节点。Straggler 也叫 Outlier，字面意思是“掉队者”。产生 Straggler 的原因有很多种，比如在某个节点的硬件或软件出错之后，节点死掉或者重启，导致计算时间多几倍。如果把 MapReduce 这样的需要同步的系统部署到廉价、可靠性低的硬件上，Straggler Effect 可能会很严重。

12.2.2 异步算法

如果把图 12.7 中的同步屏障去掉，得到的算法就叫做**异步算法 (Asynchronous Algorithm)**，如图 12.8 所示。在异步算法中，一个 Worker 节点无需等待其余节点完成计算或通信。当一个 Worker 节点完成计算，它立刻跟 Server 通信，然后开始下一轮的计算。异步算法避免了等待，节点几乎没有空闲的时间，因此系统的利用率很高。

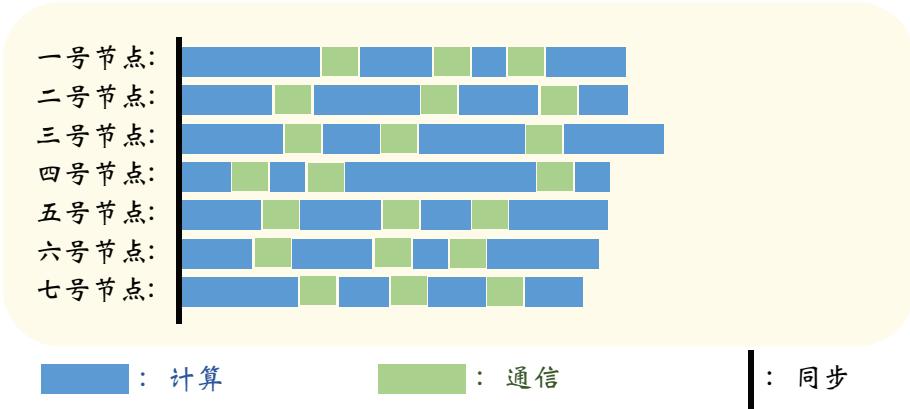


图 12.8: 异步算法中的计算、通信、同步。图中横向表示时间。

下面介绍异步梯度下降算法。我们仍然采用数据并发的方式，即把数据集 $\{(x_1, y_1), \dots, (x_n, y_n)\}$ 划分到 m 个 Worker 节点上。如图 12.9 所示，服务器可以单独与某个 Worker 节点通信：Worker 节点把计算出梯度发送给服务器，服务器把最新的参数发送给这个 Worker 节点。如果想要编程实现异步算法，可以用 Message Passing Interface (MPI) 这样底层的库，也可以借助 Ray³ 这样的框架。用户需要做的工作是编程实现 Worker 端、服务器端的计算。



图 12.9: 异步梯度下降。

Worker 端的计算： 每个 Worker 节点独立做计算，独立与服务器通信；Worker 节点之间不通信，不等待。第 k 号 Worker 节点重复下面的步骤：

1. 向服务器发出请求，索要最新的模型参数。把接收到的参数记作 w_{now} 。
2. 利用本地的数据 $\{(x_j, y_j)\}_{j \in \mathcal{I}_k}$ 和参数 w_{now} 计算本地的梯度：

$$\tilde{g}^k = \frac{1}{|\mathcal{I}_k|} \sum_{j \in \mathcal{I}_k} (x_j^T w_{\text{now}} - y_j) x_j.$$

³<https://ray.io/>

3. 把计算出的梯度 $\tilde{\mathbf{g}}^k$ 发送给服务器。

服务器端的计算：服务器上储存一份模型参数，并且用 Worker 发来的梯度更新参数。每当收到一个 Worker（比如第 k 号 Worker）发送来的梯度（记作 $\tilde{\mathbf{g}}^k$ ），服务器就立刻做梯度下降更新参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \eta \cdot \tilde{\mathbf{g}}^k.$$

服务器还需要监听 Worker 发送的请求。如果有 Worker 索要参数，就把当前的参数 \mathbf{w}_{new} 发送给这个 Worker。

12.2.3 同步与异步梯度下降的对比

上一节介绍的同步并行梯度下降完全等价于标准的梯度下降，只是把计算分配到了多个 Worker 节点上而已。然而异步梯度下降算法与标准的梯度下降是不等价的。同步与异步梯度下降不只是编程实现有区别，更是在算法上有本质区别。

1. 不难证明，**同步并行梯度下降**更新参数的方式为：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \eta \cdot \nabla_{\mathbf{w}} L(\mathbf{w}_{\text{now}}),$$

即标准的梯度下降。在同一时刻，所有 Worker 节点上的参数是相同的，都是 \mathbf{w}_{now} 。

所有 Worker 节点都基于相同的 \mathbf{w}_{now} 计算梯度。

2. 对于**异步并行梯度下降**，在同一时刻，不同 Worker 节点上的参数 \mathbf{w} 通常是不同的。

比如两个 Worker 分别在 t_1 和 t_2 时刻向服务器索要参数。在两个时刻之间，服务器可能已经对参数做了多次更新，导致在 t_1 和 t_2 时刻取回的参数不同。两个 Worker 节点会基于不同的参数计算梯度。

在理论上，异步梯度下降的收敛速度慢于同步算法，即需要更多的计算量才能达到相同的精度。但是实践中异步梯度下降远比同步算法快（指的是钟表时间），这是因为异步算法无需等待，Worker 节点几乎不会空闲，利用率很高。

12.3 并行强化学习

并行强化学习的目的在于用更少的钟表时间完成训练。第 12.3.1、12.3.2 小节分别用异步并行算法训练 DQN、Actor-Critic。本节介绍的异步算法与上一节的异步算法很类似，都是由 Worker 节点计算梯度，由服务器更新模型参数。

12.3.1 异步并行双 Q 学习

DQN 和双 Q 学习： DQN 是一个神经网络，记作 $Q(s, a; \mathbf{w})$ ，其中 s 是状态， a 是动作， \mathbf{w} 表示神经网络参数（包含多个向量、矩阵、张量）。通常用双 Q 学习等算法训练 DQN。双 Q 学习需要目标网络 $Q(s, a; \mathbf{w}^-)$ ，它的结构与 DQN 相同，但是参数不同。双 Q 学习属于异策略，即由任意策略控制智能体收集经验，事后做经验回放更新 DQN 参数。第 6 章介绍的高级技巧可以很容易地与双 Q 学习结合，此处就不详细解释了。

系统架构： 如图 12.10 所示，系统中有一个服务器和 m 个 Worker 节点。服务器可以随时给某个 Worker 发送一条信息，一个 Worker 也可以随时给服务器发送信息，但是 Worker 之间不能通信。服务器和 Worker 都存储 DQN 的参数。服务器上的参数是最新的，服务器用 Worker 发来的梯度对参数做更新。Worker 节点的参数可能是过时的，所以 Worker 需要频繁向服务器索要最新的参数。Worker 节点有自己的目标网络，而服务器上不存储目标网络。每个 Worker 节点有自己的环境，比如运行一个超级玛丽游戏，用 DQN 控制智能体与环境交互，收集经验，把 (s, a, r, s') 这样的四元组存储到本地的经验回放数组。在收集经验的同时，Worker 节点做经验回放，计算梯度，把梯度发送给服务器。

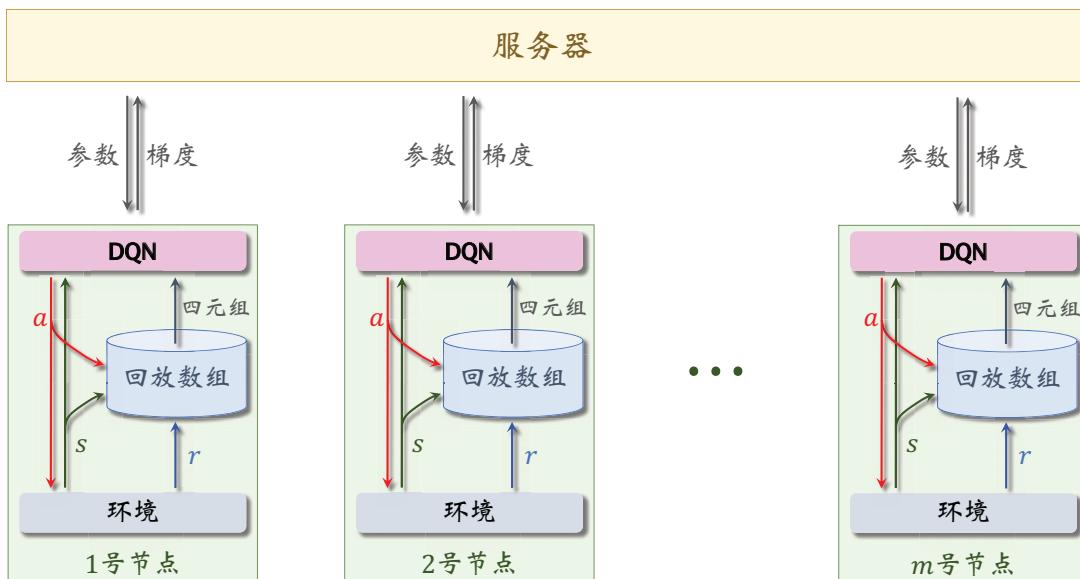


图 12.10：用异步并行算法训练 DQN。图中没有画出目标网络。

Worker 端的计算： 每个 Worker 节点本地有独立的环境，独立的经验回放数组，还有一个 DQN 和一个目标网络。（图 12.10 中没有画出目标网络。）设某个 Worker 节点当

前参数为 \mathbf{w}_{now} 。它用 ϵ -greedy 策略控制智能体与本地环境交互，收集经验。 ϵ -greedy 的定义是：

$$a_t = \begin{cases} \operatorname{argmax}_a Q(s_t, a; \mathbf{w}_{\text{now}}), & \text{以概率 } (1 - \epsilon); \\ \text{均匀抽取 } \mathcal{A} \text{ 中的一个动作,} & \text{以概率 } \epsilon. \end{cases}$$

把收集到的经验 (s_t, a_t, r_t, s_{t+1}) 存入本地的经验回放数组。

与此同时，所有的 Worker 节点都要参与异步梯度下降。Worker 节点在本地做计算，还要与服务器通信。第 k 号 Worker 节点重复下面的步骤：

1. 向服务器发出请求，索要最新的 DQN 参数。把接收到的参数记作 \mathbf{w}_{new} 。
2. 更新本地的目标网络：

$$\mathbf{w}_{\text{new}}^- \leftarrow \tau \cdot \mathbf{w}_{\text{new}} + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^-.$$

3. 在本地做经验回放，计算本地梯度：

- (a). 从本地的经验回放数组中随机抽取 b 个四元组，记作

$$(s_1, a_1, r_1, s'_1), (s_2, a_2, r_2, s'_2), \dots, (s_b, a_b, r_b, s'_b).$$

b 是批量的大小，由用户自己设定，比如 $b = 16$ 。

- (b). 用双 Q 学习计算 TD 目标。对于所有的 $j = 1, \dots, b$ ，分别计算

$$\hat{y}_j = r_j + \gamma \cdot Q(s'_j, a'_j; \mathbf{w}_{\text{new}}^-), \quad \text{其中 } a'_j = \operatorname{argmax}_a Q(s'_j, a; \mathbf{w}_{\text{new}}^-).$$

- (c). 定义目标函数：

$$L(\mathbf{w}) \triangleq \frac{1}{2b} \sum_{j=1}^b [Q(s_j, a_j; \mathbf{w}) - \hat{y}_j]^2.$$

- (d). 计算梯度：

$$\tilde{\mathbf{g}}^k = \nabla_{\mathbf{w}} L(\mathbf{w}_{\text{new}}).$$

4. 把计算出的梯度 $\tilde{\mathbf{g}}^k$ 发送给服务器。

服务器端的计算：服务器上储存有一份模型参数，记作 \mathbf{w}_{now} 。每当一个 Worker 节点发来请求，服务器就把 \mathbf{w}_{now} 发送给该 Worker 节点。每当一个 Worker 节点发来梯度 $\tilde{\mathbf{g}}^k$ ，服务器就立刻做梯度下降更新参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \tilde{\mathbf{g}}^k.$$

12.3.2 A3C: 异步并行 A2C

A2C 有一个策略网络 $\pi(a|s; \theta)$ 和一个价值网络 $v(s; \mathbf{w})$ 。通常用策略梯度更新策略网络，用 TD 算法更新价值网络。为了让 TD 算法更稳定，需要一个目标网络 $v(s; \mathbf{w}^-)$ ，它的结构与价值网络相同，但是参数不同。A2C 属于同策略，不能使用经验回放。A2C 的实现详见第 8.3 节。异步并行 A2C 被称作 **Asynchronous Advantage Actor-Critic (A3C)**。

系统架构：如图 12.10 所示，系统中有一个服务器和 m 个 Worker 节点。服务器维护策略网络和价值网络最新的参数，并用 Worker 节点发来的梯度更新参数。每个 Worker 节点有一份参数的拷贝，并每隔一段时间向服务器索要最新的参数。每个 Worker 节点有

一个目标网络，而服务器上不储存目标网络。每个 Worker 节点有独立的环境，用本地的策略网络控制智能体与环境交互，用状态、动作、奖励计算梯度。

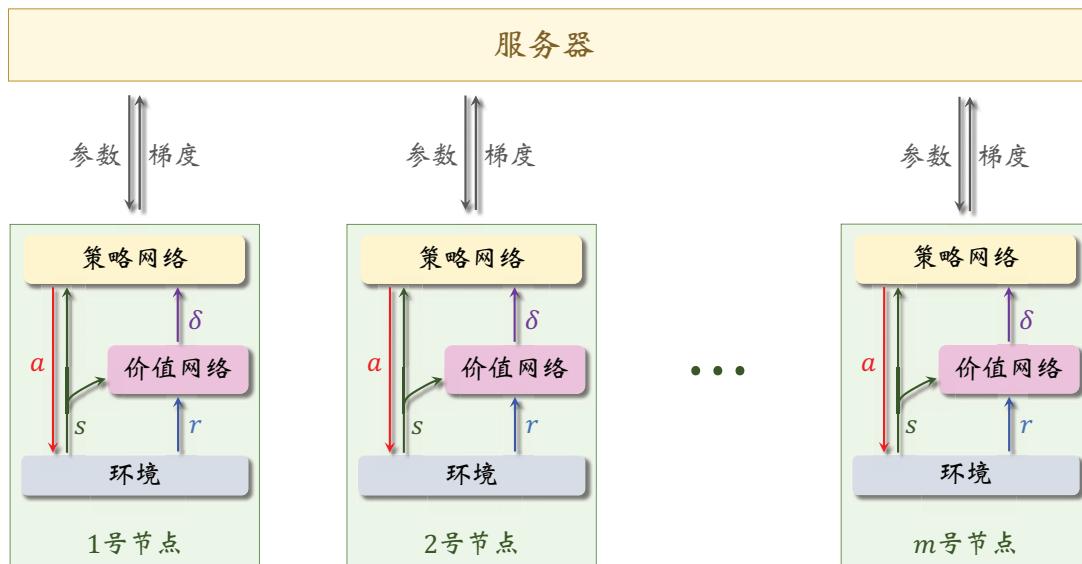


图 12.11: A3C，即异步并行 A2C。图中没有画出目标网络。

Worker 端的计算: 每个 Worker 节点有独立的环境，独立做计算，随时可以与服务器通信。每个 Worker 节点本地有一个策略网络 $\pi(a|s; \theta)$ 、一个价值网络 $v(s; w)$ 、一个目标网络 $v(s; w^-)$ 。设第 k 个 Worker 节点当前参数为 θ_{now} 、 w_{now} 、 w_{now}^- 。第 k 个 Worker 节点重复下面的步骤：

1. 向服务器发出请求，索要最新的参数。把接收到的参数记作 θ_{new} 、 w_{new} 。
2. 更新本地的目标网络：

$$w_{\text{new}}^- \leftarrow \tau \cdot w_{\text{new}} + (1 - \tau) \cdot w_{\text{now}}^-.$$

3. 重复下面的步骤 b 次 (b 是用户设置的超参数)，或是从头到尾完成一回合游戏。让智能体与环境交互，计算策略梯度，并累积策略梯度。全零初始化 $\tilde{g}_\theta^k \leftarrow \mathbf{0}$ 、 $\tilde{g}_w^k \leftarrow \mathbf{0}$ ，用它们累积梯度。
 - (a). 基于当前状态 s_t ，根据策略网络做决策 $a_t \sim \pi(\cdot | s_t, \theta)$ ，让智能体执行动作 a_t 。随后观测到奖励 r_t 和新状态 s_{t+1} 。
 - (b). 计算 TD 目标 \hat{y}_t 和 TD 误差 δ_t : ⁴

$$\begin{aligned}\hat{y}_t &= r_t + \gamma \cdot v(s_{t+1}; w_{\text{new}}^-), \\ \delta_t &= v(s_t; w_{\text{new}}) - \hat{y}_t.\end{aligned}$$

- (c). 累积梯度：

$$\begin{aligned}\tilde{g}_w^k &\leftarrow \tilde{g}_w^k + \delta_t \cdot \nabla_w v(s_t; w_{\text{new}}), \\ \tilde{g}_\theta^k &\leftarrow \tilde{g}_\theta^k + \delta_t \cdot \nabla_\theta \ln \pi(a_t | s_t; \theta_{\text{new}}).\end{aligned}$$

⁴此处可以用多步 TD 目标等技巧；详见第 5.3 节。

4. 把累积的梯度 \tilde{g}_θ^k 和 \tilde{g}_w^k 发送给服务器。

服务器端的计算： 服务器上储存有一份模型参数，记作 θ_{now} 和 w_{now} 。每当一个 Worker 节点发来请求，服务器就把 θ_{now} 和 w_{now} 发送给该 Worker 节点。每当一个 Worker 节点发来梯度 \tilde{g}_θ^k 和 \tilde{g}_w^k ，服务器就立刻做梯度下降更新参数：

$$w_{\text{new}} \leftarrow w_{\text{now}} - \alpha \cdot \tilde{g}_w^k,$$

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \beta \cdot \tilde{g}_\theta^k.$$

∽ 第十二章 相关文献 ∽

MapReduce 原本是指 Google 内部使用的软件系统，现在泛指这类系统架构。Google 的 MapReduce 系统不对外开源，但是外界可以通过 2008 年的论文 [33] 了解系统的设计。外界有多个开源项目力图实现 MapReduce 系统，其中最有名的是 Hadoop。后来基于 Hadoop 等项目开发的 Spark [123] 比 Hadoop MapReduce 的速度更快。本章介绍的异步并行算法主要基于 Parameter Server [60] 的思想。Ray [71] 是一个开源的软件系统，包含 Parameter Server 的功能。用 Ray 很容易实现异步并行算法，而且 Ray 对强化学习有很好的支持。

本章介绍的并行强化学习算法主要基于 2015 年的论文 [73] 和 2016 年的论文 [68]。两篇论文都是异步算法，主要区别在于 2015 年的论文 [73] 使用经验回放，而 2016 年的论文 [68] 不用经验回放。对于 Atari 游戏这类问题，获取经验非常容易，于是使用经验回放与否其实无关紧要。

第十三章 多智能体系统

之前章节的设定都是单智能体系统 (Single-Agent System, 缩写 SAS)。本章和后面三章介绍多智能体系统 (Multi-Agent System, 缩写 MAS) 和多智能体强化学习 (Multi-Agent Reinforcement Learning, 缩写 MARL)。本章讲解多智能体系统的基本概念，帮助大家理解 MAS 与 SAS 的区别。第 13.1 节讲解 MAS 的四种常见设定。第 13.2 节定义 MAS 的专业术语，将之前所学的观测、动作、奖励、策略、价值等概念推广到 MAS。第 13.3 节介绍几种常用的实验环境，用于对比 MARL 方法的优劣。

13.1 多智能体系统的设定

多智能体系统与单智能体系统的区别：多智能体系统 (Multi-Agent System, 缩写 MAS) 中包含 m 个智能体，智能体共享环境，智能体之间会相互影响。智能体之间是如何相互影响的呢？一个智能体的动作会改变环境状态，从而影响其余所有智能体。举个例子，股市中的每个自动交易程序就可以看做一个智能体。尽管智能体（自动交易程序）之间不会交流，它们依然会相互影响：一个交易程序的决策会影响股价，从而对其他自动交易程序有利或有害。

注意，MAS 与上一章的并行强化学习是不同的概念。上一章用 m 个节点并行计算，每个节点有独立的环境，每个环境中有一个智能体。虽然 m 个节点上一共有 m 个智能体，但是智能体之间完全独立，不会相互影响。而本章 MAS 只有一个环境，环境中有 m 个相互影响的智能体。并行强化学习的设定是 m 个单智能体系统 (Single-Agent System, 缩写 SAS) 的并集，可以视作 MAS 的一种特例。举个例子，环境中有 m 个机器人，这属于 MAS 的设定。假如把每个机器人隔绝在一个密闭的房间中，机器人之间不会通信，那么 MAS 就变成了多个 SAS 的并集。

多智能体强化学习 (Multi-Agent Reinforcement Learning, 缩写 MARL) 是指让多个智能体处于相同的环境中，每个智能体独立与环境交互，利用环境反馈的奖励改进自己的策略，以获得更高的回报（即累计奖励）。在多智能体系统中，一个智能体的策略不能简单依赖于自身的观测、动作，还需要考虑到其他智能体的观测、动作。因此，MARL 比单智能体强化学习 (Single-Agent Reinforcement Learning, 缩写 SARL) 更困难。

多智能体系统有四种常见设定：合作关系 (Fully Cooperative)、竞争关系 (Fully Competitive)、合作竞争的混合 (Mixed Cooperative & Competitive)、利己主义 (Self-Interested)。图 13.1 举例说明了四种常见设定。接下来具体讲解这些设定。

第一种设定是**完全合作关系**：智能体的利益一致，获得的奖励相同，有共同的目标。比如图 13.1 中，多个工业机器人协同装配汽车。他们的目标是相同的，都希望把汽车装好。假设一共有 m 个智能体，它们在 t 时刻获得的奖励分别是 $R_t^1, R_t^2, \dots, R_t^m$ 。（用上标表示智能体，用下标表示时刻。）在完全合作关系中，它们的奖励是相同的：

$$R_t^1 = R_t^2 = \dots = R_t^m, \quad \forall t.$$

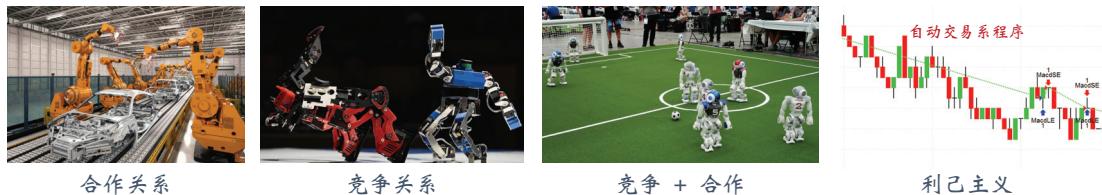


图 13.1: 多智能体强化学习的四种常见设定。四张图片来源于网络。

第二种设定是**完全竞争关系**: 一方的收益是另一方的损失。比如图 13.1 中的两个格斗机器人，它们的利益是冲突的，一方的胜利就是另一方的失败。在完全竞争的设定下，双方的奖励是负相关的：对于所有的 t , 有 $R_t^1 \propto -R_t^2$ 。如果是零和博弈，双方的获得的奖励总和等于 0 : $R_t^1 = -R_t^2$ 。

第三种设定是**合作竞争的混合**。智能体分成多个群组；组内的智能体是合作关系，它们的奖励相同；组间是竞争关系，两组的奖励是负相关的。比如图 13.1 中的足球机器人：两组是竞争关系，一方的进球是另一方的损失；而组内是合作关系，队友的利益是一致的。

第四种设定是**利己主义**。系统内有多个智能体；一个智能体的动作会改变环境状态，从而让别的智能体受益或者受损。利己主义的意思是智能体只想最大化自身的累计奖励，而不在乎他人收益或者受损。比如图 13.1 中的股票自动交易程序可以看做是一个智能体；环境（股市）中有多个智能体。这些智能体的目标都是最大化自身的收益，因此可以看做利己主义。智能体之间会相互影响：一个智能体的决策会影响股价，从而影响其他自动交易程序的收益。智能体之间有潜在而又未知的竞争与合作关系：一个智能体的决策可能会帮助其他智能体获利，也可能导致其他智能体受损。设计自动交易程序的时候，不应当把它看做孤立的系统，而应当考虑到其他自动交易程序的行为。

不同设定下学出的策略会有所不同。在**合作**的设定下，每个智能体的决策要考虑到队友的策略，要与队友做到尽量好的配合，而不是个人英雄主义；这个道理在足球、竞技电游中是显然的。在**竞争**的设定下，智能体要考虑到对手的策略，相应调整自身策略；比如在象棋游戏中，如果你很熟悉对手的套路，并相应调整自己的策略，那么你的胜算会更大。在**利己主义**的设定下，一个智能体的决策无需考虑其他智能体的利益，尽管一个智能体的动作可能会在客观上帮助或者妨害其他智能体。

13.2 多智能体系统的基本概念

本书第 3 章定义了单智能体系统的专业术语，比如状态、动作、奖励、策略、价值。在本节中，我们将这些定义推广到多智能体系统。在此后的章节中，我们用 m 表示智能体的数量，用上标 i 表示智能体的序号（ i 从 1 到 m ），依然用下标 t 表示时刻。

13.2.1 专业术语

本章依然用大写字母 S 表示**状态** (State) 随机变量，用小写字母 s 表示状态的观测值。注意，单个智能体未必能观测到完整状态。如果单个智能体的观测只是部分状态，我们就用 o^i 表示第 i 号智能体的不完全观测。

每个智能体都会做出**动作** (Action)。把第 i 号智能体的动作随机变量记作 A^i ，把动作的实际观测值记作 a^i 。如果不加上标 i ，则意味着所有智能体的动作的连接：

$$A = [A^1, A^2, \dots, A^m], \quad a = [a^1, a^2, \dots, a^m].$$

把第 i 号智能体的动作空间 (Action Space) 记作 \mathcal{A}^i ，它包含该智能体所有可能的动作。整个系统的动作空间是 $\mathcal{A} = \mathcal{A}^1 \times \dots \times \mathcal{A}^m$ 。两个智能体的动作空间 \mathcal{A}^i 和 \mathcal{A}^j 可能相同，也可能不同。比如在电子游戏中，有的士兵会远程攻击，而有的士兵只能近距离攻击，不同类型的士兵可以有不同的动作空间。

所有智能体都执行动作之后，环境依据**状态转移函数** (State-Transition Function) 给出下一时刻的状态。状态转移函数是个条件概率密度函数，记作

$$p(s_{t+1} | s_t; a_t) = \mathbb{P}[S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t].$$

它的意思是下一时刻状态 S_{t+1} 取决于当前时刻状态 S_t 、以及所有 m 个智能体的动作 $A_t = [A_t^1, A_t^2, \dots, A_t^m]$ 。

奖励 (Reward) 是环境反馈给智能体的数值。把第 i 号智能体的奖励随机变量记作 R^i ，把奖励的实际观测值记作 r^i 。在合作的设定下， $R^1 = R^2 = \dots = R^m$ ；在竞争的设定下， $R^1 \propto -R^2$ 。第 t 时刻的奖励 R_t^i 由状态 S_t 和所有智能体的动作 $A = [A^1, A^2, \dots, A^m]$ 共同决定。为什么一个智能体获得的奖励会取决于其他智能体的动作呢？举个例子，在足球比赛中，假如对方失误，自己进了个乌龙球；而你什么也没做，就获得了一分的奖励。

折扣回报 (Discounted Return) 也叫折扣累计奖励，它的定义类似于单智能体系统。第 i 号智能体的折扣回报是它自己的奖励的加权和：

$$U_t^i = R_t^i + \gamma \cdot R_{t+1}^i + \gamma^2 \cdot R_{t+2}^i + \gamma^3 \cdot R_{t+3}^i + \dots$$

此处的 $\gamma \in [0, 1]$ 是折扣率 (Discount Factor)。

13.2.2 策略网络

策略网络的意思是用神经网络近似策略函数。可以让每个智能体有自己的策略网络。对于**离散控制问题**，把第 i 号智能体的策略网络记作：

$$\hat{f} = \pi(\cdot | s; \theta^i).$$

策略网络的输入是状态 s , 输出是向量 $\hat{\mathbf{f}}$ 。向量 $\hat{\mathbf{f}}$ 的维度是动作空间的大小 $|\mathcal{A}^i|$, $\hat{\mathbf{f}}$ 的每个元素表示一个动作的概率。 $\hat{\mathbf{f}}$ 的元素都是正实数, 而且相加等于 1。做决策的时候, 根据 $\hat{\mathbf{f}}$ 做随机抽样, 得到动作 a^i , 第 i 号智能体执行这个动作。

对于连续控制问题, 即动作空间 \mathcal{A}^i 是连续集, 把第 i 号智能体的策略网络记作:

$$\mathbf{a}^i = \mu(s; \theta^i), \quad \forall i = 1, \dots, m.$$

有了这个策略网络, 第 i 号智能体就可以基于当前状态 s , 直接计算出需要执行的动作 \mathbf{a}^i 。

在上面的两种策略网络中, 每个智能体的策略网络有各自的参数: $\theta^1, \theta^2, \dots, \theta^m$ 。在有些情况下, 策略网络的角色是可以互换的, 比如同一型号无人机的功能是相同的, 那么它们的策略网络是相同的: $\theta^1 = \theta^2 = \dots = \theta^m$ 。但是在很多应用中, 策略网络不能互换。比如在足球机器人的应用中, 球员有的是负责进攻的前锋, 有的是负责防守的后卫, 还有一个守门员。它们的策略网络不能互换, 所以参数 $\theta^1, \dots, \theta^m$ 各不相同。

13.2.3 动作价值函数

上面讨论过, 第 i 号智能体在第 t 时刻得到的奖励 R_t^i 依赖于状态 S_t 、以及所有智能体的动作 $A_t = [A_t^1, \dots, A_t^m]$ 。因为(折扣)回报 U_t^i 是未来所有奖励 $R_t^i, R_{t+1}^i, \dots, R_n^i$ 之和, 所以 U_t^i 依赖于未来所有状态

$$S_t, S_{t+1}, S_{t+2}, \dots, S_n$$

与所有智能体未来的动作

$$A_t, A_{t+1}, A_{t+2}, \dots, A_n.$$

在 t 时刻, 回报 U_t^i 是个随机变量, 其随机性的来源是未来所有状态、所有智能体未来的动作。

如果用期望消掉回报 U_t^i 中的随机性, 就能得到价值函数。把 t 时刻的状态 s_t 和所有智能体的动作 $a_t = [a_t^1, \dots, a_t^m]$ 当做观测值, 用期望消掉 $t+1$ 时刻之后未知的状态和动作, 得到的结果就是动作价值函数(Action-Value Function):

$$Q_\pi^i(s_t, a_t) = \mathbb{E}[U_t^i | S_t = s_t, A_t = a_t]. \quad (13.1)$$

此处的期望是关于这些随机变量求的:

- 未来的状态 $S_{t+1}, S_{t+2}, \dots, S_n$ 。
- 未来动作 $A_{t+1}, A_{t+2}, \dots, A_n$; 这里的 $A_k = [A_k^1, \dots, A_k^m]$ 是所有智能体在 k 时刻的动作。

公式(13.1)中关于动作 $A_k = [A_k^1, \dots, A_k^m]$ 求期望, $\forall k$, 要用到动作 A_k 的概率质量函数, 即所有 m 个智能体的策略的乘积:

$$\pi(A_k^1 | S_k; \theta^1) \times \pi(A_k^2 | S_k; \theta^2) \times \dots \times \pi(A_k^m | S_k; \theta^m).$$

也就是说, 第 i 号智能体的动作价值 $Q_\pi^i(s_t, a_t)$ 依赖于所有 m 个智能体的策略。

为什么第 i 号智能体的动作价值 $Q_\pi^i(s, a)$ 会依赖于其余智能体的策略呢? 这里给一个直观的解释。在足球游戏中, 假如你有个猪队友(即策略很差), 那么你未来获得不了

多少奖励，所以你的 Q_π^i 会比较小。假如把猪队友换成靠谱的队友（即策略更好），你的 Q_π^i 会变大。虽然你没有改变自己的策略，但是你的动作价值 Q_π^i 会随着队友的策略变化。

总结一下。如果系统里有 m 个智能体，那么就有 m 个动作价值函数：

$$Q_\pi^1(s, a), \quad Q_\pi^2(s, a), \quad \dots, \quad Q_\pi^m(s, a).$$

第 i 号智能体的动作价值 $Q_\pi^i(s_t, a_t)$ 并非仅仅依赖于自己当前的动作 a_t^i 与策略 $\pi(a_t^i | s_t; \theta^i)$ 。
 $Q_\pi^i(s_t, a_t)$ 依赖于其余智能体当前的动作

$$a_t = [a_t^1, a_t^2, \dots, a_t^m]$$

与所有智能体的策略

$$\pi(a^1 | s; \theta^1), \quad \pi(a^2 | s; \theta^2), \quad \dots, \quad \pi(a^m | s; \theta^m).$$

13.2.4 状态价值函数

我们在第 3 章中学过单智能体系统的状态价值函数 (State-Value Function)，记作 $V_\pi(S)$ ，并在策略学习的方法中反复用到 $V_\pi(S)$ 。它是对动作价值函数 $Q_\pi(S, A)$ 关于当前动作 A 的期望：

$$V_\pi(s) = \mathbb{E}_A [Q_\pi(s, A)] = \sum_{a \in \mathcal{A}} \pi(A | s; \theta) \cdot Q_\pi(s, a).$$

下面我们将状态价值函数的定义推广到多智能体系统。

第 i 号智能体的动作价值函数是 $Q_\pi^i(S, A)$ 。想要对 $Q_\pi^i(S, A)$ 关于 $A = [A^1, \dots, A^m]$ 求期望，需要用到 A 的概率质量函数，即所有 m 个智能体的策略的乘积：

$$\pi(A | S; \theta^1, \dots, \theta^m) \triangleq \pi(A^1 | S; \theta^1) \times \dots \times \pi(A^m | S; \theta^m).$$

状态价值函数可以写成：

$$V_\pi^i(s) = \mathbb{E}_A [Q_\pi^i(s, A)] = \sum_{a^1 \in \mathcal{A}^1} \sum_{a^2 \in \mathcal{A}^2} \dots \sum_{a^m \in \mathcal{A}^m} \pi(a | s; \theta^1, \dots, \theta^m) \cdot Q_\pi^i(s, a).$$

很显然，第 i 号智能体的状态价值 $V_\pi^i(s)$ 依赖于所有智能体的策略：

$$\pi(a^1 | s; \theta^1), \quad \pi(a^2 | s; \theta^2), \quad \dots, \quad \pi(a^m | s; \theta^m).$$

MARL 的困难之处就在于一个智能体的价值 Q_π^i 与 V_π^i 受其他智能体策略的影响。举个例子，在足球运动中，其他所有人的策略都没变化，只有一个前锋改进了自己的策略，让他自己水平更高。那么他的队友的价值会变大，而对手的价值会变小。一个智能体 i 单独改进自己的策略，未必能让自己的价值 Q_π^i 与 V_π^i 变大，因为其他智能体的策略可能已经发生了变化。

13.3 实验环境

如果你设计出一种新的 MARL 方法，你应该将其与已有的标准方法做比较，看新的方法是否有优势。下面介绍几种 MARL 的实验环境，用于评价 MARL 方法的优劣。建议读者跳过本节内容，等到需要做 MARL 的实验的时候再阅读本节。

13.3.1 Multi-Agent Particle World

Multi-Agent Particle World 是一类简单的多智能体控制问题，其中包含很多种环境，如图 13.2 所示。这些环境由 Lowe 等人 [65] 开发，源代码公开在 GitHub 上：<https://github.com/openai/multiagent-particle-envs.git>。下面介绍图 13.2 中的四个环境。

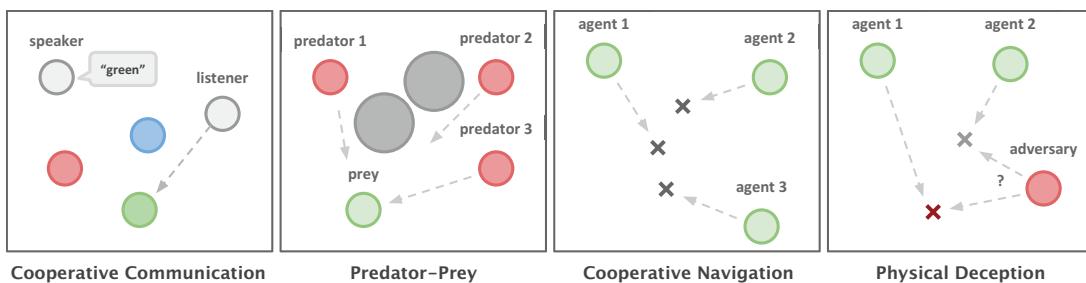


图 13.2: Multi-Agent Particle World 中的四种常用环境。图片来源于 2017 年的论文 [65]。

Cooperative Communication 这个环境中有三个点，每个点有一种颜色，这三个点不会移动。环境中有两个合作关系的智能体，一个叫做“Speaker”，另一个叫做“Listener”，它们是合作关系。任务是给定一种颜色 c ，让 Listener 移动到这种颜色的点上；离该点越近，则奖励越大。

- Speaker 的观测是 c ，即 Speaker 知道任务要求的颜色是什么。
- Speaker 的动作是发送一条信息，比如向量 $[0.1, 0.9, 0]$ 。很显然，训练 Speaker 的目的是让它发送的信息是颜色 c 的编码。
- Listener 的观测是三个点的颜色、三个点的位置（指的是相对位置）、以及 Speaker 发送的信息。比如，这是 Listener 的一个观测：

$$\left(\underbrace{[-1.5, -0.5]}_{\text{红点的位置}}, \underbrace{[-0.9, -0.9]}_{\text{绿点的位置}}, \underbrace{[-0.8, -0.2]}_{\text{蓝点的位置}}, \underbrace{[0, 1, 0]}_{\text{Speaker 发送的信息}} \right).$$

- Listener 的动作空间是这个离散集合：{不动, 上, 下, 左, 右}。

Predator-Prey 这个环境中有多个智能体，它们分为两类——多个 Predators (捕食者) 与一个 Prey (猎物)。这个问题属于混合关系，即同时存在合作与竞争关系。Predators 数量多，占有优势；为了平衡双方实力，环境的设置让 Predators 速度慢于 Prey。环境中有关碍物，智能体必须绕路。

- **奖励：**如果一个 Predator 碰到 Prey (猎物)，所有的 Predators 都会收到奖励，而 Prey 受到惩罚。

- **观测**: 每个智能体都能观测到障碍物的位置、其余智能体的位置。此处的“位置”指的是相对位置。
- **动作**: 每个智能体的动作空间都是{不动, 上, 下, 左, 右}。

Cooperative Navigation 环境中有 m 个合作关系的智能体与 m 个不动的点。

- **奖励**: 每个不动点都带有奖励，离该点最近的智能体会收集到奖励，奖励的大小与距离负相关。也就是说，最好的策略是让 m 个智能体分别覆盖 m 个点。智能体应当远离彼此；如果两个智能体碰撞，则会受到惩罚。
- **观测**: 每个智能体都能观测到其他智能体的位置、以及 m 个点的位置。此处的“位置”指的是相对位置。
- **动作**: 每个智能体的动作空间都是{不动, 上, 下, 左, 右}。

Physical Deception 这个环境中 $m+1$ 个智能体，其中 m 个是合作关系的玩家，一个是对手。这个问题属于混合关系。

- **奖励**: 环境中有 m 个点，其中一个点 x 带有奖励，离 x 距离最近的玩家获得奖励，奖励的大小与距离负相关。也就是说，应当有一个玩家到达点 x ；但这是不够的。对手也想到达点 x ；对手离 x 越近，对手得到的奖励越大，而对手的奖励是玩家的惩罚。
- **玩家的观测**: 玩家知道所有玩家的位置、所有点的位置、以及哪个点是带奖励的点 x 。此处的“位置”指的是相对位置。
- **对手的观测**: 对手知道所有玩家的位置、所有点的位置，但是不知道哪个点是 x 。此处的“位置”指的是相对位置。
- **动作**: 每个智能体的动作空间都是{不动, 上, 下, 左, 右}。

虽然只有当覆盖 x 的时候有奖励，但是玩家不能仅仅覆盖点 x ，而不覆盖其余的点。否则对手会推测出 x 是哪一个点。因此，玩家最好的策略是覆盖所有 m 个点，从而迷惑对手。

13.3.2 StarCraft Multi-Agent Challenge (SMAC)

星际争霸2(StarCraft II)是由暴雪在2010年推出的一款即时战略游戏。游戏中有很多兵种(即很多类型的智能体)，每个兵种有自己的生命值、护甲、移动速度、攻击范围、杀伤力等属性。一个士兵在生命值耗尽的时候死去，从游戏中消失。星际争霸游戏中可以有多个玩家，每个玩家控制一支军队；一支军队中有若干兵种，每个兵种有若干士兵。

StarCraft Multi-Agent Challenge (SMAC)是基于星际争霸2开发的库，是对星际争霸游戏的简化。在SMAC中，玩家控制一支军队，与游戏AI控制的军队对战。消灭掉对方所有的士兵，就算胜利；如果己方全部士兵都死亡，就算失败。SMAC由Samvelyan等人[82]开发，源代码公开在GitHub上：<https://github.com/oxwhirl/smac.git>。SMAC库中有很多对战的环境。图13.3展示了两种环境。

SMAC中每个士兵是一个智能体，有自己的观测(多个向量)，能做出离散动作。如图13.4所示，每个士兵有自己的视野，能观测到一个圆内的所有队友和对手。每个士兵



(a) 双方各有 3 只 Stalkers 和 5 只 Zealots。 (b) 一方有 7 只 Zealots，另一方有 32 只 Banelings。

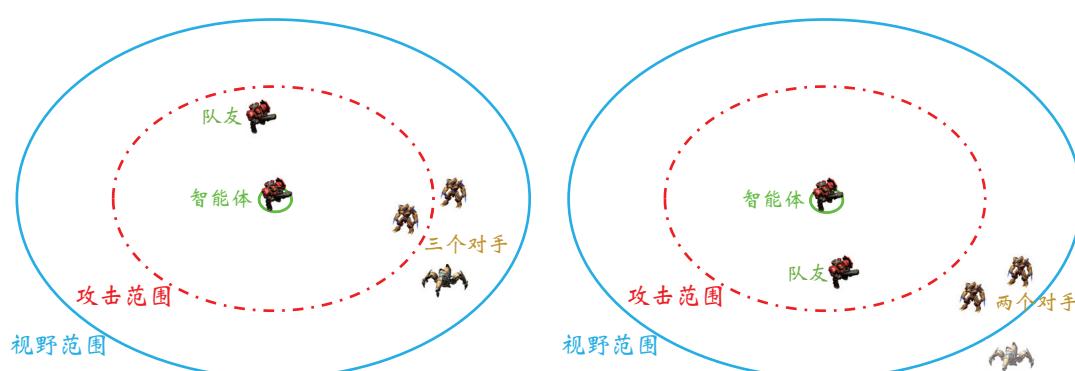
图 13.3: SMAC 库中的两种环境。

有自己的攻击范围，但仅限于攻击范围之内。每个智能体的**观测**表示为多个向量。

- 每个向量对应视野范围内的一个士兵，可以是队友或对手。
- 每个向量包含以下信息：距离、相对位置、生命值 (Health)、护甲 (Shield)、士兵类型 (Unit Type)、上一个动作（仅知道队友的上一个动作）。

每个智能体的**动作空间**是离散的，每个智能体每次可以什么也不做，或者执行下面动作中的一种：

- 向东、西、南、北四个方向中的一个移动。
- 攻击对手（或治疗队友），仅限与攻击范围之内，需要指定被攻击（或被治疗）目标的 ID。



(a) 智能体观测到 4 个士兵，表示成 4 个向量。 (b) 智能体观测到 3 个士兵，表示成 3 个向量。智能体观测不到视野范围之外的士兵。

图 13.4: 玩家控制两个士兵，对手控制三个士兵。玩家的两个士兵相当于两个智能体，它们有各自的观测和动作。

一个团队的士兵是合作关系，奖励是给予团队的，而不是给具体某个士兵。SMAC 有两种类型的奖励可供选择。一种是稀疏的奖励：最终游戏胜利获得奖励 +1，失败获得奖励 -1。另一种是稠密的奖励：杀死对方一个士兵有正奖励，己方士兵被杀有负奖励；外加游戏结束时胜利、失败的奖励。

13.3.3 Hanabi Challenge

花火 (Hanabi) 是一种合作型的卡牌游戏，玩家不能观看自己的牌，只能看其他玩家的。游戏的玩法是要将不同花色的数字牌按顺序排列。每回合中玩家只能获得有限的信息，需要做推理，从而做出决策。花火的规则较为复杂，此处不详细解释。有兴趣的读者可以在互联网上检索“花火卡牌游戏”，了解游戏规则。Hanabi Challenge 由 Bard 等人 [9] 在 2020 年开发，源代码公开在 GitHub 上：<https://github.com/deepmind/hanabi-learning-environment.git>。该程序提供花火游戏的环境，可供 MARL 学术研究。

从强化学习的角度来看，花火属于合作类型的 MARL。一局游戏结束时有奖励，奖励是给予团队的，而非玩家个人。每个玩家相当于一个智能体，他无法观测到全局状态，只能在不完全观测的情况下做出决策。玩家可以看到队友的牌，但是不能看自己的牌；玩家要靠队友提供的情报来推测自己的牌。玩家每一回合可以做出三种动作中的一种：提供情报、弃置一张牌、打出一张牌。提供情报的次数是很有限制的，玩家必须学会传递最有用的情报。玩家获得的奖励由出牌的好坏决定。综上所述，玩家需要学会两种能力：第一，将最有用的情报传递给队友；第二，根据队友传递的情报做出决策。

∽ 第十三章 相关文献 ∽

合作关系的 MARL 在自动控制领域被称作 Team Markov Games [48, 112, 122]。合作关系的 MARL 在 AI 领域最早见于论文 [17, 57]。竞争关系的 MARL 最早见于论文 [63]。混合关系的 MARL 最早见于论文 [51, 56, 64]。

很早就有论文将 Q 学习等价值学习方法推广到 MARL。1993 年的论文 [101] 研究了独立 Q 学习 (Independent Q-Learning, 缩写 IQL)，即智能体独立做 Q 学习，不共享信息。2017 年的论文 [38, 100] 将 IQL 用在深度强化学习。比较有名的多智能体价值学习方法有 Value-Decomposition Networks [94]、QMIX [79] 等方法。目前 MARL 更流行 Actor-Critic 方法，比如 [42, 37, 65, 52]。其中最有名的是 2018 年的 COMA [37] 与 2017 年的 MADDPG [65]。

对 MARL 感兴趣的读者可以阅读这些综述和书籍：Weiss 1999 [116]，Stone & Veloso 2000 [93]，Vlassis 2007 [111]，Shoham & Leyton-Brown 2008 [89]，Buşoniu *et al.* 2010 [23]，Zhang *et al.* 2019 [124]。

第十四章 合作关系设定下的多智能体强化学习

本章只考虑最简单的设定——完全合作关系——并在这种设定下研究多智能体强化学习 (MARL)。第 14.1 节定义“完全合作关系”下的策略学习。第 14.2 节介绍“完全合作关系”下的多智能体 A2C 方法，本书称之为 MAC-A2C。第 14.3 节介绍 MARL 的三种常见架构——完全去中心化、完全中心化、中心化训练 + 去中心化决策——并在三种框架下实现 MAC-A2C。

本章与上一章对状态的定义有所区别。在多智能体系统中，一个智能体未必能观测到全局状态 S 。设第 i 号智能体有一个局部观测，记作 O^i ，它是 S 的一部分。不妨假设所有的局部观测的总和构成全局状态：

$$S = [O^1, O^2, \dots, O^m],$$

MARL 的文献大多采用这种假设。本章中采用的符号如图 14.1 所示。

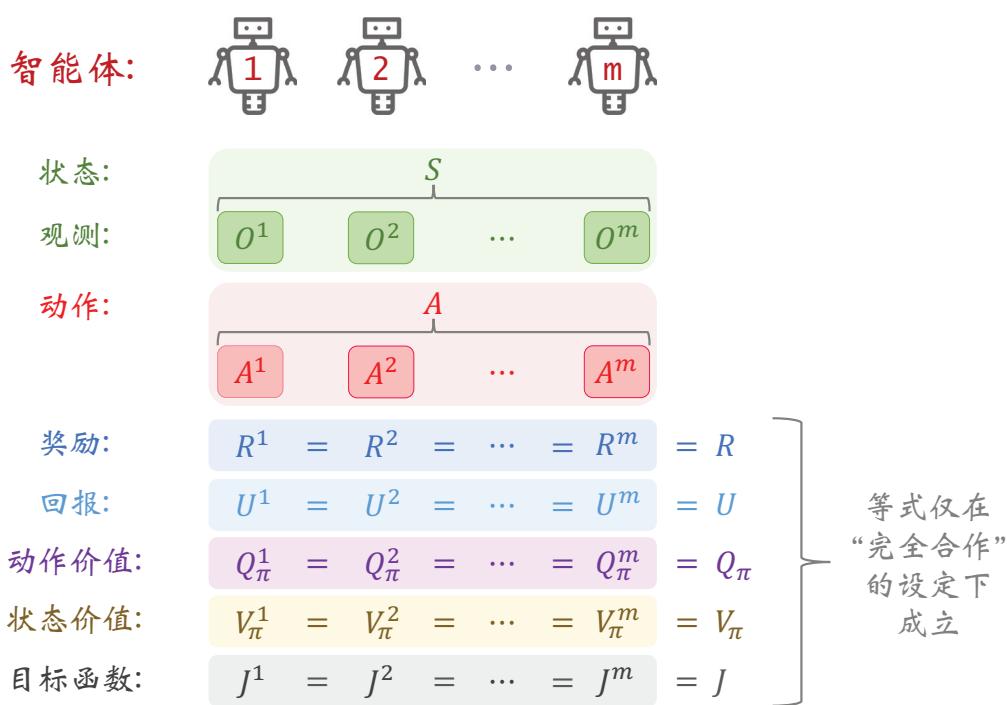


图 14.1: 多智能体强化学习 (MARL) 在“完全合作关系”设定下的符号。

14.1 合作关系设定下的策略学习

MARL 中的完全合作关系 (Fully-Cooperative) 意思是所有智能体的利益是一致的，它们具有相同的奖励：

$$R^1 = R^2 = \dots = R^m \triangleq R.$$

因此，所有的智能体都有相同的回报：

$$U^1 = U^2 = \dots = U^m \triangleq U.$$

因为价值函数是回报的期望，所以所有的智能体都有相同的价值函数。省略上标 i ，把动作价值函数记作 $Q_\pi(S, A)$ ，把状态价值函数记作 $V_\pi(S)$ 。

注意，价值函数 Q_π 和 V_π 依赖于所有智能体的策略：

$$\pi(A^1 | S; \theta^1), \quad \pi(A^2 | S; \theta^2), \quad \dots, \quad \pi(A^m | S; \theta^m).$$

举个例子，在某个竞技电游中，玩家组队打任务；每完成一个任务，团队成员（即智能体）获得相同的奖励。所以大家的 R, U, Q_π, V_π 全都是一样的。回报的期望——即价值函数 Q_π 与 V_π ——显然与所有成员的策略相关：只要有一个猪队友（即策略差）拖后腿，就有可能导致任务失败。通常来说，团队成员有分工合作，所以每个成员的策略是不同的，即 $\theta^i \neq \theta^j$ 。

如果做策略学习（即学习策略网络参数 $\theta^1, \dots, \theta^m$ ），那么所有智能体都有一个共同目标函数：

$$J(\theta^1, \dots, \theta^m) = \mathbb{E}_S[V_\pi(S)].$$

所有智能体的目的是一致的，即改进自己的策略网络参数 θ^i ，使得目标函数 J 增大。那么策略学习可以写作这样的优化问题：

$$\max_{\theta^1, \dots, \theta^m} J(\theta^1, \dots, \theta^m). \quad (14.1)$$

注意，只有“完全合作关系”这种设定下，所有智能体才会有共同的目标函数，其原因在于 $R^1 = \dots = R^m$ 。对于其它设定——“竞争关系”、“混合关系”、“利己主义”——智能体的目标函数是各不相同的（见下一章）。

合作关系设定下的策略学习的原理很简单，即让智能体各自做策略梯度上升，使得目标函数 J 增长。

$$\text{第 1 号智能体执行: } \theta^1 \leftarrow \theta^1 + \alpha^1 \cdot \nabla_{\theta^1} J(\theta^1, \dots, \theta^m),$$

$$\text{第 2 号智能体执行: } \theta^2 \leftarrow \theta^2 + \alpha^2 \cdot \nabla_{\theta^2} J(\theta^1, \dots, \theta^m),$$

$$\vdots \qquad \vdots$$

$$\text{第 } m \text{ 号智能体执行: } \theta^m \leftarrow \theta^m + \alpha^m \cdot \nabla_{\theta^m} J(\theta^1, \dots, \theta^m).$$

公式中的 $\alpha^1, \alpha^2, \dots, \alpha^m$ 是学习率。判断策略学习收敛的标准是目标函数 $J(\theta^1, \dots, \theta^m)$ 不再增长。在实践中，当平均回报不再增长，即可终止算法。由于无法直接计算策略梯度 $\nabla_{\theta^i} J$ ，我们需要对其做近似。下一节用价值网络近似策略梯度，从而推导出一种实际可行的策略梯度方法。

14.2 合作设定下的多智能体 A2C

第 8 章介绍过 Advantage Actor-Critic (A2C) 方法。本节介绍“完全合作关系”设定下的多智能体 A2C 方法 (Multi-Agent Cooperative A2C)，缩写 MAC-A2C。注意，本节介绍的方法仅适用于“完全合作关系”，也就是要求所有智能体有相同的奖励： $R^1 = \dots = R^m$ 。第 14.2.1 小节定义策略网络和价值网络。第 14.2.2 小节描述 MAC-A2C 训练和决策。第 14.2.3 小节讨论 MAC-A2C 实现中的难点。

《深度强化学习》2021-03-18 尚未完成校对，仅供参考。
如发现错误，请告知作者 shusen.wang@stevens.edu

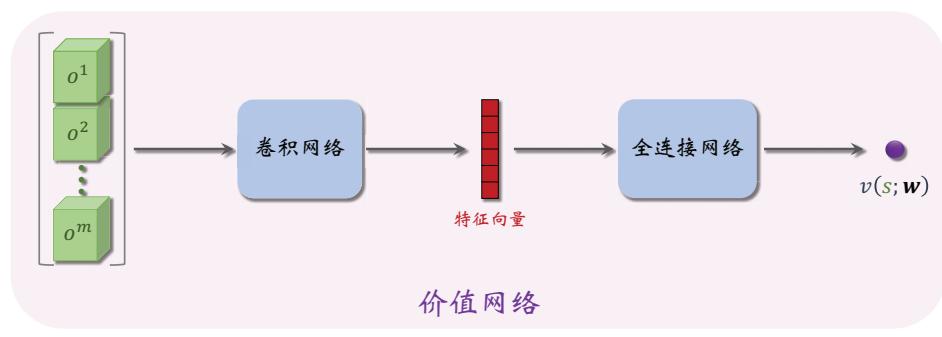


图 14.2: 图中是 MAC-A2C 中的价值网络 $v(s; \mathbf{w})$ 。所有智能体共用这个价值网络。输入是所有智能体的观测： $s = [o^1, \dots, o^m]$ 。输出是价值网络给 s 的评分。

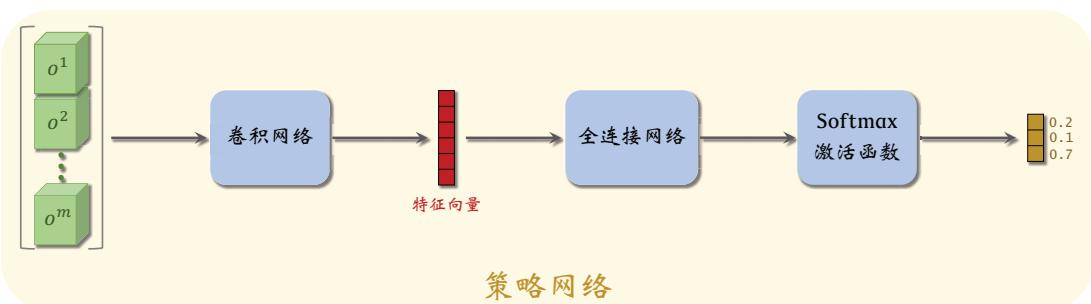


图 14.3: 图中是 MAC-A2C 中第 i 号智能体的策略网络 $\pi(\cdot | s; \theta^i)$ 。所有智能体的策略网络结构都一样，但是参数 $\theta^1, \dots, \theta^m$ 可能不一样。输入是所有智能体的观测： $s = [o^1, \dots, o^m]$ 。输出是在离散动作空间 \mathcal{A}^i 上的概率分布。

14.2.1 策略网络和价值网络

本章只考虑离散控制问题，即动作空间 $\mathcal{A}^1, \dots, \mathcal{A}^m$ 都是离散集合。MAC-A2C 使用两类神经网络：价值网络 v 与策略网络 π ；见图 14.2、图 14.3。

所有智能体共用一个价值网络，记作 $v(s; \mathbf{w})$ ，它是对状态价值函数 $V_\pi(s)$ 的近似。它把所有观测 $s = [o^1, \dots, o^m]$ 作为输入，并输出一个实数，作为对状态 s 的评分。

每个智能体有自己的策略网络。把第 i 号策略网络记作 $\pi(a^i | s; \theta^i)$ 。它的输入是所有智能体的观测 $s = [o^1, \dots, o^m]$ 。它的输出是一个向量，表示动作空间 \mathcal{A}^i 上的概率分布。比如，第 i 号智能体的动作空间是 $\mathcal{A}^i = \{\text{左}, \text{右}, \text{上}\}$ ；策略网络的输出是

$$\pi(\text{左} | s; \theta^i) = 0.2, \quad \pi(\text{右} | s; \theta^i) = 0.1, \quad \pi(\text{上} | s; \theta^i) = 0.7.$$

第 i 号智能体依据该概率分布抽样得到动作 a^i 。

MAC-A2C 属于 Actor-Critic 方法：策略网络 $\pi(A^i|S; \theta^i)$ 相当于第 i 个运动员，负责做决策；价值网络 $v(S; w)$ 相当于评委，对运动员团队的整体表现予以评价，反馈给整个团队一个分数。

训练价值网络：我们用 TD 算法训练训练价值网络 $v(s; w)$ 。观测到状态 s_t 、 s_{t+1} 和奖励 r_t ，计算 TD 目标：

$$\hat{y}_t = r_t + \gamma \cdot v(s_{t+1}; w).$$

把 \hat{y}_t 视作常数，更新 w 使得 $v(s_t; w)$ 接近 \hat{y}_t 。定义损失函数：

$$L(w) = \frac{1}{2} [v(s_t; w) - \hat{y}_t]^2.$$

损失函数的梯度等于：

$$\nabla_w L(w) = \delta_t \cdot \nabla_w v(s_t; w),$$

其中 $\delta_t = v(s_t; w) - \hat{y}_t$ 是 TD 误差。做一次梯度下降更新 w ：

$$w \leftarrow w - \alpha \cdot \delta_t \cdot \nabla_w v(s_t; w).$$

这样可以减小损失函数，也就是让 $v(s_t; w)$ 接近 \hat{y}_t 。上述 TD 算法与单智能体 A2C 的 TD 算法完全一样。

训练策略网络：完全合作关系设定下的动作价值函数记作 $Q_\pi(s, a)$ ，第 i 号智能体的策略网络为 $\pi(A^i | S; \theta^i)$ 。不难证明下面的策略梯度定理（见习题 1）：

定理 14.1. 合作关系 MARL 的策略梯度定理

设基线 b 为不依赖于 $A = [A^1, \dots, A^m]$ 的函数。那么有

$$\nabla_{\theta^i} J(\theta^1, \dots, \theta^m) = \mathbb{E}_{S,A} \left[(Q_\pi(S, A) - b) \cdot \nabla_{\theta^i} \ln \pi(A^i | S; \theta^i) \right].$$

期望中的动作 A 的概率质量函数为

$$\pi(A | S; \theta^1, \dots, \theta^m) \triangleq \pi(A^1 | S; \theta^1) \times \dots \times \pi(A^m | S; \theta^m).$$



把基线设置为状态价值： $b = V_\pi(s)$ 。定义

$$g^i(s, a; \theta^i) \triangleq (Q_\pi(s, a) - V_\pi(s)) \cdot \nabla_{\theta^i} \ln \pi(a^i | s; \theta^i).$$

定理 14.1 说明 $g^i(s, a^i; \theta^i)$ 是策略梯度的无偏估计，即

$$\nabla_{\theta^i} J(\theta^1, \dots, \theta^m) = \mathbb{E}_{S,A} [g^i(S, A; \theta^i)].$$

因此 $g^i(s, a; \theta^i)$ 可以作为策略梯度的近似。但是我们不知道公式中的 Q_π 、 V_π ，还需要进一步做近似。根据第 8.3 节 A2C 的推导，我们把 $Q_\pi(s_t, a_t)$ 近似成 $r_t + \gamma \cdot v(s_{t+1}; w)$ ，把 $V_\pi(s_t)$ 近似成 $v(s_t; w)$ 。那么近似策略梯度 $g^i(s_t, a_t; \theta^i)$ 可以进一步近似成：

$$\tilde{g}^i(s_t, a_t^i; \theta^i) \triangleq \underbrace{(r_t + \gamma \cdot v(s_{t+1}; w) - v(s_t; w))}_{\text{对 } Q_\pi(s_t, a_t) - V_\pi(s_t) \text{ 的近似}} \cdot \nabla_{\theta^i} \ln \pi(a_t^i | s_t; \theta^i).$$

观测到状态 s_t 、 s_{t+1} 、动作 a_t^i 、奖励 r_t ，这样更新策略网络参数：

$$\boldsymbol{\theta}^i \leftarrow \boldsymbol{\theta}^i + \beta \cdot \tilde{g}^i(s_t, a_t^i; \boldsymbol{\theta}^i).$$

根据 TD 误差 δ_t 的定义，不难看出 $\tilde{g}^i(s_t, a_t^i; \boldsymbol{\theta}^i) = -\delta_t \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(a_t^i | s_t; \boldsymbol{\theta}^i)$ 。因此，上面更新策略网络参数的公式可以写作：

$$\boldsymbol{\theta}^i \leftarrow \boldsymbol{\theta}^i - \beta \cdot \delta_t \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(a_t^i | s_t; \boldsymbol{\theta}^i).$$

14.2.2 训练和决策

训练： 实际实现的时候，应当使用目标网络缓解自举造成的偏差。目标网络记作 $v(s; \mathbf{w}^-)$ ，它的结构与 v 相同，但是参数不同。设当前价值网络和目标网络的参数分别是 \mathbf{w}_{now} 和 $\mathbf{w}_{\text{now}}^-$ 。设当前 m 个策略网络的参数分别是 $\boldsymbol{\theta}_{\text{now}}^1, \dots, \boldsymbol{\theta}_{\text{now}}^m$ 。MAC-A2C 重复下面的步骤更新参数：

1. 观测到当前状态 $s_t = [o_t^1, \dots, o_t^m]$ ，让每一个智能体独立做随机抽样：

$$a_t^i \sim \pi(\cdot | s_t; \boldsymbol{\theta}_{\text{now}}^i), \quad \forall i = 1, \dots, m,$$

并执行选中的动作。

2. 从环境中观测到奖励 r_t 与下一时刻状态 $s_{t+1} = [o_{t+1}^1, \dots, o_{t+1}^m]$ 。
3. 让价值网络做预测： $\hat{v}_t = v(s_t; \mathbf{w}_{\text{now}})$ 。
4. 让目标网络做预测： $\hat{v}_{t+1}^- = v(s_{t+1}; \mathbf{w}_{\text{now}}^-)$ 。
5. 计算 TD 目标与 TD 误差：

$$\hat{y}_t^- = r_t + \gamma \cdot \hat{v}_{t+1}^-, \quad \delta_t = \hat{v}_t - \hat{y}_t^-.$$

6. 更新价值网络参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} v(s_t; \mathbf{w}_{\text{now}}).$$

7. 更新目标网络参数：

$$\mathbf{w}_{\text{new}}^- \leftarrow \tau \cdot \mathbf{w}_{\text{new}} + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^-.$$

8. 更新策略网络参数：

$$\boldsymbol{\theta}_{\text{new}}^i \leftarrow \boldsymbol{\theta}_{\text{now}}^i - \beta \cdot \delta_t \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(a_t^i | s_t; \boldsymbol{\theta}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

MAC-A2C 属于同策略 (On-policy)，不能使用经验回放。

决策： 在完成训练之后，不再需要价值网络 $v(s; \mathbf{w})$ 。每个智能体可以用它自己的策略网络做决策。在时刻 t 观测到全局状态 $s_t = [o_t^1, \dots, o_t^m]$ ，然后做随机抽样得到动作：

$$a_t^i \sim \pi(\cdot | s_t; \boldsymbol{\theta}^i),$$

并执行动作。注意，智能体并不能独立做决策，因为一个智能体的策略网络需要知道其他所有智能体的观测。

14.2.3 实现中的难点

上述 MAC-A2C 的训练和决策貌似简单，然而实现起来却不容易。在 MARL 的常见设定下，第 i 号智能体只知道 o^i ，而观测不到全局状态：

$$s = [o^1, \dots, o^m].$$

这会给决策和训练造成如下的困难：

- 每个智能体有自己的策略网络 $\pi(a^i|s; \theta^i)$ ，可以依靠它做决策。但是它的决策需要全局状态 s 。
- 在训练的过程中，价值网络 $v(s; w)$ 需要知道全局状态 s 才能计算 TD 误差 δ 与梯度 $\nabla_w v(s; w)$ 。
- 在训练的过程中，每个策略网络都需要知道全局状态 s 来计算梯度 $\nabla_{\theta^i} \ln \pi(a^i|s; \theta^i)$ 。

综上所述，如果智能体之间不交换信息，那么智能体既无法做训练，也无法做决策。想要做训练和决策，有两种可行的途径：

- 一种办法是让智能体共享观测。这需要做通信，每个智能体把自己的 o^i 传输给其他智能体。这样每个智能体都有全局的状态 $s = [o^1, \dots, o^m]$ 。
- 另一种办法是对策略网络和价值函数做近似。通常使用 $\pi(a^i|o^i; \theta^i)$ 替代 $\pi(a^i|s; \theta^i)$ 。甚至可以进一步用 $v(o^i; w^i)$ 代替 $v(s; w)$ 。

共享观测的缺点在于通信会让训练和决策的速度变慢。而做近似的缺点在于不完全信息造成训练不收敛、做出错误决策。我们不得不在两种办法之间做出取舍，承受其造成的不良影响。

下一节介绍中心化 (Centralized) 与去中心化 (Decentralized) 的实现方法。中心化让智能体共享信息；优点是训练和决策的效果好，缺点是需要通信，造成延时，影响速度。去中心化需要做近似，避免通信；其优点在于速度快，而缺点则是影响训练和决策的质量。

14.3 三种架构

本节介绍 MAC-A2C 的三种实现方法。第 14.3.1 节介绍 “**中心化训练 + 中心化决策**” (Centralized Training with Centralized Execution)，它是对 MAC-A2C 的忠实实现，训练和决策都需要通信。第 14.3.2 节介绍 “**去中心化训练 + 去中心化决策**” (Decentralized Training with Decentralized Execution)，它对策略网络和价值网络都做近似，以避免训练和决策的通信。第 14.3.3 节介绍 “**中心化训练 + 去中心化决策**” (Centralized Training with Decentralized Execution)，它只近似策略网络以避免决策的通信，它的训练需要通信。

图 14.4 对比了三种架构的策略网络和价值网络。用“完全中心化”作出的决策最好，但是速度最慢，在很多问题中不适用。“中心化训练 + 去中心化决策”虽然在训练中需要通信，但是决策的时候不需要通信，可以做到实时决策。“中心化训练 + 中心化决策”是三种架构中最实用的。

	价值网络	策略网络	训练	决策
中心化训练 + 中心化决策	$v(s; \mathbf{w})$	$\pi(a^i s; \boldsymbol{\theta}^i)$	需要通信	需要通信
去中心化训练 + 去中心化决策	$v(o^i; \mathbf{w}^i)$	$\pi(a^i o^i; \boldsymbol{\theta}^i)$	无需通信	无需通信
中心化训练 + 去中心化决策	$v(s; \mathbf{w})$	$\pi(a^i o^i; \boldsymbol{\theta}^i)$	需要通信	无需通信

图 14.4：三种架构的对比。

14.3.1 中心化训练 + 中心化决策

本节用完全中心化 (Fully Centralized) 的方式实现 MAC-A2C，没有做任何近似。这种实现的缺点在于通信造成延时，使得训练和决策速度变慢。图 14.5 描述了系统的架构。最上面是中央控制器 (Central Controller)，里面部署了价值网络 $v(s; \mathbf{w})$ 与所有 m 个策略网络

$$\pi(a^1 | \boldsymbol{\theta}^1), \quad \pi(a^2 | \boldsymbol{\theta}^2), \quad \dots, \quad \pi(a^m | \boldsymbol{\theta}^m).$$

训练和决策全部由中央控制器完成。智能体负责与环境交互，执行中央控制器的决策 a^i ，并把观测到的 o^i 汇报给中央控制器。如果智能体观测到奖励 r^i ，也发给中央控制器。

中心化训练： 在时刻 t 和 $t+1$ ，中央控制器收集到所有智能体的观测值

$$s_t = [o_t^1, \dots, o_t^m] \quad \text{和} \quad s_{t+1} = [o_{t+1}^1, \dots, o_{t+1}^m].$$

在“完全合作关系”的设定下，所有智能体有相同的奖励：

$$r_t^1 = r_t^2 = \dots = r_t^m \triangleq r_t.$$

r_t 可以是中央控制器直接从环境中观测到的，也可能是所有智能体本地的奖励 \tilde{r}_t^i 的加和：

$$r_t = \tilde{r}_t^1 + \tilde{r}_t^2 + \dots + \tilde{r}_t^m.$$

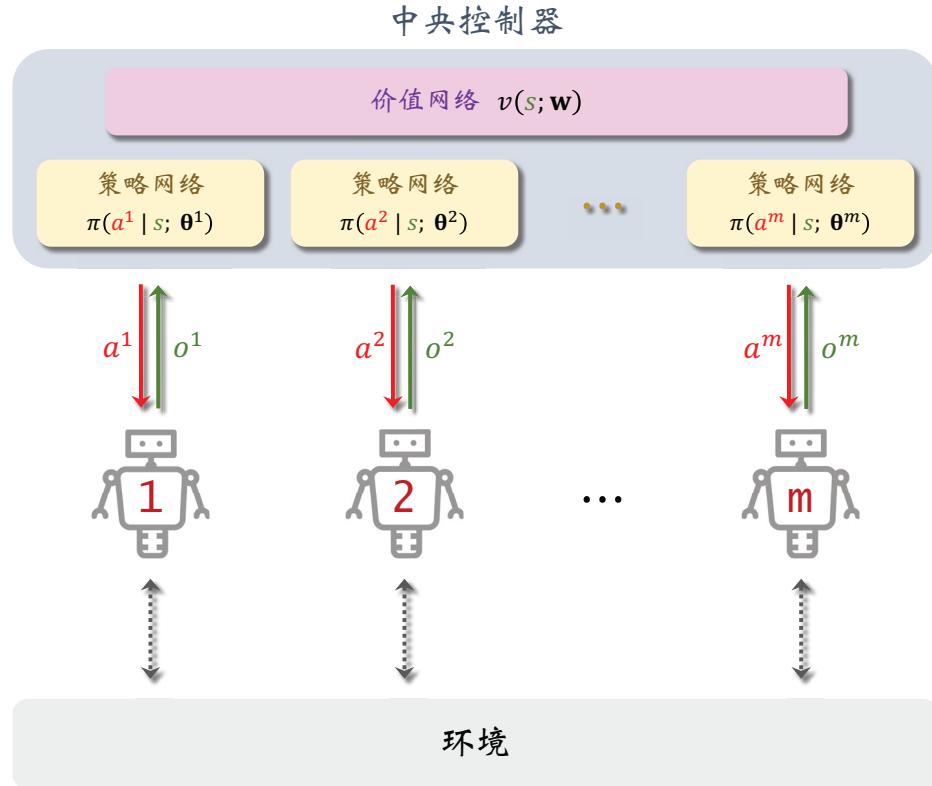


图 14.5: 中心化训练 + 中心化决策的系统架构。

决策是中央控制器上的策略网络做出的，中央控制器因此知道所有的动作：

$$a_t = [a_t^1, \dots, a_t^m].$$

综上所述，中央控制器知道如下信息：

$$s_t, s_{t+1}, a_t, r_t.$$

因此，中央控制器有足够的信息按照第 14.2.2 小节中的算法训练 MAC-A2C，更新价值网络的参数 w 和策略网络的参数 $\theta^1, \dots, \theta^m$ 。

中心化决策：在 t 时刻，中央控制器收集到所有智能体的观测值 $s_t = [o_t^1, \dots, o_t^m]$ ，然后用中央控制器上部署的策略网络做决策：

$$a_t^i \sim \pi(\cdot | s_t; \theta^i), \quad \forall i = 1, \dots, m.$$

中央控制器把决策 a_t^i 传达给第 i 号智能体，该智能体执行 a_t^i 。综上所述，智能体只需要执行中央下达的决策，而不需要自己“思考”。其原因在于策略函数 π 需要全局的状态 s_t 作为输入，而单个智能体不知道全局状态，没有能力单独做决策。

优缺点：中心化训练 + 中心化决策的优点在于完全按照 MAC-A2C 的算法实现，没有做任何改动，因此可以确保正确性。基于全局的观测 $s_t = [o_t^1, \dots, o_t^m]$ 做中心化的决策，利用完整的信息，因此作出的决策可以更好。中心化训练和决策的缺点在于延迟 (Latency) 很大，影响训练和决策的速度。在中心化执行的框架下，智能体与中央控制器要做通信。第 i 号智能体要把 o_t^i 传输给中央控制器，而控制器要在收集到所有观测 $[o_t^1, \dots, o_t^m]$ 之

后才会做决策，做出的决策 a_t^i 还得传输给第 i 号智能体。这个过程通常比较慢，使得实时决策不可能做到。机器人、无人车、无人机等应用都需要实时决策，比如在几十毫秒内做出决策；如果出现几百毫秒、甚至几秒的延迟，可能会造成灾难性的后果。

14.3.2 去中心化训练 + 去中心化决策

上一小节的“中心化训练 + 中心化决策”严格按照 MAC-A2C 的算法实现，其缺点在于训练和决策都需要智能体与中央控制器之间通信，造成训练的决策的速度慢。想要避免通信代价，就不得不对策略网络和价值网络做近似。MAC-A2C 中的策略网络

$$\pi(a^1 | s; \theta^1), \quad \pi(a^2 | s; \theta^2), \quad \dots, \quad \pi(a^m | s; \theta^m),$$

和价值网络 $v(s; w)$ 都需要全局的观测 $s = [o^1, \dots, o^m]$ 。“去中心化训练 + 去中心化决策”的基本思想是用局部观测 o^i 代替 s ，把策略网络和价值网络近似成为：

$$\pi(a^i | o^i; \theta^i) \quad \text{和} \quad v(o^i; w^i).$$

在每个智能体上部署一个策略网络和一个价值网络，它们的参数记作 θ^i 和 w^i 。智能体之间不共享参数，即 $\theta^i \neq \theta^j$, $w^i \neq w^j$ 。这样一来，训练就可以在智能体本地完成，无需中央控制器的参与，无需任何通信。见图 14.5 中的系统架构。

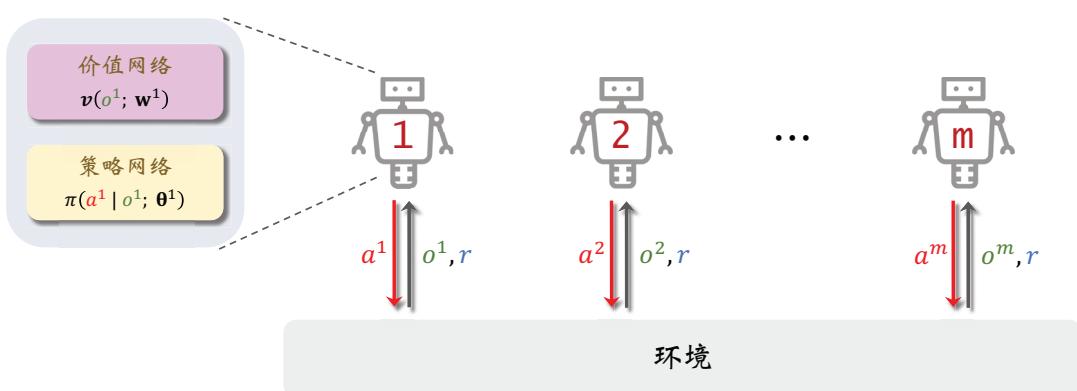


图 14.6：去中心化训练 + 去中心化决策的系统架构。这种方法也叫做 Independent Actor-Critic。

去中心化训练：假设所有智能体的奖励都是相同的，而且每个智能体都能观测到奖励 r 。每个智能体独立做训练，智能体之间不做通信，不共享观测、动作、参数。这样一来，MAC-A2C 就变成了标准的 A2C，每个智能体独立学习自己的参数 θ^i 与 w^i 。

实际实现的时候，每个智能体还需要一个目标网络，记作 $v(s; w^{i-})$ ，它的结构与 $v(s; w^i)$ 相同，但是参数不同。设第 i 号智能体的策略网络、价值网络、目标网络当前参数分别为 θ_{now}^i 、 w_{now}^i 、 w_{now}^{i-} 。该智能体重复以下步骤更新参数：

1. 在 t 时刻，智能体 i 观测到 o_t^i ，然后做随机抽样 $a_t^i \sim \pi(\cdot | o_t^i; \theta^i)$ ，并执行选中的动作 a_t^i 。
2. 环境反馈给智能体奖励 r_t 与新的观测 o_{t+1}^i 。
3. 让价值网络做预测： $\hat{v}_t^i = v(o_t^i; w_{\text{now}}^i)$ 。
4. 让目标网络做预测： $\hat{v}_{t+1}^i = v(o_{t+1}^i; w_{\text{now}}^{i-})$ 。

5. 计算 TD 目标与 TD 误差:

$$\hat{y}_t^i = r_t + \gamma \cdot \hat{v}_{t+1}^i, \quad \delta_t^i = \hat{v}_t^i - \hat{y}_t^i.$$

6. 更新价值网络参数:

$$\mathbf{w}_{\text{new}}^i \leftarrow \mathbf{w}_{\text{now}}^i - \alpha \cdot \delta_t^i \cdot \nabla_{\mathbf{w}^i} v(o_t^i; \mathbf{w}_{\text{now}}^i).$$

7. 更新目标网络参数:

$$\mathbf{w}_{\text{new}}^{i-} \leftarrow \tau \cdot \mathbf{w}_{\text{new}}^i + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^{i-}.$$

8. 更新策略网络参数:

$$\theta_{\text{new}}^i \leftarrow \theta_{\text{now}}^i - \beta \cdot \delta_t^i \cdot \nabla_{\theta^i} \ln \pi(a_t^i | o_t^i; \theta_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

注 上述算法不是 MAC-A2C，而是单智能体的 A2C。去中心化训练的本质就是单智能体强化学习 (SARL)，而非多智能体强化学习 (MARL)。在 MARL 中，智能体之间会相互影响，而本节中的“去中心化训练”把智能体视为独立个体，忽视它们之间的关联，直接用 SARL 方法独立训练每个智能体。用上述 SARL 的方法解决 MARL 问题，在实践中效果往往不佳。

去中心化决策： 在完成训练之后，智能体 i 不再需要其价值网络 $v(o^i; \mathbf{w}^i)$ 。智能体只需要用其本地部署的策略网络 $\pi(a^i | o^i; \theta^i)$ 做决策即可，决策过程无需通信。去中心化执行的速度很快，可以做到实时决策。

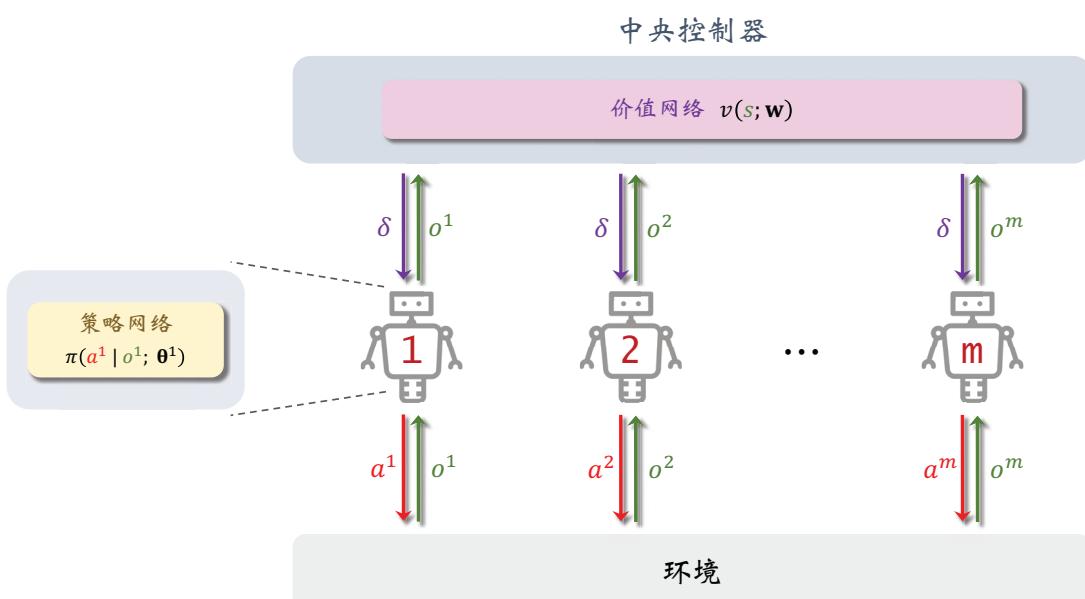


图 14.7: 中心化训练的系统架构。价值网络（以及没画出的目标网络）部署到中央控制器上，策略网络部署到每个智能体上。训练的时候，智能体 i 将观测 \mathbf{o}^i 传输到控制器上，控制器将 TD 误差 δ 传回智能体。

14.3.3 中心化训练 + 去中心化决策

前面两节讨论了完全中心化与完全去中心化，两种实现各有优缺点。当前更流行的 MARL 架构是“中心化训练 + 去中心化决策”。训练的时候使用中央控制器，辅助智能体做训练；见图 14.7。训练结束之后，不再需要中央控制器，每个智能体独立根据本地观测 o^i 做决策；见图 14.8。

本小节与“完全中心化”使用相同的价值网络 $v(s; \mathbf{w})$ 及其目标网络 $v(s; \mathbf{w}^-)$ ；本节与“完全去中心化”使用相同的策略网络：

$$\pi(a^1 | o^1; \boldsymbol{\theta}^1), \dots, \pi(a^m | o^m; \boldsymbol{\theta}^m).$$

第 i 号策略网络的输入是局部观测 o^i ，因此可以将其部署到第 i 号智能体上。价值网络 $v(s; \mathbf{w})$ 的输入是全局状态 $s = [o^1, \dots, o^m]$ ，因此需要将其部署到中央控制器上。

中心化训练：训练的过程需要所有 m 个智能体共同参与，共同改进策略网络参数 $\boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m$ 与价值网络参数 \mathbf{w} 。设当前 m 个策略网络的参数为 $\boldsymbol{\theta}_{\text{now}}^1, \dots, \boldsymbol{\theta}_{\text{now}}^m$ 。设当前价值网络和目标网络的参数分别是 \mathbf{w}_{now} 和 $\mathbf{w}_{\text{now}}^-$ 。训练的流程如下：

1. 每个智能体 i 与环境交互，获取当前观测 o_t^i ，独立做随机抽样：

$$a_t^i \sim \pi(\cdot | o_t^i; \boldsymbol{\theta}_{\text{now}}^i), \quad \forall i = 1, \dots, m, \quad (14.2)$$

并执行选中的动作。

2. 下一时刻，每个智能体 i 都观测到 o_{t+1}^i 。假设中央控制器可以从环境获取奖励 r_t ，或者向智能体询问奖励 r_t 。
3. 每个智能体 i 向中央控制器传输观测 o_t^i 和 o_{t+1}^i ；中央控制器得到状态

$$s_t = [o_t^1, \dots, o_t^m] \quad \text{和} \quad s_{t+1} = [o_{t+1}^1, \dots, o_{t+1}^m].$$

4. 中央控制器让价值网络做预测： $\hat{v}_t = v(s_t; \mathbf{w}_{\text{now}})$ 。
5. 中央控制器让目标网络做预测： $\hat{v}_{t+1}^- = v(s_{t+1}; \mathbf{w}_{\text{now}}^-)$ 。
6. 中央控制器计算 TD 目标和 TD 误差：

$$\bar{y}_t = r_t + \gamma \cdot \hat{v}_{t+1}^-, \quad \delta_t = \hat{v}_t - \bar{y}_t,$$

并将 δ_t 广播到所有智能体。

7. 中央控制器更新价值网络参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} v(s_t; \mathbf{w}_{\text{now}}).$$

8. 中央控制器更新目标网络参数：

$$\mathbf{w}_{\text{new}}^- \leftarrow \tau \cdot \mathbf{w}_{\text{new}} + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^-.$$

9. 每个智能体 i 更新策略网络参数：

$$\boldsymbol{\theta}_{\text{new}}^i \leftarrow \boldsymbol{\theta}_{\text{now}}^i - \beta \cdot \delta_t \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(a_t^i | o_t^i; \boldsymbol{\theta}_{\text{now}}^i).$$

注 此处的算法并不等价于第 14.2 节的 MAC-A2C。区别在于此处用 $\pi(a^i | o^i; \boldsymbol{\theta}^i)$ 代替 MAC-A2C 中的 $\pi(a^i | s; \boldsymbol{\theta}^i)$ 。

去中心化决策：在完成训练之后，不再需要价值网络 $v(s; \mathbf{w})$ 。智能体只需要用其

本地部署的策略网络 $\pi(a^i | o^i; \theta^i)$ 做决策，决策过程无需通信。去中心化执行的速度很快，可以做到实时决策。

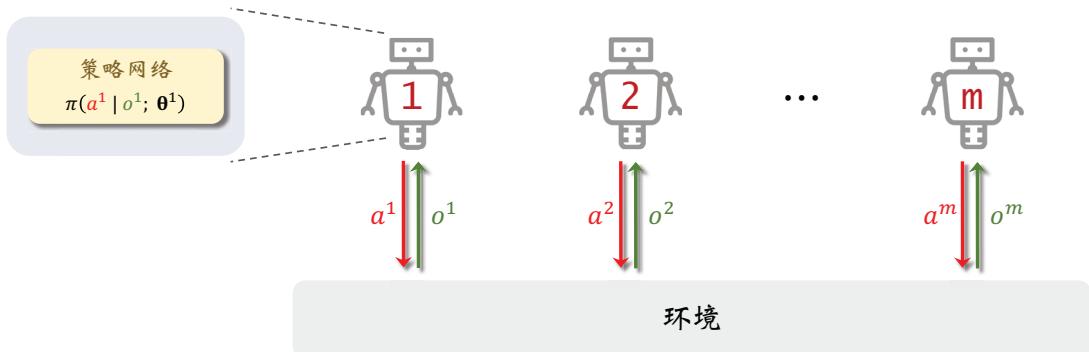


图 14.8: 去中心化决策的系统架构。在完成训练之后，智能体不再做通信，智能体用本地部署的策略网络做决策。

 第十四章 习题 

1. 设动作 $A = [A^1, \dots, A^m]$ 的概率质量函数为

$$\pi(A | S; \boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m) \triangleq \pi(A^1 | S; \boldsymbol{\theta}^1) \times \dots \times \pi(A^m | S; \boldsymbol{\theta}^m).$$

由第 8 章中带基线的策略梯度定理可得：

$$\nabla_{\boldsymbol{\theta}^i} J(\boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m) = \mathbb{E}_{S,A} \left[\left(Q_\pi(S, A) - b \right) \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(A | S; \boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m) \right].$$

公式中动作 A 的概率质量函数为 $\pi(A | S; \boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m)$ ，公式中的 b 是任意不依赖于 A 的函数。请用上面两个公式证明下面的公式：

$$\nabla_{\boldsymbol{\theta}^i} J(\boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m) = \mathbb{E}_{S,A} \left[\left(Q_\pi(S, A) - V_\pi(S) \right) \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(A^i | S; \boldsymbol{\theta}^i) \right].$$

∽ 第十四章 相关文献 ∽

完全去中心化的架构早在 1993 年就被提出 [101]，在 2017 年被用在多智能体 DQN 上 [38, 100]。中心化训练 + 去中心化执行 (Centralized Training with Decentralized Execution) 在近年来很流行 [77, 42, 37, 65, 52]。

MAC-A2C 是本书设计出来的简单方法，用于讲解 MARL 的三种架构；MAC-A2C 这个名字并没有出现在任何文献中。MAC-A2C 本质是带基线的 Actor-Critic，其中的基线是状态价值

$$V_\pi(s) \triangleq \mathbb{E}_A [Q_\pi(s, A)],$$

期望是关于动作 $A = [A^1, \dots, A^m]$ 求的。可以把基线换成

$$Q_\pi^{-i}(s, a^{-i}) \triangleq \mathbb{E}_{A^i} [Q_\pi(s, A^i, a^{-i})],$$

公式中 $a^{-i} = [a^1, \dots, a^{i-1}, a^{i+1}, \dots, a^m]$ 。公式中的期望是关于第 i 号智能体的动作 $A^i \sim \pi(\cdot | o^i, \theta^i)$ 求的。用 $Q_\pi^{-i}(s, a^{-i})$ 作为基线，代替 $V_\pi(s)$ ，得到的方法叫做 **C**ounterfactual **M**ulti-**A**gent，缩写 **COMA** [37]。此外，COMA 还在策略网络中使用 RNN；其原理见第 11 章的解释。COMA 的表现略好于 MAC-A2C，但是 COMA 的实现很复杂，不建议读者自己实现。

第十五章 非合作关系设定下的多智能体强化学习

上一章研究了多智能体强化学习 (MARL) 中最简单的设定——完全合作关系，在这种设定下，所有的智能体有相同的奖励、回报、价值、目标函数。本章研究非合作关系，那么不同智能体各自有不同的奖励、回报、价值、目标函数。本章中采用的符号如图 15.1 所示。

第 15.1 节定义非合作关系设定下的策略学习、策略梯度方法、以及收敛判别。第 15.2 节推导非合作关系下的 A2C 方法，本书称之为 Multi-Agent Noncooperative A2C，缩写 MAN-A2C，可以用于离散控制问题。第 15.3 节用三种架构实现 MAN-A2C：完全去中心化、完全中心化、中心化训练 + 去中心化决策。第 15.4 介绍多智能体确定策略梯度方法，缩写 MADDPG，可以用于连续控制问题。

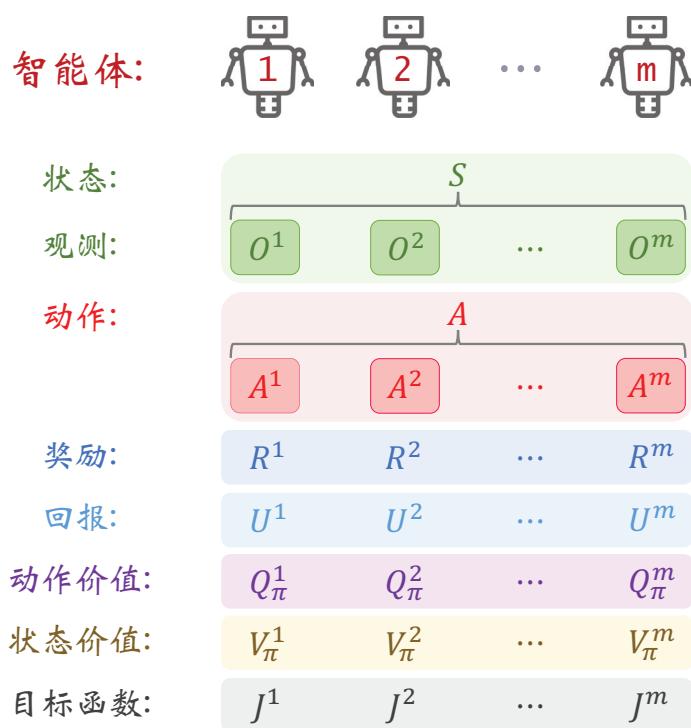


图 15.1: 多智能体强化学习 (MARL) 的符号。

15.1 非合作关系设定下的策略学习

上一章研究合作关系的 MARL，即所有智能体的奖励都相等： $R^1 = \dots = R^m$ 。在这种设定下，所有智能体有相同的状态价值函数 $V_\pi(s)$ 和目标函数

$$J(\theta^1, \dots, \theta^m) = \mathbb{E}_S[V_\pi(s)].$$

目标函数可以衡量策略网络参数 $\theta^1, \dots, \theta^m$ 的好坏。策略学习的目的是改进 $\theta^1, \dots, \theta^m$ 使得 J 变大。合作关系的设定下，策略学习的收敛标准很明确：如果找不到更好的 $\theta^1, \dots, \theta^m$ 使得 J 变大，那么当前的 $\theta^1, \dots, \theta^m$ 就是最优解。

非合作关系设定下的目标函数：如果是非合作关系，那么不存在这样的关系： $R^1 = \dots = R^m$ 。两个智能体的奖励不相等 ($R^i \neq R^j$)，那么它们的回报也不相等 ($U^i \neq U^j$)，回报的期望（价值函数）也不相等。把状态价值记作：

$$V^1(s), V^2(s), \dots, V^m(s).$$

第 i 个智能体的目标函数是状态价值的期望：

$$J^i(\theta^1, \dots, \theta^m) = \mathbb{E}_S[V_\pi^i(s)].$$

J^i 的意义是回报 U^i 的期望，所以能反映出第 i 个智能体的表现好坏。

注 目标函数 J^1, J^2, \dots, J^m 是各不相同的，也就是说智能体没有共同的目标（除非是完全合作关系）。举个例子，在 Predator-Prey（捕食者—猎物）的游戏中，捕食者的目标函数 J^1 与猎物的目标函数 J^2 负相关： $J^1 = -J^2$ 。

注 第 i 个智能体的目标函数 J^i 依赖于所有智能体的策略网络参数 $\theta^1, \dots, \theta^m$ 。为什么一个智能体的目标函数依赖于其他智能体的策略呢？举个例子，捕食者改进自己的策略 θ^1 ，而猎物没有改变策略 θ^2 。虽然猎物的策略 θ^2 没有变化，但是它的目标函数 J^2 会减小。

非合作关系设定下的策略学习：在多智能体的策略学习中，第 i 个智能体的目标是改进自己的策略参数 θ^i ，使得 J^i 尽量大。多智能体的策略学习可以描述为这样的问题：

$$\text{第 1 个智能体求解} : \max_{\theta^1} J^1(\theta^1, \dots, \theta^m),$$

$$\text{第 2 个智能体求解} : \max_{\theta^2} J^2(\theta^1, \dots, \theta^m),$$

$$\vdots \quad \vdots$$

$$\text{第 } m \text{ 个智能体求解} : \max_{\theta^m} J^m(\theta^1, \dots, \theta^m).$$

注意，目标函数 J^1, J^2, \dots, J^m 是各不相同的，也就是说智能体没有共同的目标（除非是完全合作关系）。策略学习的基本思想是让每个智能体各自做策略梯度上升：

$$\text{第 1 号智能体执行} : \theta^1 \leftarrow \theta^1 + \alpha^1 \cdot \nabla_{\theta^1} J^1(\theta^1, \dots, \theta^m),$$

$$\text{第 2 号智能体执行} : \theta^2 \leftarrow \theta^2 + \alpha^2 \cdot \nabla_{\theta^2} J^2(\theta^1, \dots, \theta^m),$$

$$\vdots \quad \vdots$$

$$\text{第 } m \text{ 号智能体执行} : \theta^m \leftarrow \theta^m + \alpha^m \cdot \nabla_{\theta^m} J^m(\theta^1, \dots, \theta^m).$$

公式中的 $\alpha^1, \alpha^2, \dots, \alpha^m$ 是学习率。由于无法直接计算策略梯度 $\nabla_{\theta^i} J^i$ ，我们需要对其做

近似。各种策略学习方法的区别就在于如何对策略梯度做近似。

收敛的判别：在合作关系设定下，所有智能体有相同的目标函数 ($J^1 = \dots = J^m$)，那么判断收敛的标准就是目标函数值不再增长。也就是说改变任何智能体的策略都无法让团队的回报增长。

在非合作关系设定下，智能体的利益是不一致的、甚至是冲突的，智能体各有各的目标函数。该如何判断策略学习的收敛呢？不能用 $J^1 + J^2 + \dots + J^m$ 作为判断收敛的标准。比如在 Predator-Prey（捕食者—猎物）的游戏中，双方的目标函数是冲突的： $J^1 = -J^2$ 。如果捕食者改进策略，那么 J^1 会增长，而 J^2 会下降。自始至终， $J^1 + J^2$ 一直等于零，不论策略学习有没有收敛。

在非合作关系设定下，收敛标准是纳什均衡。一个智能体在制定策略的时候，要考虑到其他各方的策略。在纳什均衡的情况下，每一个智能体都在以最优的方式应对其他各方的策略。在纳什均衡的情况下，谁也没有动机去单独改变自己的策略，因为改变策略不会增加自己的收益。这样就达到了一种平衡状态，所有智能体都找不到更好的策略。这种平衡状态就被认为是收敛。在实验中，如果所有智能体的平均回报都不再变化，就可以认为达到了纳什均衡。

定义 15.1. 纳什均衡

在多智能体系统中，当其余所有智能体都不改变策略的情况下，一个智能体 i 单独改变策略 θ^i ，无法让其期望回报 $J^i(\theta^1, \dots, \theta^m)$ 变大。



评价策略的优劣：有两种策略学习的方法 M_+ 和 M_- ，把它们训练出的策略网络参数分别记作 $\theta_+^1, \dots, \theta_+^m$ 和 $\theta_-^1, \dots, \theta_-^m$ 。该如何评价 M_+ 和 M_- 的优劣呢？在合作关系设定下，很容易评价两种方法的好坏。在收敛之后，把两种策略的平均回报记作 J_+ 和 J_- 。如果 $J_+ > J_-$ ，就说明 M_+ 比 M_- 好；反之亦然。

在非合作关系的设定下，不能直接用平均回报评价策略的优劣。以捕食者—猎物的游戏为例，我们用两种方法 M_+ 和 M_- 训练策略网络，把它们训练出的策略网络记作：

$$\begin{aligned} \pi(a | s, \theta_+^{\text{predator}}), & \quad \pi(a | s, \theta_+^{\text{prey}}), \\ \pi(a | s, \theta_-^{\text{predator}}), & \quad \pi(a | s, \theta_-^{\text{prey}}). \end{aligned}$$

设收敛时的平均回报为：

$$\begin{aligned} J_+^{\text{predator}} &= 0.8, & J_+^{\text{prey}} &= -0.8, \\ J_-^{\text{predator}} &= 0.1, & J_-^{\text{prey}} &= -0.1. \end{aligned}$$

请问 M_+ 和 M_- 孰优孰劣呢？假如我们的目标是学习捕食者 (Predator)，能否说明 M_+ 比 M_- 好呢？答案是否定的。 $J_+^{\text{predator}} > J_-^{\text{predator}}$ 可能是由于方法 M_+ 没有训练好猎物 (Pray) 的策略 θ_+^{prey} ，导致捕食者 (Predator) 相对有优势。 $J_+^{\text{predator}} > J_-^{\text{predator}}$ 不能说明策略 $\theta_+^{\text{predator}}$ 优于 $\theta_-^{\text{predator}}$ 。

在非合作关系的设定下，该如何评价两种方法 M_+ 和 M_- 的优劣呢？以捕食者—

猎物的游戏为例，我们让一种方法训练出的捕食者与另一种方法训练出的猎物对决：

$$\begin{array}{lll} \pi(a | s, \theta_+^{\text{predator}}) & \text{对决} & \pi(a | s, \theta_-^{\text{prey}}), \\ \pi(a | s, \theta_-^{\text{predator}}) & \text{对决} & \pi(a | s, \theta_+^{\text{prey}}). \end{array}$$

记录下两方捕食者的平均回报，记作 J_+^{predator} 、 J_-^{predator} 。两者的大小可以反映出 \mathcal{M}_+ 和 \mathcal{M}_- 的优劣。

15.2 非合作设定下的多智能体 A2C

本节研究“非合作关系”设定下的多智能体 A2C 方法 (Multi-Agent Non-cooperative A2C)，缩写 MAN-A2C。

15.2.1 策略网络和价值网络

MAN-A2C 中，每个智能体有自己的策略网络和价值网络，记作：

$$\pi(a^i | s; \theta^i) \quad \text{和} \quad v(s; w^i).$$

第 i 个策略网络需要把所有智能体的观测 $s = [o^1, \dots, o^m]$ 作为输入，并输出一个概率分布；第 i 个智能体依据该概率分布抽样得到动作 A^i 。两类神经网络的结构与上一章的 MAC-A2C 完全相同。请注意上一章 MAC-A2C 与本章 MAN-A2C 的区别：

- 上一章的 MAC-A2C 用于完全合作关系，所有智能体有相同的状态价值函数 $V_\pi(s)$ ，所以只用一个神经网络近似 $V_\pi(s)$ ，记作 $v(s; w)$ 。
- 本章的 MAN-A2C 用于非合作关系，每个智能体各有一个状态价值函数 $V_\pi^i(s)$ ，所以每个智能体各自对应一个价值网络 $v(s; w^i)$ 。

MAN-A2C 属于 Actor-Critic 方法：策略网络 $\pi(a^i | s; \theta^i)$ 相当于第 i 个运动员，负责做决策；每个运动员都有一个专属的评委 $v(s; w^i)$ ，对运动员 i 的表现予以评价。请注意，虽然评委 $v(s; w^i)$ 是对运动员 i 个人做出评价，但是评委会考虑到全局的状态 $s = [o^1, \dots, o^m]$ 。举个例子，在足球比赛中，评委 i 只对运动员 i 做评价，目的在于改进运动员 i 的技术。在比赛中，想要评价运动员 i 的跑位、传球的好坏，还需要考虑到队友、对手的位置，所以评委 i 会考虑到场上所有球员的表现 $s = [o^1, \dots, o^m]$ 。注意与上一章中 MAC-A2C 的区别：MAC-A2C 中只有一位评委，他会点评整个团队的表现，而不会给每位运动员单独一个评分。

15.2.2 算法推导

在非合作关系设定下，第 i 号智能体的动作价值函数记作 $Q_\pi^i(s, a)$ ，策略网络记作 $\pi(A^i | S; \theta^i)$ 。不难证明下面的策略梯度定理：

定理 15.1. 非合作关系 MARL 的策略梯度定理

设基线 b 为不依赖于 $A = [A^1, \dots, A^m]$ 的函数。那么有

$$\nabla_{\theta^i} J^i(\theta^1, \dots, \theta^m) = \mathbb{E}_{S, A} \left[(Q_\pi^i(S, A) - b) \cdot \nabla_{\theta^i} \ln \pi(A^i | S; \theta^i) \right].$$

期望中的动作 A 的概率质量函数为

$$\pi(A | S; \theta^1, \dots, \theta^m) \triangleq \pi(A^1 | S; \theta^1) \times \dots \times \pi(A^m | S; \theta^m).$$

我们用 $b = V_\pi^i(s)$ 作为定理中的基线，并且用价值网络 $v(s; w^i)$ 近似 $V_\pi^i(s)$ 。按照上

一章的算法推导，我们可以把策略梯度 $\nabla_{\theta^i} J^i(\theta^1, \dots, \theta^m)$ 近似成：

$$\tilde{g}^i(s_t, a_t^i; \theta^i) \triangleq \left(r_t^i + \gamma \cdot v(s_{t+1}; \mathbf{w}^i) - v(s_t; \mathbf{w}^i) \right) \cdot \nabla_{\theta^i} \pi(a_t^i | s_t; \theta^i).$$

观测到状态 s_t 、 s_{t+1} 、动作 a_t^i 、奖励 r_t^i ，这样更新策略网络参数：

$$\theta^i \leftarrow \theta^i + \beta \cdot \tilde{g}^i(s_t, a_t^i; \theta^i).$$

更新价值网络 $v(s; \mathbf{w}^i)$ 的方法与 A2C 基本一样。在观测到状态 s_t 、 s_{t+1} 、奖励 r_t^i 之后，计算 TD 目标：

$$\hat{y}_t^i = r_t^i + \gamma \cdot v(s_{t+1}; \mathbf{w}^i).$$

更新参数 \mathbf{w}^i ，使得 $v(s_t; \mathbf{w}^i)$ 更接近 \hat{y}_t^i 。

15.2.3 训练和决策

训练：实现 MAN-A2C 的时候，应当使用目标网络缓解自举造成的偏差。第 i 号智能体的目标网络记作 $v(s; \mathbf{w}^{i-})$ ，它的结构与 $v(s; \mathbf{w}^i)$ 相同，但是参数不同。设第 i 号智能体策略网络、价值网络、目标网络当前的参数是 θ_{now}^i 、 $\mathbf{w}_{\text{now}}^i$ 、 $\mathbf{w}_{\text{now}}^{i-}$ 。MAN-A2C 重复下面的步骤更新参数：

1. 观测到当前状态 $s_t = [o_t^1, \dots, o_t^m]$ ，让每一个智能体独立做随机抽样：

$$a_t^i \sim \pi(\cdot | s_t; \theta_{\text{now}}^i), \quad \forall i = 1, \dots, m,$$

并执行选中的动作。

2. 从环境中观测到奖励 r_t^1, \dots, r_t^m 与下一时刻状态 $s_{t+1} = [o_{t+1}^1, \dots, o_{t+1}^m]$ 。
3. 让价值网络做预测：

$$\hat{v}_t^i = v(s_t; \mathbf{w}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

4. 让目标网络做预测：

$$\hat{v}_{t+1}^{i-} = v(s_{t+1}; \mathbf{w}_{\text{now}}^{i-}), \quad \forall i = 1, \dots, m.$$

5. 计算 TD 目标与 TD 误差：

$$\hat{y}_t^i = r_t^i + \gamma \cdot \hat{v}_{t+1}^{i-}, \quad \delta_t^i = \hat{v}_t^i - \hat{y}_t^i, \quad \forall i = 1, \dots, m.$$

6. 更新价值网络参数：

$$\mathbf{w}_{\text{new}}^i \leftarrow \mathbf{w}_{\text{now}}^i - \alpha \cdot \delta_t^i \cdot \nabla_{\mathbf{w}^i} v(s_t; \mathbf{w}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

7. 更新目标网络参数：

$$\mathbf{w}_{\text{new}}^{i-} \leftarrow \tau \cdot \mathbf{w}_{\text{new}}^i + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^{i-}, \quad \forall i = 1, \dots, m.$$

8. 更新策略网络参数：

$$\theta_{\text{new}}^i \leftarrow \theta_{\text{now}}^i - \beta \cdot \delta_t^i \cdot \nabla_{\theta^i} \ln \pi(a_t^i | s_t; \theta_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

MAN-A2C 属于同策略 (On-policy)，不能使用经验回放。

决策：在完成训练之后，不再需要价值网络 $v(s; \mathbf{w}^1), \dots, v(s; \mathbf{w}^m)$ 。每个智能体可以用它自己的策略网络做决策。在时刻 t 观测到全局状态 $s_t = [o_t^1, \dots, o_t^m]$ ，然后做随机抽样得到动作：

$$a_t^i \sim \pi(\cdot | s_t; \theta^i),$$

并执行动作 a_t^i 。智能体并不能独立做决策，因为策略网络需要知道所有的观测 $s_t = [o_t^1, \dots, o_t^m]$ 。

15.3 三种架构

本节介绍 MAN-A2C 的三种实现方法：“中心化训练 + 中心化决策”、“去中心化训练 + 去中心化决策”、“中心化训练 + 去中心化决策”。

15.3.1 中心化训练 + 中心化决策

可以用完全中心化 (Fully Centralized) 的方式实现 MAN-A2C 的训练和决策。图 15.2 描述了系统的架构。最上面是中央控制器 (Central Controller)，里面部署了所有 m 个价值网络和策略网络：

$$\begin{aligned} v(s | \mathbf{w}^1), \quad v(s | \mathbf{w}^2), \quad \dots, \quad v(s | \mathbf{w}^m), \\ \pi(a^1 | \theta^1), \quad \pi(a^2 | \theta^2), \quad \dots, \quad \pi(a^m | \theta^m). \end{aligned}$$

训练和决策全部由中央控制器完成。智能体负责与环境交互，执行中央控制器的决策 a^i ，并把观测到的 o^i 和 r^i 汇报给中央控制器。这种中心化的方式严格实现了上一节的算法。

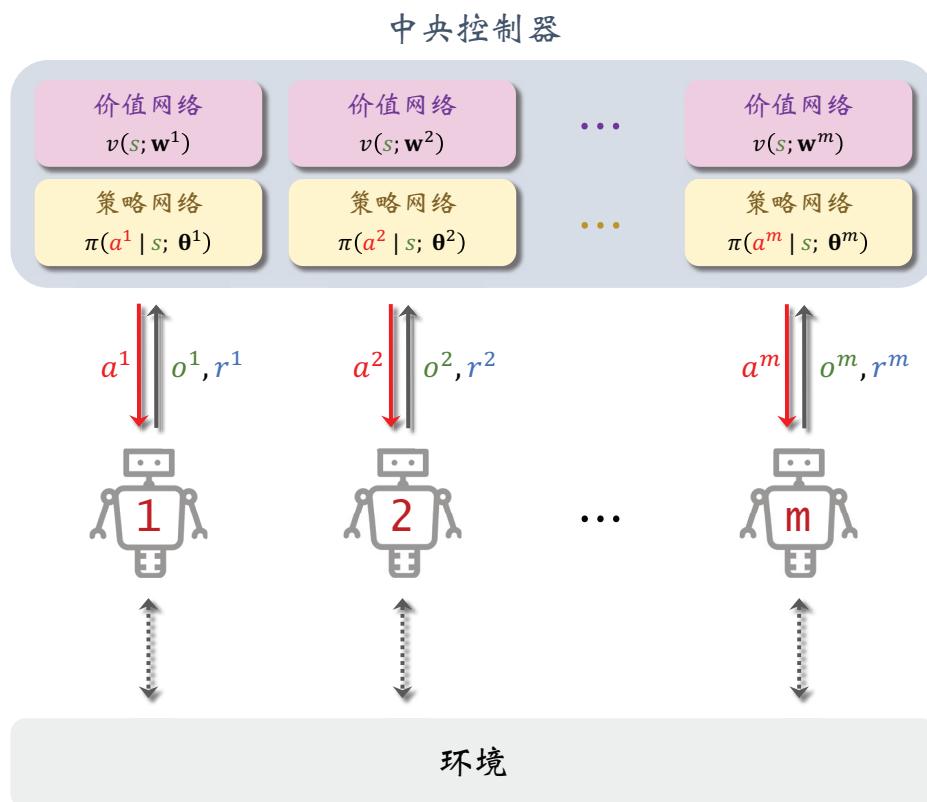


图 15.2：中心化训练 + 中心化决策的系统架构。

在上一章中，我们用完全中心化的方式实现了 MAC-A2C（见图 14.5）。请注意 MAC-A2C 与此处的 MAN-A2C 的区别。第一，MAC-A2C 的中央控制器上只有一个价值网络，而此处 MAN-A2C 则有 m 个价值网络。第二，MAC-A2C 的每一轮只有一个全局的奖励 r ，而 MAN-A2C 的每个智能体都有自己的奖励 r^i 。

15.3.2 去中心化训练 + 去中心化决策

为了避免“完全中心化”中的通信，可以对策略网络和价值网络做近似，做到“完全去中心化”。把 MAN-A2C 中的策略网络和价值网络做近似：

$$\begin{aligned}\pi(a^i | s; \theta^i) &\implies \pi(a^i | o^i; \theta^i), \\ v(s; w^i) &\implies v(o^i; w^i).\end{aligned}$$

图 15.3 描述了“完全去中心化”的系统架构。每个智能体上部署一个策略网络和一个价值网络，它们的参数记作 θ^i 和 w^i ；智能体之间不共享参数。这样一来，训练就可以在智能体本地完成，无需中央控制器的参与，也无需通信。这种实现的本质是单智能体强化学习，而非多智能体强化学习。

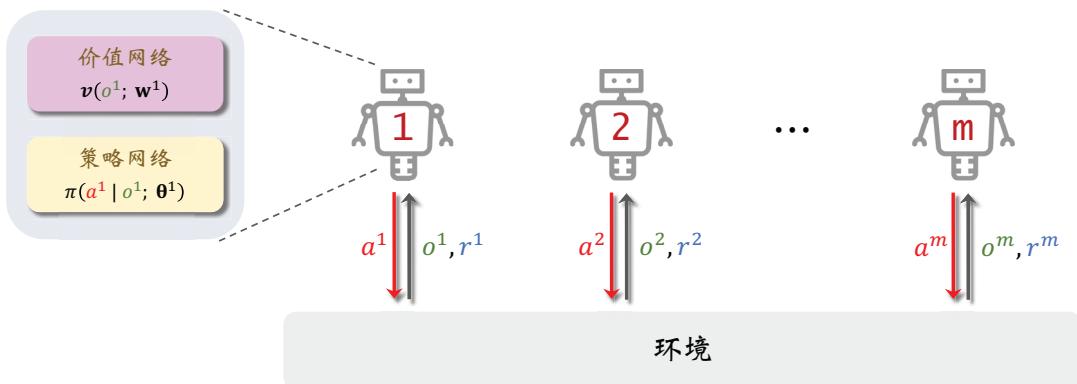


图 15.3：去中心化训练 + 去中心化决策的系统架构。这种方法也叫做 Independent Actor-Critic。

此处的实现与上一章“完全合作关系”设定下的“完全去中心化”几乎完全相同（见图 14.6）。唯一的区别在于此处每个智能体获得的奖励 r^i 是不同的，而上一章“完全合作关系”设定下的奖励是相同的 $r^1 = \dots = r^m = r$.

15.3.3 中心化训练 + 去中心化决策

第三种实现方式是“中心化训练 + 去中心化决策”。与“完全中心化”的 MAN-A2C 相比，唯一的区别在于对策略网络做近似：

$$\pi(a^i | s; \theta^i) \implies \pi(a^i | o^i; \theta^i), \quad \forall i = 1, \dots, m.$$

由于用智能体局部观测 o^i 替换了全局状态 $s = [o^1, \dots, o^m]$ ，策略网络可以部署到每个智能体上。而价值网络仍然是 $v(s; w^i)$ ，没有做近似。

图 15.4 描述了“中心化训练 + 去中心化决策”的系统架构。中央控制器上有所有的价值网络及其目标网络（图中没有画出目标网络）：

$$\begin{aligned}v(s; w^1), \quad v(s; w^2), \quad \dots, \quad v(s; w^m), \\ v(s; w^{1-}), \quad v(s; w^{2-}), \quad \dots, \quad v(s; w^{m-}).\end{aligned}$$

中央控制器用智能体发来的观测 $[o^1, \dots, o^m]$ 和奖励 $[r^1, \dots, r^m]$ 训练这些价值网络。中央控制器把 TD 误差 $\delta^1, \dots, \delta^m$ 反馈给智能体；第 i 号智能体用 δ^i 以及本地的 o^i 、 a^i 来

训练自己的策略网络。

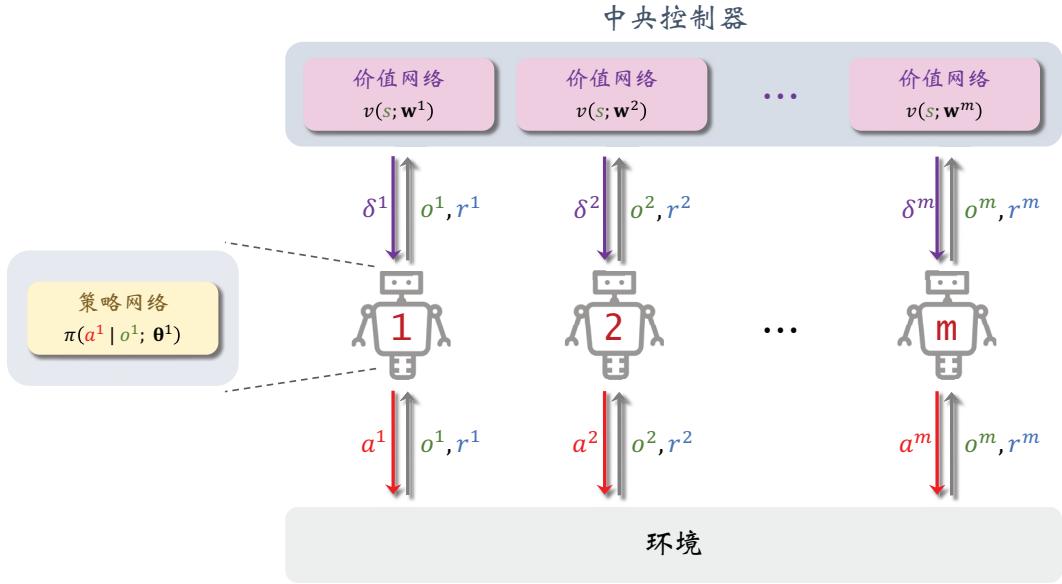


图 15.4: 中心化训练的系统架构。所有 m 个价值网络部署到中央控制器上，策略网络部署到每个智能体上。

上一章“完全合作关系”设定下的“中心化训练”只在中央控制器上部署一个价值网络 $v(s; \mathbf{w})$ 。而此处中央控制器上有 m 个价值网络，每个价值网络对应一个智能体。这是因为此处是“非合作关系”，每个智能体各自对应一个状态价值函数 $V_\pi^i(s)$ ，而非有共用的 V_π 。

中心化训练：训练的过程需要所有 m 个智能体共同参与，共同改进策略网络参数 $\theta^1, \dots, \theta^m$ 与价值网络参数 $\mathbf{w}^1, \dots, \mathbf{w}^m$ 。设第 i 号智能体的策略网络、价值网络、目标网络当前的参数分别是 θ_{now}^i 、 $\mathbf{w}_{\text{now}}^i$ 和 $\mathbf{w}_{\text{now}}^{i-}$ 。训练的流程如下：

1. 每个智能体 i 与环境交互，获取当前观测 o_t^i ，独立做随机抽样：

$$a_t^i \sim \pi(\cdot | o_t^i; \theta_{\text{now}}^i), \quad \forall i = 1, \dots, m, \quad (15.1)$$

并执行选中的动作。

2. 下一时刻，每个智能体 i 都观测到 o_{t+1}^i 和收到奖励 r_t^i 。
3. 每个智能体 i 向中央控制器传输观测 o_t^i 、 o_{t+1}^i 、 r_t^i ；中央控制器得到状态

$$s_t = [o_t^1, \dots, o_t^m] \quad \text{和} \quad s_{t+1} = [o_{t+1}^1, \dots, o_{t+1}^m].$$

4. 让价值网络做预测：

$$\hat{v}_t^i = v(s_t; \mathbf{w}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

5. 让目标网络做预测：

$$\hat{v}_{t+1}^{i-} = v(s_{t+1}; \mathbf{w}_{\text{now}}^{i-}), \quad \forall i = 1, \dots, m.$$

6. 计算 TD 目标与 TD 误差：

$$\hat{y}_t^i = r_t^i + \gamma \cdot \hat{v}_{t+1}^{i-}, \quad \delta_t^i = \hat{v}_t^i - \hat{y}_t^i, \quad \forall i = 1, \dots, m.$$

7. 更新价值网络参数:

$$\mathbf{w}_{\text{new}}^i \leftarrow \mathbf{w}_{\text{now}}^i - \alpha \cdot \delta_t^i \cdot \nabla_{\mathbf{w}^i} v(s_t; \mathbf{w}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

8. 更新目标网络参数:

$$\mathbf{w}_{\text{new}}^{i-} \leftarrow \tau \cdot \mathbf{w}_{\text{new}}^i + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^{i-}, \quad \forall i = 1, \dots, m.$$

9. 更新策略网络参数:

$$\boldsymbol{\theta}_{\text{new}}^i \leftarrow \boldsymbol{\theta}_{\text{now}}^i - \beta \cdot \delta_t^i \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(a_t^i | o_t^i; \boldsymbol{\theta}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

去中心化决策: 在完成训练之后，不再需要价值网络 $v(s; \mathbf{w}^1), \dots, v(s; \mathbf{w}^m)$ 。智能体只需要用其本地部署的策略网络 $\pi(a^i | o^i; \boldsymbol{\theta}^i)$ 做决策，决策过程无需通信，因此决策速度很快。

15.4 连续控制与 MADDPG

前两节的 MAN-A2C 仅限于离散控制。本节研究连续控制问题，即动作空间 $\mathcal{A}^1, \mathcal{A}^2, \dots, \mathcal{A}^m$ 都是连续集合，动作 $\mathbf{a}^i \in \mathcal{A}^i$ 是向量。本节介绍一种适用于连续控制的多智能体强化学习 (MARL) 方法。多智能体深度确定策略梯度 (Multi-Agent Deep Deterministic Policy Gradient, 缩写 MADDPG) 是一种很有名的 MARL 方法，它的架构是“中心化训练 + 去中心化决策”。

15.4.1 策略网络和价值网络

设系统里有 m 个智能体。每个智能体对应一个策略网络和一个价值网络：

$$\mu(o^i; \theta^i) \quad \text{和} \quad q(s, \mathbf{a}; \mathbf{w}^i).$$

策略网络是确定性的：对于确定的输入 o^i ，输出的动作 $\mathbf{a}^i = \mu(o^i; \theta^i)$ 是确定的。价值网络的输入是全局状态 $s = [o^1, \dots, o^m]$ 与所有智能体的动作 $\mathbf{a} = [\mathbf{a}^1, \dots, \mathbf{a}^m]$ ，输出是一个实数，表示“基于状态 s 执行动作 \mathbf{a} ”的好坏程度。第 i 号策略网络 $\mu(o^i; \theta^i)$ 用于控制第 i 号智能体，而价值网络 $q(s, \mathbf{a}; \mathbf{w}^i)$ 则用于评价所有动作 \mathbf{a} ，给出的分数可以指导第 i 号策略网络做出改进；见图 15.5。MADDPG 因此可以看做一种 Actor-Critic 方法。

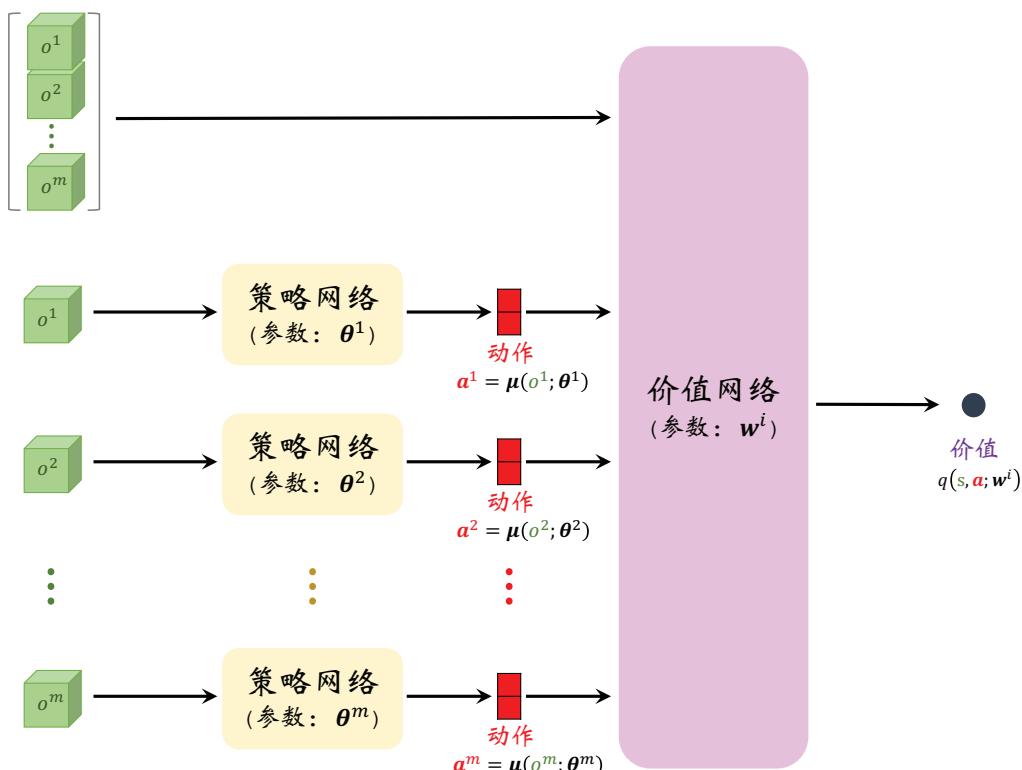


图 15.5：所有智能体的策略网络与第 i 号智能体的价值网络。

15.4.2 算法推导

训练策略网络和价值网络的算法与第 10.2 节的单智能体 DPG 非常类似：用确定策略梯度更新策略网络，用 TD 算法更新价值网络。MADDPG 是异策略(Off-policy)，我们可以使用经验回放，重复利用过去的经验。用一个经验回放数组存储收集到的经验，每一条经验都是 $(s_t, \mathbf{a}_t, r_t, s_{t+1})$ 这样一个四元组，其中

$$\begin{aligned} s_t &= [o_t^1, \dots, o_t^m], \\ \mathbf{a}_t &= [\mathbf{a}_t^1, \dots, \mathbf{a}_t^m], \\ s_{t+1} &= [o_{t+1}^1, \dots, o_{t+1}^m], \\ r_t &= [r_t^1, \dots, r_t^m]. \end{aligned}$$

训练策略网络：训练第 i 号策略网络 $\mu(o^i; \theta^i)$ 的目标是改进 θ^i ，增大第 i 号价值网络的平均打分。所以目标函数是：

$$\widehat{J}^i(\theta^1, \dots, \theta^m) = \mathbb{E}_S \left[q \left(S, [\mu(O^1; \theta^1), \dots, \mu(O^i; \theta^i), \dots, \mu(O^m; \theta^m)]; \mathbf{w}^i \right) \right].$$

公式中的期望是关于状态 $S = [O^1, \dots, O^m]$ 求的。目标函数的梯度等于：

$$\nabla_{\theta^i} \widehat{J}^i(\theta^1, \dots, \theta^m) = \mathbb{E}_S \left[\nabla_{\theta^i} q \left(S, [\mu(O^1; \theta^1), \dots, \mu(O^i; \theta^i), \dots, \mu(O^m; \theta^m)]; \mathbf{w}^i \right) \right].$$

接下来用蒙特卡洛近似公式中的期望。从经验回放数组中随机抽取一个状态：¹

$$s_t = [o_t^1, \dots, o_t^m],$$

它可以看做是随机变量 S 的一个观测值。用所有 m 个策略网络计算动作

$$\hat{\mathbf{a}}_t^1 = \mu(o_t^1; \theta^1), \dots, \hat{\mathbf{a}}_t^m = \mu(o_t^m; \theta^m).$$

那么目标函数的梯度 $\nabla_{\theta^i} \widehat{J}^i(\theta^1, \dots, \theta^m)$ 可以近似成为：

$$\begin{aligned} \mathbf{g}_{\theta}^i &= \nabla_{\theta^i} q \left(s_t, [\mu(o_t^1; \theta^1), \dots, \mu(o_t^i; \theta^i), \dots, \mu(o_t^m; \theta^m)]; \mathbf{w}^i \right) \\ &= \nabla_{\theta^i} q \left(s_t, [\hat{\mathbf{a}}_t^1, \dots, \hat{\mathbf{a}}_t^m]; \mathbf{w}^i \right). \end{aligned}$$

由于 $\hat{\mathbf{a}}_t^i = \mu(o_t^i; \theta^i)$ ，用链式法则可得：

$$\mathbf{g}_{\theta}^i = \nabla_{\theta^i} \mu(o_t^i; \theta^i) \cdot \nabla_{\hat{\mathbf{a}}^i} q \left(s_t, [\hat{\mathbf{a}}_t^1, \dots, \hat{\mathbf{a}}_t^m]; \mathbf{w}^i \right).$$

做梯度上升更新参数 θ^i ：

$$\theta^i \leftarrow \theta^i + \beta \cdot \mathbf{g}_{\theta}^i.$$

注意，在更新第 i 号策略网络的时候，除了用到全局状态 s_t ，还需要用到所有智能体的策略网络，以及第 i 号价值网络 $q(s, \mathbf{a}; \mathbf{w}^i)$ 。

训练价值网络：可以用 TD 算法训练第 i 号价值网络 $q(s, \mathbf{a}; \mathbf{w}^i)$ ，让价值网络更好拟

¹更新策略网络只需要四元组 $(s_t, \mathbf{a}_t, r_t, s_{t+1})$ 中的 s_t ，没有用其余三个。

合价值函数 $Q_\pi^i(s, \mathbf{a})$ 。给定四元组 $(s_t, \mathbf{a}_t, r_t, s_{t+1})$ ，用所有 m 个策略网络计算动作

$$\hat{\mathbf{a}}_{t+1}^1 = \mu(o_{t+1}^1; \theta^1), \dots, \hat{\mathbf{a}}_{t+1}^m = \mu(o_{t+1}^m; \theta^m).$$

设 $\hat{\mathbf{a}}_{t+1} = [\hat{\mathbf{a}}_{t+1}^1, \dots, \hat{\mathbf{a}}_{t+1}^m]$ 。然后计算 TD 目标：

$$\hat{y}_t^i = r_t^i + \gamma \cdot q(s_{t+1}, \hat{\mathbf{a}}_{t+1}; \mathbf{w}^i).$$

再计算 TD 误差：

$$\delta_t^i = q(s_t, \mathbf{a}_t; \mathbf{w}^i) - \hat{y}_t^i.$$

最后做梯度下降更新参数 \mathbf{w}^i ：

$$\mathbf{w}^i \leftarrow \mathbf{w}^i - \alpha \cdot \delta_t^i \cdot \nabla_{\mathbf{w}^i} q(s_t, \mathbf{a}_t; \mathbf{w}^i).$$

这样可以让价值网络的预测 $q(s_t, \mathbf{a}_t; \mathbf{w}^i)$ 更接近 TD 目标 \hat{y}_t^i 。

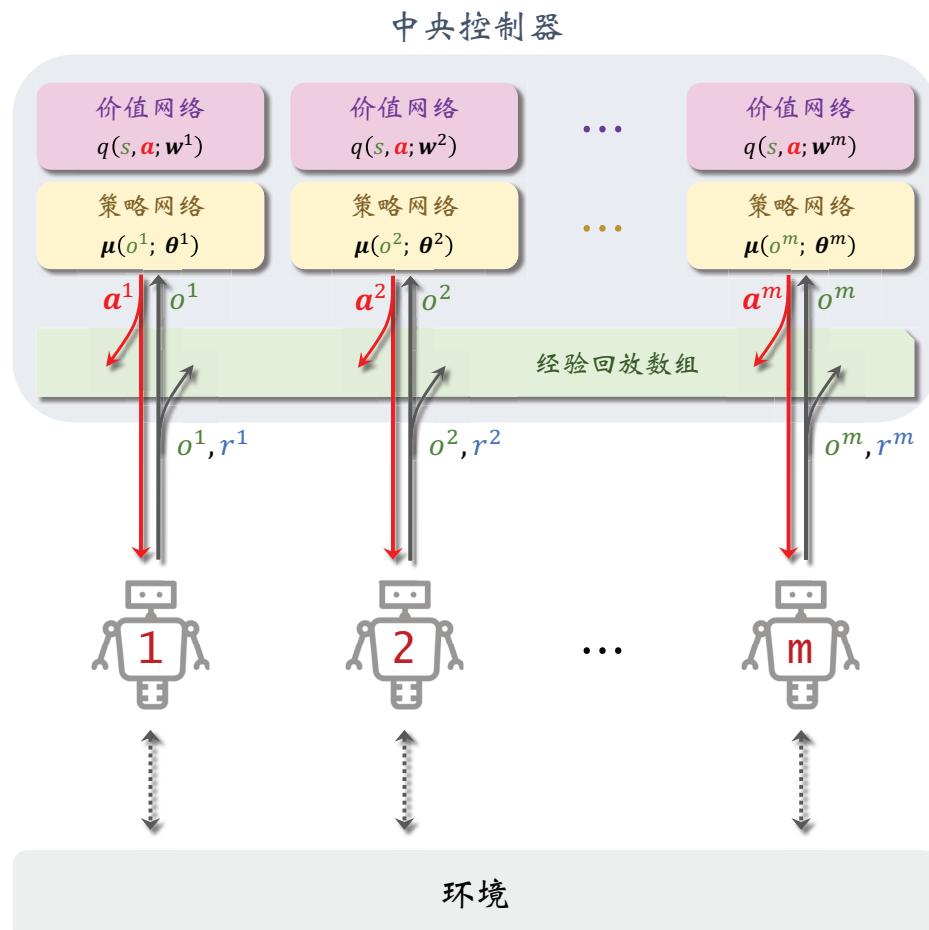


图 15.6: MADDPG 的中心化训练。

15.4.3 中心化训练

为了训练第 i 号策略网络和第 i 号价值网络，我们需要用到如下信息：从经验回放数组中取出的 $s_t, \mathbf{a}_t, s_{t+1}, r_t^i$ 、所有 m 个策略网络、以及第 i 号价值网络。很显然，一

个智能体不可能有所有这些信息，因此 MADDPG 需要“中心化训练”。

中心化训练的系统架构如图 15.6 所示。有一个中央控制器，上面部署所有的策略网络和价值网络。训练过程中，策略网络部署到中央控制器上，所以智能体不能自主做决策，智能体只是执行中央控制器发来的指令。由于训练使用异策略，可以把收集经验和更新神经网络参数分开做。

用行为策略收集经验。 行为策略 (Behavior Policy) 可以不同于目标策略 (Target Policy)，即 μ 。行为策略是什么都无所谓，比如第 i 个智能体的行为策略可以是

$$\mathbf{a}^i = \mu(o^i; \theta_{\text{old}}^i) + \epsilon,$$

其中 ϵ 是与 \mathbf{a}^i 维度相同的向量，每个元素都是从正态分布中独立抽取的，相当于随机噪声。具体实现的时候，智能体把其观测 o^i 发送给中央控制器。控制器往第 i 号策略网络输出的动作 $\mu(o^i; \theta^i)$ 中加入随机噪声 ϵ ，把动作 \mathbf{a}^i 发送给给第 i 号智能体，智能体执行 \mathbf{a}^i 。随后智能体观测到奖励 r^i ，发送给控制器。控制器把每一轮的 o^i, \mathbf{a}^i, r^i 依次存入经验回放数组。

中央控制器更新策略网络和价值网络： 实际实现的时候，中央控制器上还需要有如下目标网络（图 15.6 中没有画出）：

$$\begin{aligned} \pi(\mathbf{a}^1 | o^1; \theta^{1-}), \quad \pi(\mathbf{a}^2 | o^2; \theta^{2-}), \quad \dots, \quad \pi(\mathbf{a}^m | o^m; \theta^{m-}); \\ q(s, \mathbf{a}; \mathbf{w}^{1-}), \quad q(s, \mathbf{a}; \mathbf{w}^{2-}), \quad \dots, \quad q(s, \mathbf{a}; \mathbf{w}^{m-}). \end{aligned}$$

设第 i 号智能体当前的参数为：

$$\theta_{\text{now}}^i, \quad \theta_{\text{now}}^{i-}, \quad \mathbf{w}_{\text{now}}^i, \quad \mathbf{w}_{\text{now}}^{i-}.$$

中央控制器每次从经验回放数组中随机抽取一个四元组 $(s_t, \mathbf{a}_t, r_t, s_{t+1})$ ，然后按照下面的步骤更新所有策略网络和所有价值网络：

1. 让所有 m 个目标策略网络做预测：

$$\hat{\mathbf{a}}_{t+1}^{i-} = \mu(o_{t+1}^i; \theta_{\text{now}}^{i-}), \quad \forall i = 1, \dots, m.$$

把预测汇总成 $\hat{\mathbf{a}}_{t+1}^- = [\hat{\mathbf{a}}_{t+1}^{1-}, \dots, \hat{\mathbf{a}}_{t+1}^{m-}]$ 。

2. 让所有 m 个目标价值网络做出预测：

$$\hat{q}_{t+1}^{i-} = q(s_{t+1}, \hat{\mathbf{a}}_{t+1}^-; \mathbf{w}_{\text{now}}^{i-}), \quad \forall i = 1, \dots, m.$$

3. 计算 TD 目标：

$$\hat{y}_t^i = r_t^i + \gamma \cdot \hat{q}_{t+1}^{i-}, \quad \forall i = 1, \dots, m.$$

4. 让所有 m 个价值网络做预测：

$$\hat{q}_t^i = q(s_t, \mathbf{a}_t; \mathbf{w}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

5. 计算 TD 误差：

$$\delta_t^i = \hat{q}_t^i - \hat{y}_t^i, \quad \forall i = 1, \dots, m.$$

6. 更新所有 m 个价值网络：

$$\mathbf{w}_{\text{now}}^i \leftarrow \mathbf{w}_{\text{now}}^i - \alpha \cdot \delta_t^i \cdot \nabla_{\mathbf{w}^i} q(s_t, \mathbf{a}_t; \mathbf{w}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

7. 让所有 m 个策略网络做预测：

$$\hat{\mathbf{a}}_t^i = \mu(o_t^i; \theta_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

把预测汇总成 $\hat{\mathbf{a}}_t = [\hat{\mathbf{a}}_t^1, \dots, \hat{\mathbf{a}}_t^m]$ 。请区别 $\hat{\mathbf{a}}_t$ 与经验回放数组中抽出的 \mathbf{a}_t 。

8. 更新所有 m 个策略网络： $\forall i = 1, \dots, m$,

$$\theta_{\text{new}}^i \leftarrow \theta_{\text{now}}^i - \beta \cdot \nabla_{\theta^i} \mu(s_t; \theta_{\text{now}}^i) \cdot \nabla_{\mathbf{a}_t^i} q(s_t, \hat{\mathbf{a}}_t; \mathbf{w}_{\text{now}}^i).$$

9. 更新所有 $2m$ 个目标网络： $\forall i = 1, \dots, m$,

$$\begin{aligned} \theta_{\text{new}}^{i-} &\leftarrow \tau \cdot \theta_{\text{new}}^i + (1 - \tau) \cdot \theta_{\text{now}}^{i-}, \\ \mathbf{w}_{\text{new}}^{i-} &\leftarrow \tau \cdot \mathbf{w}_{\text{new}}^i + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^{i-}. \end{aligned}$$

改进方法：可以用三种方法改进 MADDPG。第一，用第 10.3 节中 TD3 的三种技巧改进训练的算法：

- 用截断双 Q 学习 (Clipped Double Q-Learning) 训练价值网络 $q(s, \mathbf{a}; \mathbf{w}^i)$, $\forall i = 1, \dots, m$ 。
- 往训练算法第一步中的 $\hat{\mathbf{a}}_{t+1}^{i-}$ 加入噪声。
- 减小更新策略网络和目标网络的频率，每更新 $k (> 1)$ 次价值网络，更新一次策略网络和目标网络。

第二，按照第 11 章中的方法，在策略网络和价值网络中使用 RNN，记忆历史观测。第三，在价值网络的结构中使用注意力机制，见下一章。

15.4.4 去中心化决策

在完成训练之后，不再需要价值网络，只需要策略网络做决策。如图 15.7 所示，把策略网络部署到对应的智能体上。第 i 号智能体可以基于本地观测的 o^i ，在本地独立做决策： $\mathbf{a}^i = \mu(o^i; \theta^i)$ 。

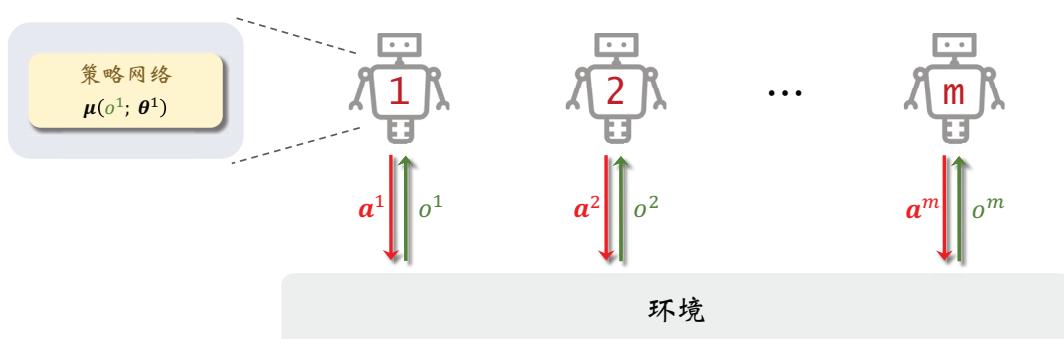


图 15.7: MADDPG 的去中心化决策。

∽ 第十五章 相关文献 ∽

MAN-A2C 是本书设计出来的简单方法，用于讲解非合作设定下的 MARL，方便读者理解。MAN-A2C 这个名字并没有出现在任何文献中。本章介绍的 MADDPG 由 Lowe 等人 2017 年的论文提出 [65]，它的改进版本叫做 MATD3。

《深度强化学习》 2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

第十六章 注意力机制与多智能体强化学习

注意力机制 (Attention) 是一种重要的深度学习方法，它最主要的用途是自然语言处理，比如机器翻译、情感分析。本章的目的不是详细解释注意力机制的原理，而是它在多智能体强化学习 (MARL) 中的应用。第 16.1 简单介绍自注意力机制 (Self-Attention)，它是一种特殊的注意力机制。第 16.2 将自注意力机制应用在 MARL，改进中心化训练或中心化决策。当智能体数量 m 较大时，自注意力机制对 MARL 有明显的效果提升。

16.1 自注意力机制

注意力机制 (Attention) 最初用于改进循环神经网络 (RNN)，提高 Sequence-to-Sequence (Seq2Seq) 模型的表现。自注意力机制 (Self-Attention) 是注意力机制的一种扩展，不局限于 Seq2Seq 模型，可以用于任意的 RNN。后来 Transformer 模型将 RNN 剥离，只保留注意力机制。与 RNN + 注意力机制相比，只用注意力机制居然表现更好，在机器翻译等任务上的效果有大幅提升。本节不深入讨论注意力机制与 RNN、Seq2Seq 之间的关系，而只介绍本章所需的一些知识点。

考虑这样一个问题：输入是长度为 m 的序列 (x^1, \dots, x^m) ，序列中的元素都是向量，要求输出长度同样为 m 的序列 (c^1, \dots, c^m) ；如图 16.1 所示。问题还有两个要求：

- 第一，序列的长度 m 是不确定的，可以动态变化。但是神经网络的参数数量不能变化。
- 第二，输出的向量 c^i 不是仅仅依赖于向量 x^i ，而是依赖于所有的输入向量 (x^1, \dots, x^m) 。

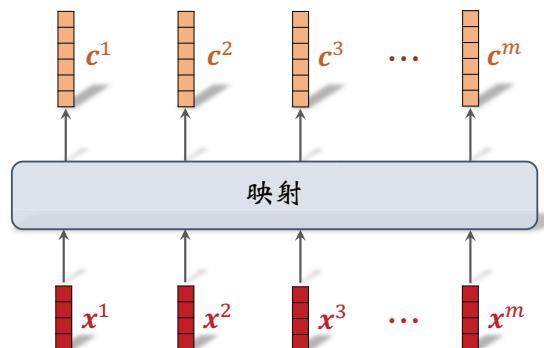


图 16.1：将一个长度为 m 的向量序列映射到另一个同等长度的向量序列。

可以用简单的全连接网络逐个把向量 x^i 映射到 c^i ，但是这样得到的 c^i 仅依赖于 x^i 一个向量而已，不满足第二个要求。第 12 章介绍的 RNN 也不满足第二个要求；RNN 输出的向量 c^i 只依赖于 (x^1, \dots, x^i) ，而不依赖于 (x^{i+1}, \dots, x^m) 。

自注意力层 (Self-Attention Layer) 可以解决上述问题。如图 16.2 所示，自注意力层的输入是序列 (x^1, \dots, x^m) ，其中的向量的大小都是 $d_{\text{in}} \times 1$ 。自注意力层有三个参数矩阵：

$$\mathbf{W}_q \in \mathbb{R}^{d_q \times d_{\text{in}}}, \quad \mathbf{W}_k \in \mathbb{R}^{d_k \times d_{\text{in}}}, \quad \mathbf{W}_v \in \mathbb{R}^{d_v \times d_{\text{in}}}.$$

序列长度 m 不会影响参数的数量。不论序列有多长，参数矩阵只有 $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v$ 。这三个参数矩阵需要从训练数据中学习。自注意力层通过以下步骤，把输入序列 (x^1, \dots, x^m) 映射到输出序列 (c^1, \dots, c^m) ，输出向量的大小都是 $d_{\text{out}} \times 1$ 。

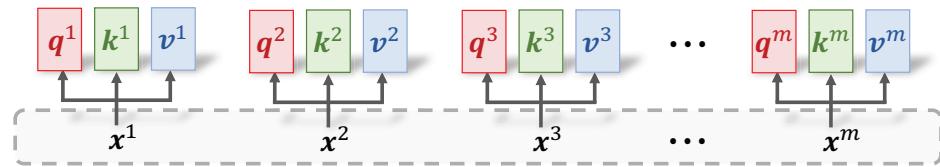


图 16.2: 首先把 x^i 映射到三元组 (q^i, k^i, v^i) , $\forall i = 1, \dots, m$ 。

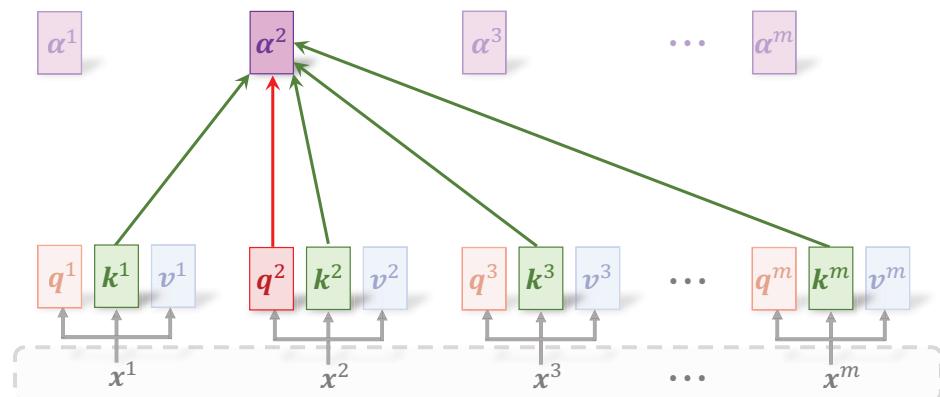


图 16.3: 然后用 q^i 和 (k^1, \dots, k^m) 计算权重向量 $\alpha^i \in \mathbb{R}^m$, $\forall i = 1, \dots, m$ 。

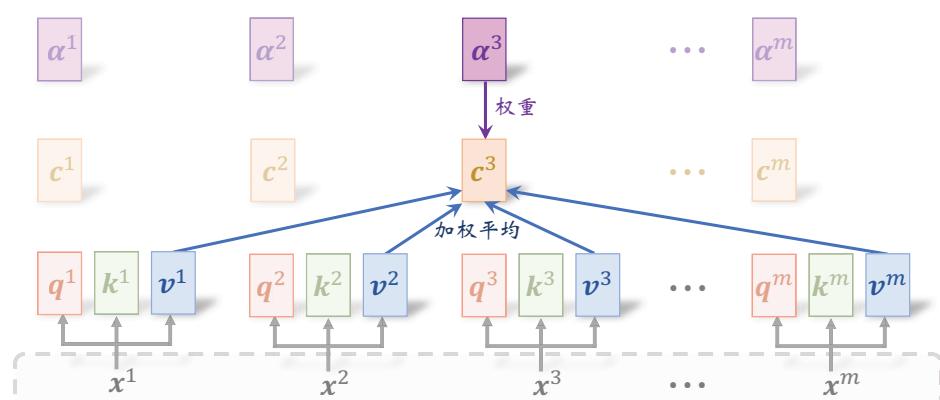


图 16.4: 最后用 α^i 和 (v^1, \dots, v^m) 计算输出向量 $c^i \in \mathbb{R}^{d_{\text{out}}}$, $\forall i = 1, \dots, m$ 。

《深度强化学习》2021-03-18 尚未完成校对，仅供参考。
如发现错误，请告知作者 shusen.wang@stevens.edu

16.1 自注意力机制

- 如图 16.2 所示，对于所有的 $i = 1, \dots, m$ ，把输入的 \mathbf{x}^i 映射到三元组 $(\mathbf{q}^i, \mathbf{k}^i, \mathbf{v}^i)$:

$$\begin{aligned}\mathbf{q}^i &= \mathbf{W}_q \mathbf{x}^i \in \mathbb{R}^{d_q}, \\ \mathbf{k}^i &= \mathbf{W}_k \mathbf{x}^i \in \mathbb{R}^{d_q}, \\ \mathbf{v}^i &= \mathbf{W}_v \mathbf{x}^i \in \mathbb{R}^{d_{\text{out}}}.\end{aligned}$$

- 如图 16.3 所示，计算权重向量 $(\alpha^1, \dots, \alpha^m)$ ，每个权重向量的大小都是 $m \times 1$ 。第 i 个权重向量 α^i 依赖于 \mathbf{q}^i 和 $(\mathbf{k}^1, \dots, \mathbf{k}^m)$:

$$\alpha^i = \text{softmax} \left(\langle \mathbf{q}^i, \mathbf{k}^1 \rangle, \langle \mathbf{q}^i, \mathbf{k}^2 \rangle, \dots, \langle \mathbf{q}^i, \mathbf{k}^m \rangle \right), \quad \forall i = 1, \dots, m.$$

公式中的 $\langle \cdot, \cdot \rangle$ 是向量内积。由于向量 α^i 是 Softmax 函数的输出，它的元素都是正实数，而且相加等于 1。向量 α^i 的第 j 个元素（记作 α_j^i ）表示 \mathbf{x}_i 与 \mathbf{x}_j 的相关性； \mathbf{x}_i 与 \mathbf{x}_j 越相关，那么元素 α_j^i 就越大。

- 如图 16.4 所示，计算输出向量 $(\mathbf{c}^1, \dots, \mathbf{c}^m)$ ，每个输出向量的维度都是 d_{out} 。第 i 个输出向量 \mathbf{c}^i 依赖于 α^i 和 $(\mathbf{v}^1, \dots, \mathbf{v}^m)$:

$$\mathbf{c}^i = [\mathbf{v}^1, \mathbf{v}^2, \dots, \mathbf{v}^m] \cdot \alpha^i = \sum_{j=1}^m \alpha_j^i \mathbf{v}^j, \quad \forall i = 1, \dots, m.$$

\mathbf{c}^i 是向量 $\mathbf{v}^1, \dots, \mathbf{v}^m$ 的加权平均，权重是 $\alpha^i = [\alpha_1^i, \dots, \alpha_m^i]$ 。

为什么这种神经网络结构叫做注意力 (Attention) 呢？如图 16.5 所示，向量 \mathbf{x}^i 位置上的输出是 \mathbf{c}^i ，它是做加权平均计算出来的：

$$\mathbf{c}^i = \alpha_1^i \mathbf{v}^1 + \alpha_2^i \mathbf{v}^2 + \dots + \alpha_m^i \mathbf{v}^m.$$

权重 $\alpha^i = [\alpha_1^i, \dots, \alpha_m^i]$ 反映出 \mathbf{c}^i 最“关注”哪些输入的 $\mathbf{v}^j = \mathbf{W}_v \mathbf{x}^j$ 。如果权重 α_j^i 大，说明 \mathbf{x}^j 对 \mathbf{c}^i 的影响较大。 \mathbf{c}^i 应当重点关注对其影响较大的 \mathbf{x}^j 。

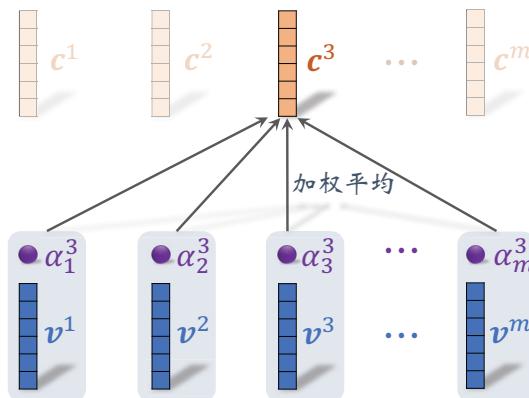


图 16.5: 第 i 个输出向量 \mathbf{c}^i 由权重 $\alpha^i = [\alpha_1^i, \dots, \alpha_m^i]$ 和向量 $(\mathbf{v}^1, \dots, \mathbf{v}^m)$ 决定。

上述自注意力层叫做单头自注意力层 (Single-Head Self-Attention Layer)，简称“单头”。实践中更常用的是多头自注意力层 (Multi-Head Self-Attention Layer)，简称“多头”，它是多个单头的组合，见图 16.6。设多头由 l 个单头组成。每个单头有自己的 3 个参数矩阵，所以多头一共有 $3l$ 个参数矩阵。它们的输入都是序列 $(\mathbf{x}_1, \dots, \mathbf{x}_m)$ ，它们的输出都是长度为 m 的向量序列。

第 1 个自注意力层输出: $(\mathbf{c}_1^1, \mathbf{c}_1^2, \mathbf{c}_1^3, \dots, \mathbf{c}_1^m)$,

第 2 个自注意力层输出: $(\mathbf{c}_2^1, \mathbf{c}_2^2, \mathbf{c}_2^3, \dots, \mathbf{c}_2^m)$,

\vdots

第 l 个自注意力层输出: $(\mathbf{c}_l^1, \mathbf{c}_l^2, \mathbf{c}_l^3, \dots, \mathbf{c}_l^m)$.

其中每个向量 c_j^i 的大小都是 $d_{\text{out}} \times 1$ 。多头的输出记作序列 (c^1, \dots, c^m) , 其中每个 c^i 都是做连接 (Concatenation) 得到的:

$$c^i = [c_1^i; c_2^i; \dots; c_l^i] \in \mathbb{R}^{l d_{\text{out}}}, \quad \forall i = 1, \dots, m.$$

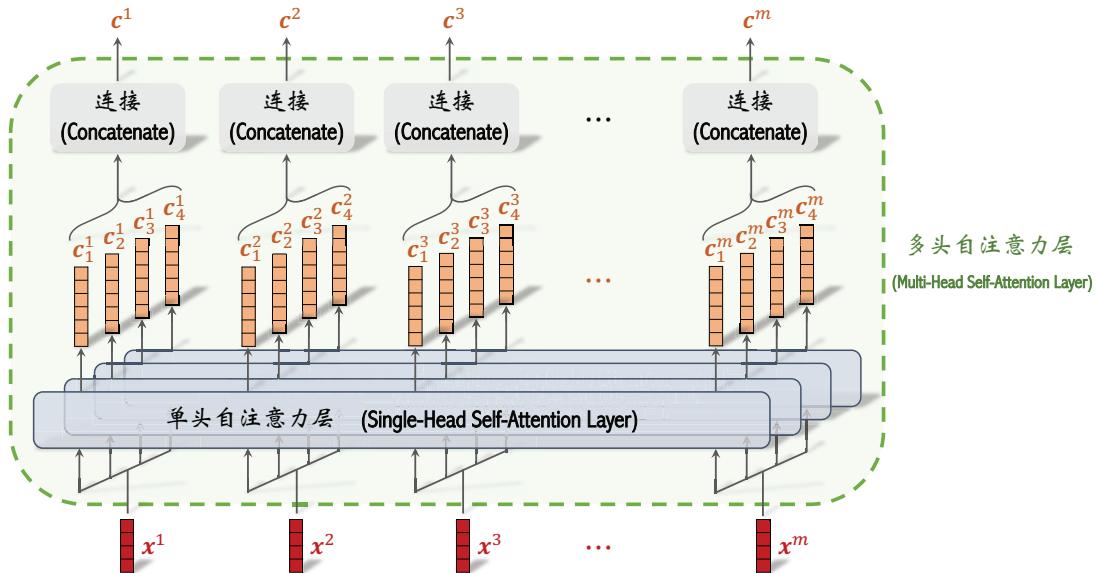


图 16.6: 这个例子中, 多头自注意力层由 $l = 4$ 个单头自注意力层组成。

总结一下, 多头自注意力层把长度为 m 的向量序列映射到同等长度的向量序列。长度 m 可以任意变化, 神经网络结构无需改变。实现一个多头自注意力层需要指定三个超参数: 单头的数量 l 、每个单头输出的大小 d_{out} 、向量 q^i 和 k^i 的大小 d_q 。多头的输出是长度为 m 的向量序列, 每个向量的大小是 $l d_{\text{out}} \times 1$ 。超参数 d_q 不影响输出的大小, 它只在计算权重向量 $\alpha^1, \dots, \alpha^m$ 的时候使用。

16.2 自注意力在中心化训练中的应用

自注意力机制 (Self-Attention) 是改进多智能体强化学习 (MARL) 的一种有效技巧，可以应用在中心化训练或中心化决策当中。多智能体系统中有 m 个智能体，每个智能体有自己的观测（记作 o^1, \dots, o^m ）和动作（记作 a^1, \dots, a^m ）。我们考虑非合作关系的 MARL。如果做中心化训练，需要用到 m 个状态价值网络

$$v([o^1, \dots, o^m]; \mathbf{w}^1), \dots, v([o^1, \dots, o^m]; \mathbf{w}^m),$$

或 m 个动作价值网络

$$q([o^1, \dots, o^m], [a^1, \dots, a^m]; \mathbf{w}^1), \dots, q([o^1, \dots, o^m], [a^1, \dots, a^m]; \mathbf{w}^m).$$

由于是非合作关系， m 个价值网络有各自的参数，而且它们的输出各不相同。我们首先以状态价值网络 v 为例讲解神经网络的结构。

不使用自注意力的状态价值网络：

图 16.7 是状态价值网络 $v(s; \mathbf{w}^i)$ 最简单的实现。每个价值网络是一个独立的神经网络，有自己的参数。底层提取特征的卷积网络可以在 m 个价值网络中共享（即复用），而上层的全连接网络不能共享。神经网络的输入是所有智能体的观测的连接 (Concatenation)，输出是实数

$$\hat{v}^i = v([o^1, \dots, o^m]; \mathbf{w}^i).$$

这种简单的神经网络结构有几个不足之处。

- 智能体数量 m 越大，神经网络的参数越多。神经网络的输入是 m 个观测的连接，它们被映射到特征向量 \mathbf{x} 。 m 越大，我们就必须把向量 \mathbf{x} 维度设置得越大，否则 \mathbf{x} 无法很好地概括 $[o^1, \dots, o^m]$ 的完整信息。 \mathbf{x} 维度越大，全连接网络的参数就越多，神经网络就越难训练（即需要收集更多的经验才能训练好神经网络）。
- 当 m 很大的时候，并非所有智能体的观测 o^1, \dots, o^m 都与第 i 号智能体密切相关。第 i 号智能体应当学会判断哪些智能体最相关，并重点关注密切相关的智能体，避免决策受无关的智能体干扰。
- 图 16.7 中价值网络的输入是 $[o^1, \dots, o^m]$ ，即所有观测的连接。如果交换其中 o^j 和 o^k 的位置，那么价值网络输出的 \hat{v}^i 会发生变化，这是没有道理的。理想情况下，只要 $j \neq i, k \neq i$ ，那么交换 o^j 和 o^k 的位置就不该改变第 i 号价值网络的输出值 \hat{v}^i 。

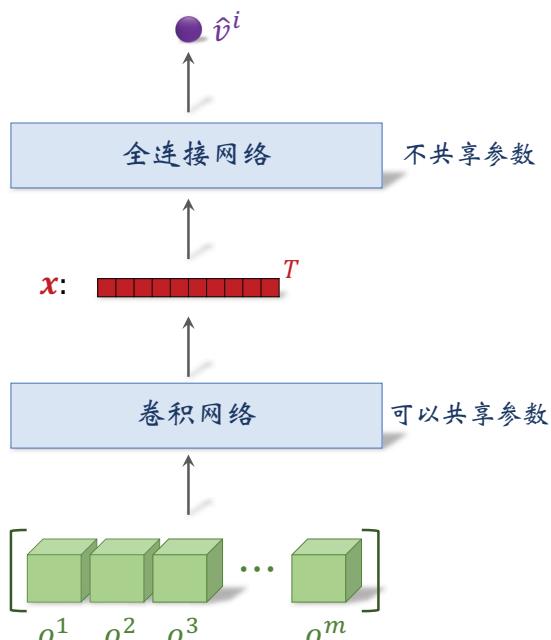


图 16.7：第 i 号状态价值网络最简单的实现。

使用自注意力的状态价值网络：图 16.8 是对状态价值网络更好的实现方式，避免了上面讨论的三种不足之处。神经网络的结构是这样的：

- 输入仍然是所有智能体的观测 o^1, \dots, o^m 。对于所有的 i ，用一个卷积网络把 o^i 映射到特征向量 x^i 。这些卷积网络的参数都是相同的。
- 自注意力层的输入是向量序列 (x^1, \dots, x^m) ，输出是序列 (c^1, \dots, c^m) 。向量 c^i 依赖于所有的观测 x^1, \dots, x^m ，但是 c^i 主要取决于最密切相关的一个或几个 x 。
- 第 i 号全连接网络把向量 c^i 作为输入，输出一个实数 \hat{v}^i ，作为第 i 号价值网络的输出。在非合作关系的设定下， m 个价值网络是不同的，因此 m 个全连接网络不共享参数。

图 16.8 中只用了一个自注意力层。其实可以重复自注意力层，比如：

$\dots \rightarrow$ 自注意力层 \rightarrow 全连接层 \rightarrow 自注意力层 \rightarrow 全连接层 $\rightarrow \dots$

自注意力的层数是一个超参数，需要用户自己调。

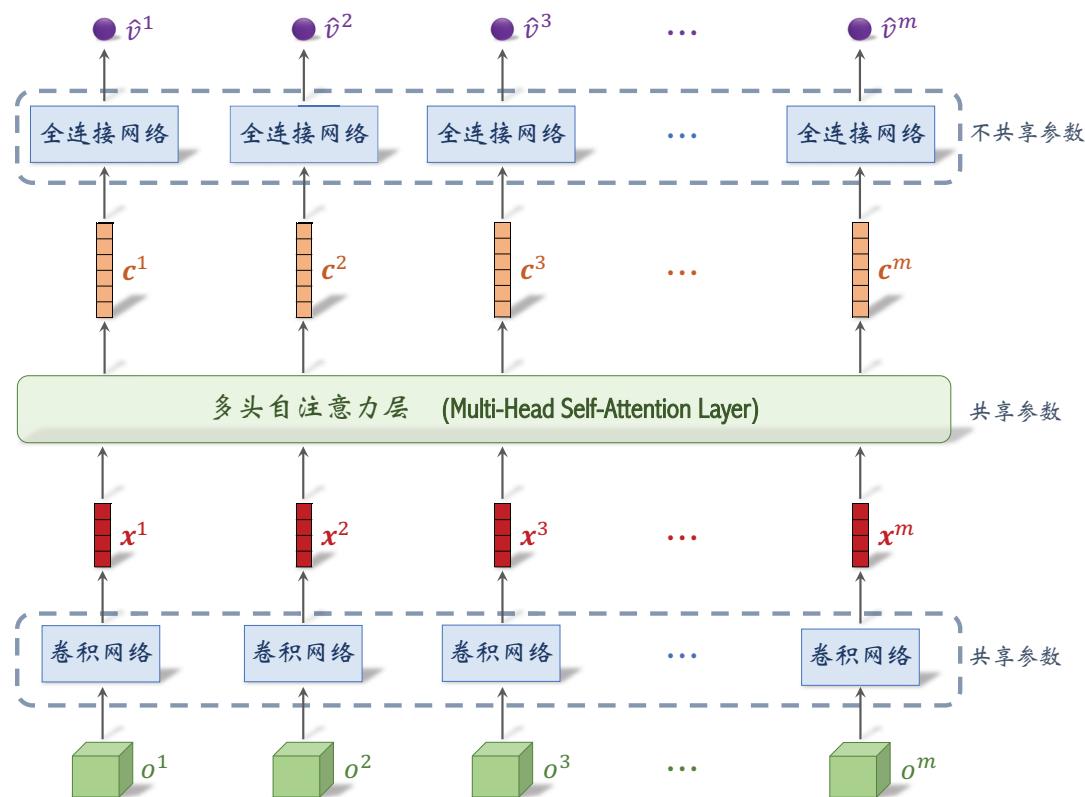


图 16.8：带有自注意力的状态价值网络。图中的 $\hat{v}^i = v([o^1, \dots, o^m]; \mathbf{w}^i)$ 是第 i 个价值网络的输出。

使用自注意力的动作价值网络：上一章介绍了 MADDPG，它是一种连续控制方法，用于非合作关系的设定。它的架构是“中心化训练 + 去中心化决策”，在中央控制器上部署 m 个动作价值网络，把第 i 个记作：

$$\hat{q}^i = q([o^1, \dots, o^m], [a^1, \dots, a^m]; \mathbf{w}^i).$$

它的输入是所有智能体的观测和动作，输出是实数 \hat{q}^i ，表示动作价值。可以按照图 16.9

16.2 自注意力在中心化训练中的应用

实现动作价值网络。在 MADDPG 中使用这样的神经网络结构可以提高 MADDPG 的表现，尤其是当 m 较大的时候，效果的提升较大。

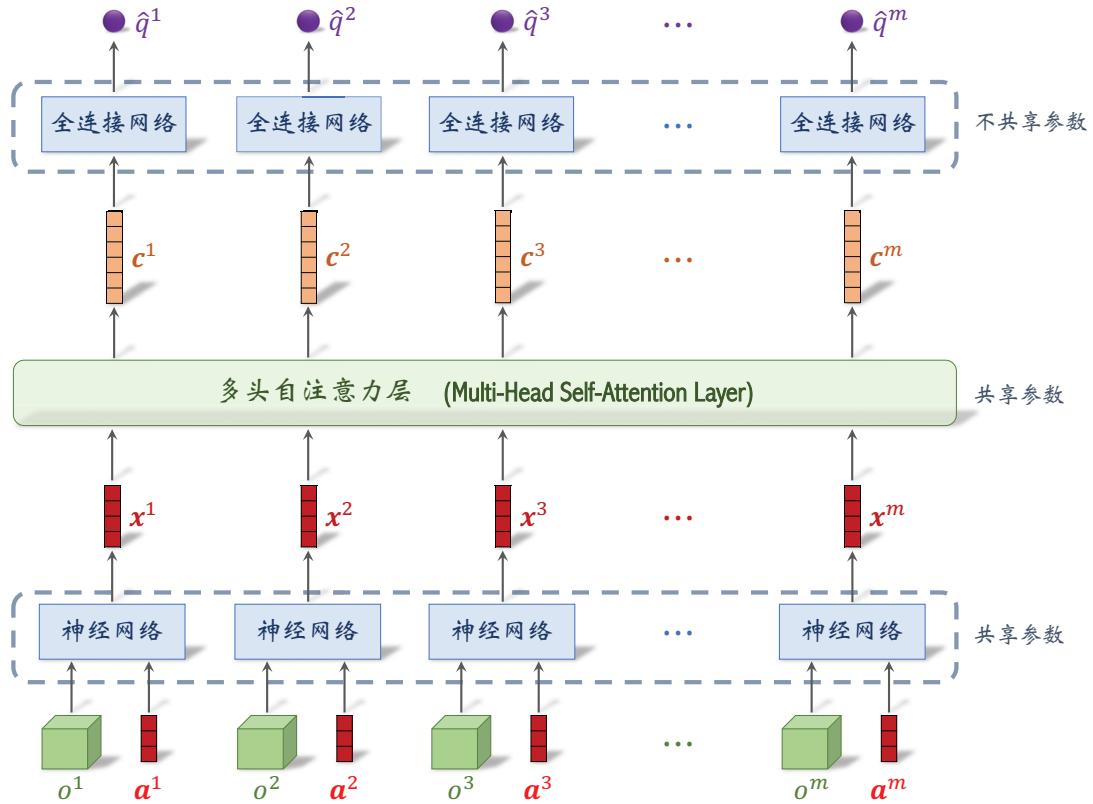


图 16.9: 带有自注意力的动作价值网络。图中的 $\hat{q}^i = q([o^1, \dots, o^m], [a^1, \dots, a^m]; \mathbf{w}^i)$ 是第 i 个动作价值网络的输出。

使用自注意力的中心化策略网络：对于“中心化训练 + 中心化决策”的系统架构，需要在中央控制器上部署 m 个策略网络，每个策略网络都需要知道所有 m 个智能体的观测 o^1, \dots, o^m 。

- 对于离散控制，第 i 号策略网络记作：

$$\hat{\mathbf{f}}^i = \pi(\cdot \mid [o^1, \dots, o^m]; \theta^i).$$

策略网络的输出是向量 $\hat{\mathbf{f}}^i$ ，它的维度是第 i 号动作空间的大小 $|\mathcal{A}^i|$ ， $\hat{\mathbf{f}}^i$ 的元素表示每种动作的概率。根据 $\hat{\mathbf{f}}^i$ 做随机抽样，得到动作 a^i ，第 i 号智能体执行这个动作。

- 对于连续控制，第 i 号策略网络记作：

$$\mathbf{a}^i = \mu([o^1, \dots, o^m]; \theta^i).$$

它的输出是动作 \mathbf{a}^i ，它是 d 维向量， d 是连续控制问题的自由度。第 i 号智能体执行动作 \mathbf{a}^i 。

不管是离散控制还是连续控制，上述两种策略网络中都可以使用自注意力层，神经网络的结构与图 16.8 中的 $v(s; \mathbf{w}^i)$ 几乎一样，唯一区别是神经网络的输出由实数 $\hat{v}^1, \dots, \hat{v}^m$

变成向量 $\hat{\mathbf{f}}^1, \dots, \hat{\mathbf{f}}^m$ 或者 $\mathbf{a}^1, \dots, \mathbf{a}^m$ 。

总结：自注意力机制在**非合作关系**的 MARL 中普遍适用。如果系统架构使用**中心化训练**，那么 m 个**价值网络**可以用一个神经网络实现，其中使用自注意力层。如果系统架构使用**中心化决策**，那么 m 个**策略网络**也可以实现成一个神经网络，其中使用自注意力层。在 m 较大的情况下，使用自注意力层对效果有较大的提升。

∽ 第十六章 相关文献 ∽

注意力机制 (Attention) 由 2015 年的论文 [6] 提出；这篇论文将注意力机制与 RNN 结合，大幅提升 RNN 在机器翻译任务上的表现。2017 年的论文 [110] 提出 Transformer 模型，去掉 RNN，只保留注意力，在机器翻译任务上取得了远优于 RNN 加注意力的表现。2019 年的论文 [52] 将注意力层用到多智能体的 Actor-Critic 中。

《深度强化学习》 2021-03-18 尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

第十七章 模仿学习

模仿学习 (Imitation Learning) 不是强化学习，而是强化学习的一种替代品。模仿学习与强化学习有相同的目的：两者的目的都是学习策略网络，从而控制智能体。模仿学习与强化学习有不同的原理：模仿学习向人类专家学习，目标是让策略网络做出的决策与人类专家相同；而强化学习利用环境反馈的奖励改进策略，目标是让累计奖励（即回报）最大化。

本章介绍三种常见的模仿学习方法：行为克隆 (Behavior Cloning)、逆向强化学习 (Inverse Reinforcement Learning)、生成判别模仿学习 (GAIL)。行为克隆不需要让智能体与环境交互，因此学习的“成本”很低；而逆向强化学习、生成判别模仿学习则需要让智能体与环境交互。

17.1 行为克隆

行为克隆 (Behavior Cloning) 是最简单的模仿学习。行为克隆的目的是模仿人的动作，学出一个随机策略网络 $\pi(a|s; \theta)$ 或者确定策略网络 $\mu(s; \theta)$ 。虽然行为克隆的目的与强化学习中的策略学习类似，但是行为克隆的本质是监督学习（分类或者回归），而不是强化学习。行为克隆通过模仿人类专家的动作来学习策略，而强化学习则是从奖励中学习策略。

模仿学习需要一个事先准备好的数据集，由（状态，动作）这样的二元组构成，记作：

$$\mathcal{X} = \{(s_1, a_1), \dots, (s_n, a_n)\}.$$

其中 s_j 是一个状态，而对应的 a_j 是人类专家基于状态 s_j 做出的动作。可以把 s_j 和 a_j 分别视作监督学习中的输入和标签。

17.1.1 连续控制问题

连续控制的意思是动作空间 \mathcal{A} 是连续集合，比如 $\mathcal{A} = [0, 360] \times [0, 180]$ 。我们搭建类似图 17.2 的确定策略网络，记作 $\mu(s; \theta)$ 。输入是状态 s ，输出是动作向量 a ，它的维度 d 是控制问题的自由度。

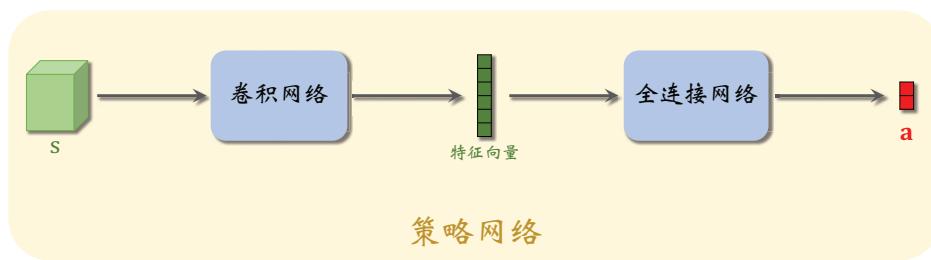


图 17.1：确定策略网络 $\mu(s; \theta)$ 的结构。输入是状态 s ，输出是动作 a 。

行为克隆用回归的方法训练确定策略网络。训练数据集 \mathcal{X} 中的二元组 (s, \mathbf{a}) 的意思是基于状态 s , 人做出动作 \mathbf{a} 。行为克隆鼓励策略网络的决策 $\mu(s; \theta)$ 接近人做出的动作 \mathbf{a} 。定义损失函数

$$L(s, \mathbf{a}; \theta) \triangleq \frac{1}{2} [\mu(s; \theta) - \mathbf{a}]^2.$$

损失函数越小, 说明策略网络的决策越接近人的动作。用梯度更新 θ :

$$\theta \leftarrow \theta - \beta \cdot \nabla_{\theta} L(s, \mathbf{a}; \theta),$$

这样可以让 $\mu(s; \theta)$ 更接近 \mathbf{a} 。

训练流程: 给定数据集 $\mathcal{X} = \{(s_j, \mathbf{a}_j)\}_{j=1}^n$ 。重复下面的随机梯度下降, 直到算法收敛:

1. 从序号 $\{1, \dots, n\}$ 中做均匀随机抽样, 把抽到的序号记作 j 。
2. 设当前策略网络参数为 θ_{now} 。把 s_j 、 \mathbf{a}_j 作为输入, 做反向传播计算梯度, 然后用梯度更新 θ :

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \beta \cdot \nabla_{\theta} L(s_j, \mathbf{a}_j; \theta_{\text{now}}).$$

17.1.2 离散控制问题

离散控制的意思是动作空间 \mathcal{A} 是离散集合, 例如 $\mathcal{A} = \{\text{左}, \text{右}, \text{上}\}$ 。我们搭建类似图 17.2 的策略网络, 记作 $\pi(a|s; \theta)$ 。输入是状态 s , 输出记作向量 f 。 f 的维度是 $|\mathcal{A}|$, 它的每个元素对应一个动作, 表示选择该动作的概率值。比如给定状态 s , 策略网络输出:

$$\begin{aligned} f_1 &= \pi(\text{左} | s; \theta) = 0.2, \\ f_2 &= \pi(\text{右} | s; \theta) = 0.1, \\ f_3 &= \pi(\text{上} | s; \theta) = 0.7. \end{aligned}$$

也就是说向量 $f = [0.2, 0.1, 0.7]^T$ 。

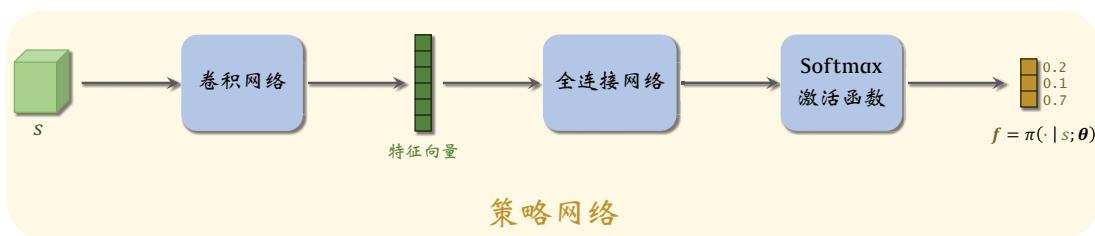


图 17.2: 策略网络 $\pi(a|s; \theta)$ 的神经网络结构。

行为克隆把策略网络 $\pi(a|s; \theta)$ 看做一个多类别分类器, 用监督学习的方法训练这个分类器。把训练数据集 \mathcal{X} 中的动作 a 看做类别标签, 用于训练分类器。需要对类别标签 a 做 One-Hot 编码, 得到 $|\mathcal{A}|$ 维的向量, 记作粗体字母 \bar{a} 。例如 $\mathcal{A} = \{\text{左}, \text{右}, \text{上}\}$, 那么

17.1 行为克隆

对动作的 One-Hot 编码就是：

$$\begin{aligned} a = \text{左} &\implies \bar{a} = [1; 0; 0], \\ a = \text{右} &\implies \bar{a} = [0; 1; 0], \\ a = \text{上} &\implies \bar{a} = [0; 0; 1]. \end{aligned}$$

向量 \bar{a} 与 f 都可以看做是离散的概率分布，可以用交叉熵 (Cross Entropy) 衡量两个分布的区别。交叉熵的定义是：

$$H(\bar{a}, f) \triangleq -\sum_{i=1}^{|A|} \bar{a}_i \cdot \ln f_i.$$

向量 \bar{a} 与 f 越接近，它们的交叉熵越小。用交叉熵作为损失函数：

$$H[\bar{a}, \pi(\cdot | s; \theta)],$$

用梯度更新参数 θ ：

$$\theta \leftarrow \theta - \beta \cdot \nabla_{\theta} H[\bar{a}, \pi(\cdot | s; \theta)].$$

这样可以使交叉熵减小，也就是说策略网络做出的决策 f 更接近人的动作 \bar{a} 。

训练流程： 给定数据集 $\mathcal{X} = \{(s_j, a_j)\}_{j=1}^n$ ，对所有的 a_j 做 One-Hot 编码，变成向量 \bar{a}_j 。重复下面的随机梯度下降，直到算法收敛：

1. 从序号 $\{1, \dots, n\}$ 中做均匀随机抽样，把抽到的序号记作 j 。
2. 设当前策略网络的参数是 θ_{now} 。把 s_j 、 \bar{a}_j 作为输入，做反向传播计算梯度，然后用梯度更新 θ ：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \beta \cdot \nabla_{\theta} H[\bar{a}_j, \pi(\cdot | s_j; \theta_{\text{now}})].$$

17.1.3 行为克隆与强化学习的对比

行为克隆不是强化学习。强化学习让智能体与环境交互，用环境反馈的奖励指导策略网络的改进，目的是最大化回报的期望。而行为克隆不需要与环境交互，而是利用事先准备好的数据集，用人类的动作指导策略网络的改进，目的是让策略网络的决策更像人类的决策。行为克隆的本质是监督学习（分类或者回归），而不是强化学习，因为行为克隆不需要与环境交互。

行为克隆训练出的策略网络通常效果不佳。人类不会探索奇怪的状态和动作，因此数据集上的状态和动作缺乏多样性。在数据集上做完行为克隆之后，智能体面对真实的环境，可能会见到陌生的状态，智能体的决策可能会很糟糕。行为克隆存在“错误累加”的缺陷。假如当前智能体的决策 a_t 不够好。那么下一时刻的状态 s_{t+1} 可能会比较罕见，于是智能体的决策 a_{t+1} 会很差；这又导致状态 s_{t+2} 非常奇怪，使得决策 a_{t+2} 更糟糕。行为克隆训练出的策略常会进入这种恶性循环。

强化学习效果通常优于行为克隆。如果用强化学习，那么智能体探索过各种各样的状态，尝试过各种各样的动作，知道面对各种状态时应该做什么决策。智能体通过探索，各种状态都见过，比行为克隆有更多的“人生经验”，因此表现会更好。强化学习在围棋、

电子游戏上的表现可以远超顶级人类玩家，而行为克隆却很难超越人类高手。

强化学习的一个缺点在于需要与环境交互，需要探索，而且会改变环境。举个例子，假如把强化学习应用到手术机器人，从随机初始化开始训练策略网络，至少要致死、致残几万个病人才能训练好策略网络。假如把强化学习应用到无人车，从随机初始化开始训练策略网络，至少要撞毁几万辆无人车才能训练好策略网络。假如把强化学习应用到广告投放，那么从初始化到训练好策略网络期间需要做探索，投放的广告会很随机，会严重降低广告收入。如果在真实物理世界应用强化学习，要考虑初始化和探索带来的成本。

行为克隆的优势在于离线训练，可以避免与真实环境的交互，不会对环境产生影响。假如用行为克隆训练手术机器人，只需要把人类医生的观测和动作记录下来，离线训练手术机器人，而不需要真的在病人身上做实验。尽管行为克隆效果不如强化学习，但是行为克隆的成本低。可以先用行为克隆初始化策略网络，而不是随机初始化，然后再做强化学习，这样可以减小对物理世界的有害影响。

17.2 逆向强化学习

逆向强化学习 (Inverse Reinforcement Learning, 缩写 IRL) 非常有名，但是在今天已经不常用了。下一节介绍的 GAIL 更简单，效果更好。本节只简单介绍 IRL 的主要思想，而不深入讲解其数学原理。

IRL 的基本设定：第一，IRL 假设智能体可以与环境交互¹，环境会根据智能体的动作更新状态，但是不会给出奖励。智能体与环境交互的轨迹是这样的：

$$s_1, a_1, s_2, a_2, s_3, a_3, \dots, s_n, a_n.$$

这种设定非常符合物理世界的实际情况。比如人类驾驶汽车，与物理环境交互，根据观测做出决策，得到上面公式中轨迹，轨迹中没有奖励。是不是汽车驾驶问题中没有奖励呢？其实是有奖励的。避免碰撞、遵守交通规则、尽快到达目的地，这些操作背后都有隐含的奖励，只是环境不会直接把奖励告诉我们而已。把奖励看做 (s_t, a_t) 的函数，记作 $R^*(s_t, a_t)$ 。

第二，IRL 假设我们可以把人类专家的策略 $\pi^*(a|s)$ 作为一个黑箱调用。黑箱的意思是我们不知道策略的解析表达式，但是可以使用黑箱策略控制智能体与环境交互，生成轨迹。IRL 假设人类学习策略 π^* 的方式与强化学习相同，都是最大化回报（即累计奖励）的期望，即

$$\pi^* = \max_{\pi} \mathbb{E}_{S_t, A_t, \dots, S_n, A_n} \left[\sum_{k=t}^n \gamma^{k-t} \cdot R^*(S_k, A_k) \right]. \quad (17.1)$$

因为 π^* 与奖励函数 $R^*(s, a)$ 密切相关，所以可以从 π^* 反推出 $R^*(s, a)$ 。

IRL 的基本思想：IRL 的目的是学到一个策略网络 $\pi(a|s; \theta)$ ，模仿人类专家的黑箱策略 $\pi^*(a|s)$ 。如图 17.3 所示，IRL 首先从 $\pi^*(a|s)$ 中学习其隐含的奖励函数 R^* ，然后利用奖励函数做强化学习，得到策略网络的参数 θ 。我们用神经网络 $R(s, a; \rho)$ 来近似奖励函数 R^* 。神经网络 R 的输入是 s 和 a ，输出是实数；我们需要学习它的参数 ρ 。



图 17.3

从黑箱策略反推奖励：假设人类专家的黑箱策略 $\pi^*(a|s)$ 满足公式 (17.1)，即 π^* 是应对奖励函数 R^* 的最优策略。对于不同的奖励函数 R^* ，则会有不同的 $\pi^*(a|s)$ 。是否能由 π^* 的决策反推出 R^* 呢？举个例子，图 17.4 是走格子的游戏，动作空间是 $\mathcal{A} = \{\text{上}, \text{下}, \text{左}, \text{右}\}$ 。两个表格表示两局游戏的状态，蓝色的箭头表示 π^* 做出的决策。请读者仔细观察，尝试推断游戏的奖励函数 R^* 。

既然蓝色箭头是最优策略做出的决策，那么沿着蓝色箭头走，可以最大化回报。我

¹注意，上一节的行为克隆无需智能体与环境交互。

们不难做出以下推断：

- 到达绿色格子有正奖励 r_+ ，原因是智能体尽量通过绿色格子。到达绿色格子的奖励只能被收集一次，否则智能体会反复回到绿色格子。
- 到达红色格子有负奖励 $-r_-$ ，因为智能体尽量避开红色格子。由于左图中智能体穿越两个红色格子去收集绿色奖励，说明 $r_+ \gtrsim 2r_-$ 。由于右图中智能体没有穿越四个红格子去收集绿色奖励，而是穿越一个红格子，说明 $r_+ \lesssim 3r_-$ 。
- 到达终点有正奖励 r_* ，因为智能体会尽力走到终点。由于右图中的智能体穿过红色格子，说明 $r_* > r_-$ 。
- 智能体尽量走最短路，说明每走一步，有一个负奖励 $-r_\rightarrow$ 。但是 r_\rightarrow 比较小，否则智能体不会绕路去收集绿色奖励。

注意，从智能体的轨迹中，只能大致推断出奖励函数，但是不可能推断出奖励 r_+ 、 $-r_-$ 、 r_* 、 r_\rightarrow 具体的大小。把四个奖励的数值同时乘以 10，根据新的奖励训练策略，最终学出的最优策略跟原来相同；这说明最优策略对应的奖励函数是不唯一的。

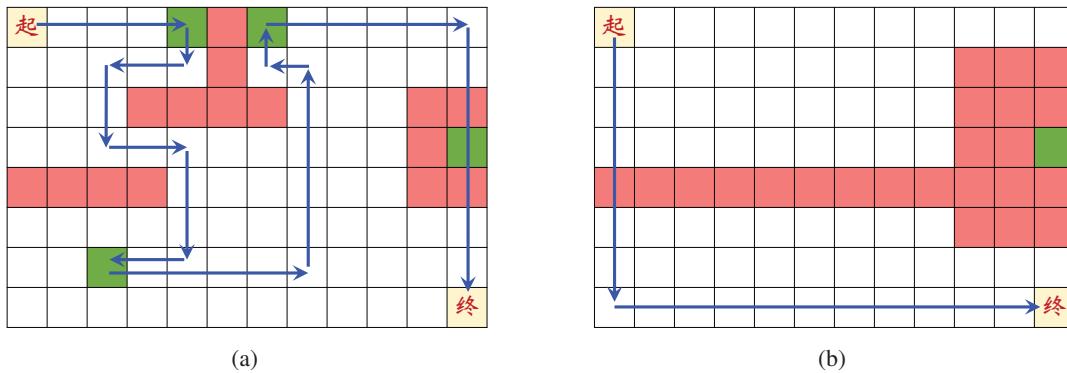


图 17.4：左右两张图表示走格子游戏的两个状态，图中蓝色箭头表示智能体的轨迹。

用奖励函数训练策略网络：假设我们已经学到了奖励函数 $R(s, a; \rho)$ ，那么就可以用它来训练一个策略网络。用策略网络 $\pi(a|s; \theta_{\text{now}})$ 控制智能体与环境交互，每次得到这样一条轨迹：

$$s_1, a_1, s_2, a_2, s_3, a_3, \dots, s_n, a_n,$$

轨迹中没有奖励。比如用策略网络控制无人车驾驶，得到的就是这样一条没有奖励的轨迹。好在我们已经从人类专家身上学到了奖励函数 $R(s, a; \rho)$ ，可以用 R 算出奖励：

$$\hat{r}_t = R(s_t, a_t; \rho), \quad \forall t = 1, \dots, n.$$

可以用任意策略学习方法更新策略网络参数 θ ，比如用 REINFORCE：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \sum_{t=1}^n \gamma^{t-1} \cdot \hat{u}_t \cdot \nabla_{\theta} \ln \pi(a|s; \theta_{\text{now}}).$$

公式中的 $\hat{u}_t \triangleq \sum_{k=t}^n \gamma^{k-t} \cdot \hat{r}_k$ 是近似回报。

更新奖励函数：具体该如何学习奖励函数 $R(s, a; \rho)$ 呢？因为我们用 $R(s, a; \rho)$ 来训练策略网络 $\pi(a|s; \theta)$ ，所以 $\pi(a|s; \theta)$ 依赖于 ρ 。IRL 的目标是让 $\pi(a|s; \theta)$ 尽量接近人类

17.2 逆向强化学习

专家的策略 $\pi^*(a|s)$ 。因此要寻找参数 ρ 使得学到的 $\pi(a|s; \theta)$ 最接近 $\pi^*(a|s)$ 。学习 ρ 的方法有很多种，本书不具体介绍了，有兴趣的读者可以阅读相关的文献。

17.3 生成判别模仿学习 (GAIL)

生成判别模仿学习 (Generative Adversarial Imitation Learning, 缩写 GAIL) 需要让智能体与环境交互，但是无法从环境获得奖励²。GAIL 还需要收集人类专家的决策记录（即很多条轨迹）。GAIL 的目标是学习一个策略网络，使得判别器无法区分一条轨迹是策略网络的决策还是人类专家的决策。

17.3.1 生成判别网络 (GAN)

GAIL 的设计基于生成判别网络 (Generative Adversarial Network, 缩写 GAN)。本小节简单介绍 GAN 的基础知识。生成器 (Generator) 和判别器 (Discriminator) 各是一个神经网络。生成器负责生成假的样本，而判别器负责判定一个样本是真是假。举个例子，在人脸数据集上训练生成器和判别器，那么生成器的目标是生成假的人脸图片，可以骗过判别器；而判别器的目标是判断一张图片是真实的还是生成的。理想情况下，当训练结束的时候，判别器的分类准确率是 50%，意味着生成器的输出已经以假乱真。

生成器记作 $a = G(s; \theta)$ ，其中 θ 是参数。它的输入是向量 s ，向量的每一个元素从均匀分布 $U(-1, 1)$ 或标准正态分布 $\mathcal{N}(0, 1)$ 中抽取。生成器的输出是数据（比如图片） x 。生成器通常是一个深度神经网络，其中可能包含卷积层 (Convolution)、反卷积层 (Transposed Convolution)、上采样层 (Upsampling)、全连接层 (Dense) 等。生成器的具体实现取决于具体的问题。

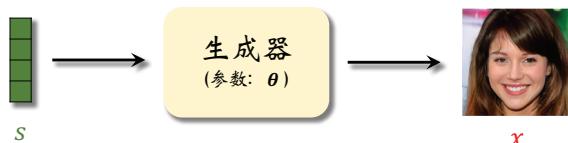


图 17.5：生成器 $a = G(s; \theta)$ 。

判别器记作 $\hat{p} = D(x; \phi)$ ，其中 ϕ 是参数。它的输入是图片 x ；输出 \hat{p} 是介于 0 到 1 之间的概率值，0 表示“假的”，1 表示“真的”。判别器的功能是二分类器，实现方法很简单。判别器主要由卷积层、池化层 (Pooling)、全连接层等组成。



图 17.6：判别器 $\hat{p} = D(x; \phi)$ 。

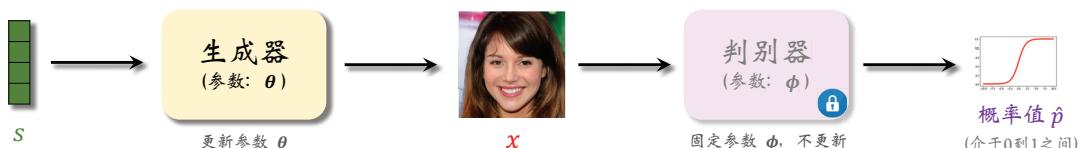


图 17.7：训练生成器 $G(s; \theta)$ 。

训练生成器：将生成器与判别器相连，如图 17.7 所示。固定住判别器的参数，只更

²GAIL 和 IRL 都需要让智能体与环境交互，而行为克隆不需要。

17.3 生成判别模仿学习 (GAIL)

新生成器的参数 θ , 使得生成的图片 $x = G(s; \theta)$ 在判别器的眼里更像真的。对于任意一个随机生成的向量 s , 应该改变 θ , 使得判别器的输出 $\hat{p} = D(x; \phi)$ 尽量接近 1。可以用交叉熵作为损失函数:

$$E(s; \theta) = \ln \left[1 - \underbrace{D(x; \phi)}_{\text{越大越好}} \right]; \quad \text{s.t. } x = G(s; \theta).$$

判别器的输出 $\hat{p} = D(x; \phi)$ 是介于 0 到 1 之间的数。 \hat{p} 越接近 1, 则损失函数 $E(s; \theta) = \ln(1 - \hat{p})$ 越小。训练生成器参数 θ 的时候, 我们希望 \hat{p} 尽量接近 1, 所以应当更新 θ 使得 $E(s; \theta)$ 减小。做一次梯度下降更新 θ :

$$\theta \leftarrow \theta - \beta \cdot \nabla_{\theta} E(s; \theta).$$

此处的 β 是学习率, 需要用户手动调。

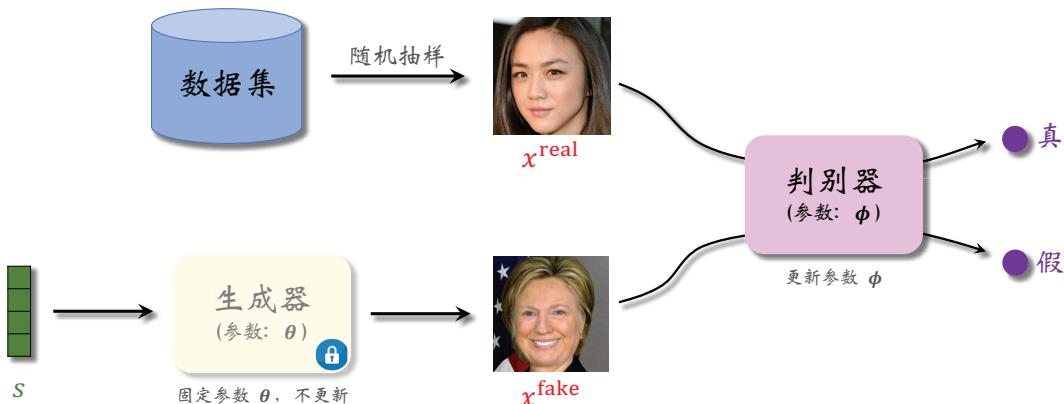


图 17.8: 训练判别器 $D(x; \phi)$ 。

训练判别器: 判别器的本质是个二分类器, 它的输出值 $\hat{p} = D(x; \phi)$ 表示对真伪的预测; \hat{p} 接近 1 表示“真”, \hat{p} 接近 0 表示“假”。判别器的训练如图 17.8 所示。从真实数据集中抽取一个样本, 记作 x^{real} 。再随机生成一个向量 s , 用生成器生成 $x^{\text{fake}} = G(s; \theta)$ 。训练判别器的目标是改进参数 ϕ , 让 $D(x^{\text{real}}; \phi)$ 更接近 1 (真), 让 $D(x^{\text{fake}}; \phi)$ 更接近 0 (假)。也就是说让判别器的分类结果更准确, 更好区分真实图片和生成的假图片。可以用交叉熵作为损失函数:

$$F(x^{\text{real}}, x^{\text{fake}}; \phi) = \ln \left[1 - \underbrace{D(x^{\text{real}}; \phi)}_{\text{越大越好}} \right] + \ln \underbrace{D(x^{\text{fake}}; \phi)}_{\text{越小越好}}.$$

判别器的判断越准确, 则损失函数 $F(x^{\text{real}}, x^{\text{fake}}; \phi)$ 越小。为什么呢?

- 判别器越相信 x^{real} 为真, 则 $D(x^{\text{real}}; \phi)$ 越大, 那么公式中 $\ln[1 - D(x^{\text{real}}; \phi)]$ 越小。
- 判别器越相信 x^{fake} 为假, 则 $D(x^{\text{fake}}; \phi)$ 越小, 那么公式中 $\ln D(x^{\text{fake}}; \phi)$ 越小。

为了减小损失函数 F , 可以做一次梯度下降更新判别器参数 ϕ :

$$\phi \leftarrow \phi - \eta \cdot \nabla_{\phi} F(x^{\text{real}}, x^{\text{fake}}; \phi).$$

此处的 η 是学习率, 需要用户手动调。

批量随机梯度 (Mini-Batch SGD): 上述训练生成器和判别器的方式其实是随机梯度下降 (SGD)，每次只用一个样本。实践中，不妨每次用一个批量 (Batch) 的样本，比如用 $b = 16$ 个，那么会计算出 b 个梯度。用 b 个梯度的平均去更新生成器和判别器。

训练流程: 实践中，要同时训练生成器和判别器，让两者同时进步。³ 每一轮要更新一次生成器，更新一次判别器。设当前生成器、判别器的参数分别为 θ_{now} 和 ϕ_{now} 。

1. (从均匀分布或正态分布中) 随机抽样 b 个向量: s_1, \dots, s_b 。
2. 用生成器生成假样本: $x_j^{\text{fake}} = G(s_j; \theta_{\text{now}})$, $\forall j = 1, \dots, b$ 。
3. 从训练数据集中随机抽样 b 个真样本: $x_1^{\text{real}}, \dots, x_b^{\text{real}}$ 。
4. 更新生成器 $G(s; \theta)$ 的参数:

(a). 计算平均梯度:

$$\mathbf{g}_\theta = \frac{1}{b} \sum_{j=1}^b \nabla_\theta E(s_j; \theta_{\text{now}}).$$

(b). 做梯度下降更新生成器参数: $\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \beta \cdot \mathbf{g}_\theta$ 。

5. 更新判别器 $D(x; \phi)$ 的参数:

(a). 计算平均梯度:

$$\mathbf{g}_\phi = \frac{1}{b} \sum_{j=1}^b \nabla_\phi F(x_j^{\text{real}}, x_j^{\text{fake}}; \phi_{\text{now}}).$$

(b). 做梯度下降更新判别器参数: $\phi_{\text{new}} \leftarrow \phi_{\text{now}} - \eta \cdot \mathbf{g}_\phi$ 。

17.3.2 GAIL 的生成器和判别器

训练数据: GAIL 的训练数据是被模仿的对象（比如人类专家）操作智能体得到的轨迹，记作

$$\tau = [s_1, a_1, s_2, a_2, \dots, s_m, a_m].$$

数据集中有 k 条轨迹，把数据集记作:

$$\mathcal{X} = \{\tau^{(1)}, \tau^{(2)}, \dots, \tau^{(k)}\}.$$

生成器: 上一小节中的 GAN 的生成器记作 $x = G(s; \theta)$ ，它的输入 s 是个随机抽取的向量，输出 x 是一个数据点（比如一张图片）。本小节的 GAIL 的生成器是策略网络 $\pi(a|s; \theta)$ ，如图 17.9 所示。策略网络的输入是状态 s ，输出是一个向量：

$$\mathbf{f} = \pi(\cdot | s; \theta).$$

输出向量 \mathbf{f} 的维度是动作空间的大小 A ，它的每个元素对应一个动作，表示执行该动作的概率。给定初始状态 s_1 ，并让智能体与环境交互，可以得到一条轨迹：

$$\tau = [s_1, a_1, s_2, a_2, \dots, s_n, a_n].$$

其中动作是根据策略网络抽样得到的: $a_t \sim \pi(\cdot | s_t; \theta)$, $\forall t = 1, \dots, n$; 下一时刻的状态

³不能让判别器比生成器进步快太多，否则训练会失败。假如判别器的准确率是 100%，那么无论生成器的输出 x 是什么，总被判别为“假”，那么生成器就不知道什么样的 x 更像真的，因而无从改进。

17.3 生成判别模仿学习 (GAIL)

是环境根据状态转移函数计算出来的: $s_{t+1} \sim p(\cdot | s_t, a_t), \forall t = 1, \dots, n$ 。

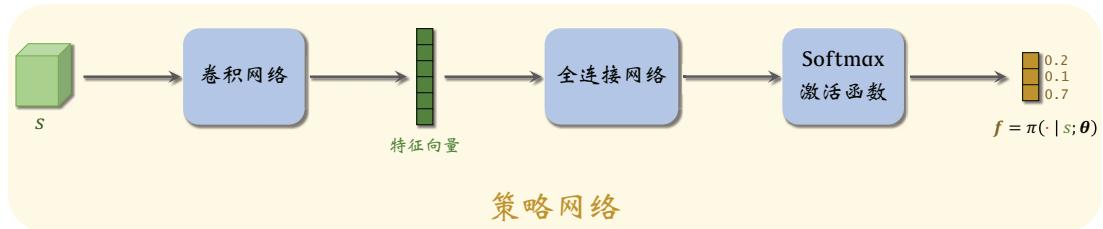


图 17.9: 策略网络 $\pi(a|s; \theta)$ 的神经网络结构。输入是状态 s , 输出是动作空间 \mathcal{A} 中每个动作的概率值。

判别器: GAIL 的判别器记作 $D(s, a; \phi)$, 它的结构如图 17.10 所示。判别器的输入是状态 s , 输出是一个向量:

$$\hat{p} = D(s, \cdot | \phi).$$

输出向量 \hat{p} 的维度是动作空间的大小 \mathcal{A} , 它的每个元素对应一个动作 a , 把一个元素记作:

$$\hat{p}_a = D(s, a; \phi) \in (0, 1), \quad \forall a \in \mathcal{A}.$$

\hat{p}_a 接近 1 表示 (s, a) 为“真”, 即动作 a 是人类专家做的。 \hat{p}_a 接近 0 表示 (s, a) 为“假”, 即策略网络生成的。



图 17.10: 判别器 $D(s, a; \phi)$ 的神经网络结构。输入是状态 s 。输出向量的维度等于 $|\mathcal{A}|$, 每个元素对应一个动作, 每个元素值都介于 0 到 1 之间。

17.3.3 GAIL 的训练

训练的目的是让生成器 (即策略网络) 生成的轨迹与数据集中的轨迹 (即被模仿对象的轨迹) 一样好。在训练结束的时候, 判别器无法区分生成的轨迹与数据集里的轨迹。

训练生成器: 设 θ_{now} 是当前策略网络的参数。用策略网络 $\pi(a|s; \theta_{\text{now}})$ 控制智能体与环境交互, 得到一条轨迹:

$$\tau = [s_1, a_1, s_2, a_2, \dots, s_n, a_n].$$

判别器可以评价 (s_t, a_t) 有多真实; $D(s_t, a_t; \phi)$ 越大, 说明 (s_t, a_t) 在判别器的眼里越真实。把

$$u_t = \ln D(s_t, a_t; \phi)$$

作为第 t 步的回报； u_t 越大，则说明 (s_t, a_t) 越真实。我们有这样一条轨迹：

$$s_1, a_1, u_1, \quad s_2, a_2, u_2, \quad \dots, \quad s_n, a_n, u_n.$$

于是可以用 TRPO 来更新策略网络。设当前策略网络的参数为 θ_{now} 。定义目标函数：

$$\tilde{L}(\theta | \theta_{\text{now}}) \triangleq \frac{1}{n} \sum_{t=1}^n \frac{\pi(a_t | s_t; \theta)}{\pi(a_t | s_t; \theta_{\text{now}})} \cdot u_t.$$

求解下面的带约束的最大化问题，得到新的参数：

$$\theta_{\text{new}} = \underset{\theta}{\operatorname{argmax}} \tilde{L}(\theta | \theta_{\text{now}}); \quad \text{s.t. } \operatorname{dist}(\theta_{\text{now}}, \theta) \leq \Delta. \quad (17.2)$$

此处的 dist 衡量 θ_{now} 与 θ 的区别， Δ 是一个需要调的超参数。TRPO 的详细解释见第 9.1 节。

训练判别器：训练判别器的目的是让它能区分真的轨迹与生成的轨迹。从训练数据集中均匀抽样一条轨迹，记作

$$\tau^{\text{real}} = [s_1^{\text{real}}, a_1^{\text{real}}, \dots, s_m^{\text{real}}, a_m^{\text{real}}].$$

用策略网络控制智能体与环境交互，得到一条轨迹，记作

$$\tau^{\text{fake}} = [s_1^{\text{fake}}, a_1^{\text{fake}}, \dots, s_n^{\text{fake}}, a_n^{\text{fake}}].$$

公式中的 m 、 n 分别是两条轨迹的长度。

训练判别器的时候，要鼓励判别器做出准确的判断。我们希望判别器知道 $(s_t^{\text{real}}, a_t^{\text{real}})$ 是真的，所以应该鼓励 $D(s_t^{\text{real}}, a_t^{\text{real}}; \phi)$ 尽量大。我们希望判别器知道 $(s_t^{\text{fake}}, a_t^{\text{fake}})$ 是假的，所以应该鼓励 $D(s_t^{\text{fake}}, a_t^{\text{fake}}; \phi)$ 尽量小。定义损失函数

$$F(\tau^{\text{real}}, \tau^{\text{fake}}; \phi) = \underbrace{\frac{1}{m} \sum_{t=1}^m \ln [1 - D(s_t^{\text{real}}, a_t^{\text{real}}; \phi)]}_{D \text{ 的输出越大, 这一项越小}} + \underbrace{\frac{1}{n} \sum_{t=1}^n \ln D(s_t^{\text{fake}}, a_t^{\text{fake}}; \phi)}_{D \text{ 的输出越小, 这一项越小}}$$

我们希望损失函数尽量小，也就是说判别器能区分开真假轨迹。可以做梯度下降来更新参数 ϕ ：

$$\phi \leftarrow \phi - \eta \cdot \nabla_{\phi} F(\tau^{\text{real}}, \tau^{\text{fake}}; \phi). \quad (17.3)$$

这样可以让损失函数减小，让判别器更能区分开真假轨迹。

训练流程：每一轮训练更新一个生成器，更新一次判别器。训练重复以下步骤，直到收敛。设当前生成器和判别器的参数分别为 θ_{now} 和 ϕ_{now} 。

1. 从训练数据集中均匀抽样一条轨迹，记作

$$\tau^{\text{real}} = [s_1^{\text{real}}, a_1^{\text{real}}, \dots, s_m^{\text{real}}, a_m^{\text{real}}].$$

2. 用策略网络 $\pi(a | s; \theta_{\text{now}})$ 控制智能体与环境交互，得到一条轨迹，记作

$$\tau^{\text{fake}} = [s_1^{\text{fake}}, a_1^{\text{fake}}, \dots, s_n^{\text{fake}}, a_n^{\text{fake}}].$$

3. 用判别器评价策略网络的决策是否真实：

$$u_t = \ln D(s_t^{\text{fake}}, a_t^{\text{fake}}; \phi_{\text{now}}), \quad \forall t = 1, \dots, n.$$

4. 把 τ^{fake} 和 u_1, \dots, u_n 作为输入，用公式 (17.2) 更新策略网络参数，得到 θ_{new} 。

5. 把 τ^{real} 和 τ^{fake} 作为输入，用公式 (17.3) 更新判别器参数，得到 ϕ_{new} 。

∽ 第十七章 相关文献 ∽

行为克隆 (Behavior Cloning) 这个概念很早就出现在人工智能领域，比如 1995 年的论文 [7]、1997 年的论文 [20]。论文 [80, 99] 研究了行为克隆的理论误差，指出行为克隆会让错误累加。行为克隆也叫做 Learning from Demonstration (LfD) [5]。LfD 这个名字最早由 1997 年的论文提出 [83]。

逆向强化学习 (Inverse Reinforcement Learning) 这个问题首先由 Ng 和 Russell 2000 年的论文提出 [74]。这个问题原本是指“从最优策略中推断出奖励函数”。Abbeel 和 Ng 2004 年的论文 [1] 提出从人类专家的策略中反向学习出奖励函数，然后用奖励函数训练策略函数；这种方法被称作学徒学习 (Apprenticeship Learning)。本书第 17.2 节的内容主要基于学徒学习的思想。逆向强化学习的方法有很多种，比如 [16, 36, 59, 98, 125]。

生成判别模仿学习 (Generative Adversarial Imitation Learning) 由 Ho 和 Ermon 在 2016 提出 [47]。它主要基于生成判别网络 (Generative Adversarial Network，缩写 GAN)。GAN 由 Goodfellow 等人在 2014 年提出 [41]。

第十八章 AlphaGo 与蒙特卡洛树搜索

之前章节介绍的强化学习方法都是无模型的强化学习 (Model-Free)，包括价值学习 (Value-Based) 和策略学习 (Policy-Based)。本章介绍的蒙特卡洛树搜索 (Monte Carlo Tree Search，缩写 MCTS) 是一种基于模型的强化学习方法 (Model-Based)。MCTS 比价值学习和策略学习更难理解，所以本章结合 AlphaGo 讲解 MCTS。

AlphaGo 的字面意思是“围棋王”，俗称“阿尔法狗”，它是世界上第一个打败人类围棋冠军的 AI。在 2015 年 10 月，AlphaGo 以 5 : 0 战胜欧洲围棋冠军、职业二段选手樊麾。在 2016 年 3 月，AlphaGo 以 4 : 1 战胜世界冠军李世石。2017 年新版的 AlphaGo Zero 更胜一筹，以 100 : 0 战胜 AlphaGo。

AlphaGo 依靠 MCTS 做决策，而决策的过程中需要策略网络和价值网络的辅助。第 18.1 节用强化学习的语言描述围棋的状态和动作，并且构造策略网络和价值网络。第 18.2 节详细讲解 MCTS 的决策过程。第 18.3 节讲解 AlphaGo 2016 版与 AlphaGo Zero 是如何训练策略网络和价值网络的。

18.1 动作、状态、策略网络、价值网络

围棋的棋盘是 19×19 的网格，可以在两条线交叉的地方放置棋子，一共有 361 个可以放置棋子的位置。两个玩家一方用黑色棋子，另一方用白色棋子，两方交替往棋盘上放置棋子。棋盘上有 361 个可以放置棋子的位置，因此动作空间是 $\mathcal{A} = \{1, \dots, 361\}$ 。比如动作 $a = 123$ 的意思是在第 123 号位置上放棋子。

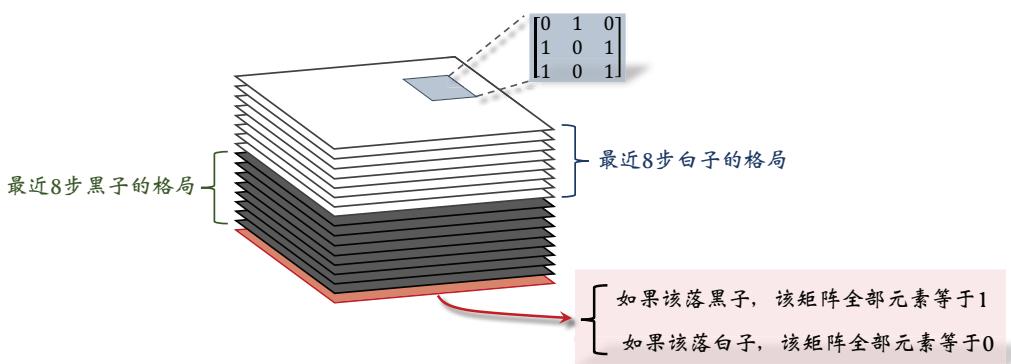


图 18.1：状态可以表示为 $19 \times 19 \times 17$ 的张量。

AlphaGo 2016 版本使用 $19 \times 19 \times 48$ 的张量 (Tensor) 表示一个状态。AlphaGo Zero 使用 $19 \times 19 \times 17$ 的张量表示一个状态。本书只解释后者；见图 18.1。下面解释 $19 \times 19 \times 17$ 的状态张量的意义。

- 张量每个切片 (Slice) 是 19×19 的矩阵，对应 19×19 的棋盘。一个 19×19 的矩阵可以表示棋盘上所有黑子的位置。如果一个位置上有黑子，矩阵对应的元素就是

1，否则就是0。同样的道理，用一个 19×19 的矩阵来表示当前棋盘上所有白子的位置。

- 张量中一共有17个这样的矩阵；17是这样得来的。记录最近8步棋盘上黑子的位置，需要8个矩阵。同理，还需要8个矩阵记录白子的位置。还另外需要一个矩阵表示该哪一方下棋；如果该下黑子，那么该矩阵元素全部等于1；如果该下白子，那么该矩阵的元素全都等于0。

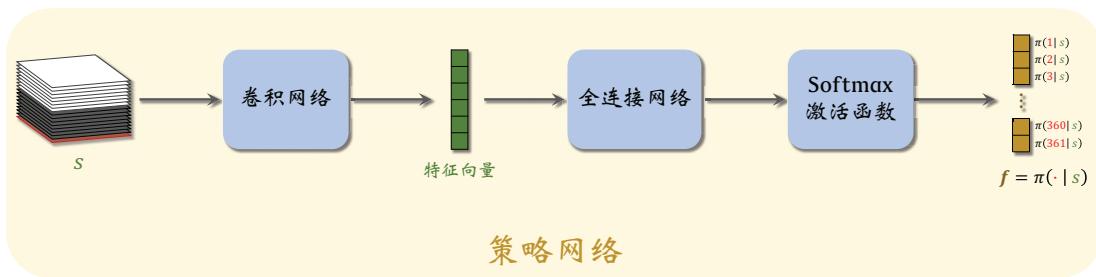


图 18.2: 策略网络的示意图。

策略网络 $\pi(a|s; \theta)$ 的结构如图 18.2 所示。策略网络的输入是 $19 \times 19 \times 17$ 的状态 s 。策略网络的输出是 361 维的向量 f ，它的每个元素对应一个动作（即在棋盘上一个位置放棋子）。向量 f 所有元素都是正数，而且相加等于 1。

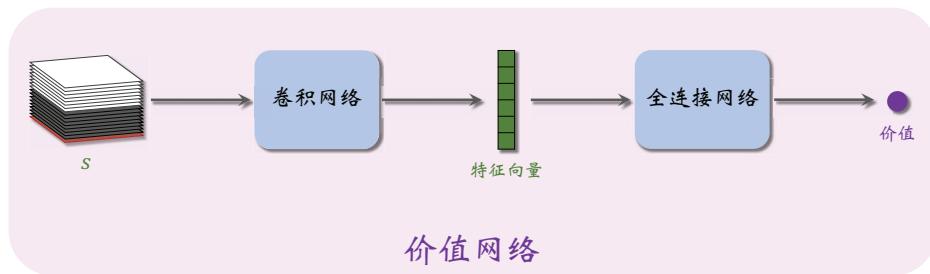


图 18.3: 价值网络的示意图。

AlphaGo 还有一个价值网络 $v(s; w)$ ，它是对状态价值函数 $V_\pi(s)$ 的近似。价值网络的结构如图 18.3 所示。价值网络的输入是 $19 \times 19 \times 17$ 的状态 s 。价值网络的输出是一个实数，它的大小评价当前状态 s 的好坏。

策略网络和价值的输入相同，都是状态 s 。它们都用多个卷积层把 s 映射到特征向量。因此可以让策略网络和价值网络共用卷积层。训练策略网络和价值网络的方法在第 18.3 节解释。

18.2 蒙特卡洛树搜索 (MCTS)

假设此时已经训练好了策略网络 $\pi(a|s; \theta)$ 和价值网络 $v(s; w)$ 。AlphaGo 真正跟人下棋的时候，做决策的不是策略网络或者价值网络，而是蒙特卡洛树搜索 (Monte Carlo Tree Search)，缩写 MCTS。MCTS 不需要训练，可以直接做决策。训练策略网络和价值网络的目的是辅助 MCTS。本节中假设策略网络和价值网络已经训练好，可以直接用；下一节再具体讲解策略网络和价值网络的训练。

18.2.1 MCTS 的基本思想

思考一个问题：人类玩家是怎么下围棋、象棋、五子棋的？人类玩家通常都会向前看几步；越是高手，看得越远。假如现在该我放棋子了，我应该思考这样的问题：当前有几个貌似可行的走法，假如我的动作是 $a_t = 234$ ，对手会怎么走呢？假如接下来对手把棋子放在 $a'_t = 30$ 的位置上，那我下一步的动作 a_{t+1} 应该是什么呢？做当前决策之前，我需要在大脑里做这样的预判，确保几步以后我很可能会占优势。如果我只根据当前格局做判断，不往前看，我肯定赢不了高手。同理，AI 下棋也应该向前看，应该枚举未来可能发生的情况，从而判断当前执行什么动作的胜算最大；这样做远好于用策略网络计算一个动作。

MCTS 的基本原理就是向前看，模拟未来可能发生的情况，从而找出当前最优的动作。AlphaGo 每走一步棋，都要用 MCTS 做成千上万次模拟，从而判断出哪个动作的胜算最大。做模拟的基本思想如下。假设当前有三种看起来很好的动作。每次模拟的时候从三种动作中选出一种，然后将一局游戏进行到底，从而知晓胜负。（只是计算机做模拟而已，不是真的跟对手下完一局。）重复成千上万次模拟，统计一下每种动作的胜负频率，发现三种动作胜率分别是 48%、56%、52%。那么 AlphaGo 应当执行第二种动作，因为它的胜算最大。以上只是 MCTS 的基本想法，实际做起来有很多难点需要解决。

18.2.2 MCTS 的四个步骤

MCTS 的每一次模拟选出一个动作 a ，执行这个动作，然后把一局游戏进行到底，用胜负来评价这个动作的好坏。MCTS 的每一次模拟分为四个步骤：选择 (Selection)、扩展 (Expansion)、求值 (Evaluation)、回溯 (Backup)。

第一步——选择 (Selection): 观测棋盘上当前的格局，找出所有空位，然后判断其中哪些位置符合围棋规则；每个符合规则的位置对应一个可行的动作。每一步至少有几十、甚至上百个可行的动作；假如挨个搜索和评估所有可行动作，计算量会大到无法承受。虽然有几十、上百个可行动作，好在只有少数几个动作有较高的胜算。第一步——选择——的目的就是找出胜算较高的动作，只搜索这些好的动作，忽略掉其他的动作。

如何判断动作 a 的好坏呢？有两个指标：第一，动作 a 的胜率；第二，策略网络给

动作 a 的评分（概率值）。用下面这个分值评价 a 的好坏：

$$\text{score}(a) \triangleq Q(a) + \frac{\eta}{1+N(a)} \cdot \pi(a|s; \theta). \quad (18.1)$$

此处的 η 是个需要调的超参数。公式中 $N(a)$ 、 $Q(a)$ 的定义如下：

- $N(a)$ 是动作 a 已经被访问过的次数。初始的时候，对于所有的 a ，令 $N(a) \leftarrow 0$ 。动作 a 每被选中一次，我们就把 $N(a)$ 加一： $N(a) \leftarrow N(a) + 1$ 。
- $Q(a)$ 是之前 $N(a)$ 次模拟算出来的动作价值，主要由胜率和价值函数决定。 $Q(a)$ 的初始值是 0；动作 a 每被选中一次，就会更新一次 $Q(a)$ ；后面会详解。

可以这样理解公式 (18.1)：

- 如果动作 a 还没被选中过，那么 $Q(a)$ 和 $N(a)$ 都等于零，因此可得

$$\text{score}(a) = \eta \cdot \pi(a|s; \theta),$$

也就是说完全由策略网络评价动作 a 的好坏。

- 如果动作 a 已经被选中过很多次，那么 $N(a)$ 就很大，导致策略网络在 $\text{score}(a)$ 中的权重降低。当 $N(a)$ 很大的时候，有

$$\text{score}(a) \approx Q(a),$$

此时主要基于 $Q(a)$ 判断 a 的好坏，而策略网络已经无关紧要。

- 系数 $\frac{1}{1+N(a)}$ 的另一个作用是鼓励探索，也就是让被选中次数少的动作有更多的机会被选中。假如两个动作有相近的 Q 分数和 π 分数，那么被选中次数少的动作的 score 会更高。

MCTS 根据公式 (18.1) 算出所有动作的分数 $\text{score}(a)$, $\forall a$ 。MCTS 选择分数最高的动作。图 18.4 的例子中有 3 个可行动作，分数分别为 0.4、0.3、0.5。第三个动作分数最高，会被选中。这一轮模拟会执行这个动作（只是模拟而已，不是 AlphaGo 真的走一步棋）。

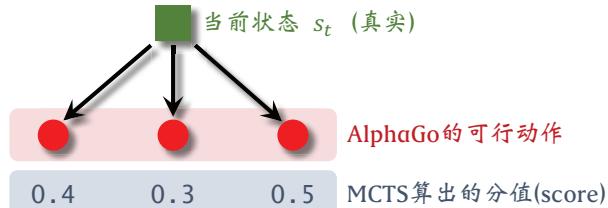


图 18.4：假设有 3 个可行动作，根据公式 (18.1) 算出它们的分数。

第二步——扩展 (Expansion): 把第一步选中的动作记作 a_t ，它只是个假想的动作，只在“模拟器”中执行，而不是 AlphaGo 真正执行的动作。AlphaGo 需要考虑这样一个问题：假如它执行动作 a_t ，那么对手会执行什么动作呢？对手肯定不会把自己的想法告诉 AlphaGo，那么 AlphaGo 只能自己猜测对手的动作。AlphaGo 可以“推己及人”：如果 AlphaGo 认为几个动作很好，对手也会这么认为。所以 AlphaGo 用策略网络模拟对手，根据策略网络随机抽样一个动作：

$$a'_t \sim \pi(\cdot | s'_t; \theta).$$

此处的状态 s' 是站在对手的角度观测到的棋盘上的格局，动作 a'_t 是（假想）对手选择

18.2 蒙特卡洛树搜索 (MCTS)

的动作。图 18.5 的例子中对手有四种可行动作，AlphaGo 用策略网络算出每个动作的概率值，然后根据概率值随机抽样一个对手的动作，记作 a'_t 。假设根据概率值 0.1, 0.3, 0.2, 0.4 做随机抽样，选中第二种动作；见图 18.6。从 AlphaGo 的角度来看，对手的动作就是 AlphaGo 新的状态。

AlphaGo 需要在模拟中跟对手将一局游戏进行下去，所以需要一个模拟器（即环境）。在模拟器中，AlphaGo 每执行一个动作 a_k ，模拟器就会返回一个新的状态 s_{k+1} 。想要搭建一个好的模拟器，关键在于使用正确的状态转移函数 $p(s_{k+1}|s_k, a_k)$ ；如果状态转移函数与事实偏离太远，那么用模拟器做 MCTS 是毫无意义的。

AlphaGo 模拟器利用了围棋游戏的对称性：AlphaGo 的策略，在对手看来是状态转移函数；对手的策略，在 AlphaGo 看来是状态转移函数。最理想的情况下，模拟器的状态转移函数是对手的真实策略；然而 AlphaGo 并不知道对手的真实策略。AlphaGo 退而求其次，用 AlphaGo 自己训练出的策略网络 π 代替对手的策略，作为模拟器的状态转移函数。

想要用 MCTS 做决策，必须要有模拟器，而搭建模拟器的关键在于构造正确的状态转移函数 $p(s_{k+1}|s_k, a_k)$ 。从搭建模拟器的角度来看，围棋是非常简单的问题：由于围棋的对称性，可以用策略网络作为状态转移函数。但是对于大多数的实际问题，构造状态转移函数是非常困难的。比如机器人、无人机等应用，状态转移的构造需要物理模型，要考虑到力、运动、以及外部世界的干扰。如果物理模型不够准确，导致状态转移函数偏离事实太远，那么 MCTS 的模拟结果就不可靠。

第三步——求值 (Evaluation): 从状态 s_{t+1} 开始，双方都用策略网络 π 做决策，在模拟器中交替落子，直到分出胜负；见图 18.7。AlphaGo 基于状态 s_k ，根据策略网络抽样得到动作

$$a_k \sim \pi(\cdot | s_k; \theta).$$

对手基于状态 s'_k （从对手角度观测到的棋盘上的格局），根据策略网络抽样得到动作

$$a'_k \sim \pi(\cdot | s'_k; \theta).$$

当这局游戏结束时，可以观测到奖励 r 。如果 AlphaGo 胜利，则 $r = +1$ ，否则 $r = -1$ 。

回顾一下，棋盘上真实的状态是 s_t ，AlphaGo 在模拟器中执行动作 a_t ，然后模拟器中的对手执行动作 a'_t ，带来新的状态 s_{t+1} 。状态 s_{t+1} 越好，则这局游戏胜算越大。

- 如果 AlphaGo 赢得这局模拟 ($r = +1$)，则说明 s_{t+1} 可能很好；如果输了 ($r = -1$)，则说明 s_{t+1} 可能不好。因此，奖励 r 可以反映出 s_{t+1} 的好坏。
- 此外，还可以用价值网络 v 评价状态 s_{t+1} 的好坏。价值 $v(s_{t+1}; \mathbf{w})$ 越大，则说明状态 s_{t+1} 越好。

奖励 r 是模拟获得的胜负，是对 s_{t+1} 很可靠的评价，但是随机性太大。价值网络的评估 $v(s_{t+1}; \mathbf{w})$ 没有 r 可靠，但是价值网络更稳定、随机性小。AlphaGo 的解决方案是把奖励 r 与价值网络的输出 $v(s_{t+1}; \mathbf{w})$ 取平均，记作：

$$V(s_{t+1}) \triangleq \frac{r + v(s_{t+1}; \mathbf{w})}{2},$$

把它记录下来，作为对状态 s_{t+1} 的评价。

实际实现的时候，AlphaGo 还训练了一个更小的神经网络，它做决策更快。MCTS 在第一步和第二步用大的策略网络，第三步用小的策略网络。读者可能好奇，为什么在且仅在第三步用小的策略网络呢？第三步两个策略网络交替落子，通常要走一两百步，导致第三步成为 MCTS 的瓶颈。用小的策略网络代替大的策略网络，可以大幅加速 MCTS。

第四步——回溯 (Backup): 第三步——求值——算出了第 $t+1$ 步某一个状态的价值，记作 $V(s_{t+1})$ ；每一次模拟都会得出这样一个价值，并且记录下来。模拟会重复很多次，于是第 $t+1$ 步每一种状态下面可以有多条记录；如图 18.8 所示。第 t 步的动作 a_t 下面有多个可能的状态（子节点），每

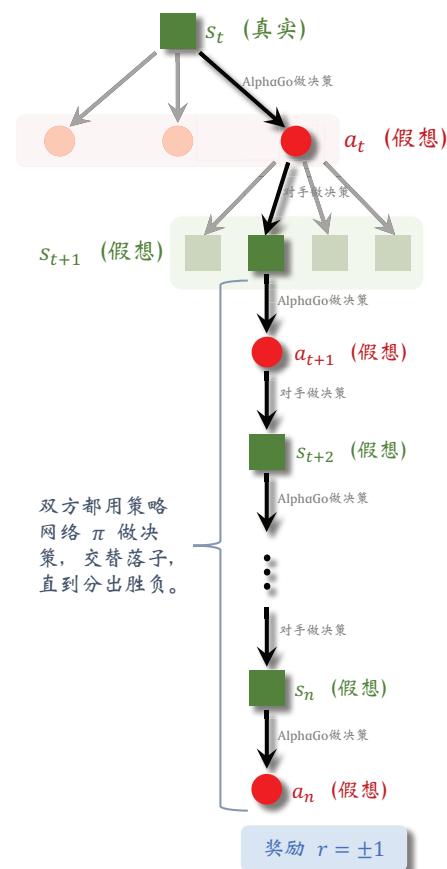


图 18.7：策略网络自我博弈。

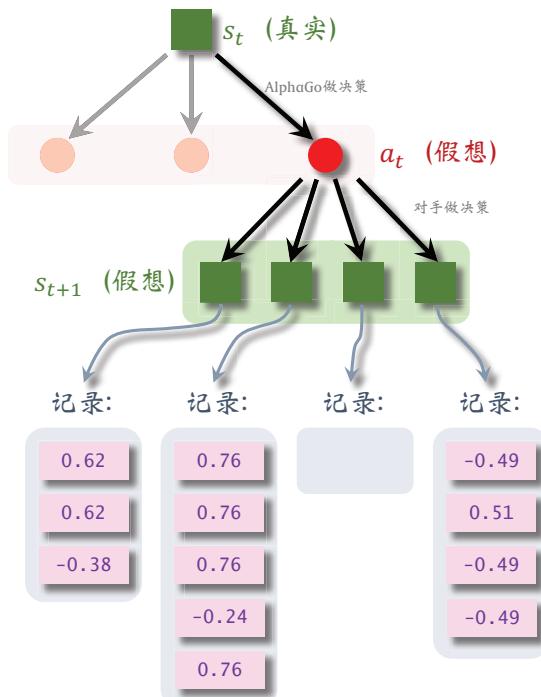


图 18.8：每一个状态 s_{t+1} 下面都有很多条记录，每一条记录是一个 $V(s_{t+1})$ 。

一个状态下面有若干条记录。把 a_t 下面所有的记录取平均，记作价值 $Q(a_t)$ ，它可以反映出动作 a_t 的好坏。在图 18.8 中， a_t 下面一共有 12 条记录， $Q(a_t)$ 是 12 条记录的均值。

给定棋盘上的真实状态 s_t ，有多个动作 a 可供选择。对于所有的 a ，价值 $Q(a)$ 的初始值是零。动作 a 每被选中一次（成为 a_t ），它下面就会多一条记录，我们就对 $Q(a)$ 做一次更新。

回顾第一步——选择 (Selection): 基于棋盘上真实的状态 s_t ，MCTS 需要从可行的动作中选出一个，作为 a_t 。MCTS 计算每一个动作 a 的分数：

$$\text{score}(a) \triangleq Q(a) + \frac{\eta}{1 + N(a)} \cdot \pi(a|s; \theta), \quad \forall a,$$

然后选择分数最高的 a 。MCTS 算出的 $Q(a)$ 的用途就是这里。

18.2.3 MCTS 的决策

上一小节讲解了单次模拟的四个步骤，注意，这只是单次模拟而已。MCTS 想要真正做出一个决策（即往真正的棋盘上落一个棋子），需要做成千上万次模拟。在做了无数次模拟之后，MCTS 做出真正的决策：

$$a_t = \operatorname{argmax}_a N(a).$$

此时 AlphaGo 才会真正往棋盘上放一个棋子。

为什么要依据 $N(a)$ 来做决策呢？在每一次模拟中，MCTS 找出所有可行的动作 $\{a\}$ ，计算它们的分数 $\text{score}(a)$ ，然后选择其中分数最高的动作，然后在模拟器里执行。如果某个动作 a 在模拟中胜率很大，那么它的价值 $Q(a)$ 就会很大，它的分数 $\text{score}(a)$ 会很高，于是它被选中的几率就大。也就是说如果某个动作 a 很好，它被选中的次数 $N(a)$ 就会大。

当观测到棋盘上当前状态 s_t ，MCTS 做成千上万次模拟，记录每个动作 a 被选中的次数 $N(a)$ ，最终做出决策 $a_t = \operatorname{argmax}_a N(a)$ 。到了下一时刻，状态变成了 s_{t+1} 。MCTS 把所有动作 a 的 $Q(a)$ 、 $N(a)$ 全部初始化为零，然后从头开始做模拟，而不能利用上一次的结果。

AlphaGo 下棋非常“暴力”：每走一步棋之前，它先在“脑海里”模拟几千、几万局，它可以预知它每一种动作带来的后果，对手最有可能做出的反应都在 AlphaGo 的算计之内。由于计算量差距悬殊，人类面对 AlphaGo 时不太可能有胜算。这样的比赛对人来说是不公平的；假如李世石下每一颗棋子之前，先跟柯洁模拟一千局，或许李世石的胜算会大于 AlphaGo。

18.3 训练策略网络和价值网络

上一节假设策略网络和价值网络已经训练好，并且用策略网络和价值网络辅助 MCTS。本节具体讲解如何训练两个神经网络。AlphaGo 有多个版本，其中最著名的是 2016、2017 年发表在 Nature 期刊的两个版本，本书称之为 2016 版和 AlphaGo Zero 版。AlphaGo Zero 实力更强：DeepMind 做了实验，让两个版本博弈 100 次，比分是 100 : 0。

18.3.1 AlphaGo 2016 版的训练

AlphaGo 2016 版的训练分为三步：第一，随机初始化策略网络 $\pi(a|s; \theta)$ 之后，用行为克隆 (Behavior Cloning) 从人类棋谱中学习策略网络；第二，让两个策略网络自我博弈，用 REINFORCE 算法改进策略网络；第三，基于已经训练好的策略网络，训练价值网络 $v(s; w)$ 。

第一步：行为克隆：一开始的时候，策略网络的参数都是随机初始化的。假如此时直接让两个策略网络自我博弈，它们会做出纯随机的动作。它们得随机摸索很多很多次，才能做出合理的动作。假如一上来就用 REINFORCE 学习策略网络，最初随机摸索的过程要花很久。这就是为什么 AlphaGo 2016 版基于人类专家的知识初步训练一个策略网络。

有一个叫 KGS 的在线围棋游戏程序，它在 2000 年的时候上线，让玩家在线比赛。KGS 会把每一局游戏都记录下来。KGS 有 16 万局是六段以上的高级玩家的记录。每一局游戏有很多步，每一步棋盘上的格局作为一个状态 s_k ，下一个棋子的位置作为动作 a_k ，这样得到数据集 $\{(s_k, a_k)\}$ 。数据集中一共有 $m = 2.94 \times 10^7$ 个 (s_k, a_k) 这样的二元组。

AlphaGo 用行为克隆训练策略网络 $\pi(a|s; \theta)$ 。第 17.1 节详细介绍了行为克隆，这里只是简单概括一下。设 361 维的向量

$$\mathbf{f}_k = \pi(\cdot | s_k; \theta) = [\pi(1 | s_k; \theta), \pi(2 | s_k; \theta), \dots, \pi(361 | s_k; \theta)]$$

是策略网络的输出，设 $\bar{\mathbf{a}}_k$ 是对动作 a_k 的 One-Hot 编码。函数 $H(\bar{\mathbf{a}}_k, \mathbf{f}_k)$ 是交叉熵 (Cross Entropy)，衡量 $\bar{\mathbf{a}}_k$ 与 \mathbf{f}_k 的差别。行为克隆可以描述成这样一个优化问题：

$$\min_{\theta} \frac{1}{m} \sum_{k=1}^m H(\bar{\mathbf{a}}_k, \mathbf{f}_k).$$

可以用随机梯度下降 (SGD) 求解这个优化问题。每次随机从 $\{1, \dots, m\}$ 中选出一个序号，记作 j 。设当前策略网络参数为 θ_{now} 。用随机梯度更新 θ ：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \eta \cdot \nabla_{\theta} H(\bar{\mathbf{a}}_j, \pi(\cdot | s_j; \theta_{\text{now}})).$$

此处的 η 是学习率。这样可以让策略网络的决策 $\pi(\cdot | s_k; \theta)$ 更接近人类高手的动作 $\bar{\mathbf{a}}_j$ 。

KGS 中的 16 万局游戏都是六段以上的高手的博弈。行为克隆得到的策略网络模仿高手的动作，可以做出比较合理的决策。它在实战中可以打败业余玩家，但是打不过职业玩家。第 17.1 节详细讨论过行为克隆的缺点。为了克服行为克隆的缺点，还需要继续用强化学习训练策略网络。在行为克隆之后再做强化学习改进策略网络，可以击败只用行为克隆的策略网络，胜算是 80%。

第二步——用 REINFORCE 训练策略网络：如图 18.9 所示，AlphaGo 让策略网络做自我博弈，用胜负作为奖励，更新策略网络。博弈的双方是两个策略网络，一个叫做“玩家”，用最新的参数，记作 θ_{now} ；另一个叫做“对手”，它的参数是从过时的参数中随机选出来的，记作 θ_{old} 。“对手”的作用相当于模拟器（环境）的状态转移函数，只是陪玩。训练的过程中，只更新“玩家”的参数，不更新“对手”的参数。



图 18.9：让两个策略网络自我博弈。

让“玩家”和“对手”博弈，将一局游戏进行到底，假设走了 n 步。游戏没结束的时候，奖励全都是零：

$$r_1 = r_2 = \cdots = r_{n-1} = 0.$$

游戏结束的时候，如果“玩家”赢了，奖励是 $r_n = +1$ ，于是所有的回报都是 $+1$: ¹

$$u_1 = u_2 = \cdots = u_n = +1.$$

如果“玩家”输了，奖励是 $r_n = -1$ ，于是所有的回报都是 -1 ：

$$u_1 = u_2 = \cdots = u_n = -1.$$

所有 n 步都用同样的回报，这相当于不区分哪一步棋走得好，哪一步走得烂；只要赢了，每一步都被视为“好棋”；假如输了，每一步都被看成“臭棋”。

REINFORCE 是一种策略梯度方法，它用观测到的回报 u 近似动作价值 Q_π 。REINFORCE 更新策略网络的公式是：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \sum_{t=1}^n u_t \cdot \nabla \ln \pi(a_t | s_t; \theta_{\text{now}}).$$

此处的 β 是学习率。

第三步——训练价值网络：价值网络 $v(s; w)$ 是对状态价值函数 $V_\pi(s)$ 的近似，用于评估状态 s 的好坏。在完成第二步——训练策略网络 π ——之后，用 π 辅助训练 v 。虽然此处有一个策略网络 π 和一个价值网络 v ，但这不属于 Actor-Critic 方法。此处先训练 π ，再训练 v ，用 π 辅助训练 v ；而 Actor-Critic 则是同时训练 π 和 v ，用 v 辅助训练 π 。

让训练好的策略网络做自我博弈，记录状态—回报二元组 (s_k, u_k) ，存到一个数组里。自我博弈需要重复非常多次，把最终得到的数据集记作 $\{(s_k, u_k)\}_{k=1}^m$ 。根据定义，状态价值 $V_\pi(s_k)$ 是回报 U_k 的期望：

$$V_\pi(s_k) = \mathbb{E}[U_k | S_k = s_k].$$

我们希望价值网络 $v(s_k; w)$ 接近 V_π ，也就是回报的期望，于是让 $v(s_k; w)$ 去拟合回报

¹回报的定义是 $u_t = r_t + r_{t+1} + \cdots + r_n$ ，折扣率是 $\gamma = 1$ 。

u_k 。定义回归问题 (Regression):

$$\min_{\mathbf{w}} \frac{1}{2m} \sum_{k=1}^m [v(s_k; \mathbf{w}) - u_k]^2.$$

可以用随机梯度下降 (SGD) 求解这个回归问题。设当前价值网络参数为 \mathbf{w}_{now} 每次随机从 $\{1, \dots, m\}$ 中选出一个序号，记作 j 。用价值网络做预测： $\hat{v}_j = v(s_j; \mathbf{w}_{\text{now}})$ 。用随机梯度更新 \mathbf{w} :

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot (\hat{v}_j - u_j) \cdot \nabla_{\mathbf{w}} v(s_j; \mathbf{w}_{\text{now}}).$$

此处的 α 是学习率。

18.3.2 AlphaGo Zero 版本的训练

AlphaGo Zero 与 2016 版本的最大区别在于训练策略网络 $\pi(a|s; \theta)$ 方式。训练 π 的时候，不再从人类棋谱学习，也不用 REINFORCE 方法，而是向 MCTS 学习。其实可以把 AlphaGo Zero 训练 π 的方法看做是模仿学习，被模仿对象是 MCTS。

自我博弈：用 MCTS 控制两个玩家对弈。每走一步棋，MCTS 需要做成千上万次模拟，并记录下每个动作被选中的次数 $N(a)$, $\forall a \in \{1, 2, \dots, 361\}$ 。设当前是 t 时刻，真实棋牌上当前状态是 s_t 。现在执行 MCTS，完成很多次模拟，得到 361 个整数（每种动作被选中的次数）：

$$N(1), N(2), \dots, N(361).$$

对这些 N 做归一化，得到的 361 个数，它们相加等于 1；把这 361 个数记作 361 维的向量：

$$\mathbf{p}_t = \text{normalize} \left(\left[N(1), N(2), \dots, N(361) \right]^T \right).$$

设这局游戏走了 n 步之后游戏分出胜负；奖励 r_n 要么等于 $+1$ ，要么等于 -1 ，取决于游戏的胜负。在游戏结束的时候，得到回报 $u_1 = \dots = u_n = r_n$ 。记录下这些数据：

$$(s_1, \mathbf{p}_1, u_1), (s_2, \mathbf{p}_2, u_2), \dots, (s_n, \mathbf{p}_n, u_n).$$

用这些数据更新策略网络 π 和价值网络 v ；对 π 和 v 的更新同时进行。

更新策略网络：上一节讨论过，MCTS 做出的决策优于策略网络 π 的决策，这就是为什么 AlphaGo 用 MCTS 做决策，而 π 只是用来辅助 MCTS。既然 MCTS 比 π 更好，那么可以把 MCTS 的决策作为目标，让 π 去模仿。这其实是行为克隆，被模仿的对象是 MCTS。我们希望 π 做出的决策

$$\mathbf{f}_t = \pi(\cdot | s_t; \theta) \in \mathbb{R}^{361}$$

尽量接近 $\mathbf{p}_t \in \mathbb{R}^{361}$ ，也就是让交叉熵 $H(\mathbf{p}_t, \mathbf{f}_t)$ 尽量小。定义优化问题：

$$\min_{\theta} \frac{1}{n} \sum_{t=1}^n H(\mathbf{p}_t, \pi(\cdot | s_t; \theta)).$$

设 π 当前参数是 θ_{now} 。做一次梯度下降更新参数：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \beta \cdot \frac{1}{n} \sum_{t=1}^n \nabla_{\theta} H(p_t, \pi(\cdot | s_t, \theta_{\text{now}})). \quad (18.2)$$

此处的 β 是学习率。

更新价值网络：训练价值网络的方法与 AlphaGo 2016 版本基本一样，都是让 $v(s_t; w)$ 拟合回报 u_t 。定义回归问题：

$$\min_w \frac{1}{2n} \sum_{t=1}^n [v(s_t; w) - u_t]^2.$$

设价值网络 v 当前参数是 w_{now} 。用价值网络做预测： $\hat{v}_t = v(s_t; w_{\text{now}})$, $\forall t = 1, \dots, n$ 。做一次梯度下降更新 w ：

$$w_{\text{new}} \leftarrow w_{\text{now}} - \alpha \cdot \frac{1}{n} \sum_{t=1}^n (\hat{v}_t - u_t) \cdot \nabla_w v(s_t; w_{\text{now}}). \quad (18.3)$$

训练流程：随机初始化策略网络参数 θ 和价值网络参数 w 。然后让 MCTS 自我博弈，玩很多局游戏；每完成一局游戏，更新一次 θ 和 w 。训练的具体流程就是重复下面三个步骤直到收敛：

1. 让 MCTS 自我博弈，完成一局游戏，收集到 n 个三元组： $(s_1, p_1, u_1), \dots, (s_n, p_n, u_n)$ 。
2. 按照公式 (18.2) 做一次梯度下降，更新策略网络参数 θ 。
3. 按照公式 (18.3) 做一次梯度下降，更新价值网络参数 w 。

∽ 第十八章 相关文献 ∽

早在很多年前，AI 就在棋类游戏中战胜了人类，比如国际象棋 (Chess) [24]，西洋跳棋 (Checker) [85, 84]，黑白棋 (Reversi 或 Othello) [22]，双陆棋 (Backgammon) [102]。这些棋类游戏的状态空间远比围棋的状态空间小，所以做搜索会相对比较容易。

AlphaGo 的论文首先发表在 Nature 2016 [90]。改进版本 AlphaGo Zero 发表在 Nature 2017 [92]。在 AlphaGo 之前一直有对围棋 AI 的探索，尽管 AI 尚无法击败人类围棋冠军。其中最有名的围棋 AI 包括 Pachi [11], Fuego [35], GNU Go (1999 年发布，2009 年停更)，Crazy Stone (2006 年发布)。Crazy Stone 虽然不及人类冠军，但是在对手让 4 子的情况下打败过 9 段高手。有兴趣的读者可以参考这些论文：[4, 72, 106, 18, 32, 35, 11]。

蒙特卡洛树搜索 (MCTS) 的名字最早在 2006 年发表的论文 [31] 中提出。另外两篇 2006 年的论文 [26, 54] 提出了类似的想法。2008 年发表的论文 [25] 将 MCTS 概括为今天我们众所周知的四个步骤。本书篇幅有限，不深入介绍 MCTS。有兴趣的读者可以阅读综述 [21] 和书 [27]。

附录 A 贝尔曼方程

定理 A.1. 贝尔曼方程 (将 Q_π 表示成 Q_π)

假设 R_t 是 S_t 、 A_t 、 S_{t+1} 的函数。那么

$$Q_\pi(s_t, a_t) = \mathbb{E}_{S_{t+1}, A_{t+1}} [R_t + \gamma \cdot Q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s_t, A_t = a_t].$$



证明 根据回报的定义 $U_t = \sum_{k=t}^n \gamma^{k-t} \cdot R_k$, 不难验证这个等式:

$$U_t = R_t + \gamma \cdot U_{t+1}.$$

用符号 $\mathcal{S}_{t+1:} = \{S_{t+1}, S_{t+2}, \dots\}$ 和 $\mathcal{A}_{t+1:} = \{A_{t+1}, A_{t+2}, \dots\}$ 表示从 $t+1$ 时刻起所有的状态和动作随机变量。根据动作价值函数 Q_π 的定义,

$$Q_\pi(s_t, a_t) = \mathbb{E}_{\mathcal{S}_{t+1:}, \mathcal{A}_{t+1:}} [U_t \mid S_t = s_t, A_t = a_t].$$

把 U_t 替换成 $R_t + \gamma \cdot U_{t+1}$, 那么

$$\begin{aligned} Q_\pi(s_t, a_t) &= \mathbb{E}_{\mathcal{S}_{t+1:}, \mathcal{A}_{t+1:}} [R_t + \gamma \cdot U_{t+1} \mid S_t = s_t, A_t = a_t] \\ &= \mathbb{E}_{\mathcal{S}_{t+1:}, \mathcal{A}_{t+1:}} [R_t \mid S_t = s_t, A_t = a_t] + \gamma \cdot \mathbb{E}_{\mathcal{S}_{t+1:}, \mathcal{A}_{t+1:}} [U_{t+1} \mid S_t = s_t, A_t = a_t]. \end{aligned} \quad (\text{A.1})$$

假设 R_t 是 S_t 、 A_t 、 S_{t+1} 的函数。那么, 给定 s_t 和 a_t , 则 R_t 随机性唯一的来源就是 S_{t+1} , 所以

$$\mathbb{E}_{\mathcal{S}_{t+1:}, \mathcal{A}_{t+1:}} [R_t \mid S_t = s_t, A_t = a_t] = \mathbb{E}_{S_{t+1}} [R_t \mid S_t = s_t, A_t = a_t]. \quad (\text{A.2})$$

等式 (A.1) 右边 U_{t+1} 的期望可以写成

$$\begin{aligned} &\mathbb{E}_{\mathcal{S}_{t+1:}, \mathcal{A}_{t+1:}} [U_{t+1} \mid S_t = s_t, A_t = a_t] \\ &= \mathbb{E}_{S_{t+1}, A_{t+1}} \left[\mathbb{E}_{S_{t+2:}, A_{t+2:}} [U_{t+1} \mid S_{t+1}, A_{t+1}] \mid S_t = s_t, A_t = a_t \right] \\ &= \mathbb{E}_{S_{t+1}, A_{t+1}} [Q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s_t, A_t = a_t]. \end{aligned} \quad (\text{A.3})$$

由公式 (A.1)、(A.2)、(A.3) 可得定理。 □

定理 A.2. 贝尔曼方程 (将 Q_π 表示成 V_π)

假设 R_t 是 S_t 、 A_t 、 S_{t+1} 的函数。那么

$$Q_\pi(s_t, a_t) = \mathbb{E}_{S_{t+1}} [R_t + \gamma \cdot V_\pi(S_{t+1}) \mid S_t = s_t, A_t = a_t].$$



证明 由于 $V_\pi(S_{t+1}) = \mathbb{E}_{A_{t+1}} [Q(S_{t+1}, A_{t+1})]$, 由定理 A.1 可得定理 A.2。 □

定理 A.3. 贝尔曼方程 (将 V_π 表示成 V_π)

假设 R_t 是 S_t 、 A_t 、 S_{t+1} 的函数。那么

$$V_\pi(s_t) = \mathbb{E}_{A_t, S_{t+1}} [R_t + \gamma \cdot V_\pi(S_{t+1}) \mid S_t = s_t].$$



证明 由于 $V_\pi(S_t) = \mathbb{E}_{A_t}[Q(S_t, A_t)]$, 由定理 A.2 可得定理 A.3。 □

定理 A.4. 最优贝尔曼方程

假设 R_t 是 S_t, A_t, S_{t+1} 的函数。那么

$$Q_\star(s_t, a_t) = \mathbb{E}_{S_{t+1} \sim p(\cdot | s_t, a_t)} [R_t + \gamma \cdot \max_{A \in \mathcal{A}} Q_\star(S_{t+1}, A) \mid S_t = s_t, A_t = a_t].$$
♡

证明 设最优策略函数为 $\pi^\star = \operatorname{argmax}_\pi Q_\pi(s, a)$, $\forall s \in \mathcal{S}, a \in \mathcal{A}$ 。由贝尔曼方程可得：

$$Q_{\pi^\star}(s_t, a_t) = \mathbb{E}_{S_{t+1}, A_{t+1}} [R_t + \gamma \cdot Q_{\pi^\star}(S_{t+1}, A_{t+1}) \mid S_t = s_t, A_t = a_t].$$

根据定义，最优动作函数是

$$Q_\star(s, a) \triangleq \max_\pi Q_\pi(s, a), \quad \forall s \in \mathcal{S}, a \in \mathcal{A}.$$

所以 $Q_{\pi^\star}(s, a)$ 就是 $Q_\star(s, a)$ 。于是

$$Q_\star(s_t, a_t) = \mathbb{E}_{S_{t+1}, A_{t+1}} [R_t + \gamma \cdot Q_\star(S_{t+1}, A_{t+1}) \mid S_t = s_t, A_t = a_t].$$

因为动作 $A_{t+1} = \operatorname{argmax}_A Q_\star(S_{t+1}, A)$ 是状态 S_{t+1} 的确定性函数，所以

$$Q_\star(s_t, a_t) = \mathbb{E}_{S_{t+1}} [R_t + \gamma \cdot \max_{A \in \mathcal{A}} Q_\star(S_{t+1}, A) \mid S_t = s_t, A_t = a_t].$$

□

参考文献

- [1] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2004.
- [2] M. S. Abdulla and S. Bhatnagar. Reinforcement learning based algorithms for average cost markov decision processes. *Discrete Event Dynamic Systems*, 17(1):23–52, 2007.
- [3] Z. Ahmed, N. Le Roux, M. Norouzi, and D. Schuurmans. Understanding the impact of entropy on policy optimization. In *International Conference on Machine Learning (ICML)*, 2019.
- [4] L. V. Allis et al. *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen, 1994.
- [5] B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.
- [6] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*, 2015.
- [7] M. Bain and C. Sammut. A framework for behavioural cloning. In *Machine Intelligence*, pages 103–129, 1995.
- [8] L. Baird. Residual algorithms: reinforcement learning with function approximation. In *Machine Learning Proceedings 1995*, pages 30–37. Elsevier, 1995.
- [9] N. Bard, J. N. Foerster, S. Chandar, N. Burch, M. Lanctot, H. F. Song, E. Parisotto, V. Dumoulin, S. Moitra, E. Hughes, et al. The Hanabi challenge: a new frontier for AI research. *Artificial Intelligence*, 280:103216, 2020.
- [10] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.
- [11] P. Baudiš and J.-l. Gailly. Pachi: State of the art open source go program. In *Advances in computer games*, pages 24–38. Springer, 2011.
- [12] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2017.
- [13] D. P. Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.
- [14] S. Bhatnagar and S. Kumar. A simultaneous perturbation stochastic approximation-based actor-critic algorithm for markov decision processes. *IEEE Transactions on Automatic Control*, 49(4):592–598, 2004.
- [15] S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee. Natural actor-critic algorithms. *Automatica*, 45(11):2471–2482, 2009.
- [16] A. Boularias, J. Kober, and J. Peters. Relative entropy inverse reinforcement learning. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011.
- [17] C. Boutilier. Planning, learning and coordination in multiagent decision processes. In *Conference on Theoretical Aspects of Rationality and Knowledge*, 1996.
- [18] B. Bouzy and B. Helmstetter. Monte-Carlo go developments. In *Advances in computer games*, pages 159–174. Springer, 2004.
- [19] S. Boyd, S. P. Boyd, and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.
- [20] I. Bratko and T. Urbancic. Transfer of control skill by machine learning. *Engineering Applications of Artificial Intelligence*, 10(1):63–71, 1997.
- [21] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.
- [22] M. Buro. From simple features to sophisticated evaluation functions. In *International Conference on Computers and Games*, pages 126–145. Springer, 1998.
- [23] L. Buşoniu, R. Babuška, and B. De Schutter. Multi-agent reinforcement learning: An overview. In *Innovations*

- in multi-agent systems and applications-1*, pages 183–221. Springer, 2010.
- [24] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.
- [25] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-Carlo tree search: A new framework for game AI. In *AIIDE*, 2008.
- [26] G. Chaslot, J.-T. Saito, B. Bouzy, J. Uiterwijk, and H. J. Van Den Herik. Monte-Carlo strategies for computer Go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, 2006.
- [27] G. M. J.-B. C. Chaslot. *Monte-Carlo tree search*. Maastricht University, 2010.
- [28] K. Cho, B. v. M. C. Gulcehre, D. Bahdanau, F. B. H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. 2014.
- [29] Y. Chow, O. Nachum, and M. Ghavamzadeh. Path consistency learning in Tsallis entropy regularized mdps. In *International Conference on Machine Learning (ICML)*, pages 979–988, 2018.
- [30] A. R. Conn, N. I. Gould, and P. L. Toint. *Trust region methods*. SIAM, 2000.
- [31] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.
- [32] R. Coulom. Computing “elo ratings” of move patterns in the game of Go. *ICGA journal*, 30(4):198–208, 2007.
- [33] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [34] T. Degris, P. M. Pilarski, and R. S. Sutton. Model-free reinforcement learning with continuous action in practice. In *American Control Conference (ACC)*, 2012.
- [35] M. Enzenberger, M. Müller, B. Arneson, and R. Segal. Fuego: an open-source framework for board games and go engine based on monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.
- [36] C. Finn, S. Levine, and P. Abbeel. Guided cost learning: deep inverse optimal control via policy optimization. In *International Conference on Machine Learning (ICML)*, 2016.
- [37] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson. Counterfactual multi-agent policy gradients. In *AAAI Conference on Artificial Intelligence*, 2018.
- [38] J. Foerster, N. Nardelli, G. Farquhar, T. Afouras, P. H. Torr, P. Kohli, and S. Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2017.
- [39] M. Fortunato, M. G. Azar, B. Piot, J. Menick, M. Hessel, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, et al. Noisy networks for exploration. In *International Conference on Learning Representations (ICLR)*, 2018.
- [40] S. Fujimoto, H. Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning (ICML)*, 2018.
- [41] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. *Advances in Neural Information Processing Systems (NIPS)*, 2014.
- [42] J. K. Gupta, M. Egorov, and M. Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems*, 2017.
- [43] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine. Reinforcement learning with deep energy-based policies. In *International Conference on Machine Learning (ICML)*, 2017.
- [44] R. Hafner and M. Riedmiller. Reinforcement learning in feedback control. *Machine learning*, 84(1-2):137–169, 2011.
- [45] M. Hausknecht and P. Stone. Deep recurrent Q-learning for partially observable MDPs. In *AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents*, 2015.
- [46] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.
- [47] J. Ho and S. Ermon. Generative adversarial imitation learning. In *Conference on Neural Information Processing*

参考文献

- Systems (NIPS), 2016.
- [48] Y.-C. Ho. Team decision theory and information structures. *Proceedings of the IEEE*, 68(6):644–654, 1980.
- [49] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
- [50] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.
- [51] J. Hu and M. P. Wellman. Nash Q-learning for general-sum stochastic games. *Journal of Machine Learning Research*, 4(Nov):1039–1069, 2003.
- [52] S. Iqbal and F. Sha. Actor-attention-critic for multi-agent reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2019.
- [53] T. Jaakkola, M. I. Jordan, and S. P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural computation*, 6(6):1185–1201, 1994.
- [54] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.
- [55] V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems (NIPS)*, 2000.
- [56] M. G. Lagoudakis and R. Parr. Learning in zero-sum team Markov games using factored value functions. *Advances in Neural Information Processing Systems (NIPS)*, 2002.
- [57] M. Lauer and M. Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *International Conference on Machine Learning (ICML)*, 2000.
- [58] K. Lee, S. Choi, and S. Oh. Sparse Markov decision processes with causal sparse Tsallis entropy regularization for reinforcement learning. *IEEE Robotics and Automation Letters*, 3(3):1466–1473, 2018.
- [59] S. Levine and V. Koltun. Continuous inverse optimal control with locally optimal examples. In *International Conference on Machine Learning (ICML)*, 2012.
- [60] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
- [61] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2016.
- [62] L.-J. Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.
- [63] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *International Conference on Machine Learning (ICML)*, 1994.
- [64] M. L. Littman. Friend-or-foe Q-learning in general-sum games. In *International Conference on Machine Learning (ICML)*, 2001.
- [65] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.
- [66] P. Marbach and J. N. Tsitsiklis. Simulation-based optimization of Markov reward processes: Implementation issues. In *IEEE Conference on Decision and Control*, 1999.
- [67] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, et al. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*, 2016.
- [68] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2016.
- [69] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [70] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*,

518(7540):529–533, 2015.

- [71] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: a distributed framework for emerging AI applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [72] M. Müller. Computer go. *Artificial Intelligence*, 134(1-2):145–179, 2002.
- [73] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.
- [74] A. Y. Ng and S. J. Russell. Algorithms for inverse reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2000.
- [75] J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.
- [76] B. O’Donoghue, R. Munos, K. Kavukcuoglu, and V. Mnih. Combining policy gradient and Q-learning. In *International Conference on Learning Representations (ICLR)*, 2017.
- [77] F. A. Oliehoek, M. T. Spaan, and N. Vlassis. Optimal and approximate Q-value functions for decentralized POMDPs. *Journal of Artificial Intelligence Research*, 32:289–353, 2008.
- [78] D. V. Prokhorov and D. C. Wunsch. Adaptive critic designs. *IEEE transactions on Neural Networks*, 8(5):997–1007, 1997.
- [79] T. Rashid, M. Samvelyan, C. Schroeder, G. Farquhar, J. Foerster, and S. Whiteson. QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.
- [80] S. Ross and D. Bagnell. Efficient reductions for imitation learning. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.
- [81] G. A. Rummery and M. Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.
- [82] M. Samvelyan, T. Rashid, C. Schroeder de Witt, G. Farquhar, N. Nardelli, T. G. Rudner, C.-M. Hung, P. H. Torr, J. Foerster, and S. Whiteson. The StarCraft Multi-Agent Challenge. In *International Conference on Autonomous Agents and MultiAgent Systems*, 2019.
- [83] S. Schaal. Learning from demonstration. In *Advances in Neural Information Processing Systems (NIPS)*, 1997.
- [84] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.
- [85] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–289, 1992.
- [86] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In *International Conference on Learning Representations (ICLR)*, 2015.
- [87] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International Conference on Machine Learning (ICML)*, 2015.
- [88] W. Shi, S. Song, and C. Wu. Soft policy gradient method for maximum entropy deep reinforcement learning. *arXiv preprint arXiv:1909.03198*, 2019.
- [89] Y. Shoham and K. Leyton-Brown. *Multiagent systems: algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.
- [90] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
- [91] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *International Conference on Machine Learning (ICML)*, 2014.
- [92] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

参考文献

- [93] P. Stone and M. Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.
- [94] P. Sunehag, G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning based on team reward. In *International Conference on Autonomous Agents and MultiAgent Systems*, 2018.
- [95] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems (NIPS)*, 1996.
- [96] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [97] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, 2000.
- [98] U. Syed, M. Bowling, and R. E. Schapire. Apprenticeship learning using linear programming. In *International Conference on Machine Learning (ICML)*, 2008.
- [99] U. Syed and R. E. Schapire. A reduction from apprenticeship learning to classification. *Advances in Neural Information Processing Systems (NIPS)*, 23, 2010.
- [100] A. Tampuu, T. Mattisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PloS one*, 12(4):e0172395, 2017.
- [101] M. Tan. Multi-agent reinforcement learning: independent vs. cooperative agents. In *International Conference on Machine Learning (ICML)*, 1993.
- [102] G. Tesauro and G. R. Galperin. On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing Systems*, pages 1068–1074, 1997.
- [103] C. Tsallis. Possible generalization of Boltzmann-Gibbs statistics. *Journal of statistical physics*, 52(1-2):479–487, 1988.
- [104] J. N. Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine learning*, 16(3):185–202, 1994.
- [105] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE transactions on automatic control*, 42(5):674–690, 1997.
- [106] H. J. Van Den Herik, J. W. Uiterwijk, and J. Van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1-2):277–311, 2002.
- [107] H. van Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems (NIPS)*, 2010.
- [108] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.
- [109] H. van Seijen. Effective multi-step temporal-difference learning for non-linear function approximation. *arXiv preprint arXiv:1608.05151*, 2016.
- [110] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.
- [111] N. Vlassis. A concise introduction to multiagent systems and distributed artificial intelligence. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 1(1):1–71, 2007.
- [112] X. Wang and T. Sandholm. Reinforcement learning to play an optimal Nash equilibrium in team Markov games. *Advances in Neural Information Processing Systems (NIPS)*, 2002.
- [113] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas. Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2016.
- [114] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.
- [115] C. J. C. H. Watkins. Learning from delayed rewards. 1989.
- [116] G. Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999.
- [117] R. J. Williams. *Reinforcement-learning connectionist systems*. College of Computer Science, Northeastern University, 1987.
- [118] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning.

- Machine learning*, 8(3-4):229–256, 1992.
- [119] R. J. Williams and J. Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.
- [120] W. Yang, X. Li, and Z. Zhang. A regularized approach to sparse optimal policy in reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 5940–5950, 2019.
- [121] Z. Yang, Y. Chen, M. Hong, and Z. Wang. Provably global convergence of actor-critic: A case for linear quadratic regulator with ergodic cost. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 8353–8365, 2019.
- [122] T. Yoshikawa. Decomposition of dynamic team decision problems. *IEEE Transactions on Automatic Control*, 23(4):627–632, 1978.
- [123] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.
- [124] K. Zhang, Z. Yang, and T. Basar. Multi-agent reinforcement learning: a selective overview of theories and algorithms. *arXiv preprint arXiv:1911.10635*, 2019.
- [125] B. D. Ziebart, A. L. Maas, J. A. Bagnell, and A. K. Dey. Maximum entropy inverse reinforcement learning. In *AAAI Conference on Artificial Intelligence*, 2008.

《深度强化学习》2021-03-18尚未完成校对，仅供预览。
如发现错误，请告知作者 shusen.wang@stevens.edu

致谢

由于本书篇幅较长，难免出现错误。真诚感谢王嘉晨、张梦娇、陈传玺、常海德、张翠娟、梅椰诚、张大康、单思远、陆浩、徐嘉诚提供的反馈与批评指正。