

还是很小。所以评论员的输出值取决于状态和演员。评论员其实都要绑定一个演员，它是在衡量某一个演员的好坏，而不是衡量一个状态的好坏。这里要强调一下，评论员的输出是与演员有关的，状态的价值其实取决于演员，当演员改变的时候，状态价值函数的输出其实也是会跟着改变的。

怎么衡量状态价值函数 $V_\pi(s)$ 呢？有两种不同的方法：基于蒙特卡洛的方法和基于时序差分的方法。基于蒙特卡洛的方法就是让演员与环境交互，我们要看演员好不好，就让演员与环境交互，让评论员评价。评论员就统计，演员如果看到状态 s_a ，接下来的累积奖励有多大；如果它看到状态 s_b ，接下来的累积奖励有多大。但是实际上，我们不可能看到所有的状态。如果我们在玩雅达利游戏，状态是图像，那么无法看到所有的状态。所以实际上 $V_\pi(s)$ 是一个网络。对一个网络来说，就算输入状态是从来都没有看过的，它也可以想办法估测一个值。怎么训练这个网络呢？如图 6.2 所示，如果在状态 s_a ，接下来的累积奖励就是 G_a 。也就是对这个价值函数，如果输入是状态 s_a ，正确的输出应该是 G_a ；如果输入状态是 s_b ，正确的输出应该是 G_b 。所以在训练的时候，它就是一个回归问题（regression problem）。网络的输出就是一个值，我们希望在输入 s_a 的时候，输出的值与 G_a 越接近越好；输入 s_b 的时候，输出的值与 G_b 越接近越好。接下来继续训练网络，这是基于蒙特卡洛的方法。

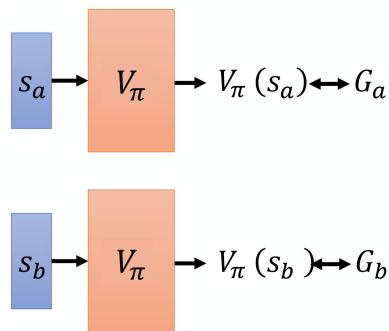


图 6.2 基于蒙特卡洛的方法

第二个方法是**时序差分**的方法，即基于时序差分的方法。在基于蒙特卡洛的方法中，每次我们都要计算累积奖励，也就是从某一个状态 s_a 一直到游戏结束的时候，得到的所有奖励的总和。如果我们要使用基于蒙特卡洛的方法，我们必须至少玩到游戏结束。但有些游戏时间非常长，我们要玩到游戏结束才能够更新网络，这花的时间太多了，因此我们会采用基于时序差分的方法。基于时序差分的方法不需要玩到游戏结束，只需要在游戏的某一个状态 s_t 的时候，采取动作 a_t 得到奖励 r_t ，接下来进入状态 s_{t+1} ，就可以使用时序差分的方法。我们可以通过式 (6.2) 来使用时序差分的方法。

$$V_\pi(s_t) = V_\pi(s_{t+1}) + r_t \quad (6.2)$$

假设我们现在用的是某一个策略 π ，在状态 s_t 时，它会采取动作 a_t ，得到奖励 r_t ，接下来进入 s_{t+1} 。状态 s_{t+1} 的值与状态 s_t 的值，它们的中间差了一项 r_t ，这是因为我们把 s_{t+1} 的值加上得到的奖励 r_t 就可以得到 s_t 的值。有了式 (6.2)，在训练的时候，我们并不是直接估测 V_π ，而是希望得到的结果 V_π 可以满足式 (6.2)。我们是这样训练的，如图 6.3 所示，我们把 s_t 输入网络，因为把 s_t 输入网络会得到 $V_\pi(s_t)$ ，把 s_{t+1} 输入网络会得到 $V_\pi(s_{t+1})$ ， $V_\pi(s_t)$ 减 $V_\pi(s_{t+1})$ 的值应该是 r_t 。我们希望它们相减的损失与 r_t 接近，训练下去，更新 V_π 的参数，我们就可以把 V_π 函数学习出来。

蒙特卡洛方法与时序差分方法有什么差别呢？如图 6.4 所示，蒙特卡洛方法最大的问题就是方差很大。因为我们在玩游戏的时候，游戏本身是有随机性的，所以我们可以把 G_a 看成一个随机变量。因为我们每次到 s_a 的时候，最后得到的 G_a 其实是不一样的。我们看到同样的状态 s_a ，最后到游戏结束的时候，因为游戏本身是有随机性的，玩游戏的模型可能也有随机性，所以我们每次得到的 G_a 是不一样的，每一次得到的 G_a 的差别其实会很大。为什么会很大呢？因为 G_a 是很多个不同的步骤的奖励的和。假设我们每一个步骤都会得到一个奖励， G_a 是从状态 s_a 开始一直到游戏结束，每一个步骤的奖励的和。

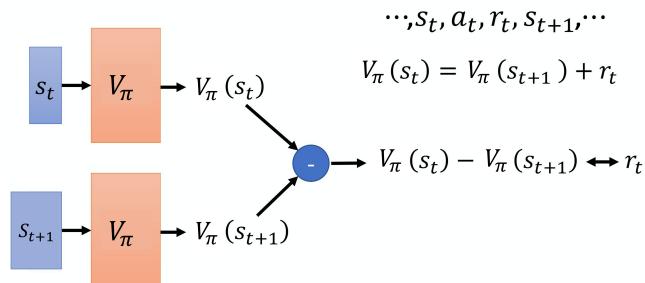


图 6.3 基于时序差分的方法

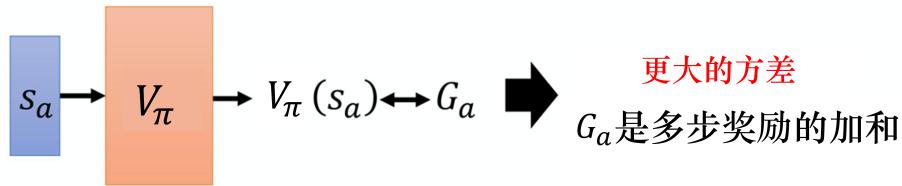


图 6.4 蒙特卡洛方法的问题

通过式 (6.3)，我们知道 G_a 的方差相较于某一个状态的奖励，它是比较大的。

$$\text{Var}[kX] = k^2 \text{Var}[X] \quad (6.3)$$

其中，Var 是指方差 (variance)。

如果用时序差分的方法，我们要去最小化

$$V_\pi(s_t) \longleftrightarrow r + V_\pi(s_{t+1}) \quad (6.4)$$

其中， r 具有随机性。因为即使我们在 s_t 采取同一个动作，得到的奖励也不一定是一样的，所以 r 是一个随机变量。但 r 的方差比 G_a 要小，因为 G_a 是很多 r 的加和，时序差分只是某一个 r 而已。 G_a 的方差会比较大， r 的方差会比较小。但是这里我们会遇到的一个问题是 V_π 的估计不一定准确。假设 V_π 的估计不准确，我们使用式 (6.4) 学习出来的结果也会是不准确的。所以蒙特卡洛方法与时序差分方法各有优劣。其实时序差分方法是比较常用的，蒙特卡洛方法其实是比较少用的。

图 6.5 所示为时序差分方法与蒙特卡洛方法的差别。假设有某一个评论员，它去观察某一个策略 π 与环境交互 8 个回合的结果。有一个策略 π 与环境交互了 8 次，得到了 8 次玩游戏的结果。接下来这个评论员去估测状态的值。

我们先计算 s_b 的值。状态 s_b 在 8 场游戏里都存在，其中有 6 场得到奖励 1，有 2 场得到奖励 0。所以如果我们要计算期望值，只算智能体看到状态 s_b 以后得到的奖励。智能体一直玩到游戏结束的时候得到的累积奖励期望值是 $3/4$ ，计算过程为

$$\frac{6 \times 1 + 2 \times 0}{8} = \frac{6}{8} = \frac{3}{4} \quad (6.5)$$

但 s_a 期望的奖励到底应该是多少呢？这里其实有两个可能的答案：0 和 $3/4$ 。为什么有两个可能的答案呢？这取决于我们用蒙特卡洛方法还是时序差分方法。用蒙特卡洛方法与用时序差分方法算出来的结果是不一样的。

假如我们用蒙特卡洛方法， s_a 就出现一次，看到状态 s_a ，接下来累积奖励就是 0，所以 s_a 期望奖励就是 0。但时序差分方法在计算的时候，需要更新

$$V_\pi(s_a) = V_\pi(s_b) + r \quad (6.6)$$

因为我们在状态 s_a 得到奖励 $r = 0$ 以后，进入状态 s_b ，所以状态 s_b 的奖励等于状态 s_b 的奖励加上从状态 s_a 进入状态 s_b 的时候可能得到的奖励 r 。而得到的奖励 r 的值是 0， s_b 期望奖励是 $3/4$ ，那么 s_a 的奖励应该是 $3/4$ 。

用蒙特卡洛方法与时序差分方法估出来的结果很有可能是不一样的。就算评论员观察到一样的训练数据，它最后估出来的结果也不一定是一样的。为什么会这样呢？哪一个结果比较对呢？其实都对。因为在第一个轨迹中， s_a 得到奖励 0 以后，再进入 s_b 也得到奖励 0。这里有两个可能。

(1) s_a 是一个标志性的状态，只要看到 s_a 以后， s_b 就不会获得奖励， s_a 可能影响了 s_b 。如果是使用蒙特卡洛方法，它会把 s_a 影响 s_b 这件事考虑进去。所以看到 s_a 以后，接下来 s_b 就得不到奖励， s_b 期望的奖励是 0。

(2) 看到 s_a 以后， s_b 的奖励是 0 这件事只是巧合，并不是 s_a 造成的，而是因为 s_b 有时候就是会得到奖励 0，这只是单纯“运气”的问题。其实平常 s_b 会得到的奖励期望值是 $3/4$ ，与 s_a 是完全没有关系的。所以假设 s_a 之后会进入 s_b ，得到的奖励按照时序差分方法来算应该是 $3/4$ 。

不同的方法考虑了不同的假设，所以运算结果不同。

评论员部分有以下8个回合

- $s_a, r = 0, s_b, r = 0$, 结束
- $s_b, r = 1$, 结束 $V_\pi(s_b) = 3/4$
- $s_b, r = 1$, 结束
- $s_b, r = 1$, 结束 $V_\pi(s_a) = ? \quad 0? \quad 3/4?$
- $s_b, r = 1$, 结束
- $s_b, r = 1$, 结束 蒙特卡罗： $V_\pi(s_a) = 0$
- $s_b, r = 1$, 结束
- $s_b, r = 0$, 结束 时序差分：

$$V_\pi(s_a) = V_\pi(s_b) + r$$

$3/4$	$3/4$	0
-------	-------	---

图 6.5 时序差分方法与蒙特卡洛方法的差别

6.2 动作价值函数

还有另外一种评论员称为 **Q 函数**，它又被称为动作价值函数。状态价值函数的输入是一个状态，它根据状态计算出这个状态以后的期望的累积奖励 (expected accumulated reward) 是多少。动作价值函数的输入是一个状态-动作对，其指在某一个状态采取某一个动作，假设我们都使用策略 π ，得到的累积奖励的期望值有多大。

Q 函数有一个需要注意的问题是，策略 π 在看到状态 s 的时候，它采取的动作不一定是 a 。**Q 函数**假设在状态 s 强制采取动作 a ，而不管我们现在考虑的策略 π 会不会采取动作 a ，这并不重要。在状态 s 强制采取动作 a 。接下来都用策略 π 继续玩下去，就只有在状态 s ，我们才强制一定要采取动作 a ，接下来就进入自动模式，让策略 π 继续玩下去，得到的期望奖励才是 $Q_\pi(s, a)$ 。

Q 函数有两种写法：

(1) 如图 6.6a 所示，输入是状态与动作，输出就是一个标量。这种 **Q 函数**既适用于连续动作（动作是无法穷举的），又适用于离散动作。

(2) 如图 6.6b 所示，输入是一个状态，输出就是多个值。这种 **Q 函数**只适用于离散动作。假设动作是离散的，比如动作就只有 3 个可能：往左、往右或是开火。**Q 函数**输出的 3 个值就分别代表 a 是往左的时候的 **Q 值**， a 是往右的时候的 **Q 值**，还有 a 是开火的时候的 **Q 值**。

如果我们去估计 **Q 函数**，看到的结果可能如图 6.7 所示。假设我们有 3 个动作：原地不动、向上、向下。假设在第一个状态，不管采取哪个动作，最后到游戏结束的时候，得到的期望奖励都差不多。因为乒乓球在这个地方时，就算我们向下，接下来我们应该还可以接到乒乓球，所以不管采取哪个动作，都相差

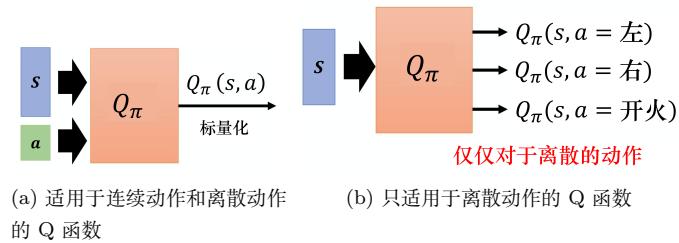
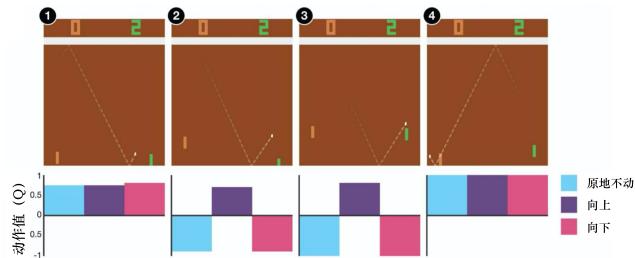


图 6.6 Q 函数

不了太多。假设在第二个状态，乒乓球已经反弹到很接近边缘的地方，这个时候我们采取向上的动作，才能接到乒乓球，才能得到正的奖励。如果我们站在原地不动或向下，接下来都会错过这个乒乓球，得到的奖励就会是负的。假设在第三个状态，乒乓球离我们的球拍很近了，所以就要采取向上的动作。假设在第四个状态，乒乓球被反弹回去，这时候采取哪个动作都差不多。这是动作价值函数的例子。

图 6.7 乒乓球例子^[1]

虽然我们学习的 Q 函数只能用来评估某一个策略 π 的好坏，但只要有了 Q 函数，我们就可以进行强化学习，就可以决定要采取哪一个动作，就可以进行策略改进。如图 6.8 所示，假设我们有一个初始的演员，也许一开始很差，随机的也没有关系。初始的演员称为 π ， π 与环境交互，会收集数据。接下来我们学习策略 π 的 Q 值，去衡量一下 π 在某一个状态强制采取某一个动作，接下来会得到的期望奖励，用时差分方法或蒙特卡洛方法都是可以的。我们学习出一个 Q 函数以后，就可以找到一个新的策略 π' ，策略 π' 会比原来的策略 π 要好（稍后会定义什么是好）。所以假设我们有一个 Q 函数和某一个策略 π ，根据策略 π 学习出策略 π 的 Q 函数，接下来可以找到一个新的策略 π' ，它会比 π 要好。我们用 π' 取代 π ，再去学习它的 Q 函数，得到新的 Q 函数以后，再去寻找一个更好的策略。这样一直循环下去，策略就会越来越好。

首先要定义的是什么好。 π' 一定会比 π 要好，什么是好呢？这里的好是指，对所有可能的状态 s 而言， $V_{\pi'}(s) \geq V_\pi(s)$ 。也就是我们到同一个状态 s 的时候，如果用 π 继续与环境交互，我们得到的奖励一定会小于等于用 π' 与环境交互得到的奖励。所以不管在哪一个状态，我们用 π' 与环境交互，得到的期望奖励一定会比较大。所以 π' 是比 π 要好的策略。

有了 Q 函数以后，我们把根据式 (6.7) 决定动作的策略称为 π' ，

$$\pi'(s) = \arg \max_a Q_\pi(s, a) \quad (6.7)$$

π' 一定比 π 好。假设我们已经学习出 π 的 Q 函数，在某一个状态 s ，把所有可能的动作 a 一一代入 Q 函数，看看哪一个 a 可以让 Q 函数的值最大，这个动作就是 π' 会采取的动作。

这里要注意，给定状态 s 和策略 π 并不一定会采取动作 a 。给定某一个状态 s 强制采取动作 a ，用 π 继续交互得到的期望奖励，这才是 Q 函数的定义。所以在状态 s 下不一定会采取动作 a 。用 π' 在状态 s 采取动作 a 与用 π 采取的动作不一定是一样的， π' 所采取的动作会让它得到比较大的奖励。所以 π' 是用 Q 函数推出来的，没有另外一个网络决定 π' 怎么与环境交互，有 Q 函数就可以找出 π' 。但是在这里

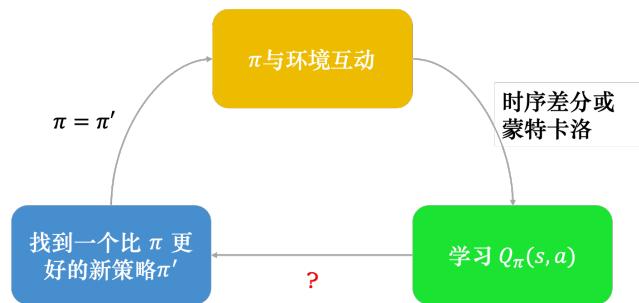


图 6.8 使用 Q 函数进行策略改进

要解决一个 $\arg \max$ 操作的问题，如果 a 是离散的，如 a 只有 3 个选项，将每个动作都代入 Q 函数，看哪个动作的 Q 值最大，这没有问题。但如果 a 是连续的，我们要解决 $\arg \max$ 操作问题，就不可行。

接下来讲一下为什么用 $Q_\pi(s, a)$ 决定的 π' 一定会比 π 好。假设有一个策略 π' ，它是由 Q_π 决定的。我们要证明对所有的状态 s ，有 $V_{\pi'}(s) \geq V_\pi(s)$ 。

怎么证明呢？ $V_\pi(s)$ 可写为

$$V_\pi(s) = Q_\pi(s, \pi(s)) \quad (6.8)$$

假设在状态 s 下按照策略 π ，会采取的动作就是 $\pi(s)$ ，我们算出来的 $Q_\pi(s, \pi(s))$ 会等于 $V_\pi(s)$ 。一般而言， $Q_\pi(s, \pi(s))$ 不一定等于 $V_\pi(s)$ ，因为动作不一定是 $\pi(s)$ 。但如果这个动作是 $\pi(s)$ ， $Q_\pi(s, \pi(s))$ 是等于 $V_\pi(s)$ 的。

$Q_\pi(s, \pi(s))$ 还满足如下的关系：

$$Q_\pi(s, \pi(s)) \leq \max_a Q_\pi(s, a) \quad (6.9)$$

因为 a 是所有动作里面可以让 Q 函数取最大值的那个动作，所以 $Q_\pi(s, a)$ 一定大于等于 $Q_\pi(s, \pi(s))$ 。 $Q_\pi(s, a)$ 中的 a 就是 $\pi'(s)$ ，因为 $\pi'(s)$ 输出的 a 可以让 $Q_\pi(s, a)$ 最大，所以我们可得

$$\max_a Q_\pi(s, a) = Q_\pi(s, \pi'(s)) \quad (6.10)$$

于是

$$V_\pi(s) \leq Q_\pi(s, \pi'(s))$$

也就是在某一个状态，如果我们按照策略 π 一直执行下去，得到的奖励一定会小于等于在状态 s 故意不按照 π 所指示的方向，而是按照 π' 的方向走一步得到的奖励。但只有第一步是按照 π' 的方向走，只有在状态 s ，才按照 π' 的指示走，接下来我们就按照 π 的指示走。虽然只有一步之差，但我们得到的奖励一定会比完全按照 π 得到的奖励要大。

接下来要证

$$Q_\pi(s, \pi'(s)) \leq V_{\pi'}(s) \quad (6.11)$$

也就是，只有一步之差，我们会得到比较大的奖励。但假设每步都是不一样的，每步都按照 π' 而不是 π ，得到的奖励一定会更大，即 $Q_\pi(s, \pi'(s))$ 是指我们在状态 s_t 采取动作 a_t ，得到奖励 r_t ，进入状态 s_{t+1} ，即

$$Q_\pi(s, \pi'(s)) = \mathbb{E}[r_t + V_\pi(s_{t+1}) | s_t = s, a_t = \pi'(s_t)] \quad (6.12)$$

有的文献上也会说：在状态 s_t 采取动作 a_t ，得到奖励 r_{t+1} 。但意思其实都是一样的。在状态 s 按照 π' 采取某一个动作 a_t ，得到奖励 r_t ，进入状态 s_{t+1} ， $V_\pi(s_{t+1})$ 是状态 s_{t+1} 根据策略 π 所估出来的值。因为在同样的状态采取同样的动作，我们得到的奖励和会进入的状态不一定一样，所以需要取期望值。

因为 $V_\pi(s) \leq Q_\pi(s, \pi'(s))$, 也就是 $V_\pi(s_{t+1}) \leq Q_\pi(s_{t+1}, \pi'(s_{t+1}))$, 所以我们可得

$$\begin{aligned} & \mathbb{E}[r_t + V_\pi(s_{t+1}) | s_t = s, a_t = \pi'(s_t)] \\ & \leq \mathbb{E}[r_t + Q_\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s, a_t = \pi'(s_t)] \end{aligned} \quad (6.13)$$

因为 $Q_\pi(s_{t+1}, \pi'(s_{t+1})) = r_{t+1} + V_\pi(s_{t+2})$, 所以我们可得

$$\begin{aligned} & \mathbb{E}[r_t + Q_\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s, a_t = \pi'(s_t)] \\ & = \mathbb{E}[r_t + r_{t+1} + V_\pi(s_{t+2}) | s_t = s, a_t = \pi'(s_t)] \end{aligned} \quad (6.14)$$

我们再把式 (6.14) 代入 $V_\pi(s) \leq Q_\pi(s, \pi'(s))$, 一直算到回合结束, 即

$$\begin{aligned} & V_\pi(s) \leq Q_\pi(s, \pi'(s)) \\ & \leq \mathbb{E}[r_t + Q_\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s, a_t = \pi'(s_t)] \\ & = \mathbb{E}[r_t + r_{t+1} + V_\pi(s_{t+2}) | s_t = s, a_t = \pi'(s_t)] \\ & \leq \mathbb{E}[r_t + r_{t+1} + Q_\pi(s_{t+2}, \pi'(s_{t+2})) | s_t = s, a_t = \pi'(s_t)] \\ & = \mathbb{E}[r_t + r_{t+1} + r_{t+2} + V_\pi(s_{t+3}) | s_t = s, a_t = \pi'(s_t)] \\ & \leq \dots \\ & \leq \mathbb{E}[r_t + r_{t+1} + r_{t+2} + \dots | s_t = s, a_t = \pi'(s_t)] \\ & = V_{\pi'}(s) \end{aligned} \quad (6.15)$$

因此

$$V_\pi(s) \leq V_{\pi'}(s) \quad (6.16)$$

我们可以估计某一个策略的 Q 函数, 接下来就可以找到另外一个策略 π' 比原来的策略 π 还要更好。

6.3 目标网络

接下来讲一些在深度 Q 网络里一定会用到的技巧。第一个技巧是目标网络 (target network)。我们在学习 Q 函数的时候, 也会用到时序差分方法的概念。我们现在收集到一个数据, 比如在状态 s_t 采取动作 a_t 以后, 得到奖励 r_t , 进入状态 s_{t+1} 。根据 Q 函数, 我们可知

$$Q_\pi(s_t, a_t) = r_t + Q_\pi(s_{t+1}, \pi(s_{t+1})) \quad (6.17)$$

所以我们在学习的时候, Q 函数输入 s_t, a_t 得到的值, 与输入 $s_{t+1}, \pi(s_{t+1})$ 得到的值之间, 我们希望它们相差 r_t , 这与时序差分方法的概念是一样的。但是实际上这样的输入并不好学习, 假设这是一个回归问题, 如图 6.9 所示, $Q_\pi(s_t, a_t)$ 是网络的输出, $r_t + Q_\pi(s_{t+1}, \pi(s_{t+1}))$ 是目标, 目标是会变动的。当然如果我们要实现这样的训练, 其实也没有问题, 就是在做反向传播的时候, Q_π 的参数会被更新, 我们会把两个更新的结果加在一起 (因为它们是同一个模型 Q_π , 所以两个更新的结果会加在一起)。但这样会导致训练变得不太稳定, 因为假设我们把 $Q_\pi(s_t, a_t)$ 当作模型的输出, 把 $r_t + Q_\pi(s_{t+1}, \pi(s_{t+1}))$ 当作目标, 我们要去拟合的目标是一直在变动的, 这是不太好训练的。

所以我们会把其中一个 Q 网络, 通常是把图 6.9 右边的 Q 网络固定住。在训练的时候, 我们只更新左边的 Q 网络的参数, 而右边的 Q 网络的参数会被固定。因为右边的 Q 网络负责产生目标, 所以被称为目标网络。因为目标网络是固定的, 所以现在得到的目标 $r_t + Q_\pi(s_{t+1}, \pi(s_{t+1}))$ 的值也是固定的。我们只调整左边 Q 网络的参数, 它就变成一个回归问题。我们希望模型输出的值与目标越接近越好, 这样会最小化它的均方误差 (mean square error)。

在实现的时候, 我们会把左边的 Q 网络更新多次, 再用更新过的 Q 网络替换目标网络。但这两个网络不要一起更新, 一起更新, 结果会很容易不好。一开始这两个网络是一样的, 在训练的时候, 我们会把右边的 Q 网络固定住, 在做梯度下降的时候, 只调整左边 Q 网络的参数。我们可能更新 100 次以后才把

参数复制到右边的网络中，把右边网络的参数覆盖，目标值就变了。就好像我们本来在做一个回归问题，训练后把这个回归问题的损失降下去以后，接下来我们把左边网络的参数复制到右边网络，目标值就变了，接下来就要重新训练。

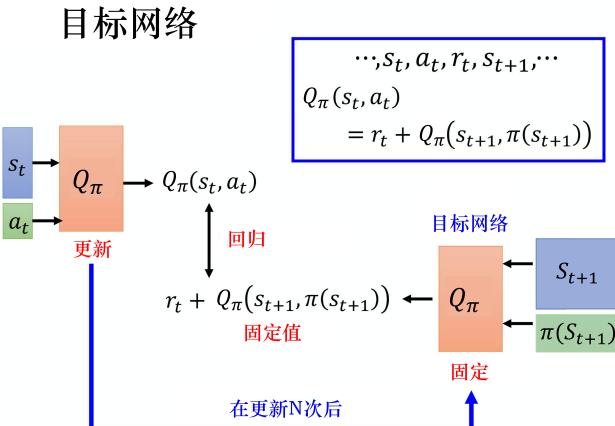


图 6.9 目标网络

如图 6.10a 所示，我们可以通过猫追老鼠的例子来直观地理解固定目标网络的目的。猫是 Q 估计，老鼠是 Q 目标。一开始，猫离老鼠很远，所以我们想让猫追上老鼠。如图 6.10b 所示，因为 Q 目标也是与模型参数相关的，所以每次优化后，Q 目标也会动。这就导致一个问题，猫和老鼠都在动。如图 6.10c 所示，猫和老鼠会在优化空间里面到处乱动，这会产生非常奇怪的优化轨迹，使得训练过程十分不稳定。所以我们可以固定 Q 网络，让老鼠动得不那么频繁，可能让它每 5 步动一次，猫则是每一步都在动。如果老鼠每 5 次动一步，猫就有足够的时间来接近老鼠，它们之间的距离会随着优化过程越来越小，最后它们就可以拟合，拟合后就可以得到一个最好的 Q 网络。



图 6.10 固定目标网络

6.4 探索

第二个技巧是探索。当我们使用 Q 函数的时候，策略完全取决于 Q 函数。给定某一个状态，我们就穷举所有的动作，采取让 Q 值最大的动作，即

$$a = \arg \max_a Q(s, a) \quad (6.18)$$

使用 Q 函数来决定动作与使用策略梯度不一样，策略梯度的输出是随机的，它会输出一个动作的分布，我们根据这个动作的分布去采样，所以在策略梯度里面，我们每次采取的动作是不一样的，是有随机性的。像 Q 函数中，如果我们采取的动作总是固定的，会遇到的问题就是这不是一个好的收集数据的方式。假设我们要估测某一个状态，可以采取动作 a_1, a_2, a_3 。我们要估测在某一个状态采取某一个动作会得到的 Q 值，一定要在那一个状态采取过那一个动作，才能估测出它的值。如果没有在那个状态采取过那个动作，我们其实是估测不出它的值的。如果 Q 函数是一个网络，这个问题可能没有那么严重。但是一般而言，假设 Q 函数是一个表格，对于没有见过的状态-动作对，它是估不出值的。如果 Q 函数是网络，也会有类似的问题，只是没有那么严重。所以假设我们在某一个状态，动作 a_1, a_2, a_3 都没有采取过，估出来的 $Q(s, a_1), Q(s, a_2), Q(s, a_3)$ 的值可能都是一样的，都是一个初始值，比如 0，即

$$\begin{aligned} Q(s, a_1) &= 0 \\ Q(s, a_2) &= 0 \\ Q(s, a_3) &= 0 \end{aligned} \quad (6.19)$$

但是如图 6.11 所示，假设我们在状态 s 采取动作 a_2 ，它得到的值是正的奖励， $Q(s, a_2)$ 就会比其他动作的 Q 值要大。在采取动作的时候，谁的 Q 值最大就采取谁，所以之后永远都只会采取 a_2 ，其他的动作就再也不会被采取了，这就会有问题。比如我们去一个餐厅吃饭。假设我们点了某一样菜，比如椒麻鸡，我们觉得还可以。接下来我们每次去就都会点椒麻鸡，再也不点别的菜了，那我们就不知道别的菜是不是会比椒麻鸡好吃，这是一样的问题。

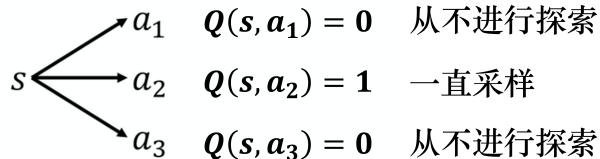


图 6.11 探索

如果我们没有好的探索，在训练的时候就会遇到这种问题。例如，假设我们用深度 Q 网络来玩 slither.io 网页游戏。我们有一条蛇，它在环境里面走来走去，吃到星星，就加分。假设游戏一开始，蛇往上走，然后吃到星星，就可以得到分数，它就知道往上走可以得到奖励。接下来它就再也不会采取往上走以外的动作了，以后就会变成每次游戏一开始，它就往上走，然后游戏结束。所以需要有探索的机制，让智能体知道，虽然根据之前采样的结果， a_2 好像是不错的，但我们至少偶尔也试一下 a_1 与 a_3 ，说不定它们更好。

这个问题就是探索-利用窘境 (exploration-exploitation dilemma) 问题，有两个方法可以解决这个问题： ϵ -贪心和玻尔兹曼探索 (Boltzmann exploration)。

ϵ -贪心是指我们有 $1 - \epsilon$ 的概率会按照 Q 函数来决定动作，可写为

$$a = \begin{cases} \arg \max_a Q(s, a) & , \text{有 } 1 - \epsilon \text{ 的概率} \\ \text{随机} & , \text{否则} \end{cases} \quad (6.20)$$

通常将 ϵ 设为一个很小的值， $1 - \epsilon$ 可能是 0.9，也就是 0.9 的概率会按照 Q 函数来决定动作，但是我们有 0.1 的概率是随机的。通常在实现上 ϵ 会随着时间递减。在最开始的时候，因为不知道哪个动作是

比较好的，所以我们会花比较大的力气探索。接下来，随着训练的次数越来越多，我们已经比较确定哪个动作是比较好的。我们就会减少探索，会把 ϵ 的值变小，主要根据 Q 函数来决定动作，比较少随机决定动作，这就是 ϵ -贪心。

还有一个方法称为玻尔兹曼探索。在玻尔兹曼探索中，我们假设对于任意的 s, a , $Q(s, a) \geq 0$ ，因此 a 被选中的概率与 $e^{Q(s,a)/T}$ 呈正比，即

$$\pi(a | s) = \frac{e^{Q(s,a)/T}}{\sum_{a' \in A} e^{Q(s,a')/T}} \quad (6.21)$$

其中， $T > 0$ 称为温度系数。如果 T 很大，所有动作几乎以等概率选择（探索）；如果 T 很小，Q 值大的动作更容易被选中（利用）；如果 T 趋于 0，我们就只选择最优动作。

6.5 经验回放

第三个技巧是经验回放（experience replay）。如图 6.12 所示，经验回放会构建一个回放缓冲区（replay buffer），回放缓冲区又被称为回放内存（replay memory）。回放缓冲区是指现在有某一个策略 π 与环境交互，它会去收集数据，我们把所有的数据放到一个数据缓冲区（buffer）里面，数据缓冲区里面存储了很多数据。比如数据缓冲区可以存储 5 万笔数据，每一笔数据就是记得说，我们之前在某一个状态 s_t ，采取某一个动作 a_t ，得到了奖励 r_t ，进入状态 s_{t+1} 。我们用 π 去与环境交互多次，把收集到的数据放到回放缓冲区里面。回放缓冲区里面的经验可能来自不同的策略，我们每次用 π 与环境交互的时候，可能只交互 10000 次，接下来我们就更新 π 了。但是回放缓冲区里面可以放 5 万笔数据，所以 5 万笔数据可能来自不同的策略。回放缓冲区只有在它装满的时候，才会把旧的数据丢掉。所以回放缓冲区里面其实装了很多不同的策略的经验。

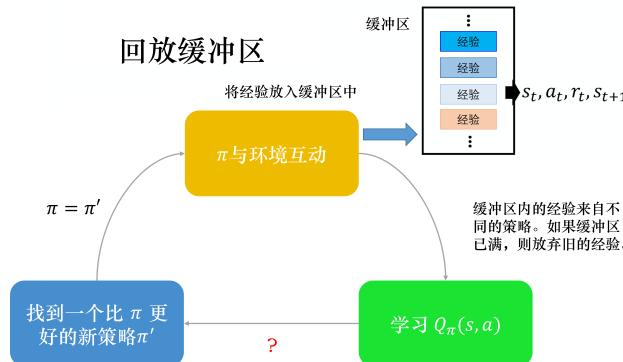


图 6.12 经验回放

如图 6.13 所示，有了回放缓冲区以后，我们怎么训练 Q 模型、怎么估 Q 函数呢？我们会迭代地训练 Q 函数，在每次迭代里面，从回放缓冲区中随机挑一个批量（batch）出来，即与一般的网络训练一样，从训练集里面挑一个批量出来。我们采样该批量出来，里面有一些经验，我们根据这些经验去更新 Q 函数。这与时序差分学习要有一个目标网络是一样的。我们采样一个批量的数据，得到一些经验，再去更新 Q 函数。

如果某个算法使用了经验回放这个技巧，该算法就变成了一个异策略的算法。因为本来 Q 是要观察 π 的经验的，但实际上存储在回放缓冲区里面的这些经验不是通通来自于 π ，有些是过去其他的策略所留下的经验。因为我们不会用某一个 π 就把整个回放缓冲区装满，拿去测 Q 函数， π 只是采样一些数据放到回放缓冲区里面，接下来就让 Q 去训练。所以 Q 在采样的时候，它会采样到过去的一些数据。

这么做有两个好处。第一个好处是，在进行强化学习的时候，往往最花时间的步骤是与环境交互，训练网络反而是比较快的。因为我们用 GPU 训练其实很快，真正花时间的往往是与环境交互。用回放缓冲区可以减少与环境交互的次数，因为在做训练的时候，经验不需要通通来自于某一个策略。一些过去的策

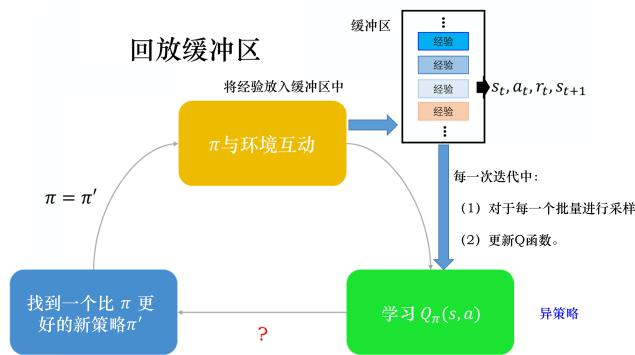


图 6.13 使用回放缓冲区训练 Q 函数

略所得到的经验可以放在回放缓冲区里面被使用很多次，被反复的再利用，这样可以比较高效地采样经验。第二个好处是，在训练网络的时候，其实我们希望一个批量里面的数据越多样（diverse）越好。如果批量里面的数据都是同样性质的，我们训练下去，训练结果是容易不好的。如果批量里面都是一样的数据，训练的时候，性能会比较差。我们希望批量里的数据越多样越好。如果回放缓冲区里面的经验通通来自于不同的策略，我们采样到的一个批量里面的数据会是比较多样化的。

Q：我们观察 π 的值，发现里面混杂了一些不是 π 的经验，这有没有关系？

A：没关系。这并不是因为过去的策略与现在的策略很像，就算过去的策略与现在的策略不是很像，也是没有关系的。主要的原因是我们并不是去采样一个轨迹，我们只采样了一笔经验，所以与是不是异策略这件事是没有关系的。就算是异策略，就算是这些经验不是来自于 π ，我们还是可以用这些经验来估测 $Q_{\pi}(s, a)$ 。

6.6 深度 Q 网络

图 6.14 所示为一般的深度 Q 网络算法。深度 Q 网络算法是这样的，我们初始化两个网络—— Q 和 \hat{Q} ， \hat{Q} 就等于 Q 。一开始目标 Q 网络与原来的 Q 网络是一样的。在每一个回合中，我们用演员与环境交互，在每一次交互的过程中，都会得到一个状态 s_t ，会采取某一个动作 a_t 。怎么知道采取哪一个动作 a_t 呢？我们就根据现在的 Q 函数，但是要有探索的机制。比如我们用玻尔兹曼探索或是 ϵ -贪心探索，接下来得到奖励 r_t ，进入状态 s_{t+1} 。所以现在收集到一笔数据 (s_t, a_t, r_t, s_{t+1}) ，我们将其放到回放缓冲区里面。如果回放缓冲区满了，我们就把一些旧的数据丢掉。接下来我们就从回放缓冲区里面去采样数据，采样到的是 (s_i, a_i, r_i, s_{i+1}) 。这笔数据与刚放进去的不一定是同一笔，我们可能抽到旧的。要注意的是，我们采样出来不是一笔数据，采样出来的是一个批量的数据，采样一些经验出来。接下来就是计算目标。假设我们采样出一笔数据，根据这笔数据去计算目标。目标要用目标网络 \hat{Q} 来计算。目标是：

$$y = r_i + \max_a \hat{Q}(s_{i+1}, a) \quad (6.22)$$

其中， a 是让 \hat{Q} 值最大的动作。因为我们在状态 s_{i+1} 会采取的动作 a 就是可以让 Q 值最大的那一个动作。接下来我们要更新 Q 值，就把它当作一个回归问题。我们希望 $Q(s_i, a_i)$ 与目标越接近越好。假设已经更新了一定的次数，比如 C 次，设 $C = 100$ ，那我们就把 \hat{Q} 设成 Q ，这就是深度 Q 网络算法。

Q：深度 Q 网络和 Q 学习有什么不同？

A：整体来说，深度 Q 网络与 Q 学习的目标价值以及价值的更新方式都非常相似。主要的不同点在于：深度 Q 网络将 Q 学习与深度学习结合，用深度网络来近似动作价值函数，而 Q 学习则是采用表格存储；深度 Q 网络采用了经验回放的训练方法，从历史数据中随机采样，而 Q 学习直接采用下一个状态的数据进行学习。

- 初始化函数 Q 、目标函数 \hat{Q} ，令 $\hat{Q} = Q$ 。
- 对于每一个回合。
 - 对于每一个时间步 t 。
 - 对于给定的状态 s_t ，基于 Q (ϵ -贪婪) 执行动作 a_t 。
 - 获得反馈 r_t ，并获得新的状态 s_{t+1} 。
 - 将 (s_t, a_t, r_t, s_{t+1}) 存储到缓冲区中。
 - 从缓冲区中采样（通常以批量形式） (s_i, a_i, r_i, s_{i+1}) 。
 - 目标值是 $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$ 。
 - 更新 Q 的参数使得 $Q(s_i, a_i)$ 尽可能接近于 y （回归）。
 - 每 C 次更新重置 $\hat{Q} = Q$ 。

图 6.14 深度 Q 网络算法

6.7 关键词

深度 Q 网络 (deep Q-network, DQN): 基于深度学习的 Q 学习算法，其结合了价值函数近似 (value function approximation) 与神经网络技术，并采用目标网络和经验回放等方法进行网络的训练。

状态-价值函数 (state-value function): 其输入为演员某一时刻的状态，输出为一个标量，即当演员在对应的状态时，预期的到过程结束时间段内所能获得的价值。

状态-价值函数贝尔曼方程 (state-value function Bellman equation): 基于状态-价值函数的贝尔曼方程，它表示在状态 s_t 下对累积奖励 G_t 的期望。

Q 函数 (Q-function): 其也被称为动作价值函数 (action-value function)。其输入是一个状态-动作对，即在某一具体的状态采取对应的动作，假设我们都使用某个策略 π ，得到的累积奖励的期望值有多大。

目标网络 (target network) : 其可解决在基于时序差分的网络中，优化目标 $Q_\pi(s_t, a_t) = r_t + Q_\pi(s_{t+1}, \pi(s_{t+1}))$ 左右两侧会同时变化使得训练过程不稳定，从而增大回归的难度的问题。目标网络选择将右边部分，即 $r_t + Q_\pi(s_{t+1}, \pi(s_{t+1}))$ 固定，通过改变左边部分，即 $Q_\pi(s_t, a_t)$ 中的参数进行回归，这也是深度 Q 网络应用中比较重要的技巧。

探索 (exploration): 我们在使用 Q 函数的时候，我们的策略完全取决于 Q 函数，这有可能导致出现对应的动作是固定的某几个数值的情况，而不像策略梯度中的输出是随机的，我们再从随机分布中采样选择动作。这会导致我们继续训练的输入值一样，从而“加重”输出的固定性，导致整个模型的表达能力急剧下降，这就是探索-利用窘境 (exploration-exploitation dilemma) 问题。我们可以使用 ϵ -贪心和玻尔兹曼探索 (Boltzmann exploration) 等探索方法进行优化。

经验回放 (experience replay): 其会构建一个回放缓冲区 (replay buffer) 来保存许多经验，每一个经验的形式如下：在某一个状态 s_t ，采取某一个动作 a_t ，得到奖励 r_t ，然后进入状态 s_{t+1} 。我们使用 π 与环境交互多次，把收集到的经验都存储在回放缓冲区中。当我们的缓冲区“装满”后，就会自动删去最早进入缓冲区的经验。在训练时，对于每一轮迭代都有相对应的批量 (batch)（与我们训练普通的网络一样，都是通过采样得到的），然后用这个批量中的经验去更新我们的 Q 函数。综上，Q 函数在采样和训练的时候，会用到过去的经验，所以这里称这个方法为经验回放，其也是深度 Q 网络应用中比较重要的技巧。

6.8 习题

6-1 为什么在深度 Q 网络中采用价值函数近似的表示方法？

6-2 评论员的输出通常与哪几个值直接相关？

6-3 我们通常怎么衡量状态价值函数 $V_\pi(s)$ ？其优势和劣势分别有哪些？

6-4 基于本章正文介绍的基于蒙特卡洛的网络方法，我们怎么训练模型呢？或者我们应该将其看作机器学习中什么类型的问题呢？

6-5 基于本章正文中介绍的基于时序差分的网络方法，具体地，我们应该怎么训练模型呢？

6-6 动作价值函数和状态价值函数的有什么区别和联系？

6-7 请介绍 Q 函数的两种表示方法。

6-8 当得到了 Q 函数后，我们应当如何找到更好的策略 π' 呢？或者说 π' 的本质是什么？

6-9 解决探索-利用窘境问题的探索的方法有哪些？

6-10 我们使用经验回放有什么好处？

6-11 在经验回放中我们观察 π 的价值，发现里面混杂了一些不是 π 的经验，这会有影响吗？

6.9 面试题

6-1 友善的面试官：请问深度 Q 网络是什么？其两个关键性的技巧分别是什么？

6-2 友善的面试官：那我们继续分析！你刚才提到的深度 Q 网络中的两个技巧——目标网络和经验回放，其具体作用是什么呢？

6-3 友善的面试官：深度 Q 网络和 Q 学习有什么异同点？

6-4 友善的面试官：请问，随机性策略和确定性策略有什么区别吗？

6-5 友善的面试官：请问不打破数据相关性，神经网络的训练效果为什么就不好？

参考文献

- [1] MNIH V, KAVUKCUOGLU K, SILVER D, et al. Human-level control through deep reinforcement learning[J]. nature, 2015, 518(7540): 529-533.

第 7 章 深度 Q 网络进阶技巧

7.1 双深度 Q 网络

本章我们介绍训练深度 Q 网络的一些技巧。第一个技巧是双深度 Q 网络 (double DQN, DDQN)。为什么要有 DDQN 呢？因为在实现上，Q 值往往是被高估的。如图 7.1 所示，这里有 4 个不同的小游戏，横轴代表迭代轮次，红色锯齿状的一直在变的线表示 Q 函数对不同的状态估计的平均 Q 值，有很多不同的状态，每个状态我们都进行采样，算出它们的 Q 值，然后进行平均。这条红色锯齿状的线在训练的过程中会改变，但它是不断上升的。因为 Q 函数是取决于策略的，在学习的过程中策略越来越强，我们得到的 Q 值会越来越大。在同一个状态，我们得到奖励的期望会越来越大，所以一般而言，Q 值都是上升的，但这是深度 Q 网络预估出来的值。接下来我们就用策略去玩游戏，玩很多次，比如 100 万次，然后计算在某一个状态下，我们得到的 Q 值是多少。我们会得到在某一个状态采取某一个动作的累积奖励是多少。预估出来的值远比真实值大，且大很多，在每一个游戏中都是这样。所以 DDQN 的方法可以让预估值与真实值比较接近。

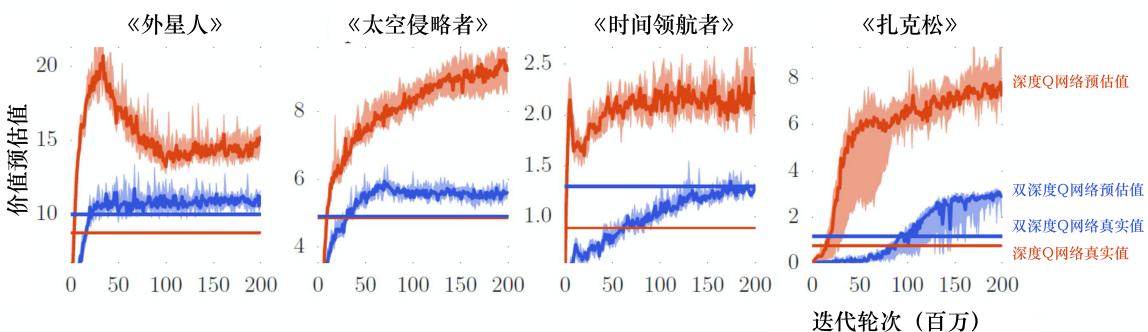


图 7.1 被高估的 Q 值^[1]

图 7.1 中蓝色的锯齿状的线是 DDQN 的 Q 网络所估测出来的 Q 值，蓝色的无锯齿状的线是真正的 Q 值，它们是比较接近的。我们不用管用网络估测的值，它比较没有参考价值。我们用 DDQN 得出的真正的 Q 值在图 7.1 的 3 种情况下都是比原来的深度 Q 网络高的，代表 DDQN 学习出来的策略比较强，所以实际上得到的奖励是比较大的。虽然一般的深度 Q 网络的 Q 网络高估了自己会得到的奖励，但实际上它得到的奖励是比较低的。

Q: 为什么 Q 值总是被高估了？

A: 因为实际在训练的时候，如式 (7.1) 所示，我们要让左式与右式（目标）越接近越好。但目标的值很容易被设得太高，因为在计算目标的时候，我们实际上在做的，是看哪一个 a 可以得到最大的 Q 值，就把它加上去变成目标。

$$Q(s_t, a_t) \longleftrightarrow r_t + \max_a Q(s_{t+1}, a) \quad (7.1)$$

例如，假设我们现在有 4 个动作，本来它们得到的 Q 值都是差不多的，它们得到的奖励也是差不多的。但是在估计的时候，网络是有误差的。如图 7.2 (a) 所示，假设是第一个动作被高估了，绿色代表是被高估的量，智能体就会选这个动作，就会选这个高估的 Q 值来加上 r_t 来当作目标。如图 7.2 (b) 所示，如果第四个动作被高估了，智能体就会选第四个动作来加上 r_t 当作目标。所以智能体总是会选那个 Q 值被高估的动作，总是会选奖励被高估的动作的 Q 值当作最大的结果去加上 r_t 当作目标，所以目标值总是太大。

Q: 怎么解决目标值总是太大的问题呢？

A: 在 DDQN 里面，选动作的 Q 函数与计算值的 Q 函数不是同一个。在原来的深度 Q 网络里面，我们穷举所有的 a ，把每一个 a 都代入 Q 函数，看哪一个 a 可以得到的 Q 值最高，就把那个 Q 值加上 r_t 。

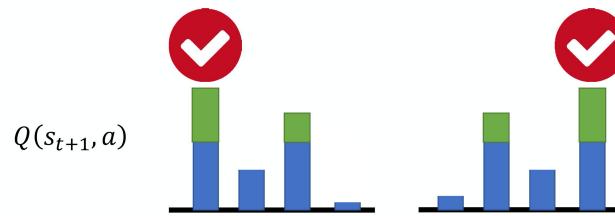


图 7.2 Q 值被高估的问题

但是在 DDQN 里面有两个 Q 网络，第一个 Q 网络 Q 决定哪一个动作的 Q 值最大（我们把所有的 a 代入 Q 函数中，看看哪一个 a 的 Q 值最大）。我们决定动作以后，Q 值是用 Q' 算出来的。

如式 (7.2) 所示，假设我们有两个 Q 函数—— Q 和 Q' ，如果 Q 高估了它选出来的动作 a ，只要 Q' 没有高估动作 a 的值，算出来的就还是正常的值。假设 Q' 高估了某一个动作的值，也是没问题的，因为只要 Q 不选这个动作就可以，这就是 DDQN 神奇的地方。

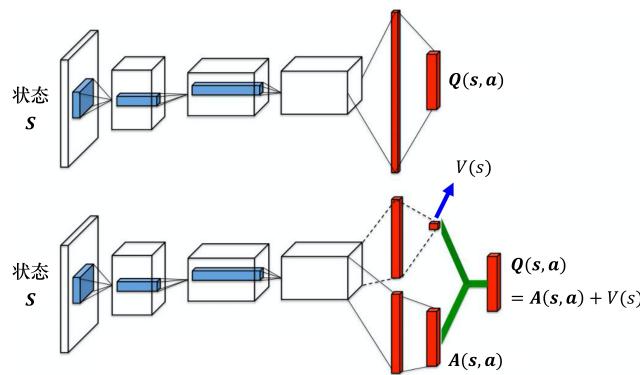
$$Q(s_t, a_t) \longleftrightarrow r_t + Q' \left(s_{t+1}, \arg \max_a Q(s_{t+1}, a) \right) \quad (7.2)$$

我们动手实现的时候，有两个 Q 网络：会更新的 Q 网络和目标 Q 网络。所以在 DDQN 里面，我们会用会更新参数的 Q 网络去选动作，用目标 Q 网络（固定住的网络）计算值。

DDQN 相较于原来的深度 Q 网络的更改是最少的，它几乎没有增加任何的运算量，也不需要新的网络，因为原来就有两个网络。我们只需要做一件事：本来是用目标网络 Q' 来找使 Q 值最大的 a ，现在改用另外一个会更新的 Q 网络来找使 Q 值最大的 a 。如果只选一个技巧，我们一般都会选 DDQN，因为其很容易实现。

7.2 竞争深度 Q 网络

第二个技巧是竞争深度 Q 网络（dueling DQN），相较于原来的深度 Q 网络，它唯一的差别是改变了网络的架构。Q 网络输入状态，输出的是每一个动作的 Q 值。如图 7.3 所示，原来的深度 Q 网络直接输出 Q 值，竞争深度 Q 网络不直接输出 Q 值，而是分成两条路径运算。第一条路径会输出一个标量 $V(s)$ ，因为它与输入 s 是有关系的，所以称为 $V(s)$ 。第二条路径会输出一个向量 $A(s, a)$ ，它的每一个动作都有一个值。我们再把 $V(s)$ 和 $A(s, a)$ 加起来就可以得到 Q 值 $Q(s, a)$ 。

图 7.3 竞争深度 Q 网络的网络结构^[2]

我们知道状态是离散的（实际上状态不是离散的），为了说明方便，我们假设就只有 4 个不同的状态，只有 3 个不同的动作，所以 $Q(s, a)$ 可以看成一个表格，如图 7.4 所示。

我们知道

$$Q(s, a) = V(s) + A(s, a)$$

其中， $V(s)$ 对不同的状态，都有一个值。 $A(s, a)$ 对不同的状态、不同的动作都有一个值。我们把 $V(s)$ 的每一列的值加到 $A(s, a)$ 的每一列就可以得到 Q 值，以第一列为为例，有 $2+1$ 、 $2+(-1)$ 、 $2+0$ ，可以得到 3 、 1 、 2 ，以此类推。

如图 7.4 所示，假设我们在训练网络的时候，目标是希望 Q 表格中第一行第二列的值变成 4 ，第二行第二列的值变成 0 。但是我们实际上能修改的并不是 Q 值，能修改的是 $V(s)$ 与 $A(s, a)$ 的值。根据网络的参数， $V(s)$ 与 $A(s, a)$ 的值输出以后，就直接把它们加起来，所以其实不是修改 Q 值。在学习网络的时候，假设我们希望 Q 表格中的 3 增加 1 变成 4 、 -1 增加 1 变成 0 。最后我们在训练网络的时候，我们可能就不用修改 $A(s, a)$ 的值，就修改 $V(s)$ 的值，把 $V(s)$ 的值从 0 变成 1 。从 0 变成 1 有什么好处呢？本来只想修改两个值，但 Q 表格中的第三个值也被修改了： -2 变成了 -1 。所以有可能我们在某一个状态下，只采样到这两个动作，没采样到第三个动作，但也可以更改第三个动作的 Q 值。这样的好处就是我们不需要把所有的状态-动作对都采样，可以用比较高效的方式去估计 Q 值。因为有时候我们更新的时候，不一定是更新 Q 表格，而是只更新了 $V(s)$ ，但更新 $V(s)$ 的时候，只要修改 $V(s)$ 的值，Q 表格的值也会被修改。竞争深度 Q 网络是一个使用数据比较有效率的方法。



图 7.4 竞争深度 Q 网络训练

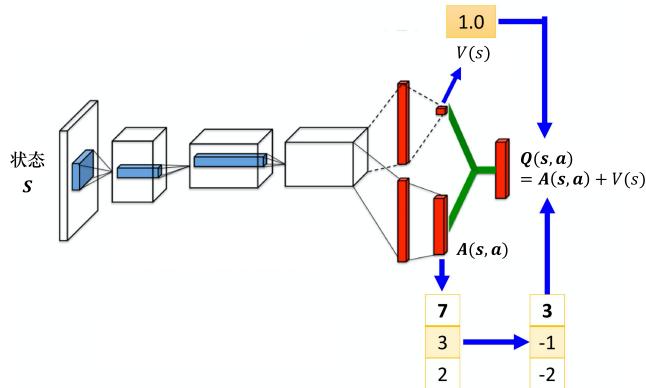
可能会有人认为使用竞争深度 Q 网络会有一个问题，竞争深度 Q 网络最后学习的结果可能是这样的：智能体就学到 $V(s)$ 等于 0 ， $A(s, a)$ 等于 Q ，使用任何竞争深度 Q 网络就没有任何好处，就和原来的深度 Q 网络一样。为了避免这个问题出现，实际上我们要给 $A(s, a)$ 一些约束，让 $A(s, a)$ 的更新比较麻烦，让网络倾向于使用 $V(s)$ 来解决问题。

例如，我们有不同的约束，一个最直觉的约束是必须要让 $A(s, a)$ 的每一列的和都是 0 ，所以看我这边举的例子，列的和都是 0 。如果这边列的和都是 0 ，我们就可以把 $V(s)$ 的值想成是上面 Q 的每一列的平均值。这个平均值，加上 $A(s, a)$ 的值才会变成是 Q 的值。所以假设在更新参数的时候，要让整个列一起被更新，更新 $A(s, a)$ 的某一列比较麻烦，所以我们就不会想要更新 $A(s, a)$ 的某一列。因为 $A(s, a)$ 的每一列的和都要是 0 ，所以我们无法让 $A(s, a)$ 的某列的值都加 1 ，这是做不到的，因为它的约束就是和永远都是 0 ，所以不可以都加 1 ，这时候就会强迫网络去更新 $V(s)$ 的值，让我们可以用比较有效率的方法去使用数据。

实现时，我们要给这个 $A(s, a)$ 一个约束。例如，如图 7.5 所示，假设有 3 个动作，输出的向量是 $[7, 3, 2]^T$ ，我们在把 $A(s, a)$ 与 $V(s)$ 加起来之前，先进行归一化（normalization）。归一化的过程如下：

- (1) 计算均值 $(7+3+2) / 3 = 4$ ；
- (2) 向量 $[7, 3, 2]^T$ 的每个元素的值都减去均值 4 ，于是归一化的向量为 $[3, -1, 2]^T$ 。

接着我们将向量 $[3, -1, 2]^T$ 中的每个元素的值加上 1 ，就可以得到最后的 Q 值。这个归一化的步骤就是网络的其中一部分，在训练的时候，我们也使用反向传播，只是归一化是没有参数的，它只是一个操作，可以把它放到网络里面，与网络的其他部分共同训练，这样 $A(s, a)$ 就会有比较大的约束，网络就会给它一些好处，让它倾向于去更新 $V(s)$ 的值，这就是竞争深度 Q 网络。

图 7.5 竞争深度 Q 网络约束^[2]

7.3 优先级经验回放

第三个技巧称为优先级经验回放 (prioritized experience replay, PER)。如图 7.6 所示，我们原来在采样数据训练 Q 网络的时候，会均匀地从回放缓冲区里面采样数据。这样不一定是最好的，因为也许有一些数据比较重要。假设有一些数据，我们之前采样过，发现这些数据的时序差分误差特别大（时序差分误差就是网络的输出与目标之间的差距），这代表我们在训练网络的时候，这些数据是比较不好训练的。既然比较不好训练，就应该给它们比较大的概率被采样到，即给它优先权 (priority)。这样在训练的时候才会多考虑那些不好训练的数据。实际上在做 PER 的时候，我们不仅会更改采样的过程，还会因为更改了采样的过程，而更改更新参数的方法。所以 PER 不仅改变了采样数据的分布，还改变了训练过程。

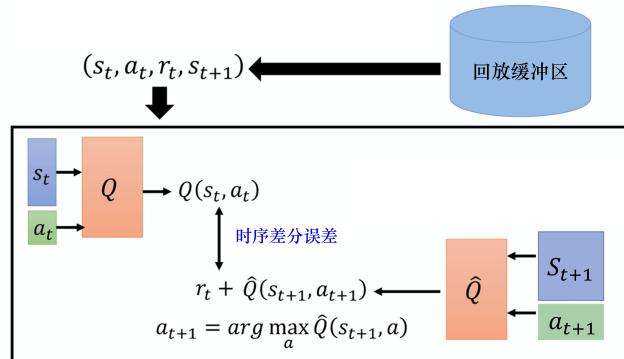


图 7.6 优先级经验回放

7.4 在蒙特卡洛方法和时序差分方法中取得平衡

蒙特卡洛方法与时序差分方法各有优劣，因此我们可以在蒙特卡洛方法和时序差分方法中取得平衡，这个方法也被称为多步方法。我们的做法如图 7.7 所示，在时序差分方法里面，在某一个状态 s_t 采取某一个动作 a_t 得到奖励 r_t ，接下来进入状态 s_{t+1} 。但是我们可以不只保存一个步骤的数据，可保存 N 个步骤的数据。

我们记录在 s_t 采取 a_t ，得到 r_t 时，会进入的 s_{t+1} 。一直记录到第 N 个步骤以后，在 s_{t+N} 采取 a_{t+N} ，得到 r_{t+N} ，进入 s_{t+N+1} 的这些经验，把它们保存下来。实际上在做更新的时候，在做 Q 网络学习的时候，我们要让 $Q(s_t, a_t)$ 与目标值越接近越好。 \hat{Q} 所计算的不是 s_{t+1} 的，而是 s_{t+N+1} 的奖励。我们会把 N 个步骤以后的状态 s_{t+N+1} 输入到 \hat{Q} 中去计算 N 个步骤以后会得到的奖励。如果要算目标值，要再加上多步 (multi-step) 的奖励 $\sum_{t'=t}^{t+N} r_{t'}$ ，多步的奖励是从时间 t 一直到 $t+N$ 的 $N+1$ 个奖励的和。我们希望 $Q(s_t, a_t)$ 和目标值越接近越好。

多步方法就是蒙特卡洛方法与时序差分方法的结合，因此它不仅有蒙特卡洛方法的好处与坏处，还有时序差分方法的好处与坏处。我们先看看多步方法的好处，之前只采样了某一个步骤，所以得到的数据是真实的，接下来都是 Q 值估测出来的。现在采样比较多的步骤，采样 N 个步骤才估测值，所以估测的部分所造成的影响就会比较小。当然多步方法的坏处就与蒙特卡洛方法的坏处一样，因为 r 有比较多项，所以我们把 N 项的 r 加起来，方差就会比较大。但是我们可以调整 N 的值，在方差与不精确的 Q 值之间取得一个平衡。 N 就是一个超参数，我们可以对其进行调整。

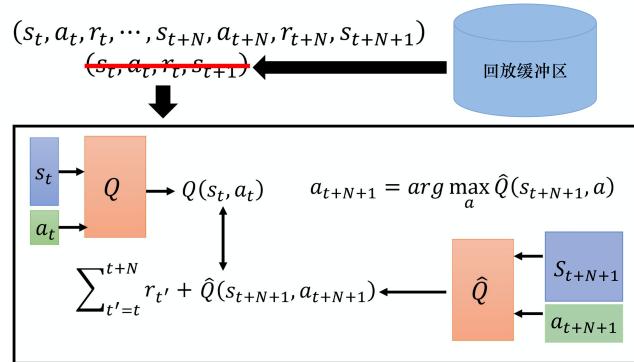


图 7.7 在蒙特卡洛方法和时序差分方法中取得平衡

7.5 噪声网络

我们还可以改进探索。 ϵ -贪心这样的探索就是在动作的空间上加噪声，但是有一个更好的方法称为**噪声网络 (noisy net)**，它是在参数的空间上加噪声。噪声网络是指，每一次在一个回合开始的时候，在智能体要与环境交互的时候，智能体使用 Q 函数来采取动作，Q 函数里面就是一个网络，我们在网络的每一个参数上加上一个高斯噪声 (Gaussian noise)，就把原来的 Q 函数变成 \tilde{Q} 。因为我们已经用 \hat{Q} 来表示目标网络，所以我们用 \tilde{Q} 来表示**噪声 Q 函数 (noisy Q-function)**。我们把每一个参数都加上一个高斯噪声，就得到一个新的网络 \tilde{Q} 。使用噪声网络执行的动作

$$a = \arg \max_a \tilde{Q}(s, a) \quad (7.3)$$

这里要注意，在每个回合开始的时候，与环境交互之前，我们就采样噪声。接下来我们用固定的噪声网络玩游戏，直到游戏结束，才重新采样新的噪声，噪声在一个回合中是不能被改变的。OpenAI 与 DeepMind 在同时间提出了几乎一模一样的噪声网络方法，并且对应的两篇论文都发表在 ICLR 2018 会议中。不一样的地方是，他们用不同的方法加噪声。OpenAI 的方法比较简单，直接加一个高斯噪声，也就是把每一个参数、每一个权重 (weight) 都加一个高斯噪声。DeepMind 的方法比较复杂，该方法中的噪声是由一组参数控制的，网络可以自己决定噪声要加多大。但是两种方法的概念都是一样的，总之，我们就是对 Q 函数里面的网络加上一些噪声，把它变得有点儿不一样，即与原来的 Q 函数不一样，然后与环境交互。两篇论文里面都强调，参数虽然会被加上噪声，但在同一个回合里面参数是固定的。我们在换回合、玩另一场新的游戏的时候，才会重新采样噪声。在同一场比赛里面就是同一个噪声 Q 网络在玩该场比赛，这非常重要。因为这导致了噪声网络与原来的 ϵ -贪心或其他在动作上做采样的方法的本质上的差异。

有什么本质上的差异呢？在原来采样的方法中，比如 ϵ -贪心中，就算给定同样的状态，智能体采取的动作也不一定是一样的。因为智能体通过采样来决定动作，给定同一个状态，智能体根据 Q 函数的网络来输出一个动作，或者采样到随机来输出一个动作。所以给定相同的状态，如果是用 ϵ -贪心的方法，智能体可能会执行不同的动作。但实际上策略并不是这样的，一个真实世界的策略，给定同样的状态，它应该有同样的回应。而不是给定同样的状态，它有时候执行 Q 函数，有时候又是随机的，这是一个不正常的动作，是在真实的情况下不会出现的动作。但是如果我们是在 Q 函数的网络的参数上加噪声，就不会出现

这种情况。因为如果在 Q 函数的网络的参数上加噪声，在整个交互的过程中，在同一个回合里面，它的网络的参数总是固定的，所以看到相同或类似的状态，就会采取相同动作，这是比较正常的。这被称为**依赖状态的探索 (state-dependent exploration)**，我们虽然会做探索这件事，但是探索是与状态有关系的，看到同样的状态，就会采取同样的探索的方式，而噪声 (noisy) 的动作只是随机乱试。但如果我们是在参数下加噪声，在同一个回合里面，参数是固定的，我们就是系统地尝试。比如，我们每次在某一个状态，都向左试试看。在下一次在玩同样游戏的时候，看到同样的状态，我再向右试试看，是系统地在探索环境。

7.6 分布式 Q 函数

还有一个技巧称为**分布式 Q 函数 (distributional Q-function)**。分布式 Q 函数是比较合理的，但是它难以实现。Q 函数是累积奖励的期望值，所以我们算出来的 Q 值其实是一个期望值。如图 7.8 所示，因为环境是有随机性的，所以在某一个状态采取某一个动作的时候，我们把所有的奖励在游戏结束的时候进行统计，得到的是一个分布。也许在奖励得到 0 的概率很高，得到 -10 的概率比较低，得到 +10 的概率也比较低，但是它是一个分布。(我们对这个分布计算它的平均值才是这个 Q 值，算出来是累积奖励的期望。所以累积奖励是一个分布，对它取期望，对它取平均值，得到 Q 值)。但不同的分布可以有同样的平均值。也许真正的分布是图 7.8 所示右边的分布，它的平均值与左边的分布的平均值其实是一样的，但它们背后所代表的分布其实是不一样的。假设我们只用 Q 值的期望来代表整个奖励，可能会丢失一些信息，无法对奖励的分布进行建模。

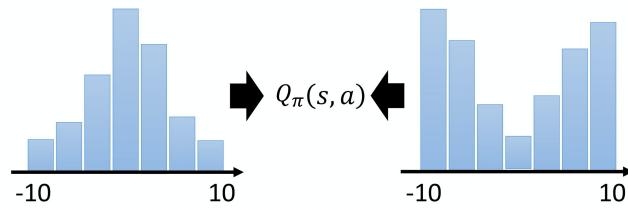


图 7.8 奖励分布

分布式 Q 函数是对分布 (distribution) 建模，怎么做呢？如图 7.9a 所示，在原来的 Q 函数里面，假设我们只能采取 a_1, a_2, a_3 这 3 个动作，我们输入一个状态，输出 3 个值。这 3 个值分别代表 3 个动作的 Q 值，但是这些 Q 值是一个分布的期望值。所以分布式 Q 函数就是直接输出分布。实际上的做法如图 7.9b 所示，假设分布的值就分布在某一个范围里面，比如 $-10 \sim 10$ ，把 $-10 \sim 10$ 拆成一个一个的长条。例如，每一个动作的奖励空间拆成 5 个长条。假设奖励空间可以拆成 5 个长条，Q 函数的输出就是要预测我们在某一个状态采取某一个动作得到的奖励，其落在某一个长条里面的概率。所以绿色长条概率的和应该是 1，其高度代表在某一个状态采取某一个动作的时候，它落在某一个长条内的概率。绿色的代表动作 a_1 ，红色的代表动作 a_2 ，蓝色的代表动作 a_3 。所以我们就可以用 Q 函数去估计 a_1 的分布、 a_2 的分布、 a_3 的分布。实际上在做测试的时候，我们选平均值最大的动作执行。

除了选平均值最大的动作以外，我们还可以对分布建模。例如，我们可以考虑动作的分布，如果分布方差很大，这代表采取这个动作虽然平均而言很不错，但也许风险很高，我们可以训练一个网络来规避风险。在两个动作平均值都差不多的情况下，也许可以选一个风险较小的动作来执行，这就是分布式 Q 函数的好处。

7.7 彩虹

最后一个技巧称为**彩虹 (rainbow)**，如图 7.10 所示，假设每个方法有一种自己的颜色（如果每一个单一颜色的线代表只用某一个方法），把所有的颜色组合起来，就变成“彩虹”，我们把原来的深度 Q 网络也算作一种方法，故有 7 种颜色。横轴代表训练过程的帧数，纵轴代表玩十几个雅达利小游戏的平均分数

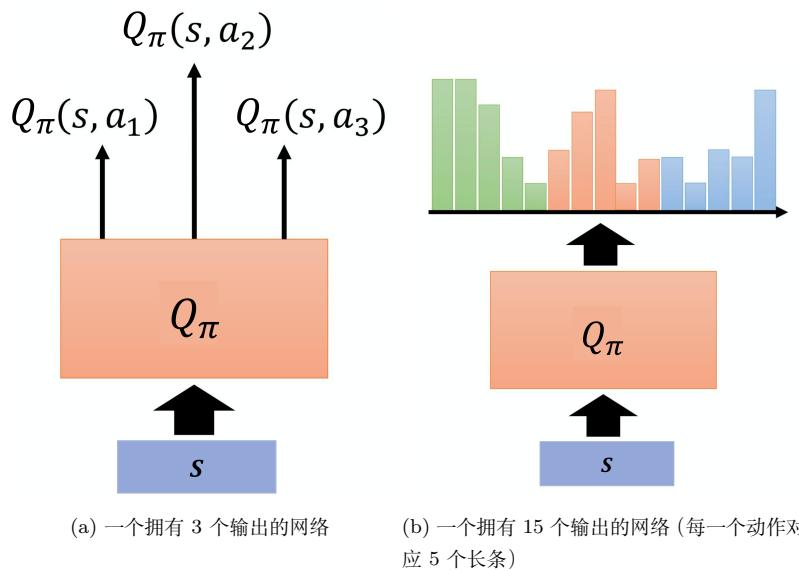
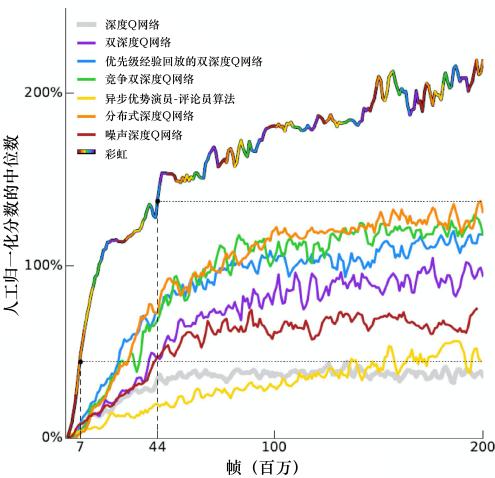


图 7.9 分布式 Q 函数

的和，但它取的是分数的中位数。为什么是取中位数而不是直接取平均呢？因为不同小游戏的分数差距很大，如果取平均，某几个游戏可能会控制结果，因此我们取中位数。如果我们使用的是一般的深度 Q 网络（灰色的线），深度 Q 网络的性能不是很好。噪声深度 Q 网络（noisy DQN）比 DQN 的性能好很多。紫色的线代表 DDQN，DDQN 还挺有效的。优先级经验回放的双深度 Q 网络（prioritized DDQN）、竞争双深度 Q 网络（dueling DDQN）和分布式深度 Q 网络（distributional DQN）性能也挺高的。异步优势演员-评论员（asynchronous advantage actor-critic, A3C）是演员-评论员的方法，A3C 算法又被译作异步优势动作评价算法，我们在第九章详细介绍异步优势演员-评论员算法。单纯的异步优势演员-评论员算法看起来是比深度 Q 网络强的。图 7.10 中没有多步方法，这是因为异步优势演员-评论员算法本身内部就有多步方法，所以实现异步优势演员-评论员算法就等同于实现多步方法，我们可以把异步优势演员-评论员算法的结果看成多步方法的结果。这些方法本身之间是没有冲突的，我们把全部方法都用上就变成了七彩的方法，即彩虹方法，彩虹方法的性能很好。

图 7.10 彩虹方法^[3]

我们把所有的方法加在一起，模型的表现会提高很多，但会不会有些方法其实是没有用的呢？我们可以去掉其中一种方法来判断这个方法是否有用。如图 7.11 所示，虚线就是彩虹方法去掉某一种方法以后的结果，黄色的虚线去掉多步方法后“掉”很多。彩虹是彩色的实线，去掉多步方法会“掉下来”，去掉优先级经验回放后会“掉下来”，去掉分布也会“掉下来”。这边有一个有趣的地方，在开始的时候，分布训练的

方法与其他方法速度差不多。但是我们去掉分布训练方法的时候，训练不会变慢，但是性能 (performance) 最后会收敛在比较差的地方。我们去掉噪声网络后性能也差一点儿，去掉竞争深度 Q 网络后性能也差一点儿，去掉双深度 Q 网络却没什么差别。所以我们把全部方法组合在一起的时候，去掉双深度 Q 网络是影响比较小的。当我们使用分布式深度 Q 网络的时候，本质上就不会高估奖励。我们是为了避免高估奖励才加了 DDQN。如果我们使用了分布式深度 Q 网络，就可能不会有高估的结果，多数的情况是低估奖励的，所以变成 DDQN 没有用。

为什么分布式深度 Q 网络不会高估奖励奖励，反而会低估奖励呢？因为分布式深度 Q 网络输出的是一个分布的范围，输出的范围不可能是无限的，我们一定会设一个限制，比如最大输出范围就是从 $-10 \sim 10$ 。假设得到的奖励超过 10，比如 100 怎么办？我们就当作没看到这件事，所以奖励很极端的值、很大的值是会被丢弃的，用分布式深度 Q 网络的时候，我们不会高估奖励，反而会低估奖励。

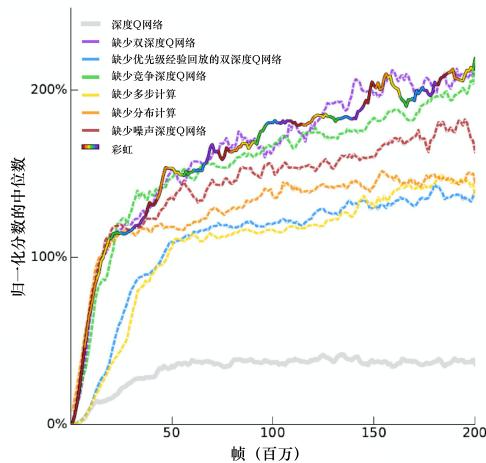


图 7.11 彩虹：去掉其中一种方法^[3]

7.8 使用深度 Q 网络解决推车杆问题

在学习本节之前，可以先回顾一下之前的项目实战，即使用 Q 学习解决悬崖寻路问题。本节将具体实现深度 Q 网络算法来解决推车杆问题，对应的模拟环境为 OpenAI Gym 中的 CartPole-v0，我们同样先对该环境做一个简要说明。

7.8.1 CartPole-v0 简介

CartPole-v0 是一个经典的入门环境，如图 7.12 所示，它通过向左（动作 = 0）或向右（动作 = 1）推动推车来实现推车杆的平衡。每次实施一个动作后，如果杆能够继续保持平衡，就会得到一个 +1 的奖励，否则杆将无法保持平衡而导致游戏结束。理论上最优算法情况下，推车杆是能够一直保证平衡的，但是如果每回合无限制地进行下去，会影响到算法的训练，所以环境一般设置每回合的最大步数为 200。另外 Gym 官方也推出了另外一版的推车杆环境，名为 CartPole-v1，相比 v0 版本，v1 每回合最大步数为 500，其他基本不变，可以说是 v0 的难度升级版。

我们来看看这个环境的一些参数，执行以下代码：

```
import gym
env = gym.make('CartPole-v0') # 建立环境
env.seed(1) # 随机种子
n_states = env.observation_space.shape[0] # 状态数
n_actions = env.action_space.n # 动作数
print(f"状态数: {n_states}, 动作数: {n_actions}")
```

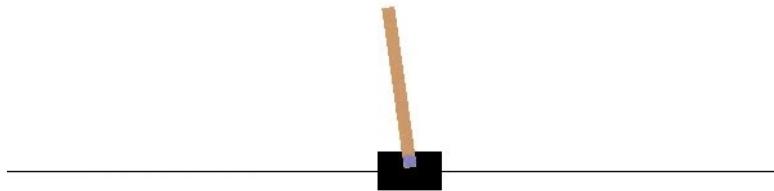


图 7.12 CartPole-v0 环境

可以得到结果：

状态数：4， 动作数：2

该环境的状态数是 4 个，分别为车的位置、车的速度、杆的角度以及杆顶部的速度；动作数为 2 个，并且是离散的向左或者向右。

我们也可以直接重置或者初始化环境看看初始状态，代码如下：

```
state = env.reset() # 初始化环境
print(f"初始状态: {state}")
```

结果为：

初始状态: [0.03073904 0.00145001 -0.03088818 -0.03131252]

7.8.2 深度 Q 网络基本接口

介绍完环境之后，我们沿用接口的概念，通过分析伪代码来实现深度 Q 网络的基本训练模式。其实所有的强化学习算法都遵循同一个训练思路，执行动作，环境反馈，然后智能体更新，只是不同算法需要的一些要素不同，我们需要分析出这些要素，比如建立什么网络需要什么模块，以进一步完善算法。

我们现在常用的深度 Q 网络伪代码如图 7.13 所示。

- 初始化经验缓冲区 D ，容量为 N
- 初始化状态-动作函数，即带有初始权重 θ 的 Q 网络
- 初始化状态-动作函数，即带有初始权重 $\hat{\theta}$ 的 \hat{Q} 网络
- 执行 M 个回合循环，对于每个回合
 - 初始化环境，得到初始状态 s_1
 - 循环 T 个时间步长，对于每个时步 t
 - 使用 ϵ -贪心策略选择动作 a_t
 - 环境根据 a_t 反馈奖励 r_t 和下一个状态 s_{t+1}
 - 更新状态 $s_{t+1} = s_t$
 - 存储转移即 (s_t, a_t, r_t, s_{t+1}) 到经验回放 D 中
 - 更新策略如下：
 - 1. 从 D 中随机采样一个小批量的转移
 - 2. 计算实际的 Q 值 $y_j = \begin{cases} r_j, & \text{如果回合在时步 } j+1 \text{ 终止} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \hat{\theta}), & \text{否则} \end{cases}$
 - 3. 对损失函数 $(y_j - Q(\phi_j, a_j; \theta))^2$ 关于参数 θ 做随机梯度下降
 - 4. 每 C 步重置 $\hat{Q} = Q$

图 7.13 深度 Q 网络算法伪代码

用代码实现如下：

```

rewards = [] # 记录奖励
ma_rewards = [] # 记录滑动平均奖励
for i_ep in range(cgf.train_eps):
    state = env.reset() # 初始化环境
    done = False
    ep_reward = 0
    while True:
        action = agent.choose_action(state)
        next_state, reward, done, _ = env.step(action)
        ep_reward += reward
        agent.memory.push(state, action, reward, next_state, done)
        state = next_state
        agent.update()
        if done:
            break
    if (i_ep+1) % cfg.target_update == 0:
        agent.target_net.load_state_dict(agent.policy_net.state_dict())
    if (i_ep+1)%10 == 0:
        print('回合: {}/{}, 奖励: {}'.format(i_ep+1, cfg.train_eps, ep_reward))
    rewards.append(ep_reward)
    if ma_rewards:
        ma_rewards.append(0.9*ma_rewards[-1]+0.1*ep_reward)
    else:
        ma_rewards.append(ep_reward)

```

可以看到，深度 Q 网络的训练模式其实和大多强化学习算法是一样的思路，但与传统的 Q 学习算法相比，深度 Q 网络使用神经网络来代替之前的 Q 表格从而存储更多的信息，且由于使用了神经网络，因此我们一般需要利用随机梯度下降来优化 Q 值的预测。此外深度 Q 网络多了回放缓冲区，并且使用两个网络，即目标网络和当前网络。

7.8.3 回放缓冲区

从伪代码中可以看出，回放缓冲区的功能有两个：一个是将每一步采集的经验（包括状态、动作、奖励、下一时刻的状态）存储到缓冲区中，并且缓冲区具有一定的容量（capacity）；另一个是在更新策略的时候需要随机采样小批量的经验进行优化。因此我们可以定义一个 ReplayBuffer 类，包括 push() 和 sample() 两个函数，用于存储和采样。

```

import random
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity # 回放缓冲区的容量
        self.buffer = [] # 缓冲区
        self.position = 0

    def push(self, state, action, reward, next_state, done):
        ''' 缓冲区是一个队列，容量超出时删除开始存入的经验
        ...
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.position] = (state, action, reward, next_state, done)
        self.position = (self.position + 1) % self.capacity

```

```

def sample(self, batch_size):
    batch = random.sample(self.buffer, batch_size) # 随机采小批量经验
    state, action, reward, next_state, done = zip(*batch) # 解压成状态、动作等
    return state, action, reward, next_state, done
def __len__(self):
    """ 返回当前存储的量
    """
    return len(self.buffer)

```

7.8.4 Q 网络

在深度 Q 网络中我们使用神经网络替代原有的 Q 表格，从而能够存储更多的 Q 值，实现更为高级的策略以便用于复杂的环境。这里我们用的是一个三层的感知机或者称之为连接网络：

```

class MLP(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim=128):
        """ 初始化Q网络，为全连接网络
            input_dim: 输入的特征数即环境的状态数
            output_dim: 输出的动作维度
        """
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim) # 输入层
        self.fc2 = nn.Linear(hidden_dim, hidden_dim) # 隐藏层
        self.fc3 = nn.Linear(hidden_dim, output_dim) # 输出层

    def forward(self, x):
        # 各层对应的激活函数
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

```

学过深度学习的读者应该对全连接十分熟悉。在强化学习中，网络的输入一般是状态，输出则是一个动作，假如总共有两个动作，那么这里的动作维度就是 2，可能的输出就是 0 或 1，一般我们用 ReLU 作为激活函数。根据实际需要也可以改变神经网络的模型结构等等，比如若我们使用图像作为输入，则可以使用卷积神经网络（convolutional neural network，CNN）。

7.8.5 深度 Q 网络算法

与前面的项目实战一样，深度 Q 算法一般也包括选择动作和更新策略两个函数，首先我们看选择动作：

```

def choose_action(self, state):
    """选择动作
    """
    self.frame_idx += 1
    if random.random() > self.epsilon(self.frame_idx):
        with torch.no_grad():
            state = torch.tensor([state], device=self.device, dtype=torch.float32)
            q_values = self.policy_net(state)
            action = q_values.max(1)[1].item() # 选择Q值最大的动作
    else:

```

```
action = random.randrange(self.action_dim)
```

可以看到跟深度 Q 网络算法与 Q 学习算法其实是一样的，都是用的 ε -贪心策略，只是深度 Q 网络算法我们需要通过 PyTorch 或者 TensorFlow 工具来处理相应的数据。

而深度 Q 网络算法更新策略的步骤稍微复杂一点儿，主要包括三个部分：随机采样、计算期望 Q 值和梯度下降，如下：

```
def update(self):
    if len(self.memory) < self.batch_size: # 当memory中不满足一个批量时，不更新策略
        return
    # 从回放缓冲区中随机采样一个批量的经验
    state_batch, action_batch, reward_batch, next_state_batch, done_batch = self.memory.sample(
        self.batch_size)
    # 转为张量
    state_batch = torch.tensor(state_batch, device=self.device, dtype=torch. float)
    action_batch = torch.tensor(action_batch, device=self.device).unsqueeze(1)
    reward_batch = torch.tensor(reward_batch, device=self.device, dtype=torch. float)
    next_state_batch = torch.tensor(next_state_batch, device=self.device, dtype=torch. float)
    done_batch = torch.tensor(np.float32(done_batch), device=self.device)

    q_values = self.policy_net(state_batch).gather(dim=1, index=action_batch) # 计算当前状态(s_t,a)对应的Q(s_t,a)
    next_q_values = self.target_net(next_state_batch).
        max(1)[0].detach() # 计算下一时刻的状态(s_t_,a)对应的Q值
    # 计算期望的Q值，对于终止状态，此时done_batch[0]=1，对应的expected_q_values等于reward
    expected_q_values = reward_batch + self.gamma * next_q_values * (1-done_batch)
    loss = nn.MSELoss()(q_values, expected_q_values.unsqueeze(1)) # 计算均方根损失
    # 优化更新模型
    self.optimizer.zero_grad()
    loss.backward()
    for param in self.policy_net.parameters(): # clip防止梯度爆炸
        param.grad.data.clamp_(-1, 1)
    self.optimizer.step()
```

7.8.6 结果分析

实现代码之后，我们先来看看深度 Q 网络算法的训练效果，如图 7.14 所示。

从图 7.14 中可以看出，算法其实已经在 60 回合左右达到收敛，最后一直维持在最佳奖励 200 左右，可能会有轻微的波动，这是因为我们在收敛的情况下依然保持了一定的探索率，即 $\text{epsilon_end}=0.01$ 。现在我们可以载入模型看看测试的效果，如图 7.15 所示。

我们测试了 30 个回合，每个回合奖励都保持在 200 左右，说明我们的模型学习得不错！

7.9 关键词

双深度 Q 网络 (double DQN): 在双深度 Q 网络中存在两个 Q 网络，第一个 Q 网络决定哪一个动作的 Q 值最大，从而决定对应的动作。另一方面，Q 值是用 Q' 计算得到的，这样就可以避免过度估计的问题。具体地，假设我们有两个 Q 函数并且第一个 Q 函数高估了它现在执行的动作 a 的值，这没关系，只要第二个 Q 函数 Q' 没有高估动作 a 的值，那么计算得到的就还是正常的值。

竞争深度 Q 网络 (dueling DQN): 将原来的深度 Q 网络的计算过程分为两步。第一步计算一个与输入有关的标量 $V(s)$ ；第二步计算一个向量 $A(s, a)$ 对应每一个动作。最后的网络将两步的结果相加，得到

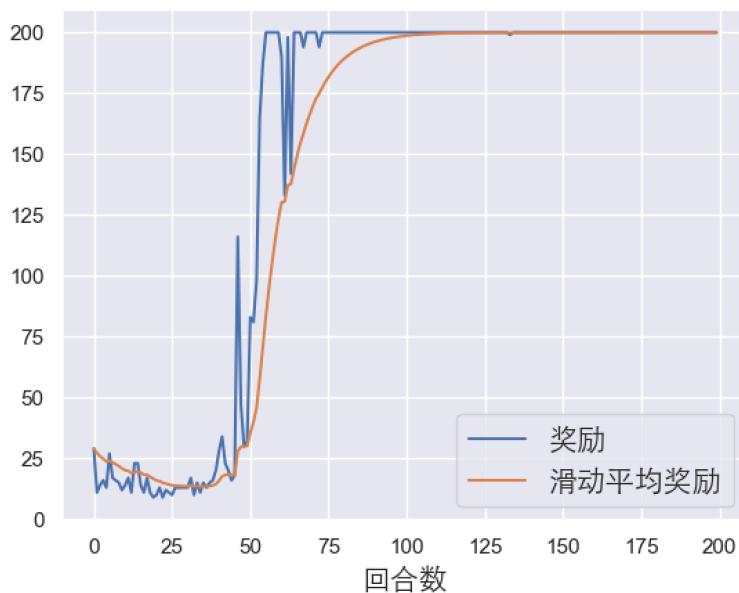


图 7.14 CartPole-v0 环境下深度 Q 网络算法的训练曲线

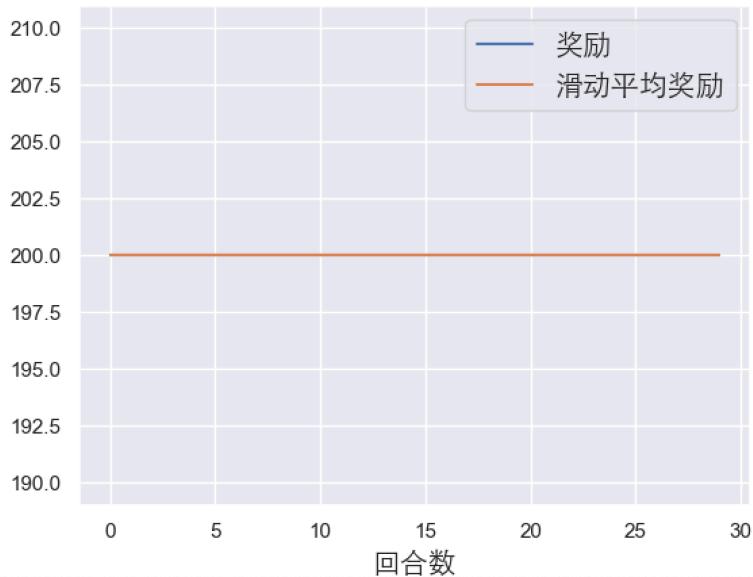


图 7.15 CartPole-v0 环境下深度 Q 网络算法的测试曲线

我们最终需要的 Q 值。用一个公式表示就是 $Q(s, a) = V(s) + A(s, a)$ 。另外，竞争深度 Q 网络，使用状态价值函数与动作价值函数来评估 Q 值。

优先级经验回放 (prioritized experience replay, PER): 这个方法是为了解决我们在第 6 章中提出的经验回放方法的不足而提出的。我们在使用经验回放时，均匀地取出回放缓冲区 (reply buffer) 中的采样数据，这里并没有考虑数据间的权重大小。但是我们应该将那些训练效果不好的数据对应的权重加大，即其应该有更大的概率被采样到。综上，优先级经验回放不仅改变了被采样数据的分布，还改变了训练过程。

噪声网络 (noisy net): 其在每一个回合开始的时候，即智能体要和环境交互的时候，在原来的 Q 函数的每一个参数上加上一个高斯噪声 (Gaussian noise)，把原来的 Q 函数变成 \tilde{Q} ，即噪声 Q 函数。同样，我们把每一个网络的权重等参数都加上一个高斯噪声，就得到一个新的网络 \tilde{Q} 。我们会使用这个新的网络与环境交互直到结束。

分布式 Q 函数 (distributional Q-function): 对深度 Q 网络进行模型分布，将最终网络的输出的每一类别的动作再进行分布操作。

彩虹 (rainbow): 将第 6、7 章 7 个技巧综合起来的方法，7 个技巧分别是——深度 Q 网络、双深度

Q 网络、优先级经验回放的双深度 Q 网络、竞争深度 Q 网络、异步优势演员-评论员算法 (A3C)、分布式 Q 函数、噪声网络，进而考察每一个技巧的贡献度或者与环境的交互是否是正反馈的。

7.10 习题

7-1 为什么传统的深度 Q 网络的效果并不好？可以参考其公式 $Q(s_t, a_t) = r_t + \max_a Q(s_{t+1}, a)$ 来描述。

7-2 在传统的深度 Q 网络中，我们应该怎么解决目标值太大的问题呢？

7-3 请问双深度 Q 网络中所谓的 Q 与 Q' 两个网络的功能是什么？

7-4 如何理解竞争深度 Q 网络的模型变化带来的好处？

7-5 使用蒙特卡洛和时序差分平衡方法的优劣分别有哪些？

7.11 面试题

7-1 友善的面试官：深度 Q 网络都有哪些变种？引入状态奖励的是哪种？

7-2 友善的面试官：请简述双深度 Q 网络原理。

7-3 友善的面试官：请问竞争深度 Q 网络模型有什么优势呢？

参考文献

- [1] VAN HASSELT H, GUEZ A, SILVER D. Deep reinforcement learning with double q-learning[C]// Proceedings of the AAAI conference on artificial intelligence: volume 30. 2016: 2094-2100.
- [2] WANG Z, SCHAUER T, HESSEL M, et al. Dueling network architectures for deep reinforcement learning[C]//International conference on machine learning. PMLR, 2016: 1995-2003.
- [3] HESSEL M, MODAYIL J, VAN HASSELT H, et al. Rainbow: Combining improvements in deep reinforcement learning[C]//Thirty-second AAAI conference on artificial intelligence. 2018: 3215-3222.

第 8 章 针对连续动作的深度 Q 网络

与基于策略梯度的方法相比，深度 Q 网络比较稳定，策略梯度比较不稳定，玩大部分游戏不能使用策略梯度。在没有近端策略优化之前，我们很难用策略梯度做什么事情。最早 DeepMind 的论文拿深度强化学习来玩雅达利的游戏，用的就是深度 Q 网络。深度 Q 网络比较容易训练的一个原因是：在深度 Q 网络里面，我们只要能够估计出 Q 函数，就保证一定可以找到一个比较好的策略。也就是我们只要能够估计出 Q 函数，就保证可以改进策略。而估计 Q 函数是比较容易的，因为它就是一个回归问题。在回归问题里面，我们可以通过观察回归的损失有没有下降，就可以知道模型学习得好不好，所以估计 Q 函数相较于学习一个策略是比较容易的。我们只要估计 Q 函数，就可以保证现在一定会得到比较好的策略，所以一般而言深度 Q 网络比较容易操作。

但深度 Q 网络其实存在一些问题，最大的问题是它很难处理连续动作。很多时候动作是连续的，比如我们玩雅达利的游戏时，智能体只需要决定如上、下、左、右这 4 个动作，这种动作是离散的。很多时候动作是连续的，例如，假设智能体要开车，它要决定方向盘要左转几度、右转几度，这种动作就是连续的。假设智能体是一个机器人，身上有 50 个关节，它的每一个动作就对应身上 50 个关节的角度，而这些角度也是连续的。所以很多时候动作并不是离散的，它是一个向量，这个向量的每一个维度都有一个对应的值，这些值都是实数，它是连续的。如果动作是连续的，我们使用深度 Q 网络就会有困难。因为在使用深度 Q 网络时很重要的一步是我们要能够解决优化问题，也就是估计出 Q 函数 $Q(s, a)$ 以后，我们必须找到一个 a ，它可以让 $Q(s, a)$ 最大，即

$$a = \arg \max_a Q(s, a) \quad (8.1)$$

假设 a 是离散的，即 a 的可能性是有限的。例如，在雅达利的小游戏里面， a 就是上、下、左、右与开火，它是有限的，我们可以把每一个可能的动作都代入 Q 里面算它的 Q 值。但假如 a 是连续的，我们无法穷举所有可能的连续动作，试试看哪一个连续动作可以让 Q 值最大。

怎么解决这个问题呢？我们有多种不同的方案，下面一一介绍。

8.1 方案 1：对动作进行采样

第 1 个方案是什么呢？我们可以采样出 N 个可能的 a : $\{a_1, a_2, \dots, a_N\}$ ，把它们一个一个地代入 Q 函数，看谁的 Q 值最大。这个方案不会太低效，因为我们在运算的时候会使用 GPU，一次把 N 个连续动作都代入 Q 函数，一次得到 N 个 Q 值，看谁最大。当然这不是一个非常精确的方案，因为我们没有办法进行太多的采样，所以估计出来的 Q 值、最后决定的动作可能不是非常精确。

8.2 方案 2：梯度上升

第 2 个方案是什么呢？既然要解决的是一个优化问题（optimization problem），我们就要最大化目标函数（objective function）。要最大化目标函数，我们就可以用梯度上升。我们把 a 当作参数，要找一组 a 去最大化 Q 函数，就用梯度上升去更新 a 的值，最后看看能不能找到一个 a 最大化 Q 函数（目标函数）。但我们会遇到全局最大值（global maximum）的问题，不一定能够找到最优的结果，而且运算量显然很大，因为要迭代地更新 a ，训练一个网络就很花时间了。如果我们使用梯度上升的方案来处理连续的问题，每次决定采取哪一个动作的时候，还要训练一次网络，显然运算量是很大的。

8.3 方案 3：设计网络架构

第 3 个方案是特别设计网络的架构，特别设计 Q 函数来使得解决 $\arg \max$ 操作的问题变得非常容易。

如图 8.1 所示，通常输入状态 s 是图像，我们可以用向量或矩阵来表示它。输入 s ，Q 函数会输出向量 $\mu(s)$ 、矩阵 $\Sigma(s)$ 和标量 $V(s)$ 。Q 函数根据输入 s 与 a 来决定输出值。到目前为止，Q 函数只有输入

s , 它还没有输入 a , a 在哪里呢? 接下来我们可以输入 a , 用 a 与 $\mu(s)$ 、 $\Sigma(s)$ 和 $V(s)$ 互相作用。Q 函数 $Q(s, a)$ 可定义为

$$Q(s, a) = -(a - \mu(s))^T \Sigma(s)(a - \mu(s)) + V(s) \quad (8.2)$$

注意, a 现在是连续的动作, 所以它是一个向量。假设我们要操作机器人, 向量 a 的每一个维度可能就对应机器人的每一个关节, 它的数值就是关节的角度。假设 a 和 $\mu(s)$ 是列向量, 那么 $(a - \mu(s))^T$ 是一个行向量。 $\Sigma(s)$ 是一个正定矩阵 (positive-definite matrix), 因为 $\Sigma(s) = LL^T$, 其中 L 为下三角矩阵 (lower-triangular matrix)。 $a - \mu(s)$ 也是一个列向量。所以 Q 值即 $-(a - \mu(s))^T \Sigma(s)(a - \mu(s)) + V(s)$ 是标量。

我们要怎么找到一个 a 来最大化 Q 值呢? 因为 $(a - \mu(s))^T \Sigma(s)(a - \mu(s))$ 一定是正的, 它前面有一个负号, 假设我们不看负号, 所以第一项 $(a - \mu(s))^T \Sigma(s)(a - \mu(s))$ 的值越小, 最终的 Q 值就越大。因为我们是把 $V(s)$ 减掉第一项, 所以第一项的值越小, 最后的 Q 值就越大。怎么让第一项的值最小呢? 我们直接令 $\mu(s)$ 等于 a , 让第一项变成 0, 就可以让第一项的值最小。因此, 令 $\mu(s)$ 等于 a , 我们就可以得到最大值, 解决 arg max 操作的问题就变得非常容易。所以深度 Q 网络也可以用在连续的情况下, 只是有一些局限: 函数不能随便设置。

如果 n 阶对称矩阵 A 对于任意非零的 n 维向量 x 都有 $x^T A x > 0$, 则称矩阵 A 为正定矩阵。

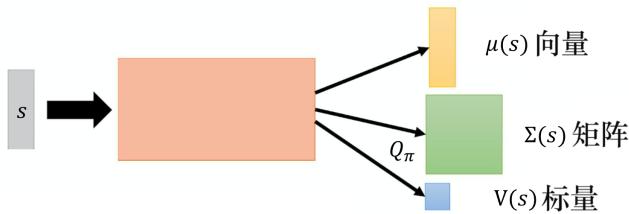


图 8.1 方案 3: 设计网络架构

8.4 方案 4: 不使用深度 Q 网络

第 4 个方案就是不使用深度 Q 网络, 用深度 Q 网络处理连续动作是比较麻烦的。如图 8.2 所示, 我们将基于策略的方法——PPO 和基于价值的方法——深度 Q 网络结合在一起, 就可以得到演员-评论员的方法。

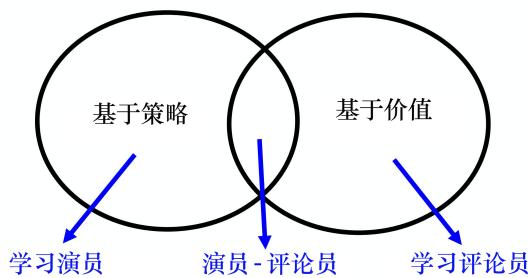


图 8.2 方案 4: 不使用深度 Q 网络

8.5 习题

8-1 深度 Q 网络相比基于策略梯度的方法为什么训练效果更好、更平稳?

8-2 深度 Q 网络在处理连续动作时存在什么样的问题呢? 对应的解决方法有哪些呢?

第 9 章 演员-评论员算法

在 REINFORCE 算法中，每次需要根据一个策略采集一条完整的轨迹，并计算这条轨迹上的回报。这种采样方式的方差比较大，学习效率也比较低。我们可以借鉴时序差分学习的思想，使用动态规划方法来提高采样效率，即从状态 s 开始的总回报可以通过当前动作的即时奖励 $r(s, a, s')$ 和下一个状态 s' 的值函数来近似估计。

演员-评论员算法是一种结合策略梯度和时序差分学习的强化学习方法，其中，演员是指策略函数 $\pi_\theta(a|s)$ ，即学习一个策略以得到尽可能高的回报。评论员是指价值函数 $V_\pi(s)$ ，对当前策略的值函数进行估计，即评估演员的好坏。借助于价值函数，演员-评论员算法可以进行单步参数更新，不需要等到回合结束才进行更新。在演员-评论员算法里面，最知名的算法就是异步优势演员-评论员算法。如果我们去掉异步，则为**优势演员-评论员 (advantage actor-critic, A2C) 算法**。A2C 算法又被译作优势演员-评论员算法。如果我们加了异步，变成异步优势演员-评论员算法。

9.1 策略梯度回顾

我们复习一下策略梯度，在更新策略参数 θ 的时候，我们可以通过

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \left(\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n - b \right) \nabla \log p_\theta(a_t^n | s_t^n) \quad (9.1)$$

来计算梯度。式 (9.1) 表示我们首先通过智能体与环境的交互，可以计算出在某一个状态 s 采取某一个动作 a 的概率 $p_\theta(a_t|s_t)$ 。接下来，我们计算在某一个状态 s 采取某一个动作 a 之后直到游戏结束的累积奖励。 $\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n$ 表示我们把从时间 t 到时间 T 的奖励相加，并且在前面乘一个折扣因子，通常将折扣因子设置为 0.9 或 0.99 等数值，与此同时也会减去一个基线值 b ，减去值 b 的目的是希望括号里面这一项是有正有负的。如果括号里面这一项是正的，我们就要增大在这个状态采取这个动作的概率；如果括号里面是负的，我们就要减小在这个状态采取这个动作的概率。

我们使用 G 表示累积奖励， G 是非常不稳定的。因为交互的过程本身具有随机性，所以在某一个状态 s 采取某一个动作 a 时计算得到的累积奖励，每次结果都是不同的，因此 G 是一个随机变量。对于同样的状态 s 和同样的动作 a ， G 可能有一个固定的分布。但由于我们采取采样的方式，因此我们在某一个状态 s 采取某一个动作 a 一直到游戏结束，统计一共得到了多少的奖励，我们就把它当作 G 。

如图 9.1 所示，如果我们把 G 想成一个随机变量，实际上是在对 G 做采样，用这些采样的结果去更新参数。但实际上在某一个状态 s 采取某一个动作 a ，接下来会发生什么事，其本身是有随机性的。虽然说有一个固定的分布，但其方差可能会非常大。智能体在同一个状态采取同一个动作时，最后得到的结果可能会是很不一样的。当然，假设我们在每次更新参数之前，都可以采样足够多次，那当然就没有以上的问题了。但我们每次做策略梯度，每次更新参数之前都要做一些采样时，采样的次数是不可能太多的，我们只能做非常少量的采样。如果我们正好采样到差的结果，比如采样到 $G = 100$ 、采样到 $G = -10$ ，显然结果会是很差的。

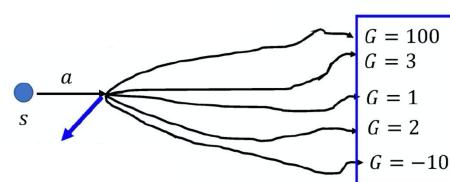


图 9.1 策略梯度回顾

9.2 深度 Q 网络回顾

Q：我们能不能让整个训练过程变得稳定，能不能直接估测随机变量 G 的期望值？

A：我们直接用一个网络去估测在状态 s 采取动作 a 时 G 的期望值。如果这样是可行的，那么在随后的训练中我们就用期望值代替采样的值，这样就会让训练变得更加稳定。

Q：怎么使用期望值代替采样的值呢？

A：这里就需要引入基于价值的（value-based）的方法。基于价值的方法就是深度 Q 网络。深度 Q 网络有两种函数，有两种评论员。如图 9.2 所示，第一种评论员是 $V_\pi(s)$ 。即假设演员的策略是 π ，使用 π 与环境交互，当智能体看到状态 s 时，接下来累积奖励的期望值是多少。第二种评论员是 $Q_\pi(s, a)$ 。 $Q_\pi(s, a)$ 把 s 与 a 当作输入，它表示在状态 s 采取动作 a ，接下来用策略 π 与环境交互，累积奖励的期望值是多少。 V_π 接收输入 s ，输出一个标量。 Q_π 接收输入 s ，它会给每一个 a 都分配一个 Q 值。

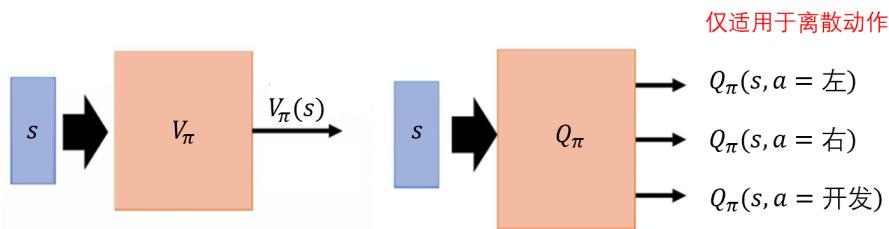


图 9.2 深度 Q 网络

9.3 演员-评论员算法

如图 9.3 所示，随机变量 G 的期望值正好就是 Q 值，即

$$\mathbb{E}[G_t^n] = Q_{\pi_\theta}(s_t^n, a_t^n) \quad (9.2)$$

此也为 Q 函数的定义。Q 函数的定义就是在某一个状态 s ，采取某一个动作 a ，假设策略是 π 的情况下所能得到的累积奖励的期望值，即 G 的期望值。累积奖励的期望值就是 G 的期望值。所以假设用 $\mathbb{E}[G_t^n]$ 来代表 $\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n$ 这一项，把 Q 函数套在这里就结束了，我们就可以把演员与评论员这两个方法结合起来。

有不同的方法表示基线，一个常见的方法是用价值函数 $V_{\pi_\theta}(s_t^n)$ 来表示基线。价值函数的定义为，假设策略是 π ，其在某个状态 s 一直与环境交互直到游戏结束，期望奖励有多大。 $V_{\pi_\theta}(s_t^n)$ 没有涉及动作， $Q_{\pi_\theta}(s_t^n, a_t^n)$ 涉及动作。 $V_{\pi_\theta}(s_t^n)$ 是 $Q_{\pi_\theta}(s_t^n, a_t^n)$ 的期望值， $Q_{\pi_\theta}(s_t^n, a_t^n) - V_{\pi_\theta}(s_t^n)$ 会有正有负，所以 $\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n - b$ 这一项就会有正有负。所以我们就把策略梯度里面 $\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n - b$ 这一项换成了 $Q_{\pi_\theta}(s_t^n, a_t^n) - V_{\pi_\theta}(s_t^n)$ 。

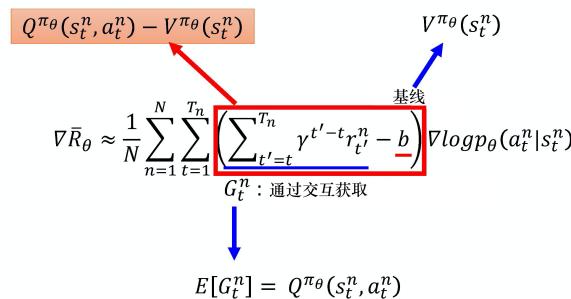


图 9.3 演员-评论员算法

9.4 优势演员-评论员算法

如果我们这么实现，有一个缺点，即我们需要估计两个网络——Q 网络和 V 网络，估计不准的风险就变成原来的两倍。所以我们何不只估计一个网络呢？事实上，在演员-评论员算法中，我们可以只估计网络 V，并利用 V 的值来表示 Q 的值， $Q_\pi(s_t^n, a_t^n)$ 可以写成 $r_t^n + V_\pi(s_{t+1}^n)$ 的期望值，即

$$Q_\pi(s_t^n, a_t^n) = \mathbb{E}[r_t^n + V_\pi(s_{t+1}^n)] \quad (9.3)$$

在状态 s 采取动作 a ，我们会得到奖励 r ，进入状态 s_{t+1} 。但是我们会得到什么样的奖励 r ，进入什么样的状态 s_{t+1} ，这件事本身是有随机性的。所以要把 $r_t^n + V_\pi(s_{t+1}^n)$ 取期望值才会等于 Q 函数的值。但我们现在把取期望值去掉，即

$$Q_\pi(s_t^n, a_t^n) = r_t^n + V_\pi(s_{t+1}^n) \quad (9.4)$$

我们就可以把 Q 函数的值用 $r_t^n + V_\pi(s_{t+1}^n)$ 取代，可得

$$r_t^n + V_\pi(s_{t+1}^n) - V_\pi(s_t^n) \quad (9.5)$$

把取期望值去掉的好处就是我们不需要估计 Q 了，只需要估计 V。但与此同时我们会引入一个随机的参数 r 。 r 是有随机性的，它是一个随机变量，但是 r 相较于累积奖励 G 是一个较小的值，因为它是某一个步骤得到的奖励，而 G 是所有未来会得到的奖励的总和， G 的方差比较大。 r 虽然也有一些方差，但它的方差比 G 的要小。所以把原来方差比较大的 G 换成方差比较小的 r 也是合理的。

Q：为什么我们可以直接把取期望值去掉？

A：原始的异步优势演员-评论员算法的论文尝试了各种方法，最后发现这个方法最好。当然有人可能会有疑问，说不定估计 Q 和 V 也可以估计得很好，但实际做实验的时候，最后结果就是这个方法最好，所以来大家都使用了这个方法。

因为 $r_t^n + V_\pi(s_{t+1}^n) - V_\pi(s_t^n)$ 被称为**优势函数**，所以该算法被称为优势演员-评论员算法。优势演员-评论员算法的流程如图 9.4 所示，我们有一个 π ，有个初始的演员与环境交互，先收集资料。在策略梯度方法里收集资料以后，就来更新策略。但是在演员-评论员算法里面，我们不是直接使用那些资料来更新策略。我们先用这些资料去估计价值函数，可以用时序差分方法或蒙特卡洛方法来估计价值函数。接下来，我们再基于价值函数，使用式 (9.6) 更新 π 。

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (r_t^n + V_\pi(s_{t+1}^n) - V_\pi(s_t^n)) \nabla \log p_\theta(a_t^n | s_t^n) \quad (9.6)$$

有了新的 π 以后，再与环境交互，收集新的资料，去估计价值函数。再用新的价值函数更新策略，更新演员。整个优势演员-评论员算法就是这么运作的。



图 9.4 优势评论员-评论员算法流程

实现优势演员-评论员算法的时候，有两个一定会用到的技巧。第一个技巧是，我们需要估计两个网络： V 网络和策略的网络（也就是演员）。评论员网络 $V_\pi(s)$ 接收一个状态，输出一个标量。演员的策略

$\pi(s)$ 接收一个状态，如果动作是离散的，输出就是一个动作的分布，如果动作是连续的，输出就是一个连续的向量。

图 9.5 所示为离散动作的例子，连续动作的情况也是一样的。输入一个状态，网络决定现在要采取哪一个动作。演员网络和评论员网络的输入都是 s ，所以它们前面几个层（layer）是可以共享的。

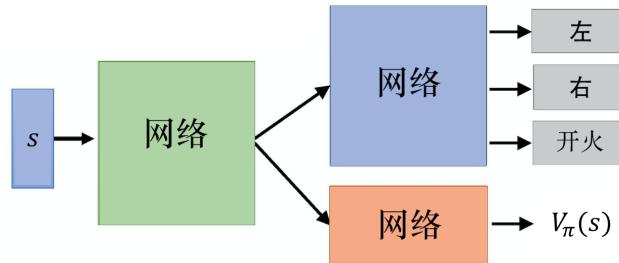


图 9.5 离散动作的例子

尤其当我们在玩雅达利游戏时，输入都是图像。输入的图像非常复杂，通常我们在前期都会用一些卷积神经网络来处理它们，把图像抽象成高级（high level）的信息。把像素级别的信息抽象成高级信息的特征提取器，对于演员与评论员来说是可以共用的。所以通常我们会让演员与评论员共享前面几层，并且共用同一组参数，这一组参数大部分都是卷积神经网络的参数。先把输入的像素变成比较高级的信息，再让演员决定要采取什么样的动作，让评论员即价值函数计算期望奖励。

第二个技巧是我们需要探索的机制。在演员-评论员算法中，有一个常见的探索的方法是对 π 输出的分布设置一个约束。这个约束用于使分布的熵（entropy）不要太小，也就是希望不同的动作被采用的概率平均一些。这样在测试的时候，智能体才会多尝试各种不同的动作，才会把环境探索得比较好，从而得到比较好的结果。

9.5 异步优势演员-评论员算法

强化学习有一个问题，就是它很慢，怎么提高训练的速度呢？例如，如图 9.6 所示，在动漫《火影忍者》中，有一次鸣人想要在一周之内打败晓，所以要加快修行的速度，鸣人的老师就教他一个方法：用影分身进行同样的修行。两个一起修行，经验值累积的速度就会变成两倍，所以鸣人就使用了 1000 个影分身来进行修行。这就是异步优势演员-评论员算法的体现。



图 9.6 影分身例子

异步优势演员-评论员算法同时使用很多个进程（worker），每一个进程就像一个影分身，最后这些影分身会把所有的经验值集合在一起。如果我们没有很多 CPU，也是不好实现的。我们可以实现优势演员-评论员算法就可以。

异步优势演员-评论员算法的运作流程，如图 9.7 所示，异步优势演员-评论员算法一开始有一个全局网络（global network）。全局网络包含策略网络和价值网络，这两个网络是绑定（tie）在一起的，它们的

前几个层会被绑在一起。假设全局网络的参数是 θ_1 ，我们使用多个进程，每个进程用一张 CPU 去跑。比如我们有 8 个进程，则至少 8 张 CPU。每一个进程在工作前都会把全局网络的参数复制过来。接下来演员就与环境交互，每一个演员与环境交互的时候，都要收集到比较多样化的数据。例如，如果是走迷宫，可能每一个演员起始的位置都会不一样，这样它们才能够收集到比较多样化的数据。每一个演员与环境交互完之后，我们就会计算出梯度。计算出梯度以后，要用梯度去更新参数。我们就计算一下梯度，用梯度去更新全局网络的参数。就是这个进程算出梯度以后，就把梯度传回给中央的控制中心，中央的控制中心就会用这个梯度去更新原来的参数。注意，所有的演员都是平行跑的，每一个演员各做各的，不管彼此。所以每个演员都是去要了一个参数以后，做完就把参数传回去。当第一个进程做完想要把参数传回去的时候，本来它要的参数是 θ_1 ，等它要把梯度传回去的时候，可能别人已经把原来的参数覆盖掉，变成 θ_2 了。但是没有关系，它一样会把这个梯度就覆盖过去。

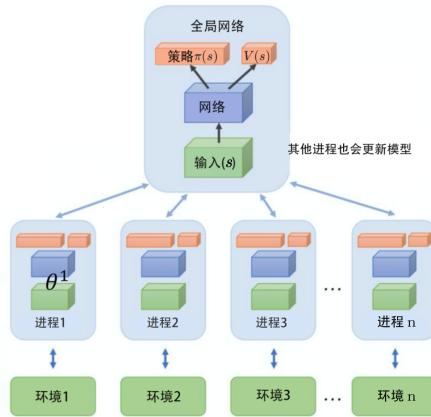


图 9.7 异步优势演员-评论员算法的运作流程

9.6 路径衍生策略梯度

接下来我们来了解路径衍生策略梯度 (**pathwise derivative policy gradient**) 方法。这个方法可以看成深度 Q 网络解连续动作的一种特别的方法，也可以看成一种特别的演员-评论员的方法。用动漫《棋魂》来比喻，阿光就是一个演员，佐为就是一个评论员。阿光落某一子以后，如果佐为是一般的演员-评论员算法的评论员，他会告诉阿光这时候不应该下小马步飞。佐为会告诉我们，我们现在采取的这一步算出来的值到底是好还是不好，但这样就结束了，他只告诉我们好还是不好。因为一般的演员-评论员算法的评论员就是输入状态或输入状态-动作对，给演员一个值，所以对演员来说，它只知道它做的这个动作到底是好还是不好。

但在路径衍生策略梯度里面，评论员会直接告诉演员采取什么样的动作才是好的。所以佐为不只是告诉阿光，这个时候不要下小马步飞，同时还告诉阿光这个时候应该要下大马步飞，这就是路径衍生策略梯度中的评论员所做的。评论员会直接告诉演员做什么样的动作才可以得到比较大的值。

从深度 Q 网络的观点来看，深度 Q 网络的一个问题是在使用深度 Q 网络时，考虑连续向量会比较麻烦，没有通用的解决方法 (general solution)，那我们应该怎么解这个优化问题呢？我们用一个演员来解决这个优化的问题。本来在深度 Q 网络里面，如果是一个连续的动作，我们要解决这个优化问题。但是现在这个优化问题由演员来解决，假设演员就是一个解决者 (solver)，这个解决者的工作就是对于给定的状态 s ，解出来哪一个动作可以得到最大的 Q 值，这是从另外一个观点来看路径衍生策略梯度。在生成对抗网络中也有类似的说法。我们学习一个判别器 (discriminator) 并用于评估时，是非常困难的，因为我们要解决的 arg max 的问题非常的困难，所以用生成器 (generator) 来生成。所以概念是一样的，Q 就是那个判别器。根据这个判别器决定动作非常困难，怎么办？另外学习一个网络来解决这个优化问题，这个网络就是演员。所以两个不同的观点是同一件事。从两个不同的观点来看，一个观点是：我们可以对原来的

深度 Q 网络加以改进，学习一个演员来决定动作以解决 $\arg \max$ 不好解的问题。另外一个观点是：原来的演员-评论员算法的问题是评论员并没有给演员足够的信息，评论员只告诉演员好或不好的，没有告诉演员什么样是好，现在有新的方法可以直接告诉演员什么样的好。路径衍生策略梯度算法如图 9.8 所示，假设我们学习了一个 Q 函数，Q 函数的输入是 s 与 a ，输出是 $Q_\pi(s, a)$ 。接下来，我们要学习一个演员，这个演员的工作就是解决 $\arg \max$ 的问题，即输入一个状态 s ，希望可以输出一个动作 a 。 a 被代入 Q 函数以后，它可以让 $Q_\pi(s, a)$ 尽可能大，即

$$\pi'(s) = \arg \max_a Q_\pi(s, a) \quad (9.7)$$

实际上在训练的时候，我们就是把 Q 与演员连接起来变成一个比较大的网络。Q 是一个网络，接收输入 s 与 a ，输出一个值。演员在训练的时候，它要做的事就是接收输入 s ，输出 a 。把 a 代入 Q 中，希望输出的值越大越好。我们会固定住 Q 的参数，只调整演员的参数，用梯度上升的方法最大化 Q 的输出，这就是一个生成对抗网络，即有条件的生成对抗网络（conditional GAN）。Q 就是判别器，但在强化学习里就是评论员，演员在生成对抗网络里面就是生成器。

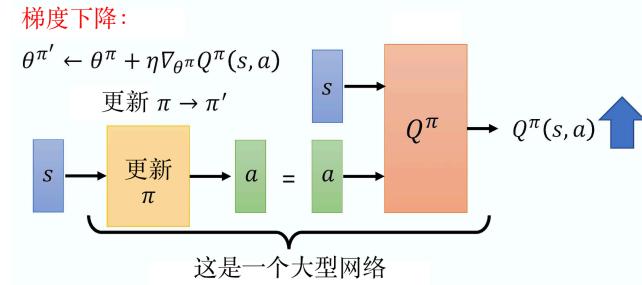


图 9.8 路径衍生策略梯度

我们来看一下路径衍生策略梯度算法。如图 9.9 所示，一开始会有一个策略 π ，它与环境交互并估计 Q 值。估计完 Q 值以后，我们就把 Q 值固定，只去学习一个演员。假设这个 Q 值估得很准，它知道在某一个状态采取什么样的动作会得到很大的 Q 值。接下来就学习这个演员，演员在给定 s 的时候，采取了 a ，可以让最后 Q 函数算出来的值越大越好。我们用准则（criteria）去更新策略 π ，用新的 π 与环境交互，再估计 Q 值，得到新的 π 去最大化 Q 值的输出。深度 Q 网络里面的技巧，在这里也几乎都用得上，比如经验回放、探索等技巧。

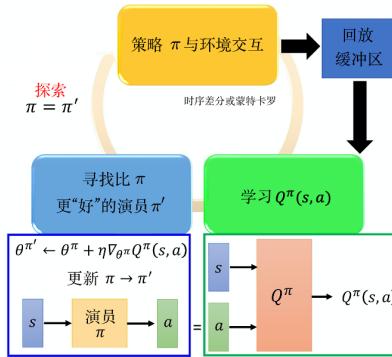


图 9.9 路径衍生策略梯度算法

图 9.10 所示为原来深度 Q 网络的算法。我们有一个 Q 函数 Q 和另外一个目标 Q 函数 \hat{Q} 。每一次训练，在每一个回合的每一个时间点，我们会看到一个状态 s_t ，会采取某一个动作 a_t 。至于采取哪一个动作是由 Q 函数所决定的。如果是离散动作，我们看哪一个动作 a 可以让 Q 值最大，就采取哪一个动作。当然，我们需要加一些探索，这样表现才会好。我们会得到奖励 r_t ，进入新的状态 s_{t+1} ，然后把 (s_t, a_t, r_t, s_{t+1})

放到回放缓冲区里。接下来，我们会从回放缓冲区中采样一个批量的数据，在这个批量数据里面，可能某一笔数据是 (s_i, a_i, r_i, s_{i+1}) 。接下来我们会算一个目标 y ， $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$ 。怎么学习 Q 呢？我们希望 $Q(s_i, a_i)$ 与 y 越接近越好，这是一个回归问题，最后每 C 步，要用 Q 替代 \hat{Q} 。

- 初始化 Q 函数 Q ，目标 Q 函数 $\hat{Q} = Q$
- 对于每一个回合
 - 对于每一个时间步 t
 - 对于给定的状态 s_t ，基于 Q (ε -贪心) 执行动作 a_t 。
 - 获得反馈 r_t ，并获得新的状态 s_{t+1} 。
 - 将 (s_t, a_t, r_t, s_{t+1}) 存储到缓冲区中。
 - 从缓冲区中采样（通常以批量形式） (s_i, a_i, r_i, s_{i+1}) 。
 - 目标值是 $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$ 。
 - 更新 Q 的参数使得 $Q(s_i, a_i)$ 尽可能接近于 y （回归）。
 - 每 C 步重置 $\hat{Q} = Q$ 。

图 9.10 深度 Q 网络算法

接下来我们把深度 Q 网络改成路径衍生策略梯度，需要做 4 个改变，如图 9.11 所示。

(1) 第一个改变是，我们要把 Q 换成 θ ，本来是用 Q 来决定在状态 s_t 执行一个动作 a_t ，现在直接用 θ 来执行动作。我们直接学习了一个演员。这个演员的输入 s_t 会告诉我们应该采取哪一个 a_t 。所以本来输入 s_t ，采取哪一个 a_t 是 Q 决定的，而在路径衍生策略梯度里面，我们会直接用 θ 来决定。

(2) 第二个改变是，本来我们要计算在 s_{i+1} ，根据策略采取某一个动作 a 会得到的 Q 值，我们会采取让 \hat{Q} 最大的那个动作 a 。现在因为我们直接把 s_{i+1} 代入 θ ，就会知道给定 s_{i+1} ，哪个动作会给我们最大的 Q 值，就采取哪个动作。在 Q 函数里面，有两个 Q 网络：真正的 Q 网络和目标 Q 网络。实际上我们在实现路径衍生策略梯度算法的时候，也有两个演员：真正要学习的演员 θ 和目标演员 $\hat{\theta}$ 。这个原理就与为什么要有目标 Q 网络一样，我们在算目标值的时候，并不希望它一直的变动，所以我们会有一个目标演员和一个目标 Q 函数，它们平常的参数就是固定住的，这样可以让目标的值不会一直地变化。总结一下，第二个改变是我们用策略取代原来要解 $\arg \max$ 的地方。

(3) 第三个改变是，之前只要学习 Q 函数，现在我们多学习了一个 θ ，学习 θ 的目的是最大化 Q 函数，希望得到的演员可以让 Q 函数的输出尽可能大，这与学习生成对抗网络里面的生成器的概念是类似的。

(4) 第四个改变是，我们不仅要取代目标的 Q 网络，还要取代目标策略。

- 初始化 Q 函数 Q ，目标 Q 函数 $\hat{Q} = Q$ ，演员 π ，目标演员 $\hat{\pi} = \pi$
- 在每个回合中
 - 对于每个时间步 t
 - ① ■ 获取状态 s_t ，根据 π 执行动作 a_t （探索）
 - 获取奖励 r_t ，到达新状态 s_{t+1}
 - 存储 (s_t, a_t, r_t, s_{t+1}) 到缓冲区
 - 从缓冲区采样 (s_t, a_t, r_t, s_{t+1}) （通常是一个批量）
 - ② ■ 目标 $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$ $\hat{Q}(s_{i+1}, \hat{\pi}(s_{i+1}))$
 - 更新 Q 的参数使得 $Q(s_i, a_i)$ 接近于 y （回归）
 - ③ ■ 更新 π 的参数使 $Q(s_i, \pi(s_i))$ 最大
 - 每 C 步重置 $\hat{Q} = Q$
 - ④ ■ 每 C 步重置 $\hat{\pi} = \pi$

图 9.11 从深度 Q 网络到路径衍生策略梯度

9.7 与生成对抗网络的联系

如表 9.1 所示，GAN 与演员-评论员的方法是非常类似的。如果大家感兴趣，可以参考一篇论文：“Connecting Generative Adversarial Network and Actor-Critic Methods”。

生成对抗网络与演员-评论员都挺难训练，所以在文献上就有各式各样的方法，告诉我们怎么样可以训练生成对抗网络。知道生成对抗网络与演员-评论员非常相似后，我们就可以知道怎样训练演员-评论员。但是因为做生成对抗网络与演员-评论员的人是两群人，所以这篇论文里面就列出说在生成对抗网络上面有哪些技术是有人做过的，在演员-评论员上面，有哪些技术是有人做过的。也许训练生成对抗网络的技术，我们可以试着应用在演员-评论员上，在演员-评论员上用过的技术，也可以试着应用在生成对抗网络上。

表 9.1 与生成对抗网络的联系

方法	生成对抗网络	演员-评论员
冻结学习	有	有
标签平滑	有	无
历史平均	有	无
小批量判别	有	无
批量归一化	有	有
目标网络	不适用	有
经验回放	无	有
熵正则化	无	有
兼容性	无	有

9.8 关键词

优势演员-评论员 (advantage actor-critic, A2C) 算法：一种改进的演员-评论员 (actor-critic) 算法。

异步优势演员-评论员 (asynchronous advantage actor-critic, A3C) 算法：一种改进的演员-评论员算法，通过异步的操作，实现强化学习模型训练的加速。

路径衍生策略梯度 (pathwise derivative policy gradient)：一种使用 Q 学习来求解连续动作的算法，也是一种演员-评论员算法。其会对演员提供价值最大的动作，而不仅仅是提供某一个动作的好坏程度。

9.9 习题

9-1 完整的优势演员-评论员算法的工作流程是怎样的？

9-2 在实现演员-评论员算法的时候有哪些技巧？

9-3 异步优势演员-评论员算法在训练时有很多的进程进行异步的工作，最后再将他们所获得的“结果”集合到一起。那么其具体是如何运作的呢？

9-4 对比经典的 Q 学习算法，路径衍生策略梯度有哪些改进之处？

9.10 面试题

9-1 友善的面试官：请简述一下异步优势演员-评论员算法 (A3C)，另外 A3C 是同策略还是异策略的模型呀？

9-2 友善的面试官：请问演员-评论员算法有何优点呢？

9-3 友善的面试官：请问异步优势演员-评论员算法具体是如何异步更新的？

9-4 友善的面试官：演员-评论员算法中，演员和评论员两者的区别是什么？

9-5 友善的面试官：演员-评论员算法框架中的评论员起了什么作用？

9-6 友善的面试官：简述异步优势演员-评论员算法的优势函数。

第 10 章 稀疏奖励

实际上用强化学习训练智能体的时候，多数时候智能体都不能得到奖励。在不能得到奖励的情况下，训练智能体是非常困难的。例如，假设我们要训练一个机器臂，桌上有一个螺丝钉与一个螺丝起子，要训练它用螺丝起子把螺丝钉栓进去很难，因为一开始智能体是什么都不知道，它唯一能够做不同的动作的原因是探索。例如，我们在做 Q 学习的时候会有一些随机性，让它去采取一些过去没有采取过的动作，要随机到，它把螺丝起子捡起来，再把螺丝栓进去，就会得到奖励 1，这件事情是永远不可能发生的。所以，不管演员做了什么事情，它得到的奖励永远都是 0，对它来说不管采取什么样的动作都是一样糟或者是一样好。所以，它最后什么都不会学到。

如果环境中的奖励非常稀疏，强化学习的问题就会变得非常困难，但是人类可以在非常稀疏的奖励上去学习。人生通常多数的时候，就只是活在那里，都没有得到什么奖励或是惩罚。但是，人还是可以采取各种各样的行为。所以，一个真正厉害的人工智能应该能够在稀疏奖励的情况下也学到怎么与环境交互。

我们可以通过 3 个方向来解决稀疏奖励的问题，下面一一介绍。

10.1 设计奖励

第一个方向是设计奖励 (reward shaping)。环境有一个固定的奖励，它是真正的奖励，但是为了让智能体学到的结果是我们想要的，所以我们刻意设计了一些奖励来引导智能体。

例如，如图 10.1 所示，如果我们把小孩当成一个智能体，他可以采取两个动作：玩耍或者学习。如果他玩耍，在下一个时间点就会得到奖励 1。但是他在月考的时候，成绩可能会很差，所以在 100 个小时之后，他会得到奖励 -100。他也可以决定要学习，在下一个时间点，因为他没有玩耍，所以觉得很不爽，所以得到奖励 -1。但是在 100 个小时后，他可以得到奖励 100。对于一个小孩来说，他可能就会想要采取玩耍的动作而不是学习的动作。我们计算的是累积奖励，但也许对小孩来说，折扣因子会很大，所以他就不在意未来的奖励。而且因为他是一个小孩，还没有很多经验，所以他的 Q 函数估计是非常不精准的。所以要他去估计很远以后会得到的累积奖励，他是估计不出来的。这时候大人就要引导他，对他说：“如果你学习，我就给你一根棒棒糖。”对小孩来说，下一个时间点他得到的奖励就变成正的，他也许就会认为学习是比玩耍好的。虽然这并不是真正的奖励，而是其他人引导他的奖励。设计奖励的概念是一样的，简单来说，就是我们自己想办法设计一些奖励，这些奖励不是环境真正的奖励。在玩雅达利游戏时，真正的奖励是游戏主机给的奖励，但我们自己可以设计一些奖励引导智能体，让智能体做我们想要它做的事情。

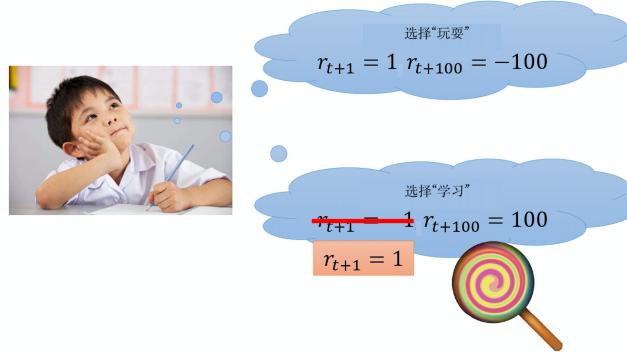


图 10.1 设计奖励

举个 Meta (原 Facebook) 玩 ViZDoom 的智能体的例子。ViZDoom 是一个第一人称射击游戏，在这个射击游戏中，杀了敌人得到正奖励，被杀得到负奖励。研究人员设计了一些新的奖励，用新的奖励来引导智能体让它们做得更好，这不是游戏中真正的奖励。比如掉血就扣分，弹药减少就扣分，捡到补给包就加分，待在原地就扣分，移动就加分。活着会扣一个很小的分数，因为如果不这样做，智能体会只想活着，

一直躲避敌人，这样会让智能体好战一些。

设计奖励是有问题的，因为我们需要领域知识 (domain knowledge)。例如，如图 10.2 所示，机器人想要学会把蓝色的板子从柱子穿过。机器人很难学会，我们可以设计奖励。一个貌似合理地说法是，蓝色的板子离柱子越近，奖励越大。但是机器人靠近的方式会有问题，它会用蓝色的板子打柱子。而机器人要把蓝色板子放在柱子上面，才能让蓝色板子穿过柱子。因此，这种设计奖励的方式是有问题的。至于哪种设计奖励的方式有问题，哪种设计奖励的方式没问题，会变成一个领域知识，是我们要去调整的。

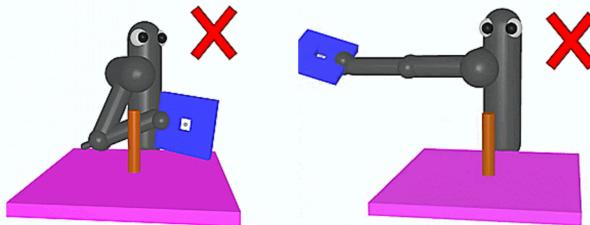


图 10.2 设计奖励的问题

10.2 好奇心

接下来介绍各种我们可以自己加入并且一般看起来是有用的奖励。例如，一种技术是给智能体加上好奇心 (curiosity)，称为好奇心驱动的奖励 (curiosity driven reward)。如图 10.3 所示，我们输入某个状态和某个动作到奖励函数中，奖励函数就会输出在这个状态采取这个动作会得到的奖励，总奖励越大越好。

在好奇心驱动的技术里面，我们会加上一个新的奖励函数——内在好奇心模块 (intrinsic curiosity module, ICM)，它用于给智能体加上好奇心。内在好奇心模块需要 3 个输入：状态 s_1 、动作 a_1 和状态 s_2 。根据输入，它会输出另外一个奖励 r_1^i 。对智能体来说，总奖励并不是只有 r ，还有 r^i 。它不是只把所有的 r 都加起来，它还把所有 r^i 加起来当作总奖励。所以在与环境交互的时候，它不是只希望 r 越大越好，它还同时希望 r^i 越大越好，它希望从内在好奇心模块里面得到的奖励越大越好。内在好奇心模块代表一种好奇心。

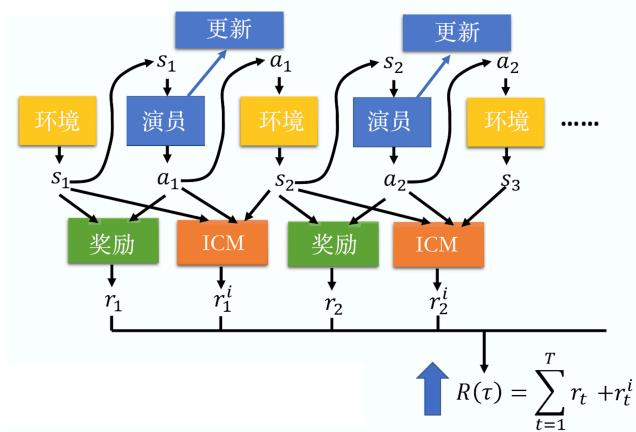


图 10.3 好奇心

怎么设计内在好奇心模块？最原始的设计如图 10.4 所示，内在好奇心模块的输入是现在的状态 s_t 、在这个状态采取的动作 a_t 以及下一个状态 s_{t+1} ，输出一个奖励 r_t^i 。那么 r_t^i 是怎么算出来的呢？在内在好奇心模块里面，我们有一个网络，这个网络会接收输入 a_t 与 s_t ，输出 \hat{s}_{t+1} ，也就是这个网络根据 a_t 和 s_t

去预测 \hat{s}_{t+1} 。然后再看这个网络的预测 \hat{s}_{t+1} 与真实的情况 s_{t+1} 的相似度，越不相似得到的奖励就越大。所以奖励 r_t^i 的意思是，未来状态越难被预测，得到的奖励就越大。这就是鼓励智能体去冒险、去探索，现在采取这个动作，未来会发生什么越难被预测，这个动作的奖励就越大。所以如果有这样的内在好奇心模块，智能体就会倾向于采取一些风险较大的动作，它想要去探索未知的世界。假设某一个状态是它没有办法预测的，它就会特别想要接近该状态，这可以提高智能体探索的能力。

网络 1 是另外训练出来的。训练的时候，我们会给网络 1 输入 a_t 、 s_t 、 s_{t+1} ，让网络 1 学习根据给定 a_t 、 s_t 预测 \hat{s}_{t+1} 。在智能体与环境交互的时候，我们要把内在好奇心模块固定住。这个想法有一个问题：某些状态很难被预测并不代表它就是好的、它就是应该要被尝试的。例如，俄罗斯轮盘的结果也是没有办法预测的，这并不代表人应该每天去玩俄罗斯轮盘。所以只鼓励智能体去冒险是不够的，因为如果仅仅只有这个网络的架构，智能体只知道什么东西无法预测。如果在某一个状态采取某一个动作，智能体无法预测接下来结果，它就会采取那个动作，但这并不代表这样的结果一定是好的。例如，可能在某个游戏里面，背景会有风吹草动、会有树叶飘动这种无关紧要的事情。也许树叶飘动这件事，是很难被预测的，对智能体来说，它在某一个状态什么都不做，就看着树叶飘动，发现树叶飘动是没有办法预测的，接下来它就会一直看树叶飘动。所以智能体仅有好奇心是不够的，还要让它知道，什么事情是真正重要的。

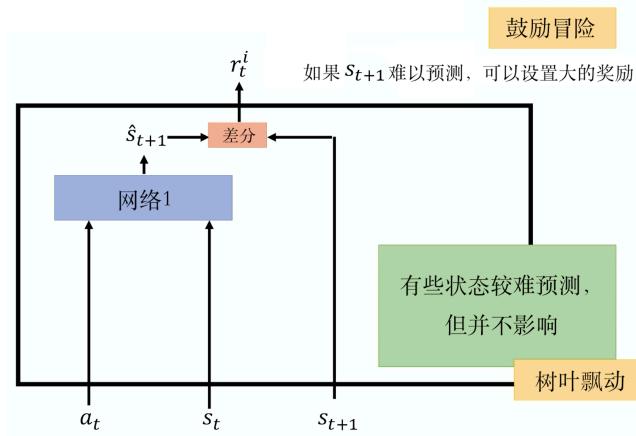


图 10.4 内在好奇心模块设计

怎么让智能体知道什么事情是真正重要的呢？我们要加上另外一个模块，我们要学习一个**特征提取器 (feature extractor)**。如图 10.5 所示，黄色的格子代表特征提取器，它输入一个状态，输出一个特征向量来代表这个状态，我们期待特征提取器可以把没有意义的画面，状态里面没有意义的东西过滤掉，比如风吹草动、白云的飘动以及树叶飘动。

假设特征提取器可以把无关紧要的信息过滤掉，网络 1 实际上做的事情是，给它一个演员和一个状态 s_t 的特征表示 (feature representation)，让它预测状态 s_{t+1} 的特征表示。接下来我们再来评价，这个预测的结果与真正的状态 s_{t+1} 的特征表示像不像，越不像，奖励就越大。怎么学习特征提取器呢？怎么让特征提取器把无关紧要的信息过滤掉呢？我们可以学习另外一个网络，即网络 2。网络 2 把向量 $\phi(s_t)$ 和 $\phi(s_{t+1})$ 作为输入，它要预测动作 a 是什么，它希望这个动作 a 与真正的动作 a 越接近越好。网络 2 会输出一个动作 a_t ，它会输出，从状态 s_t 到状态 s_{t+1} ，要采取的动作与真正的动作越接近越好。加上网络 2 是因为要用 $\phi(s_t)$ 、 $\phi(s_{t+1})$ 预测动作。所以，我们提取出来的特征与预测动作这件事情是有关的，风吹草动等与智能体要采取的动作无关的就会被过滤掉，就不会在被提取出来的向量中被表示。

10.3 课程学习

第二个方向是**课程学习 (curriculum learning)**。课程学习不是强化学习独有的概念，在机器学习尤其是深度学习中，我们都会用到课程学习的概念。具体来说，课程学习是指我们为智能体的学习做规划，给他“喂”的训练数据是有顺序的，通常都是由简单到难的。比如，假设我们要教一个小朋友学微积分，

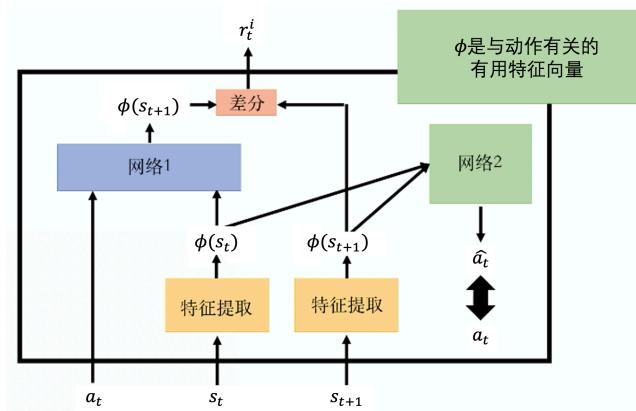


图 10.5 好奇心模块

他做错题就惩罚他，这样他很难学会。我们应该先教他乘法，再教他微积分。所以课程学习就是指在训练智能体的时候，训练数据从简单到困难。就算不是强化学习，一般在训练深度网络的时候，我们有时候也会这么做。例如，在训练循环神经网络的时候，已经有很多的文献都证明，给智能体先看短的序列，再慢慢给它看长的序列，通常它可以学得比较好。在强化学习里面，我们就是要帮智能体规划它的课程，课程难度从易到难。

例如，Meta 玩 *ViZDoom* 的智能体表现非常好，它参加 *ViZDoom* 的比赛得了第一名。对于 Meta 玩 *ViZDoom* 的智能体，是有为智能体规划课程的，从课程 0 一直到课程 7。在不同的课程里面，怪物的速度与血量是不一样的。所以，在越进阶的课程里面，怪物的速度越快，血量越多。如果直接上课程 7，智能体是无法学习的。要从课程 0 开始，一点一点增加难度，这样智能体才学得起来。

再例如，对于把蓝色的板子穿过柱子的任务，怎么让机器人一直从简单学到难呢？如图 10.6（左）所示，也许一开始，板子就已经在柱子上了。这时候，机器人只要把蓝色的板子压下去就可以了。这种情况比较简单，机器人应该很快就能学会。因为机器人只有往上与往下这两个选择，往下就得到奖励，任务就结束了，所有它也不知道学的是什么。如图 10.6（中）所示，我们把板子放高一点儿，机器人有时候会笨拙地往上拉板子，然后把板子拿出来。如果机器人可以学会压板子，拿板子也有很大的可能可以学会。假设机器人现在已经学到，只要板子接近柱子，它就可以把板子压下去。接下来，我们再让它学更一般的情况。如图 10.6（右）所示，一开始，让板子离柱子远一点儿。然后，板子放到柱子上面的时候，机器人就知道把板子压下去，这就是课程学习的概念。当然课程学习有点儿特别，它需要人去为智能体设计课程。

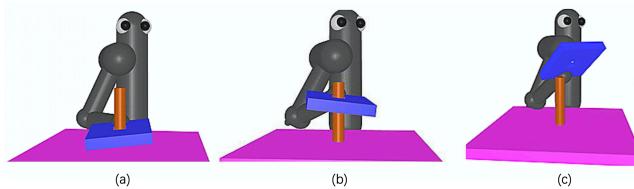


图 10.6 课程学习

有一个比较通用的方法：**逆向课程生成 (reverse curriculum generation)**。我们可以用一个比较通用的方法来帮智能体设计课程。如图 10.7 所示，假设我们一开始有一个状态 s_g ，这是**黄金状态 (gold state)**，也就是最后最理想的结果。如果以板子和柱子的实验为例，黄金状态就是把板子穿过柱子。如果我们以训练机械臂抓东西为例，抓到东西就称为黄金状态。接下来我们根据黄金状态去找其他的状态，这些其他的状态与黄金状态是比较接近的。例如，在让机械臂抓东西的例子里面，机械臂可能还没有抓到东西。假设与黄金状态很接近的状态称为 s_1 。机械臂还没有抓到东西，但它与黄金状态很接近，这种状态可称为 s_1 。至于什么是接近，这取决于具体情况。我们要根据任务来设计怎么从 s_g 采样出 s_1 。接下来，智能体再从 s_1 开始与环境交互，看它能不能够达到黄金状态 s_g ，在每一个状态下，智能体与环境交互的时

候，都会得到一个奖励。

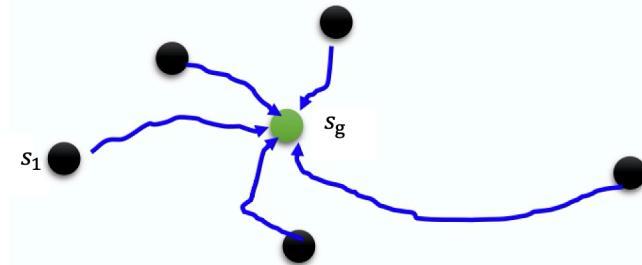


图 10.7 逆向课程生成

接下来，我们把奖励特别极端的情况去掉。奖励特别极端的情况的意思是这些情况太简单或是太难了。如果奖励很大，就代表这个情况太简单了，就不用学习了，因为智能体已经会了，它可以得到很大的奖励。如果奖励太小，就代表这个情况太难了，依照智能体体现的能力它学不会，所以就不学这个，只学一些奖励适中的情况。

接下来，再根据这些奖励适中的情况采样出更多的状态。假设一开始，机械臂在某个位置可以抓得到后。接下来，机械臂就再离远一点儿，看看能不能抓到；又能抓到后，再离远一点儿，看看能不能抓到。这是一个有用的方法，称为逆课程学习 (reverse curriculum learning)。前面讲的是课程学习，就是我们要为智能体规划学习的顺序。而逆课程学习是从黄金状态反推，如图 10.8 所示，就是从目标反推，所以这称为逆课程学习。

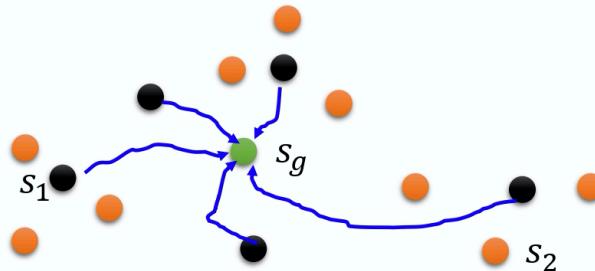


图 10.8 逆课程学习

10.4 分层强化学习

第三个方向是分层强化学习 (hierarchical reinforcement learning, HRL)。分层强化学是指，我们有多个智能体，一些智能体负责比较高级的东西，它们负责定目标，定完目标以后，再将目标分配给其他的智能体，让其他智能体来执行目标。这样的想法也是很合理的。例如，假设我们想写一篇论文，首先我们要想创新点，想完创新点后，还要做实验。做实验以后，我们要写论文。写完论文以后还要投稿、发表。每一个动作下面又会再细分，比如怎么做实验呢？我们要先收集数据，收集完数据以后，要标注标签，还要设计一个网络，然后又训练不起来，要训练很多次。重新设计网络架构好几次，最后才把网络训练起来。所以，我们要完成一个很大的任务的时候，并不是从非常底层的动作开始做，其实是一个计划的。我们会先想，如果要完成这个最大的任务，要将之拆解成哪些小任务，每一个小任务要怎么拆解成更小的任务。例如，我们直接写一本书可能很困难，但先把一本书拆成几章，每章拆成几节，每节又拆成几段，每段又拆成几个句，这样可能就比较好写，这就是分层强化学。

如图 10.9 所示，例如，假设校长、教授和研究生都是智能体，并且我们所在的学校只要进入百大大学校 (QS 排名前 100 的学校) 就可以得到奖励。假设进入百大大学校，校长就要提出愿景并告诉其他的智能体，现在我们要达到什么样的目标。校长的愿景可能是教授每年都要发 3 篇期刊。这些智能体都是分层的，所

以上的智能体，他的动作就是提出愿景。他把他的愿景传给下一层的智能体，下一层的智能体会接收这个愿景。如果他下面还有其他智能体，他就会提出新的愿景。比如，校长要教授发期刊论文，但教授自己没时间实验，他也只能够让下面的研究生做实验。所以教授就提出愿景，做出实验的规划，研究生才是执行这个实验的人。把实验做出来以后，大家就可以得到奖励。现在是这样的，在学习的时候，每一个智能体都会学习，他们的整体目标就是要得到最后的奖励。前面的智能体，他们提出来的动作就是愿景。但是，假设他们提出来的愿景是下面的智能体达不到的，就会被讨厌。例如，教授都一直让研究生做一些很困难的实验，研究生做不出来，教授就会得到一个惩罚。所以如果下层的智能体没有办法达到上层智能体所提出的目标，上层的智能体就会被讨厌，它就会得到一个负奖励。所以他要避免提出的那些愿景是下层的智能体做不到的。每一个智能体都把上层的智能体所提出的愿景当作输入，决定他自己要产生什么输出。

但是就算看到上面的愿景让我们做某件事情，最后也不一定能做成这件事情。假设本来教授的目标是要发期刊论文，但他突然切换目标，要变成一个 YouTuber。这时，我们需要把原来的愿景改成变成 YouTuber。因为虽然本来的愿景是发期刊论文，但是后来变成 YouTuber，这些动作是没有被浪费的。我们就假设，本来的愿景就是要成为 YouTuber，我们就知道成为 YouTuber 要怎做了。这就是分层强化学习，是可以实现的技巧。

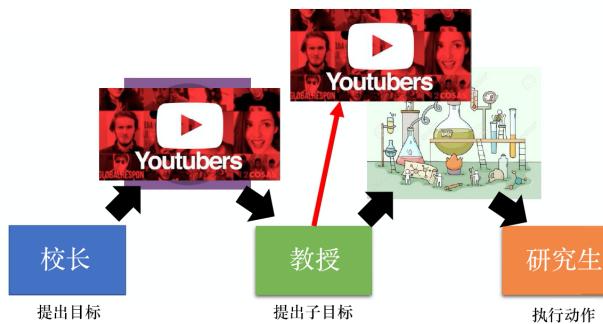


图 10.9 分层强化学习例子^[1]

图 10.10 是真实游戏的例子。第一个游戏是走迷宫，蓝色的是智能体，蓝色的智能体要走到黄色的目标。第二个游戏是单摆，单摆要碰到黄色的球。愿景是什么呢？在走迷宫游戏里面，只有两个智能体，下层的智能体负责决定要怎么走，上层的智能体负责提出愿景。虽然，实际上我们可以用很多层，但这只用了两层。走迷宫的游戏中粉红色的点代表的就是愿景。上层的智能体告诉蓝色的智能体，我们现在的第一个目标是先走到某个位置。蓝色的智能体到达以后，再说新的目标是走到另一个位置。蓝色的智能体再到达以后，新的目标会在其他位置。接下来蓝色的智能体又到达这个位置，最后希望蓝色的智能体可以到达黄色的位置。单摆的例子也一样，粉红色的点代表的是上层的智能体所提出的愿景，所以这个智能体先摆到这边，接下来，新的愿景又跑到某个位置，所以它又摆到对应的位置。然后，新的愿景又跑到上面。然后又摆到上面，最后就走到黄色的位置。这就是分层强化学习。

最后，我们对分层强化学习进行总结。分层强化学习是指将一个复杂的强化学习问题分解成多个小的、简单的子问题，每个子问题都可以单独用马尔可夫决策过程来建模。这样，我们可以将智能体的策略分为高层次策略和低层次策略，高层次策略根据当前状态决定如何执行低层次策略。这样，智能体就可以解决一些非常复杂的任务。

10.5 关键词

设计奖励 (reward shaping): 当智能体与环境进行交互时，我们人为设计一些奖励，从而“指挥”智能体，告诉其采取哪一个动作是最优的。需要注意的是，这个奖励区别于环境的奖励。其可以提高我们估算 Q 函数时的准确性。

内在好奇心模块 (intrinsic curiosity module, ICM): 其代表好奇心驱动这个技术中的增加新的奖励

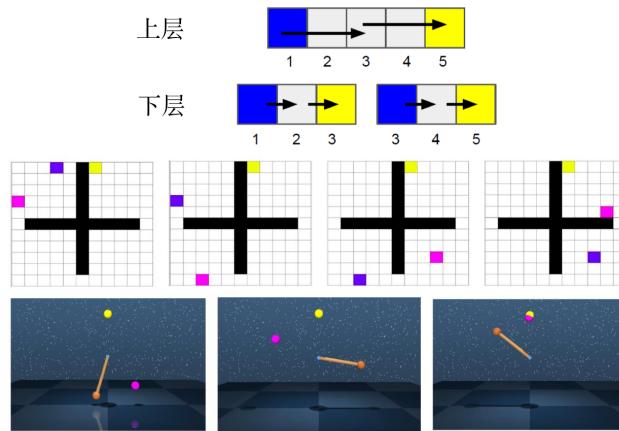


图 10.10 走迷宫和单摆的例子

函数以后的奖励函数。

课程学习 (curriculum learning)：一种广义的用在强化学习中训练智能体的方法，其在输入训练数据的时候，采取由易到难的顺序进行输入，也可以人为设计它的学习过程。这个方法在机器学习和强化学习中普遍使用。

逆课程学习 (reverse curriculum learning)：相较于课程学习，逆课程学习为更广义的方法。其从最终最理想的状态 [我们称之为黄金状态 (gold state)] 开始，依次去寻找距离黄金状态最近的状态作为想让智能体达到的阶段性的“理想”状态。当然，我们会在此过程中有意地去掉一些极端的状态，即太简单、太难的状态。综上，逆课程学习是从黄金状态反推的方法。

分层强化学习 (hierarchical reinforcement learning)：将一个大型的任务，横向或者纵向地拆解成由多个智能体去执行的子任务。其中，有一些智能体负责比较高层次的任务，如负责定目标，定完目标后，再将目标分配给其他的智能体执行。

10.6 习题

10-1 解决稀疏奖励的方法有哪些？

10-2 设计奖励存在什么主要问题？

10-3 内在好奇心模块是什么？我们应该如何设计内在好奇心模块？

参考文献

- [1] LEVY A, PLATT R, SAENKO K. Hierarchical reinforcement learning with hindsight[C]//Proceedings of the International Conference on Learning Representations. ICLR, 2019.

第 11 章 模仿学习

模仿学习 (imitation learning, IL) 讨论的问题是，假设我们连奖励都没有，要怎么进行更新以及让智能体与环境交互呢？模仿学习又被称为示范学习 (learning from demonstration)，学徒学习 (apprenticeship learning)，观察学习 (learning by watching)^[1]。在模仿学习中，有一些专家的示范，智能体也可以与环境交互，但它无法从环境里得到任何的奖励，它只能通过专家的示范来学习什么是好的，什么是不好的。其实，在多数情况下，我们都无法从环境里得到非常明确的奖励。例如，如果是棋类游戏或者是电玩，我们将会有非常明确的奖励。但是多数的情况都是没有奖励的，以聊天机器人为例，机器人与人聊天，聊得怎样算是好，聊得怎样算是不好，我们是无法给出明确的奖励的。

当然，虽然我们无法给出明确的奖励，但是我们可以收集专家的示范。例如，在自动驾驶汽车方面，虽然我们无法给出自动驾驶汽车的奖励，但我们可以收集很多人类开车的记录。在聊天机器人方面，我们可能无法定义什么是好的对话，什么是不好的对话，但我们可以收集很多人的对话当作范例。因此模仿学习的实用性非常高。假设我们不知道该怎么定义奖励，就可以收集专家的示范。如果我们可以收集到一些示范，可以收集到一些很厉害的智能体（比如人）与环境实际上的交互，就可以考虑采用模仿学习。在模仿学习里面，我们介绍两个方法：行为克隆 (behavior cloning, BC) 和逆强化学习 (inverse reinforcement learning, IRL)。逆强化学习也被称为逆最优控制 (inverse optimal control)。

11.1 行为克隆

其实行为克隆与监督学习较为相似。以自动驾驶汽车为例，如图 11.1 所示，我们可以收集到人开自动驾驶汽车的数据，比如可以通过行车记录器进行收集。看到图 11.1 所示的观测的时候，人会决定向前，智能体也采取与人一样的行为，即也向前。专家做什么，智能体就做一模一样的事，这就称为行为克隆。

怎么让智能体学会与专家一模一样的行为呢？我们可以把它当作一个监督学习的问题，先收集很多行车记录器的数据，再收集人在具体情境下会采取什么样的行为（训练数据）。我们知道人在状态 s_1 会采取动作 a_1 ，人在状态 s_2 会采取动作 a_2 ，人在状态 s_3 会采取动作 a_3 ……接下来，我们就学习一个网络。这个网络就是演员，输入 s_i 的时候，我们希望它的输出是 a_i 。

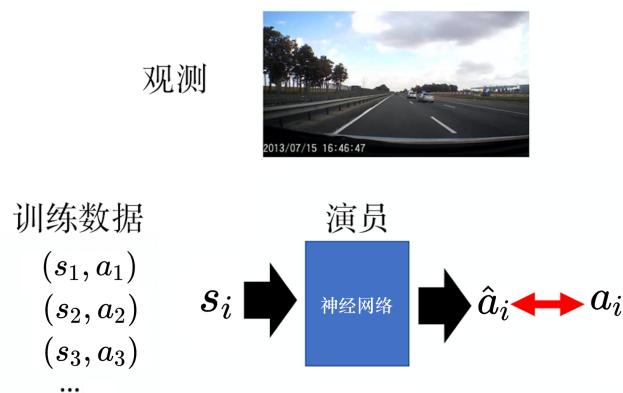


图 11.1 自动驾驶汽车例子

行为克隆虽然非常简单，但它的问题是，如果我们只收集专家的示范，可能我们看过的观测以及状态是非常有限的。例如，如图 11.2 所示，假设我们要学习自动驾驶一辆汽车通过图中的弯道。如果是专家，它将顺着红线通过弯道。但假设智能体很笨，它开车的时候撞墙了，它永远不知道撞墙这种状况要怎么处理。因为训练数据里面从来没有撞墙相关的数据，所以它根本就不知道撞墙这种情况要怎么处理。打电玩也是一样的，让专家去玩《超级马里奥》，专家可能非常强，它从来不会跳不上水管，所以智能体根本不知道跳不上水管时要怎么处理。所以仅仅使用行为克隆是不够的，只观察专家的示范是不够的，还需要结合另一个方法：数据集聚合 (dataset aggregation, DAgger)。

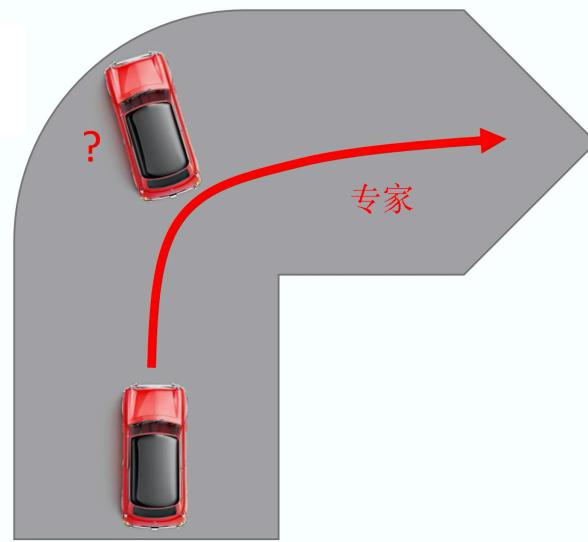


图 11.2 行为克隆的问题

我们希望收集更多样的数据，而不是只收集专家所看到的观测。我们希望能够收集专家在各种极端的情况下所采取的行为。如图 11.3 所示，以自动驾驶汽车为例，一开始我们有演员 θ_1 ，并且让其去驾驶这辆车，同时车上坐了一个专家。这个专家会不断地告诉智能体，如果在这个情境里面，我会怎么样开。所以 θ_1 自己开自己的，但是专家会不断地表达它的想法。比如，一开始的时候，专家可能说往前走。在拐弯的时候，专家可能就会说往右转。但 θ_1 是不管专家的指令的，所以它会继续撞墙。虽然专家说往右转，但是不管他怎么下指令都是没有用的， θ_1 会做自己的事情，因为我们要做的记录的是说，专家在 θ_1 看到这种观测的情况下，它会做什么样的反应。这个方法显然是有一些问题的，因为我们每开一次自动驾驶汽车就会牺牲一个专家。我们用这个方法，牺牲一个专家以后，就会知道，人类在快要撞墙的时候，会采取什么样的行为。再用这些数据训练新的演员 θ_2 ，并反复进行这个过程，这个方法称为数据集聚合。

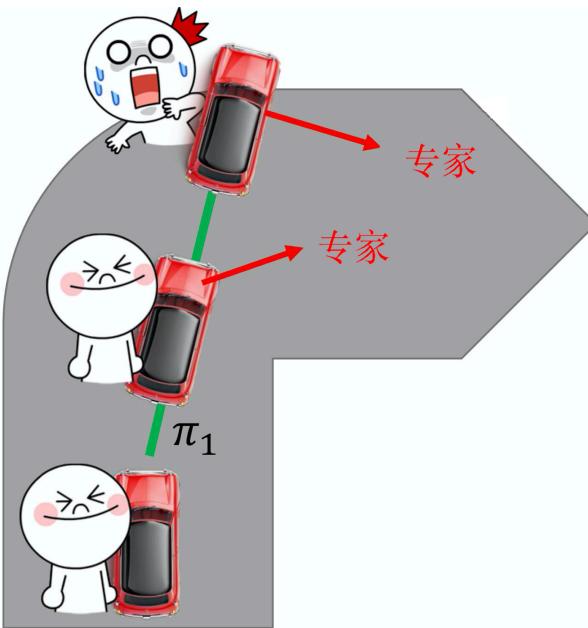


图 11.3 数据集聚合

行为克隆还有一个问题：智能体会完全模仿专家的行为，不管专家的行为是否有道理，就算没有道理，没有什么用，就算这是专家本身的习惯，智能体也会把它记下来。如果智能体确实可以记住所有专家的行

为，也许还好。因为如果专家这么做，有些行为是多余的。但是没有问题，假设智能体的行为可以完全仿造专家行为，也就算了，它就是与专家一样得好，只是做一些多余的事。但问题是智能体是一个网络，网络的容量是有限的。就算给网络训练数据，它在训练数据上得到的正确率往往也不是 100%，它有些事情是学不起来的。这个时候，什么该学，什么不该学就变得很重要。

例如，如图 11.4 所示，在学习中文的时候，老师有语音、行为和知识，但其实只有语音部分是重要的，知识部分是不重要的。也许智能体只能学一件事，如果它只学到了语音，没有问题。如果它只学到了手势，这样就有问题了。所以让智能体学习什么东西是需要模仿的、什么东西是不需要模仿的，这件事情是很重要的。而单纯的行为克隆没有学习这件事情，因为智能体只是复制专家所有的行为而已，它不知道哪些行为是重要的，是对接下来有影响的，哪些行为是不重要的、是对接下来没有影响的。

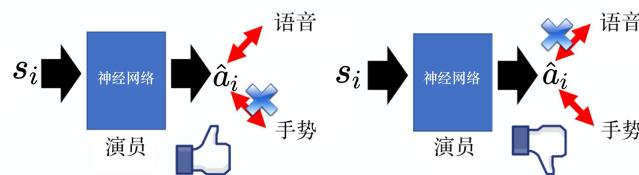


图 11.4 智能体学习中文

行为克隆的问题还在于：我们使用行为克隆的时候，训练数据与测试数据是不匹配的。我们可以用数据集聚合的方法来缓解这个问题。在训练与测试的时候，数据分布是不一样的。因为在强化学习中，动作会影响到接下来的状态。我们先有状态 s_1 ，然后采取动作 a_1 ， a_1 会决定接下来的状态 s_2 。所以在强化学习里有一个很重要的特征，就是我们采取的动作会影响我们接下来的状态，也就是会影响状态的分布。如果有行为克隆，我们只能观察到专家 $\hat{\theta}$ 的一些状态-动作对 (s, a) 。

我们希望可以学习一个 θ^* ，并且希望 θ^* 与 $\hat{\theta}$ 越接近越好。如果 θ^* 可以与 $\hat{\theta}$ 一模一样，训练的时候看到的状态与测试的时候看到的状态会是一样的。因为虽然动作会影响我们看到的状态，但假设两个策略一模一样，在同一个状态都会采取同样的动作，我们接下来看到的状态都会是一样的。但问题就是我们很难让学习出来的策略与专家的策略一模一样。专家是一个自然人，网络要与人一模一样，有点儿困难。

如果 θ^* 与 $\hat{\theta}$ 有一点儿误差，在一般监督学习问题里面，每一个样本 (example) 都是独立的，也许没什么问题。但对强化学习的问题来说，可能在某个地方就是失之毫厘，谬以千里。可能在某个地方，也许智能体无法完全复制专家的行为，它复制的差了一点儿，也许最后得到的结果就会差很多。所以行为克隆并不能够完全解决模仿学习的问题，我们还有另外一个比较好的方法，称为逆强化学习。

11.2 逆强化学习

为什么叫逆强化学习？因为原来的强化学习里，有一个环境和一个奖励函数。根据环境和奖励函数，通过强化学习这一技术，我们会找到一个演员，并会学习出一个最优演员。但逆强化学习刚好是相反的，它没有奖励函数，只有一些专家的示范，但还是有环境的。逆强化学习假设现在有一些专家的示范，用 τ 来代表专家的示范。如果是在玩电玩，每一个 τ 就是一个很会玩电玩的人玩一场游戏的记录。如果是自动驾驶汽车，就是人开自动驾驶汽车的记录。这些就是专家的示范，每一个 τ 是一个轨迹。

把所有专家的示范收集起来，再使用逆强化学习这一技术。使用逆强化学习技术的时候，智能体是可以与环境交互的。但它得不到奖励，它的奖励必须从专家那里推出来。有了环境和专家的示范以后，可以反推出奖励函数。强化学习是由奖励函数反推出什么样的动作、演员是最好的。逆强化学习则反过来，我们有专家的示范，我们相信它是不错的，我就反推，专家是因为什么样的奖励函数才会采取这些行为。有了奖励函数以后，接下来，我们就可以使用一般的强化学习的方法去找出最优演员。所以逆强化学习是先找出奖励函数，找出奖励函数以后，再用强化学习找出最优演员。

把这个奖励函数找出来，相较于原来的强化学习有什么好处呢？一个可能的好处是也许奖励函数是比

较简单的。即虽然专家的行为非常复杂，但也许简单的奖励函数就可以导致非常复杂的行为。一个例子就是人类本身的奖励函数就只有活着这样，每多活一秒，我们就加一分。但人类有非常复杂的行为，但是这些复杂的行为都只是围绕着要从这个奖励函数里面得到分数而已。有时候很简单的奖励函数也许可以推导出非常复杂的行为。

逆强化学习实际上是怎么做的呢？如图 11.5 所示，首先，我们有一个专家 $\hat{\theta}$ ，这个专家与环境交互，产生很多轨迹 $\{\hat{t}_1, \hat{t}_2, \dots, \hat{t}_N\}$ 。如果我们玩游戏，就让某个电玩高手去玩 N 场游戏，把 N 场游戏的状态与动作的序列都记录下来。接下来，我们有一个演员 θ ，一开始演员很烂，这个演员也与环境交互。它也去玩了 N 场游戏，它也有 N 场游戏的记录。接下来，我们要反推出奖励函数。怎么反推出奖励函数呢？原则就是专家永远是最棒的，是先射箭，再画靶的概念。专家去玩一玩游戏，得到这些游戏的记录，演员也去玩一玩游戏，得到这些游戏的记录。接下来，我们要定一个奖励函数，这个奖励函数的原则就是专家得到的分数要比演员得到的分数高（先射箭，再画靶），所以我们就学习出一个奖励函数，这个奖励函数会使专家得到的奖励大于演员得到的奖励。有了新的奖励函数以后，我们就可以使用一般强化学习的方法学习一个演员，这个演员会针对奖励函数最大化它的奖励。它也会采取一些的动作。但是这个演员虽然可以最大化奖励函数，采取大量的动作，得到大量游戏的记录。

但接下来，我们更改奖励函数。这个演员就会很生气，它已经可以在这个奖励函数得到高分。但是它得到高分以后，我们就改奖励函数，仍然让专家可以得到比演员更高的分数。这就是逆强化学习。有了新的奖励函数以后，根据这个新的奖励函数，我们就可以得到新的演员，新的演员再与环境交互。它与环境交互以后，我们又会重新定义奖励函数，让专家得到的奖励比演员的大。

怎么让专家得到的奖励大过演员呢？如图 11.5 所示，我们在学习的时候，奖励函数也许就是神经网络。神经网络的输入为 τ ，输出就是应该要给 τ 的分数。或者假设我们认为输入整个 τ 太难了，因为 τ 是 s 和 a 的一个很强的序列。也许就向它输入一个 s 和 a 的对，它会输出一个实数。把整个 τ 会得到的实数加起来就得到 $R(\tau)$ 。在训练的时候，对于 $\{\hat{t}_1, \hat{t}_2, \dots, \hat{t}_N\}$ ，我们希望它输出的 R 值越大越好。对于 $\{\tau_1, \tau_2, \dots, \tau_N\}$ ，我们就希望 R 值越小越好。

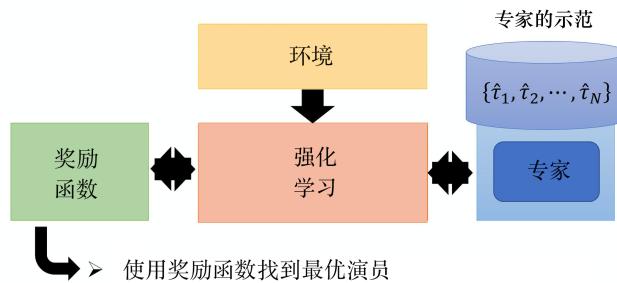


图 11.5 逆强化学习实际的做法

什么可以被称为一个最好的奖励函数呢？最后我们学习出来的奖励函数应该是专家和演员在这个奖励函数上都会得到一样高的分数。最终的奖励函数无法分辨出谁应该会得到比较高的分数。通常在训练的时候，我们会迭代地去做。最早的逆强化学习对奖励函数有些限制，它是假设奖励函数是线性的（linear）。如果奖励函数是线性，我们可以证明这个算法会收敛（converge）。但是如果奖励函数不是线性的，我们就无法证明它会收敛。

逆强化学习的框架如图 11.6 所示，其实我们只要把逆强化学习中的演员看成生成器，把奖励函数看成判别器，它就是生成对抗网络。所以逆强化学习会不会收敛就等于生成对抗网络会不会收敛。如果我们已经实现过，就会知道逆强化学习不一定会收敛。但除非我们对 R 执行一个非常严格的限制，否则如果 R 是一个一般的网络，我们就会有很大的麻烦。

我们可以把逆强化学习与生成对抗网络详细地比较一下。如图 11.7 所示，在生成对抗网络里面，我们有一系列很好的图、一个生成器和一个判别器。一开始，生成器不知道要产生什么样的图，它就会乱画。判别器的工作就是给画的图打分，专家画的图得高分，生成器画的图得低分。生成器会想办法去骗过判别

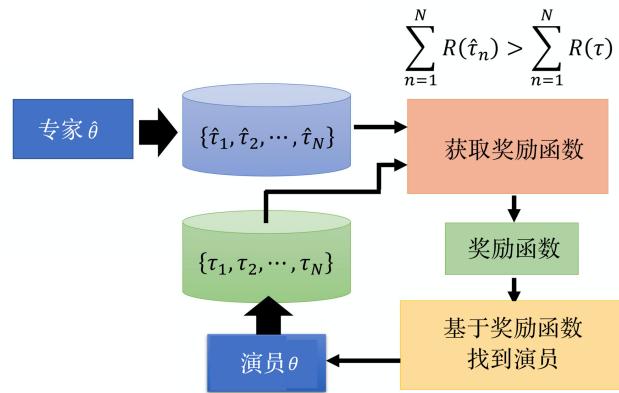


图 11.6 逆强化学习的框架

器，生成器希望判别器也给它画的图打高分。整个过程与逆强化学习是一模一样的。专家画的图就是专家的示范。生成器就是演员，生成器画很多图，演员与环境交互，产生很多轨迹。演员与环境交互的记录其实就等价于生成对抗网络里面的这些图。然后我们学习一个奖励函数。奖励函数就是判别器。奖励函数要给专家的示范打高分，给演员交互的结果打低分。接下来，演员会想办法，从已经学习出的奖励函数中得到高分，然后迭代地循环。

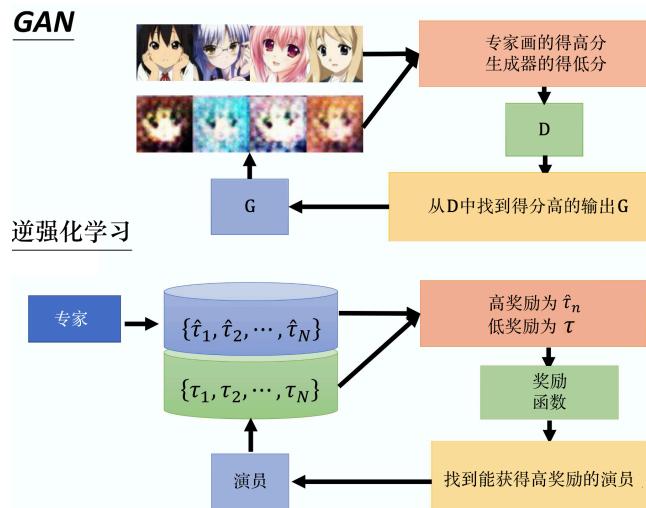


图 11.7 生成对抗网络与逆强化学习的区别

逆强化学习有很多的应用，比如可以用于自动驾驶汽车，有人用这个技术来学开自动驾驶汽车的不同风格。每个人在开车的时候会有不同风格，例如，能不能压到线、能不能倒退、要不要遵守交通规则等。每个人的风格是不同的，用逆强化学习可以让自动驾驶汽车学会各种不同的开车风格。

逆强化学习有一个有趣的地方：通常我们不需要太多的训练数据，训练数据往往都是个位数。因为逆强化学习只是一种示范，实际上智能体可以与环境交互多次，所以我们往往会看到只用几笔数据就可以训练出一些有趣的结果。图 11.8 所示为让自动驾驶汽车学会在停车场中安全停车的例子。这个例子的示范是这样的：蓝色是终点，自动驾驶汽车要开到蓝色终点停车。给智能体只看一行的 4 个示范，让它学习怎么开车，最后它就可以学出，如果它要在红色的终点位置停车，应该这样开。给智能体看不同的示范，最后它学出来的开车的风格就会不太一样。例如，图 11.8 第二行所示为不守规矩的开车方式，因为它会开到道路之外，并且还会穿过其他的车。所以智能体就会学到一些不符合交通规范的行为，例如不一定非要走在道路上、可以走非道路的地方等。图 11.8 第三行所示为倒退停车，智能体也会学会说，它可以倒退。

我们也可以用逆强化学习训练机器人，我们可以让机器人做一些我们人类想要它做的动作。过去，如果我们要训练机器人，让它做我们想要它做的动作，其实是比较麻烦的。例如，如果我们要操控机械臂，

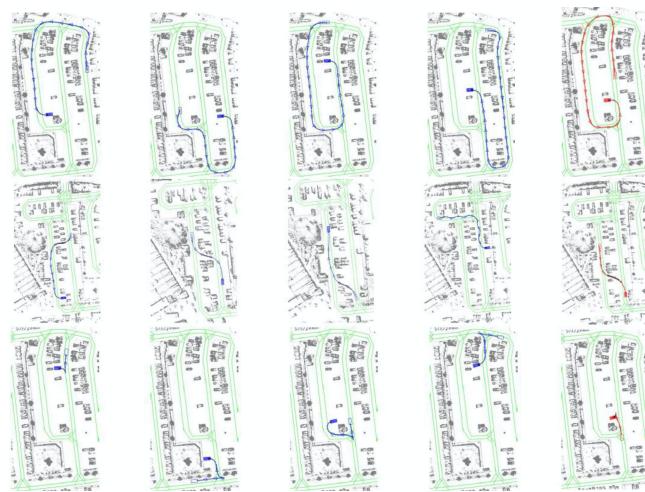


图 11.8 自动驾驶汽车停车例子^[2]

就需要花很多精力编写程序，这样才能让机械臂做一件很简单的事情。有了逆强化学习技术，我们自身可以做示范，机器人就通过示范来学习。比如，让机器人学会摆盘子，拉着机器人的手臂去摆盘子，机器自己动。再如，让机器人学会倒水，人只教它 20 次，杯子每次放的位置不太一样。

11.3 第三人称视角模仿学习

其实还有很多与训练机器人相关的研究，如图 11.9 所示，例如，我们在教机器人做动作的时候，要注意的是也许机器人的视角与我们人类的视角是不太一样的。在上面的例子中，人与机器人的动作是一样的。但是在未来的世界里面，也许机器人是看着人的行为学习的。刚才是人拉着，假设我们要让机器人学会打高尔夫球，如果与上面的例子相似，就是人拉着机器人手臂去打高尔夫球，但是在未来有没有可能，机器人看着人打高尔夫球，它自己就学会打高尔夫球了呢？这个时候，要注意的是机器人的视角与它真正去采取这个行为的视角是不一样的。机器人必须了解到当它是第三人称视角的时候，看到另外一个人在打高尔夫球，与它实际上自己去打高尔夫球的视角显然是不一样的。但它怎么把它是第三人称视角所观察到的经验泛化到它是第一人称视角的时候所采取的行为，这就需要用到**第三人称视角模仿学习 (third person imitation learning)** 技术。



图 11.9 第三人称视角模仿学习例子

这怎么做呢？第三人称视角模仿学习技术其实不只用到了模仿学习，它还用到了**领域对抗训练 (domain-adversarial training)**。领域对抗训练也是一种生成对抗网络的技术。如图 11.10 所示，我们希望有一个特征提取器，有两幅不同领域 (domain) 的图像，通过特征提取器以后，无法分辨出图像来自哪一个领域。第一人称视角和第三人称视角模仿学习用的技术是一样的，希望学习一个特征提取器，智能体在第三人称的时候与它在第一人称的时候的视角其实是一样的，就是把最重要的东西抽出来就好了。