

图 2.25 马尔可夫决策过程控制：策略迭代示例

左、右随机改变，而是会选取最佳的策略进行改变。

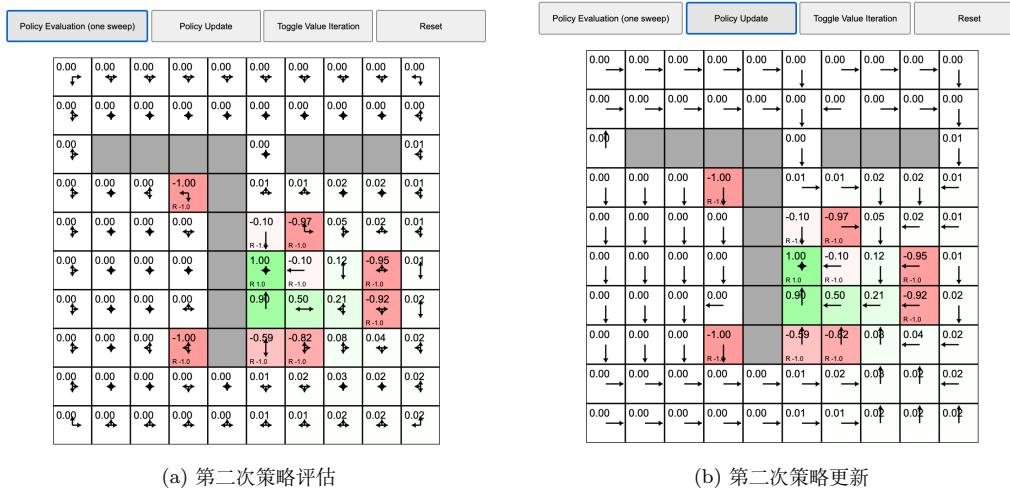


图 2.26 马尔可夫决策过程控制：策略迭代示例

如图 2.27a 所示，我们再次执行策略评估，格子的值又在不停地变化，变化之后又收敛了。如图 2.27b 所示，我们再执行一次策略更新。现在格子的值又会有变化，每一个状态中格子的最佳策略也会产生一些改变。如图 2.28a 所示，我们再执行一遍策略更新，格子的值没有发生变化，这说明整个马尔可夫决策过程已经收敛了。所以现在每个状态的值就是当前最佳的价值函数的值，当前状态对应的策略就是最佳的策略。

通过上面的例子，我们知道策略迭代可以把网格世界“解决掉”。“解决掉”是指，不管在哪个状态，我们都可以利用状态对应的最佳的策略到达可以获得最多奖励的状态。

如图 2.28b 所示，我们再用价值迭代来解马尔可夫决策过程，单击“切换成价值迭代”。当格子的值确定后，就会产生它的最佳状态，最佳状态提取的策略与策略迭代得出的最佳策略是一致的。在每个状态，我们使用最佳策略，就可以到达得到最多奖励的状态。

我们再来对比策略迭代和价值迭代，这两个算法都可以解马尔可夫决策过程的控制问题。策略迭代分两步。首先进行策略评估，即对当前已经搜索到的策略函数进行估值。得到估值后，我们进行策略改进，即把 Q 函数算出来，进行进一步改进。不断重复这两步，直到策略收敛。价值迭代直接使用贝尔曼最优方程进行迭代，从而寻找最佳的价值函数。找到最佳价值函数后，我们再提取最佳策略。



图 2.27 马尔可夫决策过程控制：策略迭代示例

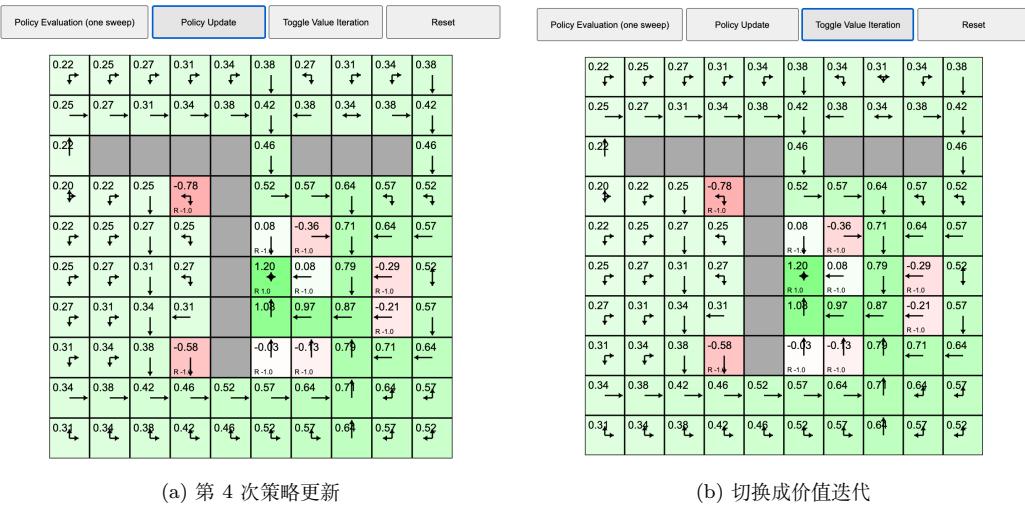


图 2.28 马尔可夫决策过程控制：策略迭代示例

2.3.14 马尔可夫决策过程中的预测和控制总结

总结如表 2.1 所示，我们使用动态规划算法来解马尔可夫决策过程里面的预测和控制，并且采取不同的贝尔曼方程。对于预测问题，即策略评估的问题，我们不停地执行贝尔曼期望方程，这样就可以估计出给定的策略，然后得到价值函数。对于控制问题，如果我们采取的算法是策略迭代，使用的就是贝尔曼期望方程；如果我们采取的算法是价值迭代，使用的就是贝尔曼最优方程。

2.4 关键词

马尔可夫性质 (Markov property, MP)：如果某一个过程未来的状态与过去的状态无关，只由现在的状态决定，那么其具有马尔可夫性质。换句话说，一个状态的下一个状态只取决于它的当前状态，而与它当前状态之前的状态都没有关系。

马尔可夫链 (Markov chain)：概率论和数理统计中具有马尔可夫性质且存在于离散的指数集 (index set) 和状态空间 (state space) 内的随机过程 (stochastic process)。

状态转移矩阵 (state transition matrix)：状态转移矩阵类似于条件概率 (conditional probability)，其表示当智能体到达某状态后，到达其他所有状态的概率。矩阵的每一行描述的是从某节点到达所有其他

表 2.1 动态规划算法

问题	贝尔曼方程	算法
预测	贝尔曼方程	迭代策略评估
控制	贝尔曼期望方程	策略迭代
控制	贝尔曼最优方程	价值迭代

节点的概率。

马尔可夫奖励过程 (Markov reward process, MRP)：本质是马尔可夫链加上一个奖励函数。在马尔可夫奖励过程中，状态转移矩阵和它的状态都与马尔可夫链的一样，只多了一个奖励函数。奖励函数是一个期望，即在某一个状态可以获得多大的奖励。

范围 (horizon)：定义了同一个回合 (episode) 或者一个完整轨迹的长度，它是由有限个步数决定的。

回报 (return)：把奖励进行折扣 (discounted)，然后获得的对应的奖励。

贝尔曼方程 (Bellman equation)：其定义了当前状态与未来状态的迭代关系，表示当前状态的价值函数可以通过下个状态的价值函数来计算。贝尔曼方程因其提出者、动态规划创始人理查德·贝尔曼 (Richard Bellman) 而得名，同时也被叫作“动态规划方程”。贝尔曼方程即 $V(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s)V(s')$ ，特别地，其矩阵形式为 $V = R + \gamma PV$ 。

蒙特卡洛算法 (Monte Carlo algorithm, MC algorithm)：可用来计算价值函数的值。使用本节中小船的例子，当得到一个马尔可夫奖励过程后，我们可以从某一个状态开始，把小船放到水中，让它随波流动，这样就会产生一个轨迹，从而得到一个折扣后的奖励 g 。当积累该奖励到一定数量后，用它直接除以轨迹数量，就会得到其价值函数的值。

动态规划算法 (dynamic programming, DP)：其可用来计算价值函数的值。通过一直迭代对应的贝尔曼方程，最后使其收敛。当最后更新的状态与上一个状态差距不大的时候，动态规划算法的更新就可以停止。

Q 函数 (Q-function)：其定义的是某一个状态和某一个动作所对应的有可能得到的回报的期望。

马尔可夫决策过程中的预测问题：即策略评估问题，给定一个马尔可夫决策过程以及一个策略 π ，计算它的策略函数，即每个状态的价值函数值是多少。其可以通过动态规划算法解决。

马尔可夫决策过程中的控制问题：即寻找一个最佳策略，其输入是马尔可夫决策过程，输出是最佳价值函数 (optimal value function) 以及最佳策略 (optimal policy)。其可以通过动态规划算法解决。

最佳价值函数：搜索一种策略 π ，使每个状态的价值最大， V^* 就是到达每一个状态的极值。在极值中，我们得到的策略是最佳策略。最佳策略使得每个状态的价值函数都取得最大值。所以当我们说某一个马尔可夫决策过程的环境可解时，其实就是我们可以得到一个最佳价值函数。

2.5 习题

2-1 为什么在马尔可夫奖励过程中需要有折扣因子？

2-2 为什么矩阵形式的贝尔曼方程的解析解比较难求得？

2-3 计算贝尔曼方程的常见方法有哪些，它们有什么区别？

2-4 马尔可夫奖励过程与马尔可夫决策过程的区别是什么？

2-5 马尔可夫决策过程中的状态转移与马尔可夫奖励过程中的状态转移的结构或者计算方面的差异有哪些？

2-6 我们如何寻找最佳策略，寻找最佳策略方法有哪些？

2.6 面试题

2-1 友善的面试官: 请问马尔可夫过程是什么? 马尔可夫决策过程又是什么? 其中马尔可夫最重要的性质是什么呢?

2-2 友善的面试官: 请问我一般怎么求解马尔可夫决策过程?

2-3 友善的面试官: 请问如果数据流不具备马尔可夫性质怎么办? 应该如何处理?

2-4 友善的面试官: 请分别写出基于状态价值函数的贝尔曼方程以及基于动作价值函数的贝尔曼方程。

2-5 友善的面试官: 请问最佳价值函数 V^* 和最佳策略 π^* 为什么等价呢?

2-6 友善的面试官: 能不能手写一下第 n 步的价值函数更新公式呀? 另外, 当 n 越来越大时, 价值函数的期望和方差是分别变大还是变小呢?

参考文献

- [1] 邱锡鹏. 神经网络与深度学习[M]. 北京: 机械工业出版社, 2020.
- [2] 周志华. 机器学习[M]. 北京: 清华大学出版社, 2016.
- [3] SUTTON R S, BARTO A G. Reinforcement learning: An introduction(second edition)[M]. London: The MIT Press, 2018.

第 3 章 表格型方法

本章我们通过最简单的表格型方法（tabular method）来讲解如何使用基于价值的方法求解强化学习问题。

3.1 马尔可夫决策过程

强化学习是一个与时间相关的序列决策的问题。例如，如图 3.1 所示，在 $t - 1$ 时刻，我看到熊对我招手，下意识的动作就是逃跑。熊看到有人逃跑，就可能觉得发现了猎物，并开始发动攻击。而在 t 时刻，我如果选择装死的动作，可能熊咬咬我、摔几下就觉得挺无趣的，可能会走开。这个时候我再逃跑，可能就成功了，这就是一个序列决策过程。

在输出每一个动作之前，我们可以选择不同的动作。比如在 t 时刻，我选择逃跑的时候，可能熊已经追上来了。如果在 t 时刻，我没有选择装死，而是选择逃跑，这个时候熊已经追上来了，那么我就会转移到不同的状态。有一定的概率我会逃跑成功，也有一定的概率我会逃跑失败。我们用状态转移概率 $p[s_{t+1}, r_t | s_t, a_t]$ 来表示在状态 s_t 选择动作 a_t 的时候，转移到转态 s_{t+1} ，而且得到奖励 r_t 的概率是多少。状态转移概率是具有马尔可夫性质的（系统下一时刻的状态仅由当前时刻的状态决定，不依赖于以往任何状态）。因为在这个过程中，下一时刻的状态取决于当前的状态 s_t ，它和之前的 s_{t-1} 和 s_{t-2} 没有关系。再加上这个过程也取决于智能体与环境交互的 a_t ，所以包含了决策的过程，我们称这样的过程为马尔可夫决策过程。马尔可夫决策过程就是序列决策的经典的表现方式。马尔可夫决策过程也是强化学习里面一个非常基本的学习框架。状态、动作、状态转移概率和奖励 (S, A, P, R)，这 4 个合集就构成了强化学习马尔可夫决策过程的四元组，后面也可能会再加上折扣因子构成五元组。

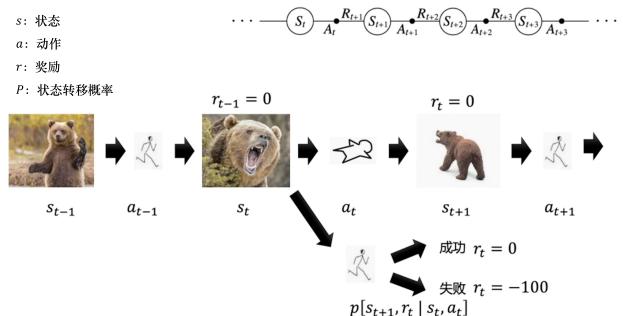


图 3.1 马尔可夫决策过程四元组

3.1.1 有模型

如图 3.2 所示，我们把这些可能的动作和可能的状态转移的关系画成树状。它们之间的关系就是从 s_t 到 a_t ，再到 s_{t+1} ，再到 a_{t+1} ，再到 s_{t+2} 的过程。我们与环境交互时，只能走一条完整的通路，这里面产生了一系列决策的过程，我们与环境交互产生了经验。我们会使用概率函数（probability function） $P[s_{t+1}, r_t | s_t, a_t]$ 和奖励函数 $R[s_t, a_t]$ 来描述环境。概率函数就是状态转移的概率，它反映的是环境的随机性。

如果我们知道概率函数和奖励函数，马尔可夫决策过程就是已知的，我们可以通过策略迭代和价值迭代来找最佳的策略。比如，在熊发怒的情况下，我如果选择装死，假设熊看到人装死就一定会走开，我们就称这里面的状态转移概率是 1。但如果在熊发怒的情况下，我选择逃跑而导致可能成功以及失败两种情况，转移到跑成功情况的概率大概 0.1，跑失败的概率大概是 0.9。

如果我们知道环境的状态转移概率和奖励函数，就可以认为这个环境是已知的，因为我们用这两个函数来描述环境。如果环境是已知的，我们其实可以用动态规划算法去计算，如果要逃脱，那么能够逃脱的概率最大的最佳策略是什么。

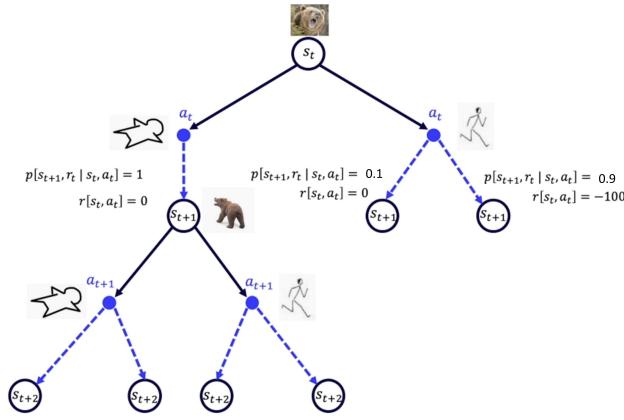


图 3.2 状态转移与序列决策

3.1.2 免模型

很多强化学习的经典算法都是免模型的，也就是环境是未知的。因为现实世界中人类第一次遇到熊时，我们根本不知道能不能逃脱，所以 0.1、0.9 的概率都是虚构出来的概率。熊到底在什么时候往什么方向转变，我们通常是不知道的。我们处在未知的环境里，也就是这一系列的决策的概率函数和奖励函数是未知的，这就是有模型与免模型的最大的区别。

如图 3.3 所示，强化学习可以应用于完全未知的和随机的环境。强化学习像人类一样学习，人类通过尝试不同的路来学习，通过尝试不同的路，人类可以慢慢地了解哪个状态会更好。强化学习用价值函数 $V(S)$ 来表示状态是好的还是坏的，用 Q 函数来判断在什么状态下采取什么动作能够取得最大奖励，即用 Q 函数来表示状态-动作值。

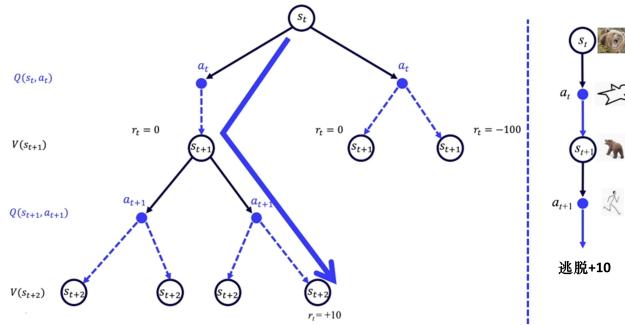


图 3.3 免模型试错探索

3.1.3 有模型与免模型的区别

如图 3.4 所示，策略迭代和价值迭代都需要得到环境的转移和奖励函数，所以在这个过程中，智能体没有与环境进行交互。在很多实际的问题中，马尔可夫决策过程的模型有可能是未知的，也有可能因模型太大不能进行迭代的计算，比如雅达利游戏、围棋、控制直升飞机、股票交易等问题，这些问题的状态转移非常复杂。

如图 3.5 所示，当马尔可夫决策过程的模型未知或者模型很大时，我们可以使用免模型强化学习的方法。免模型强化学习方法没有获取环境的状态转移和奖励函数，而是让智能体与环境进行交互，采集大量的轨迹数据，智能体从轨迹中获取信息来改进策略，从而获得更多的奖励。

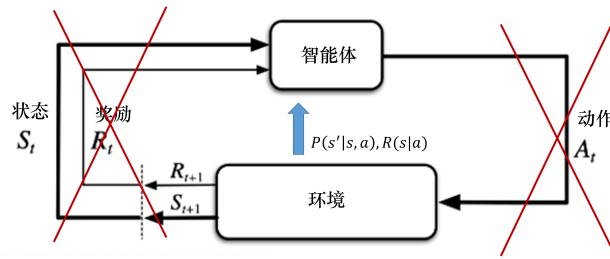


图 3.4 有模型强化学习方法

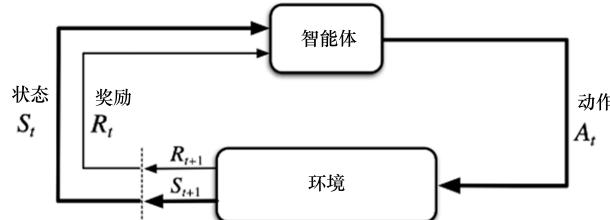


图 3.5 免模型强化学习方法

3.2 Q 表格

在多次尝试和熊打交道之后，我们就可以对熊的不同的状态做出判断，用状态动作价值来表达在某个状态下某个动作的好坏。如图 3.6 所示，如果 **Q 表格**是一张已经训练好的表格，这张表格就像是一本生活手册。通过查看这本手册，我们就知道在熊发怒的时候，装死的价值会高一点；在熊离开的时候，我们偷偷逃跑会比较容易获救。这张表格里面 Q 函数的意义就是我们选择了某个动作后，最后能不能成功，就需要我们去计算在某个状态下选择某个动作，后续能够获得多少总奖励。如果可以预估未来的总奖励的大小，我们就知道在当前的状态下选择哪个动作价值更高。我们选择某个动作是因为这样未来可以获得的价值会更高。所以强化学习的目标导向性很强，环境给出的奖励是非常重要的反馈，它根据环境的奖励来做选择。

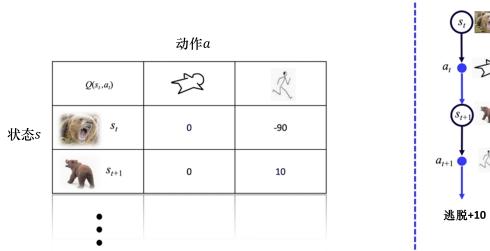


图 3.6 Q 表格

Q: 为什么我们可以用未来的总奖励来评价当前动作是好是坏？

A: 例如，如图 3.7 所示，假设一辆车在路上，当前是红灯，我们直接闯红灯的奖励就很低，因为这违反了交通规则，我们得到的奖励是当前的单步奖励。可是如果我们的车是一辆救护车，我们正在运送病人，把病人快速送达医院的奖励非常高，而且越快奖励越高。在这种情况下，我们可能要闯红灯，因为未来的远期奖励太高了。这是因为在现实世界中奖励往往是延迟的，所以强化学习需要学习远期的奖励。我们一般会从当前状态开始，把后续有可能会收到的所有奖励加起来计算当前动作的 Q 值，让 Q 值可以真正代表当前状态下动作的真正价值。

但有的时候我们把目光放得太长远并不好。如果任务很快就结束，那么考虑到最后一步的奖励无可厚非。但如果任务是一个持续的没有尽头的任务，即**持续式任务 (continuing task)**，我们把未来的奖励全

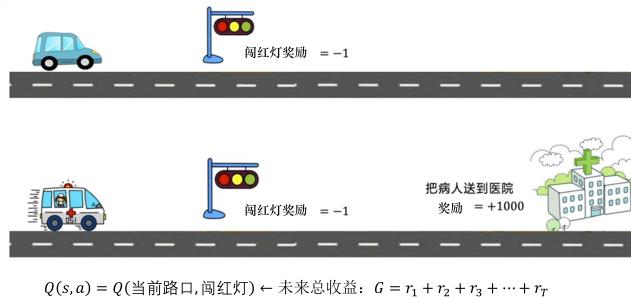


图 3.7 未来的总奖励示例

部相加作为当前的状态价值就很不合理。股票就是一个典型的例子，如图 3.8 所示，我们关注的是累积的股票奖励，可是如果 10 年之后股票才有一次大涨大跌，我们肯定不会把 10 年后的奖励也作为当前动作的考虑因素。这个时候，我们就可以引入折扣因子 γ 来计算未来总奖励， $\gamma \in [0, 1]$ ，越往后 γ^n 就会越小，越后面的奖励对当前价值的影响就会越小。

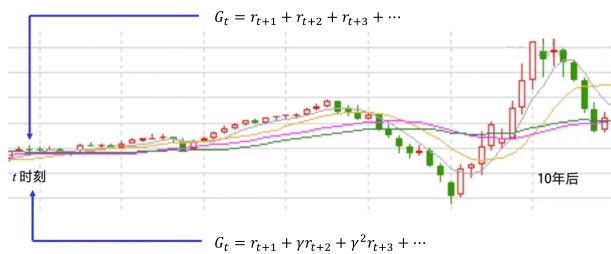


图 3.8 股票的例子

悬崖行走问题是强化学习的一个经典问题，如图 3.9 所示，该问题需要智能体从出发点 S 出发，到达目的地 G，同时避免掉进悬崖（cliff），每走一步就有 -1 分的惩罚，掉进悬崖会有 -100 分的惩罚，但游戏不会结束，智能体会回到出发点，游戏继续，直到到达目的地结束游戏。智能体需要尽快地到达目的地。为了到达目的地，智能体可以沿着例如蓝线和红线的路线行走。

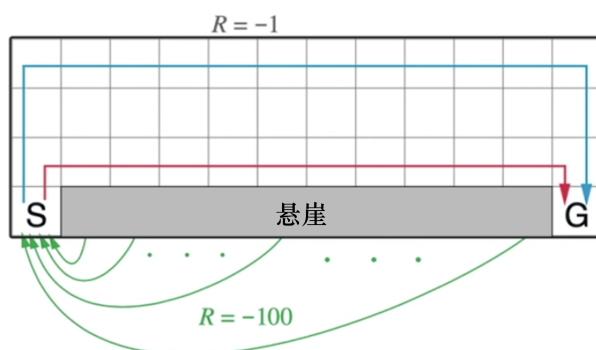


图 3.9 悬崖行走问题

在悬崖行走问题的环境中，我们怎么计算状态动作价值（未来的总奖励）呢？我们可以选择一条路线，计算出这条路线上每个状态动作的价值。在悬崖行走问题里面，智能体每走一步都会拿到 -1 分的奖励，只有到达目的地之后，智能体才会停止。

- 如果 $\gamma = 0$ ，如图 3.10a 所示，我们考虑的就是单步的奖励，我们可以认为它是目光短浅的计算的方法。
- 如果 $\gamma = 1$ ，如图 3.10b 所示，就等于把后续所有的奖励全部加起来，我们可以认为它是目光过于长

远的方法。如果智能体走的不是红色的路线，而是蓝色的路线，算出来的 Q 值可能如图中所示。因此，我们就可以知道，当小乌龟在 -12 的时候，往右走是 -11，往上走是 -15，它知道往右走的价值更大，它就会往右走。

- 如果 $\gamma = 0.6$ ，如图 3.10c 所示，我们的目光没有放得太长远，计算结果如式 (3.1) 所示。我们可以利用公式 $G_t = r_{t+1} + \gamma G_{t+1}$ 从后往前推。

$$\begin{aligned}
 G_{13} &= 0 \\
 G_{12} &= r_{13} + \gamma G_{13} = -1 + 0.6 \times 0 = -1 \\
 G_{11} &= r_{12} + \gamma G_{12} = -1 + 0.6 \times (-1) = -1.6 \\
 G_{10} &= r_{11} + \gamma G_{11} = -1 + 0.6 \times (-1.6) = -1.96 \\
 G_9 &= r_{10} + \gamma G_{10} = -1 + 0.6 \times (-1.96) = -2.176 \approx -2.18 \\
 G_8 &= r_9 + \gamma G_9 = -1 + 0.6 \times (-2.176) = -2.3056 \approx -2.3
 \end{aligned} \tag{3.1}$$

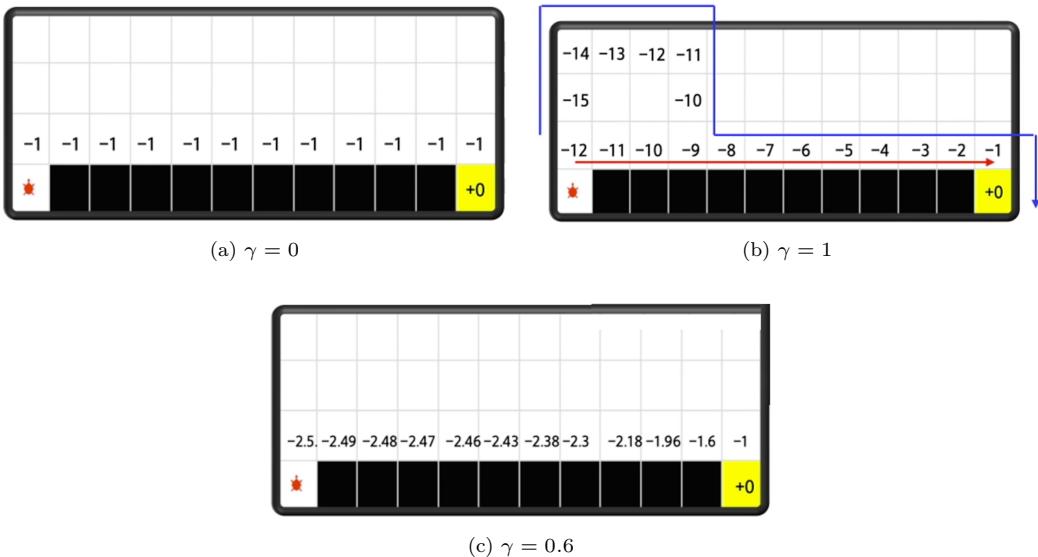


图 3.10 折扣因子

类似于图 3.11，最后我们要求解的就是一张 Q 表格，它的行数是所有状态的数量，一般可以用坐标来表示格子的状态，也可以用 1、2、3、4、5、6、7 来表示不同的位置。Q 表格的列表示上、下、左、右 4 个动作。最开始的时候，Q 表格会全部初始化为 0。智能体会不断和环境交互得到不同的轨迹，当交互的次数足够多的时候，我们就可以估算出每一个状态下，每个动作的平均总奖励，进而更新 Q 表格。Q 表格的更新就是接下来要引入的强化概念。

状态	上	下	左	右
坐标 (1, 1)	0	0	0	0
坐标 (1, 2)	0	0	0	0
坐标 (1, 3)	0	0	0	0
坐标 (1, 4)	0	0	0	0
坐标 (1, 5)	0	0	0	0
坐标 (1, 6)	0	0	0	0
...

图 3.11 Q 表格

强化是指我们可以用下一个状态的价值来更新当前状态的价值，其实就是强化学习里面自举的概念。

在强化学习里面，我们可以每走一步更新一次 Q 表格，用下一个状态的 Q 值来更新当前状态的 Q 值，这种单步更新的方法被称为时序差分方法。

3.3 免模型预测

在无法获取马尔可夫决策过程的模型情况下，我们可以通过蒙特卡洛方法和时序差分方法来估计某个给定策略的价值。

3.3.1 蒙特卡洛策略评估

蒙特卡洛方法是基于采样的方法，给定策略 π ，我们让智能体与环境进行交互，可以得到很多轨迹。每个轨迹都有对应的回报：

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \quad (3.2)$$

我们求出所有轨迹的回报的平均值，就可以知道某一个策略对应状态的价值，即

$$V_\pi(s) = \mathbb{E}_{\tau \sim \pi} [G_t \mid s_t = s] \quad (3.3)$$

蒙特卡洛仿真是指我们可以采样大量的轨迹，计算所有轨迹的真实回报，然后计算平均值。蒙特卡洛方法使用经验平均回报 (empirical mean return) 的方法来估计，它不需要马尔可夫决策过程的状态转移函数和奖励函数，并且不需要像动态规划那样用自举的方法。此外，蒙特卡洛方法有一定的局限性，它只能用在有终止的马尔可夫决策过程中。

接下来，我们对蒙特卡洛方法进行总结。为了得到评估 $V(s)$ ，我们采取了如下的步骤。

(1) 在每个回合中，如果在时间步 t 状态 s 被访问了，那么

- 状态 s 的访问数 $N(s)$ 增加 1， $N(s) \leftarrow N(s) + 1$ 。
- 状态 s 的总的回报 $S(s)$ 增加 G_t ， $S(s) \leftarrow S(s) + G_t$ 。

(2) 状态 s 的价值可以通过回报的平均来估计，即 $V(s) = S(s)/N(s)$ 。

根据大数定律，只要我们得到足够多的轨迹，就可以趋近这个策略对应的价值函数。当 $N(s) \rightarrow \infty$ 时， $V(s) \rightarrow V_\pi(s)$ 。

假设现在有样本 x_1, x_2, \dots, x_t ，我们可以把经验均值 (empirical mean) 转换成增量均值 (incremental mean) 的形式：

$$\begin{aligned} \mu_t &= \frac{1}{t} \sum_{j=1}^t x_j \\ &= \frac{1}{t} \left(x_t + \sum_{j=1}^{t-1} x_j \right) \\ &= \frac{1}{t} (x_t + (t-1)\mu_{t-1}) \\ &= \frac{1}{t} (x_t + t\mu_{t-1} - \mu_{t-1}) \\ &= \mu_{t-1} + \frac{1}{t} (x_t - \mu_{t-1}) \end{aligned} \quad (3.4)$$

通过这种转换，我们就可以把上一时刻的平均值与现在时刻的平均值建立联系，即

$$\mu_t = \mu_{t-1} + \frac{1}{t} (x_t - \mu_{t-1}) \quad (3.5)$$

其中， $x_t - \mu_{t-1}$ 是残差， $\frac{1}{t}$ 类似于学习率 (learning rate)。当我们得到 x_t 时，就可以用上一时刻的值来更新现在的值。

我们可以把蒙特卡洛方法更新的方法写成增量式蒙特卡洛 (incremental MC) 方法。我们采集数据，得到一个新的轨迹 $(s_1, a_1, r_1, \dots, s_t)$ 。对于这个轨迹，我们采用增量的方法进行更新：

$$\begin{aligned} N(s_t) &\leftarrow N(s_t) + 1 \\ V(s_t) &\leftarrow V(s_t) + \frac{1}{N(s_t)} (G_t - V(s_t)) \end{aligned} \quad (3.6)$$

我们可以直接把 $\frac{1}{N(s_t)}$ 变成 α (学习率), 即

$$V(s_t) \leftarrow V(s_t) + \alpha (G_t - V(s_t)) \quad (3.7)$$

其中, α 代表更新的速率, 我们可以对其进行设置。

我们再来看一下动态规划方法和蒙特卡洛方法的差异。动态规划也是常用的估计价值函数的方法。在动态规划方法里面, 我们使用了自举的思想。自举就是我们基于之前估计的量来估计一个量。此外, 动态规划方法使用贝尔曼期望备份 (Bellman expectation backup), 通过上一时刻的值 $V_{i-1}(s')$ 来更新当前时刻的值 $V_i(s)$, 即

$$V_i(s) \leftarrow \sum_{a \in A} \pi(a | s) \left(R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) V_{i-1}(s') \right) \quad (3.8)$$

将其不停迭代, 最后可以收敛。如图 3.12 所示, 贝尔曼期望备份有两层加和, 即内部加和和外部加和, 计算两次期望, 得到一个更新。

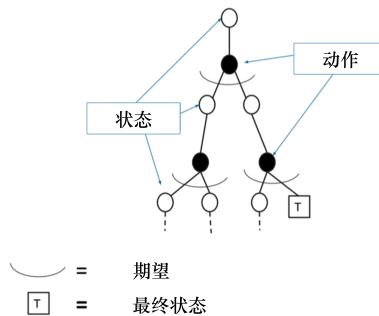


图 3.12 贝尔曼期望备份

蒙特卡洛方法通过一个回合的经验平均回报 (实际得到的奖励) 来进行更新, 即

$$V(s_t) \leftarrow V(s_t) + \alpha (G_{i,t} - V(s_t)) \quad (3.9)$$

如图 3.13 所示, 我们使用蒙特卡洛方法得到的轨迹对应树上蓝色的轨迹, 轨迹上的状态已经是决定的, 采取的动作也是已经决定的。我们现在只更新这条轨迹上的所有状态, 与这条轨迹没有关系的状态都不进行更新。

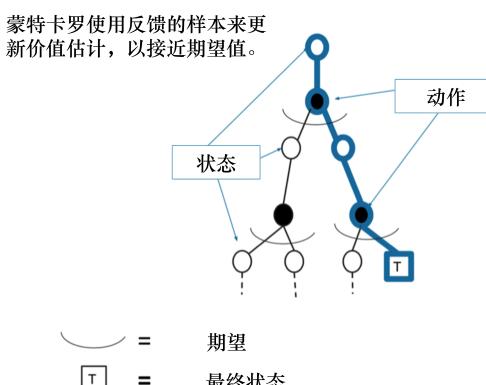


图 3.13 蒙特卡洛方法更新

蒙特卡洛方法相比动态规划方法是有一些优势的。首先，蒙特卡洛方法适用于环境未知的情况，而动态规划是有模型的方法。蒙特卡洛方法只需要更新一条轨迹的状态，而动态规划方法需要更新所有的状态。状态数量很多的时候（比如 100 万个、200 万个），我们使用动态规划方法进行迭代，速度是非常慢的。这也是基于采样的蒙特卡洛方法相对于动态规划方法的优势。

3.3.2 时序差分

为了让读者更好地理解时序差分这种更新方法，我们给出它的“物理意义”。我们先了解一下巴甫洛夫的条件反射实验，如图 3.14 所示，这个实验讲的是小狗会对盆里面的食物无条件产生刺激，分泌唾液。一开始小狗对于铃声这种中性刺激是没有反应的，可是我们把铃声和食物结合起来，每次先给它响一下铃，再给它喂食物，多次重复之后，当铃声响起的时候，小狗也会开始流口水。盆里的肉可以认为是强化学习里面那个延迟的奖励，声音的刺激可以认为是有奖励的那个状态之前的状态。多次重复实验之后，最后的奖励会强化小狗对于声音的条件反射，它会让小狗知道这个声音代表着有食物，这个声音对于小狗也就有了价值，它听到这个声音就会流口水。

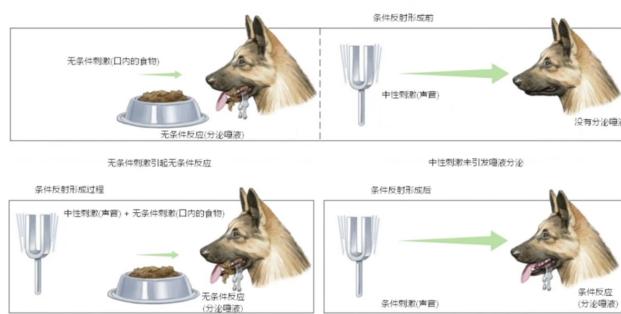


图 3.14 强化概念：巴甫洛夫的条件反射实验

如图 3.15 所示，巴甫洛夫效应揭示的是，当中性刺激（铃声）与无条件刺激（食物）相邻反复出现的时候，中性刺激也可以引起无条件刺激引起的唾液分泌，然后形成条件刺激。我们称这种中性刺激与无条件刺激在时间上面的结合为强化，强化的次数越多，条件反射就会越巩固。小狗本来不觉得铃声有价值，经过强化之后，小狗就会慢慢地意识到铃声也是有价值，它可能带来食物。更重要的是当一种条件反射巩固之后，我们再用另外一种新的刺激和条件反射相结合，还可以形成第二级条件反射，同样地还可以形成第三级条件反射。

在人的身上也可以建立多级的条件反射，例如，比如我们遇到熊可能是这样一个顺序：看到树上有熊爪，然后看到熊，突然熊发怒并扑过来了。经历这个过程之后，我们可能最开始看到熊才会害怕，后面可能看到树上有熊爪就已经有害怕的感觉了。在不断的重复实验后，下一个状态的价值可以不断地强化影响上一个状态的价值。

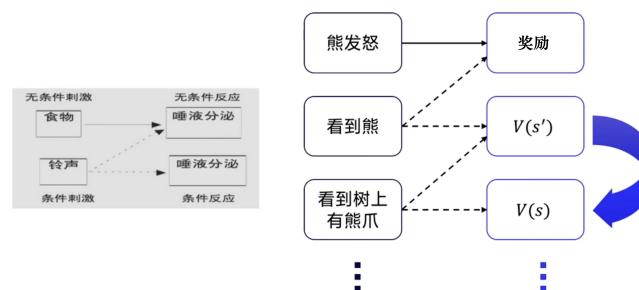


图 3.15 强化示例

为了让读者更加直观地感受下一个状态会如何影响上一个状态（状态价值迭代），我们推荐时序差分

学习网格世界演示 (https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_td.html)。如图 3.16 所示，我们先初始化，然后开始时序差分方法的更新过程。在训练的过程中，小黄球在不断地试错，在探索中会先迅速地发现有奖励的格子。最开始的时候，有奖励的格子才有价值。当小黄球不断地重复走这些路线的时候，有价值的格子可以慢慢地影响它附近的格子的价值。反复训练之后，有价值的格子周围的格子的状态就会慢慢被强化。强化就是价值最终收敛到最优的情况之后，小黄球就会自动往价值高的格子走，就可以走到能够拿到奖励的格子。

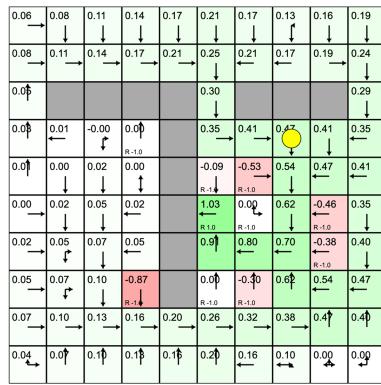


图 3.16 时序差分学习网格世界演示

下面我们开始正式介绍时序差分方法。时序差分是介于蒙特卡洛和动态规划之间的方法，它是免模型的，不需要马尔可夫决策过程的转移矩阵和奖励函数。此外，时序差分方法可以从不完整的回合中学习，并且结合了自举的思想。

接下来，我们对时序差分方法进行总结。时序差分方法的目的是对于某个给定的策略 π ，在线 (online) 地算出它的价值函数 V_π ，即一步一步地 (step-by-step) 算。最简单的算法是一步时序差分 (one-step TD)，即 **TD(0)**。每往前走一步，就做一步自举，用得到的估计回报 (estimated return) $r_{t+1} + \gamma V(s_{t+1})$ 来更新上一时刻的值 $V(s_t)$ ：

$$V(s_t) \leftarrow V(s_t) + \alpha (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \quad (3.10)$$

估计回报 $r_{t+1} + \gamma V(s_{t+1})$ 被称为**时序差分目标 (TD target)**，时序差分目标是带衰减的未来奖励的总和。时序差分目标由两部分组成：

- (1) 我们走了某一步后得到的实际奖励 r_{t+1} ；
- (2) 我们利用了自举的方法，通过之前的估计来估计 $V(s_{t+1})$ ，并且加了折扣因子，即 $\gamma V(s_{t+1})$ 。

时序差分目标是估计有两个原因：

- (1) 时序差分方法对期望值进行采样；
- (2) 时序差分方法使用当前估计的 V 而不是真实的 V_π 。

时序差分误差 (TD error) $\delta = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ 。类比增量式蒙特卡洛方法，给定一个回合 i ，我们可以更新 $V(s_t)$ 来逼近真实的回报 G_t ，具体更新公式为

$$V(s_t) \leftarrow V(s_t) + \alpha (G_{i,t} - V(s_t)) \quad (3.11)$$

式 (3.10) 体现了强化的概念。

我们对比一下蒙特卡洛方法和时序差分方法。在蒙特卡洛方法里面， $G_{i,t}$ 是实际得到的值（可以看成目标），因为它已经把一条轨迹跑完了，可以算出每个状态实际的回报。时序差分不等轨迹结束，往前走一步，就可以更新价值函数。如图 3.17 所示，时序差分方法只执行一步，状态的值就更新。蒙特卡洛方法全部执行完之后，到了终止状态之后，再更新它的值。

接下来，进一步比较时序差分方法和蒙特卡洛方法。

(1) 时序差分方法可以在线学习 (online learning)，每走一步就可以更新，效率高。蒙特卡洛方法必须等游戏结束时才可以学习。

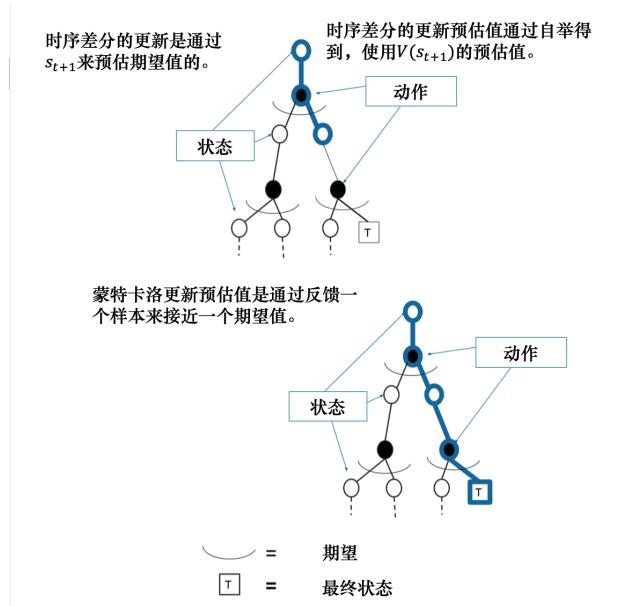


图 3.17 时序差分方法相比蒙特卡洛方法的优势

(2) 时序差分方法可以从不完整序列上进行学习。蒙特卡洛方法只能从完整的序列上进行学习。

(3) 时序差分方法可以在连续的环境下（没有终止）进行学习。蒙特卡洛方法只能在有终止的情况下学习。

(4) 时序差分方法利用了马尔可夫性质，在马尔可夫环境下有更高的学习效率。蒙特卡洛方法没有假设环境具有马尔可夫性质，利用采样的价值来估计某个状态的价值，在不是马尔可夫的环境下更加有效。

例如来解释时序差分方法和蒙特卡洛方法的区别。时序差分方法是指在不清楚马尔可夫状态转移概率的情况下，以采样的方式得到不完整的状态序列，估计某状态在该状态序列完整后可能得到的奖励，并通过不断地采样持续更新价值。蒙特卡洛则需要经历完整的状态序列后，再来更新状态的真实价值。例如，我们想获得开车去公司的时间，每天上班开车的经历就是一次采样。假设我们今天在路口 A 遇到了堵车，时序差分方法会在路口 A 就开始更新预计到达路口 B、路口 C ……，以及到达公司的时间；而蒙特卡洛方法并不会立即更新时间，而是在到达公司后，再更新到达每个路口和公司的时间。时序差分方法能够在知道结果之前就开始学习，相比蒙特卡洛方法，其更快速、灵活^[1]。

如图 3.18 所示，我们可以把时序差分方法进行进一步的推广。之前是只往前走一步，即 $TD(0)$ 。我们可以调整步数 (step)，变成 **n 步时序差分 (n-step TD)**。比如 $TD(2)$ ，即往前走两步，利用两步得到的回报，使用自举来更新状态的价值。这样我们就可以通过步数来调整算法需要的实际奖励和自举。

$$\begin{aligned} n = 1(TD) \quad G_t^{(1)} &= r_{t+1} + \gamma V(s_{t+1}) \\ n = 2 \quad G_t^{(2)} &= r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+2}) \\ &\vdots \\ n = \infty(MC) \quad G_t^{\infty} &= r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{T-t-1} r_T \end{aligned} \tag{3.12}$$

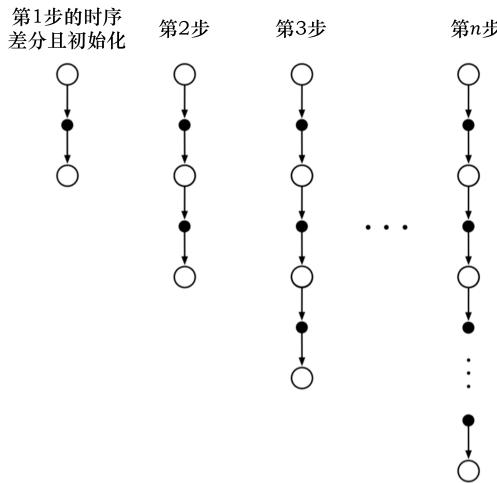
如式 (3.12) 所示，通过调整步数，可以进行蒙特卡洛方法和时序差分方法之间的权衡。如果 $n = \infty$ ，即整个游戏结束后，再进行更新，时序差分方法就变成了蒙特卡洛方法。

n 步时序差分可写为

$$G_t^n = r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n}) \tag{3.13}$$

得到时序差分目标之后，我们用增量式学习 (incremental learning) 的方法来更新状态的价值：

$$V(s_t) \leftarrow V(s_t) + \alpha (G_t^n - V(s_t)) \tag{3.14}$$

图 3.18 n 步时序差分

3.3.3 动态规划方法、蒙特卡洛方法以及时序差分方法的自举和采样

自举是指更新时使用了估计。蒙特卡洛方法没有使用自举，因为它根据实际的回报进行更新。动态规划方法和时序差分方法使用了自举。

采样是指更新时通过采样得到一个期望。蒙特卡洛方法是纯采样的方法。动态规划方法没有使用采样，它是直接用贝尔曼期望方程来更新状态价值的。时序差分方法使用了采样。时序差分目标由两部分组成，一部分是采样，一部分是自举。

如图 3.19 所示，动态规划方法直接计算期望，它把所有相关的状态都进行加和，即

$$V(s_t) \leftarrow \mathbb{E}_\pi [r_{t+1} + \gamma V(s_{t+1})] \quad (3.15)$$

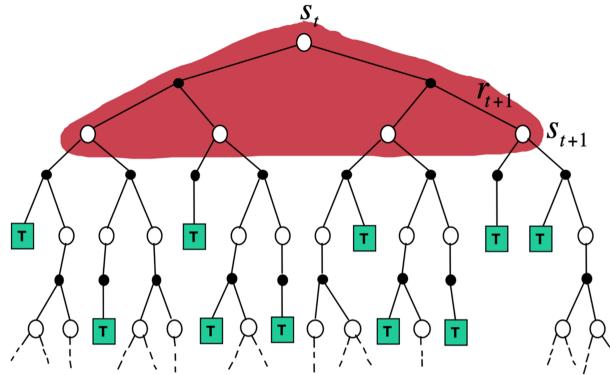


图 3.19 统一视角：动态规划方法备份

如图 3.20 所示，蒙特卡洛方法在当前状态下，采取一条支路，在这条路径上进行更新，更新这条路径上的所有状态，即

$$V(s_t) \leftarrow V(s_t) + \alpha (G_t - V(s_t)) \quad (3.16)$$

如图 3.21 所示，时序差分从当前状态开始，往前走了一步，关注的是非常局部的步骤，即

$$\text{TD}(0) : V(s_t) \leftarrow V(s_t) + \alpha (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \quad (3.17)$$

如图 3.22 所示，如果时序差分方法需要更广度的更新，就变成了动态规划方法（因为动态规划方法是把所有状态都考虑进去来进行更新）。如果时序差分方法需要更深度的更新，就变成了蒙特卡洛方法。

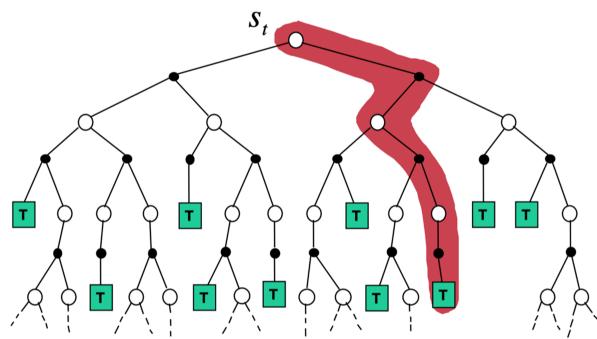


图 3.20 统一视角：蒙特卡洛备份

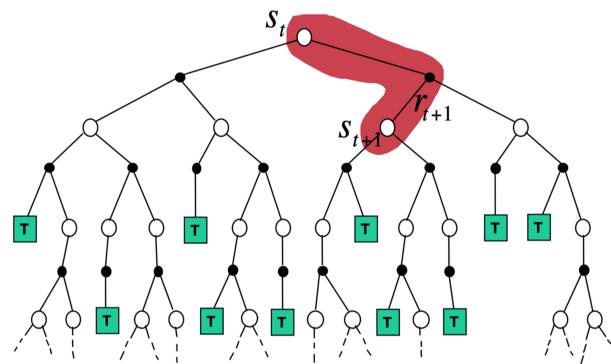


图 3.21 统一视角：时序差分方法备份

图 3.22 右下角是穷举搜索的方法（exhaustive search），穷举搜索的方法不仅需要很深度的信息，还需要很广度的信息。

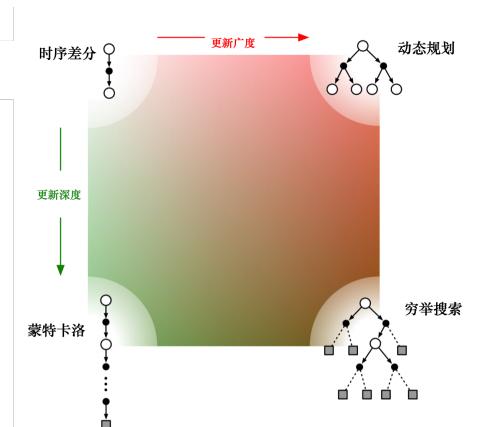


图 3.22 强化学习的统一视角

3.4 免模型控制

在我们不知道马尔可夫决策过程模型的情况下，如何优化价值函数，得到最佳的策略呢？我们可以把策略迭代进行广义的推广，使它能够兼容蒙特卡洛和时序差分的方法，即带有蒙特卡洛方法和时序差分方法的广义策略迭代（generalized policy iteration, GPI）。

如图 3.23 所示，策略迭代由两个步骤组成。第一，我们根据给定的当前策略 π 来估计价值函数；第

二，得到估计的价值函数后，我们通过贪心的方法来改进策略，即

$$\pi' = \text{贪心函数}(V_\pi) \quad (3.18)$$

这两个步骤是一个互相迭代的过程。

$$\pi_{i+1}(s) = \arg \max_a Q_{\pi_i}(s, a) \quad (3.19)$$

我们可以计算出策略 π 的动作价值函数，并且可以根据式 (3.19) 来计算针对状态 $s \in S$ 的新策略 π_{i+1} 。但得到状态价值函数后，我们并不知道奖励函数 $R(s, a)$ 和状态转移 $P(s'|s, a)$ ，所以就无法估计 Q 函数

$$Q_{\pi_i}(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) V_{\pi_i}(s') \quad (3.20)$$

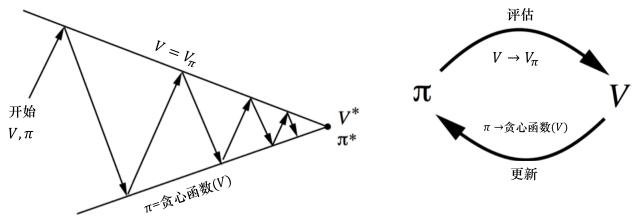


图 3.23 策略迭代

这里有一个问题：当我们不知道奖励函数和状态转移时，如何进行策略的优化？

如图 3.24 所示，针对上述情况，我们引入了广义的策略迭代的方法。我们对策略评估部分进行修改，使用蒙特卡洛的方法代替动态规划的方法估计 Q 函数。我们首先进行策略评估，使用蒙特卡洛方法来估计策略 $Q = Q_\pi$ ，然后进行策略更新，即得到 Q 函数后，我们就可以通过贪心的方法去改进它：

$$\pi(s) = \arg \max_a Q(s, a) \quad (3.21)$$

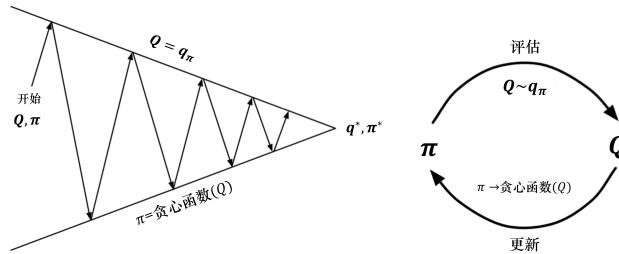


图 3.24 广义策略迭代

图 3.25 所示为蒙特卡洛方法估计 Q 函数的算法。一个保证策略迭代收敛的假设是回合有**探索性开始** (**exploring start**)。假设每一个回合都有一个探索性开始，探索性开始保证所有的状态和动作都在无限步的执行后能被采样到，这样才能很好地进行估计。算法通过蒙特卡洛方法产生很多轨迹，每条轨迹都可以算出它的价值。然后，我们可以通过平均的方法去估计 Q 函数。Q 函数可以看成一个 Q 表格，我们通过采样的方法把表格的每个单元的值都填上，然后使用策略改进来选取更好的策略。如何用蒙特卡洛方法来填 Q 表格是这个算法的核心。

为了确保蒙特卡洛方法能够有足够的探索，我们使用了 ε -贪心 (ε -greedy) 探索。 ε -贪心是指我们有 $1 - \varepsilon$ 的概率会按照 Q 函数来决定动作，通常 ε 就设一个很小的值， $1 - \varepsilon$ 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 的值会随着时间递减。

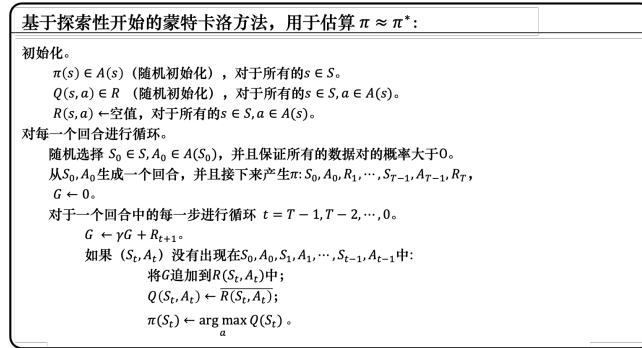


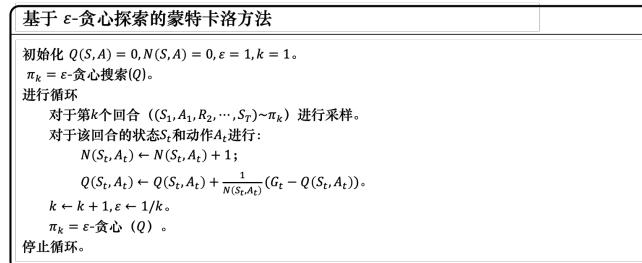
图 3.25 基于探索性开始的蒙特卡洛方法

在最开始的时候，因为我们还不知道哪个动作是比较好的，所以会花比较多的时间探索。接下来随着训练的次数越来越多，我们已经比较确定哪一个动作是比较好的，就会减少探索，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。

当我们使用蒙特卡洛方法和 ε -贪心探索的时候，可以确保价值函数是单调的、改进的。对于任何 ε -贪心策略 π ，关于 Q_π 的 ε -贪心策略 π' 都是一个改进，即 $V_\pi(s) \leq V_{\pi'}(s)$ ，证明过程如下：

$$\begin{aligned}
 Q_\pi(s, \pi'(s)) &= \sum_{a \in A} \pi'(a | s) Q_\pi(s, a) \\
 &= \frac{\varepsilon}{|A|} \sum_{a \in A} Q_\pi(s, a) + (1 - \varepsilon) \max_a Q_\pi(s, a) \\
 &\geq \frac{\varepsilon}{|A|} \sum_{a \in A} Q_\pi(s, a) + (1 - \varepsilon) \sum_{a \in A} \frac{\pi(a | s) - \frac{\varepsilon}{|A|}}{1 - \varepsilon} Q_\pi(s, a) \\
 &= \sum_{a \in A} \pi(a | s) Q_\pi(s, a) = V_\pi(s)
 \end{aligned} \tag{3.22}$$

基于 ε -贪心探索的蒙特卡洛方法如图 3.26 所示。

图 3.26 基于 ε -贪心探索的蒙特卡洛方法

与蒙特卡洛方法相比，时序差分方法有如下几个优势：低方差，能够在线学习，能够从不完整的序列中学习。所以我们可以把时序差分方法也放到控制循环（control loop）里面去估计 Q 表格，再采取 ε -贪心探索改进。这样就可以在回合没结束的时候更新已经采集到的状态价值。

偏差 (bias): 描述的是预测值（估计值）的期望与真实值之间的差距。偏差越高，越偏离真实数据，如图 3.27 第 2 行所示。
方差 (variance): 描述的是预测值的变化范围、离散程度，也就是离其期望值的距离。方差越高，数据的分布越分散，如图 3.27 右列所示。

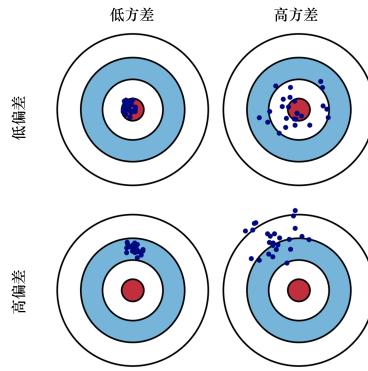


图 3.27 偏差-方差

3.4.1 Sarsa: 同策略时序差分控制

时序差分方法是给定一个策略，然后我们去估计它的价值函数。接着我们要考虑怎么使用时序差分方法的框架来估计 Q 函数，也就是 Sarsa 算法。

Sarsa 所做出的改变很简单，它将原本时序差分方法更新 V 的过程，变成了更新 Q，即

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad (3.23)$$

式 (3.23) 是指我们可以用下一步的 Q 值 $Q(s_{t+1}, a_{t+1})$ 来更新这一步的 Q 值 $Q(s_t, a_t)$ 。Sarsa 直接估计 Q 表格，得到 Q 表格后，就可以更新策略。

为了理解式 (3.23)，如图 3.28 所示，我们先把 $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$ 当作目标值，即 $Q(s_t, a_t)$ 想要逼近的目标值。 $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$ 就是时序差分目标。

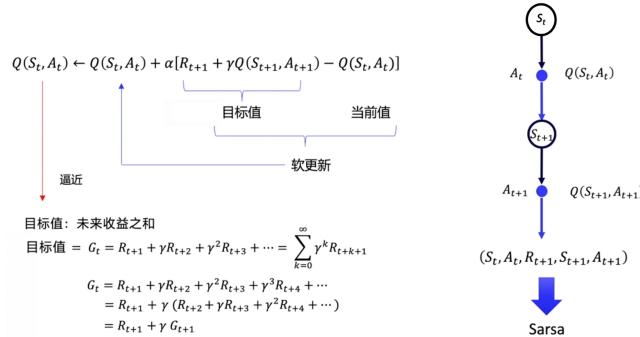


图 3.28 时序差分单步更新

我们想要计算的就是 $Q(s_t, a_t)$ 。因为最开始 Q 值都是随机初始化或者是初始化为 0，所以它需要不断地去逼近它理想中真实的 Q 值（时序差分目标）， $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$ 就是时序差分误差。我们用 $Q(s_t, a_t)$ 来逼近 G_t ，那么 $Q(s_{t+1}, a_{t+1})$ 其实就是近似 G_{t+1} 。我们就可以用 $Q(s_{t+1}, a_{t+1})$ 近似 G_{t+1} ，把 $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$ 当成目标值。 $Q(s_t, a_t)$ 要逼近目标值，我们用软更新的方式来逼近。软更新的方式就是每次我们只更新一点点， α 类似于学习率。最终 Q 值是可以慢慢地逼近真实的目标值的。这样更新公式只需要用到当前时刻的 s_t 、 a_t ，还有获取的 r_{t+1} 、 s_{t+1} 、 a_{t+1} 。

该算法由于每次更新值函数时需要知道当前的状态 (state)、当前的动作 (action)、奖励 (reward)、下一步的状态 (state)、下一步的动作 (action)，即 $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ 这几个值，因此得名 Sarsa 算法。它走了一步之后，获取了 $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ 之后，就可以做一次更新。

如图 3.29 所示，Sarsa 的更新公式可写为

$$Q(S, A) \leftarrow Q(S, A) + \alpha (R + \gamma Q(S', A') - Q(S, A)) \quad (3.24)$$

Sarsa 的更新公式与时序差分方法的公式是类似的。 S' 就是 s_{t+1} 。我们就是用下一步的 Q 值 $Q(S', A')$ 来更新这一步的 Q 值 $Q(S, A)$ ，不断地强化每一个 Q 值。

$$\begin{aligned} n = 1(\text{Sarsa}) \quad Q_t^1 &= r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) \\ n = 2 \quad Q_t^2 &= r_{t+1} + \gamma r_{t+2} + \gamma^2 Q(s_{t+2}, a_{t+2}) \\ &\vdots \\ n = \infty(\text{MC}) \quad Q_t^\infty &= r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{T-t-1} r_T \end{aligned} \quad (3.25)$$

我们考虑 n 步的回报 ($n = 1, 2, \dots, \infty$)，如式 (3.25) 所示。Sarsa 属于单步更新算法，每执行一个动作，就会更新一次价值和策略。如果不进行单步更新，而是采取 n 步更新或者回合更新，即在执行 n 步之后再更新价值和策略，这样我们就得到了 **n 步 Sarsa (n -step Sarsa)**。

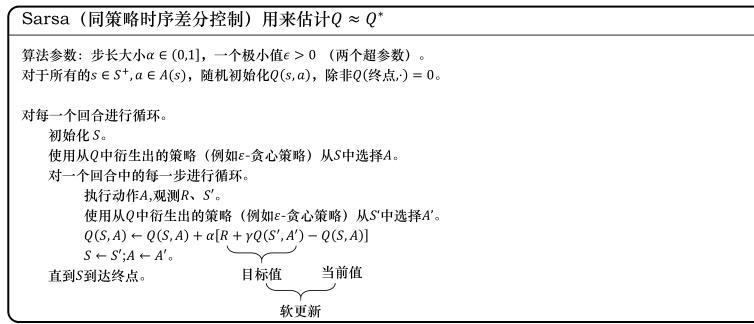


图 3.29 Sarsa 算法

比如 2 步 Sarsa 就是执行两步后再来更新 Q 函数的值。对于 Sarsa，在 t 时刻的价值为

$$Q_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) \quad (3.26)$$

而对于 n 步 Sarsa，它的 n 步 Q 回报为

$$Q_t^n = r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{n-1} r_{t+n} + \gamma^n Q(s_{t+n}, a_{t+n}) \quad (3.27)$$

如果给 Q_t^n 加上资格迹衰减参数 (decay-rate parameter for eligibility traces) λ 并进行求和，即可得到 Sarsa(λ) 的 Q 回报

$$Q_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} Q_t^n \quad (3.28)$$

因此， n 步 Sarsa(λ) 的更新策略为

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha (Q_t^\lambda - Q(s_t, a_t)) \quad (3.29)$$

总之，Sarsa 和 Sarsa(λ) 的差别主要体现在价值的更新上。

了解单步更新的基本公式之后，代码实现就很简单了。如图 3.30 所示，右边是环境，左边是智能体。智能体每与环境交互一次之后，就可以学习一次，向环境输出动作，从环境当中获取状态和奖励。智能体主要实现两个方法：

- (1) 根据 Q 表格选择动作，输出动作；
- (2) 获取 $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ 这几个值更新 Q 表格。

3.4.2 Q 学习：异策略时序差分控制

Sarsa 是一种同策略 (on-policy) 算法，它优化的是它实际执行的策略，它直接用下一步会执行的动作去优化 Q 表格。同策略在学习的过程中，只存在一种策略，它用一种策略去做动作的选取，也用一种

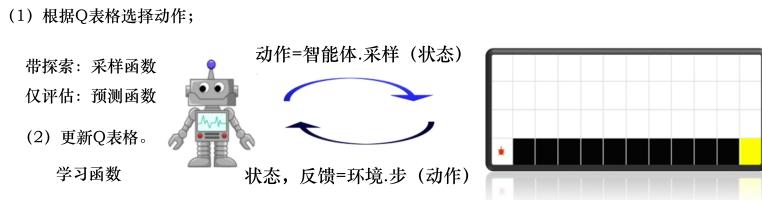


图 3.30 Sarsa 代码实现示意

策略去做优化。所以 Sarsa 知道它下一步的动作有可能会跑到悬崖那边去，它就会在优化自己的策略的时候，尽可能离悬崖远一点。这样子就会保证，它下一步哪怕是有随机动作，它也还是在安全区域内。

Q 学习是一种异策略 (off-policy) 算法。如图 3.31 所示，异策略在学习的过程中，有两种不同的策略：**目标策略 (target policy)** 和 **行为策略 (behavior policy)**。目标策略是我们需要去学习的策略，一般用 π 来表示。目标策略就像是在后方指挥战术的一个军师，它可以根据自己的经验来学习最优的策略，不需要去和环境交互。行为策略是探索环境的策略，一般用 μ 来表示。行为策略可以大胆地去探索到所有可能的轨迹，采集轨迹，采集数据，然后把采集到的数据“喂”给目标策略学习。而且“喂”给目标策略的数据中并不需要 a_{t+1} ，而 Sarsa 是要有 a_{t+1} 的。行为策略像是一个战士，可以在环境里面探索所有的动作、轨迹和经验，然后把这些经验交给目标策略去学习。比如目标策略优化的时候， Q 学习不会管我们下一步去往哪里探索，它只选取奖励最大的策略。



图 3.31 异策略

再例如，如图 3.32 所示，比如环境是波涛汹涌的大海，但学习策略 (learning policy) 太“胆小”了，无法直接与环境交互学习，所以我们有了探索策略 (exploratory policy)，探索策略是一个不畏风浪的海盗，它非常激进，可以在环境中探索。因此探索策略有很多经验，它可以把这些经验“写成稿子”，然后“喂”给学习策略。学习策略可以通过稿子进行学习。



图 3.32 异策略例子

在异策略学习的过程中，轨迹都是行为策略与环境交互产生的，产生这些轨迹后，我们使用这些轨迹来更新目标策略 π 。异策略学习有很多好处。首先，我们可以利用探索策略来学到最佳的策略，学习效率高；其次，异策略学习可以让我们学习其他智能体的动作，进行模仿学习，学习人或者其他智能体产生的轨迹；最后，异策略学习可以让我们重用旧的策略产生的轨迹，探索过程需要很多计算资源，这样可以节省资源。

Q 学习有两种策略：行为策略和目标策略。目标策略 π 直接在 Q 表格上使用贪心策略，取它下一步

能得到的所有状态，即

$$\pi(s_{t+1}) = \arg \max_{a'} Q(s_{t+1}, a') \quad (3.30)$$

行为策略 μ 可以是一个随机的策略，但我们采取 ε -贪心策略，让行为策略不至于是完全随机的，它是基于 Q 表格逐渐改进的。

我们可以构造 Q 学习目标，Q 学习的下一个动作都是通过 $\arg \max$ 操作选出来的，于是我们可得

$$\begin{aligned} r_{t+1} + \gamma Q(s_{t+1}, A') &= r_{t+1} + \gamma Q(s_{t+1}, \arg \max Q(s_{t+1}, a')) \\ &= r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') \end{aligned} \quad (3.31)$$

接着我们可以把 Q 学习更新写成增量学习的形式，时序差分目标变成了 $r_{t+1} + \gamma \max_a Q(s_{t+1}, a)$ ，即

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \quad (3.32)$$

如图 3.33 所示，我们再通过对比的方式来进一步理解 Q 学习。Q 学习是异策略的时序差分学习方法，Sarsa 是同策略的时序差分学习方法。Sarsa 在更新 Q 表格的时候，它用到的是 A' 。我们要获取下一个 Q 值的时候， A' 是下一个步骤一定会执行的动作，这个动作有可能是 ε -贪心方法采样出来的动作，也有可能是最大化 Q 值对应的动作，也有可能是随机动作，但这是它实际执行的动作。但是 Q 学习在更新 Q 表格的时候，它用到的是 Q 值 $Q(S', a)$ 对应的动作，它不一定是下一个步骤会执行的实际的动作，因为我们下一个实际会执行的那个动作可能会探索。Q 学习默认的下一个动作不是通过行为策略来选取的，Q 学习直接看 Q 表格，取它的最大化的值，它是默认 A' 为最佳策略选取的动作，所以 Q 学习在学习的时候，不需要传入 A' ，即 a_{t+1} 的值。

事实上，Q 学习算法被提出的时间更早，Sarsa 算法是 Q 学习算法的改进。^[2]

Sarsa (同策略的时序差分控制) 用来估计 $Q \approx Q^*$

算法参数：步长大 $\alpha \in (0, 1]$ ，一个极小值 $\epsilon > 0$ 。
对于所有的 $s \in S^+, a \in A(s)$ ，随机初始化 $Q(s, a)$ ，除非 $Q(\text{终点}, \cdot) = 0$ 。
对于每一个回合进行循环。
 初始化 S 。
 使用从 Q 中衍生出的策略（例如 ε -贪心策略）从 S 中选择 A 。
 对一个回合中的每一步进行循环。
 执行动作 A ，观测 R, S' 。
 使用从 Q 中衍生出的策略（例如 ε -贪心策略）从 S' 中选择 A' 。
 $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$ 。
 $S \leftarrow S'; A \leftarrow A'$ 。
 直到 S 到达终点。

(a) Sarsa

Q 学习 (异策略的时序差分控制) 用来估计 $\pi \approx \pi^*$

算法参数：步长大 $\alpha \in (0, 1]$ ，一个极小值 $\epsilon > 0$ 。
对于所有的 $s \in S^+, a \in A(s)$ ，随机初始化 $Q(s, a)$ ，除非 $Q(\text{终点}, \cdot) = 0$ 。
对每一个回合进行循环。
 初始化 S 。
 对一个回合中的每一步进行循环。
 使用从 Q 中衍生出的策略（例如 ε -贪心策略）从 S 中选择 A 。
 执行动作 A ，观测 R, S' 。
 $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$ 。
 $S \leftarrow S'$ 。
 直到 S 到达终点。

(b) Q 学习

图 3.33 Sarsa 与 Q 学习的伪代码

Sarsa 和 Q 学习的更新公式是一样的，区别只在目标计算的部分，Sarsa 是 $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$ ，Q 学习是 $r_{t+1} + \gamma \max_a Q(s_{t+1}, a)$ 。

如图 3.34a 所示，Sarsa 用自己的策略产生了 S, A, R, S', A' 这条轨迹，然后用 $Q(s_{t+1}, a_{t+1})$ 去更新原本的 Q 值 $Q(s_t, a_t)$ 。但是 Q 学习并不需要知道我们实际上选择哪一个动作，它默认下一个动作就是 Q 值最大的那个动作。Q 学习知道实际上行为策略可能会有 0.1 的概率选择别的动作，但 Q 学习并不担心受到探索的影响，它默认按照最佳的策略去优化目标策略，所以它可以更大胆地去寻找最优的路径，它表现得比 Sarsa 大胆得多。

如图 3.34b 所示，我们对 Q 学习进行逐步拆解，Q 学习与 Sarsa 唯一不一样的就是并不需要提前知道 A_2 ，就能更新 $Q(S_1, A_1)$ 。在一个回合的训练当中，Q 学习在学习之前也不需要获取下一个动作 A' ，它只需要前面的 (S, A, R, S') ，这与 Sarsa 很不一样。

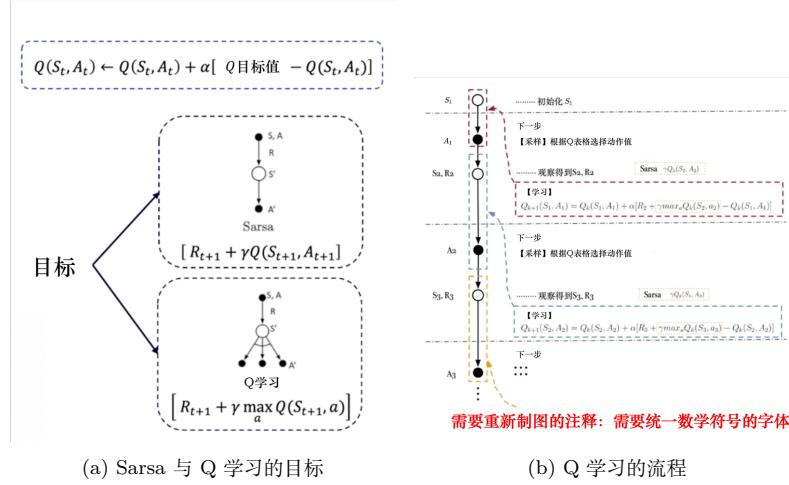


图 3.34 Sarsa 与 Q 学习的区别

3.4.3 同策略与异策略的区别

总结一下同策略和异策略的区别。Sarsa 是一个典型的同策略算法，它只用了一个策略 π ，它不仅使用策略 π 学习，还使用策略 π 与环境交互产生经验。如果策略采用 ε -贪心算法，它需要兼顾探索，为了兼顾探索和利用，它训练的时候会显得有点“胆小”。它在解决悬崖行走问题的时候，会尽可能地远离悬崖边，确保哪怕自己不小心探索了一点儿，也还是在安全区域内。此外，因为采用的是 ε -贪心算法，策略会不断改变（ ε 值会不断变小），所以策略不稳定。Q 学习是一个典型的异策略算法，它有两种策略——目标策略和行为策略，它分离了目标策略与行为策略。Q 学习可以大胆地用行为策略探索得到的经验轨迹来优化目标策略，从而更有可能探索到最佳策略。行为策略可以采用 ε -贪心算法，但目标策略采用的是贪心算法，它直接根据行为策略采集到的数据来采用最佳策略，所以 Q 学习不需要兼顾探索。我们比较一下 Q 学习和 Sarsa 的更新公式，就可以发现 Sarsa 并没有选取最大值的最大化操作。因此，Q 学习是一个非常激进的方法，它希望每一步都获得最大的利益；Sarsa 则相对较为保守，它会选择一条相对安全的迭代路线。

表格型方法总结如图 3.35 所示。

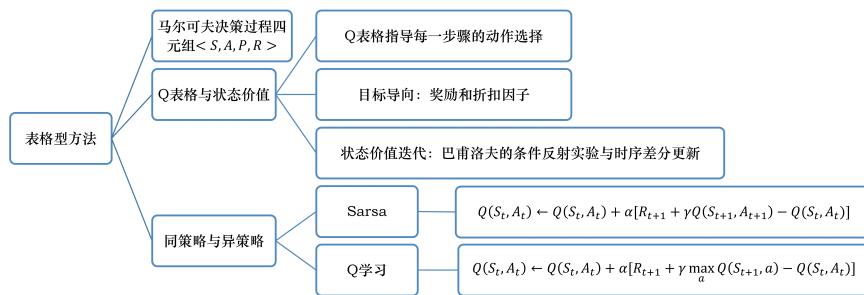


图 3.35 表格型方法总结

3.5 使用 Q 学习解决悬崖寻路问题

强化学习在运动规划方面也有很大的应用前景，目前有很多对应的仿真环境，小到迷宫游戏，大到贴近真实的自动驾驶环境 CARLA。本节使用 OpenAI Gym 开发的 **CliffWalking-v0** 环境，带读者入门 Q 学习算法的代码实战。

3.5.1 CliffWalking-v0 环境简介

我们首先简单介绍 CliffWalking-v0 环境，该环境中文名为悬崖寻路（cliff walking），是一个迷宫类问题。如图 3.36 所示，在一个 4×12 的网格中，智能体以网格的左下角位置为起点，以网格的右下角位置为终点，目标是移动智能体到达终点位置，智能体每次可以在上、下、左、右这 4 个方向中移动一步，每移动一步会得到 -1 单位的奖励。

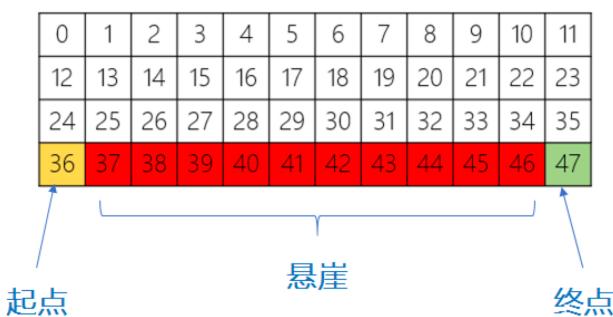


图 3.36 CliffWalking-v0 环境

起终点之间是一段悬崖，即编号为 37~46 的网格，智能体移动过程中会有如下的限制：

- (1) 智能体不能移出网格，如果智能体想执行某个动作移出网格，那么这一步智能体不会移动，但是这个操作依然会得到 $-1W$ 单位的奖励；
- (2) 如果智能体“掉入悬崖”，会立即回到起点位置，并得到 -100 单位的奖励；
- (3) 当智能体移动到终点时，该回合结束，该回合总奖励为各步奖励之和。

我们的目标是以最少的步数到达终点，容易看出最少需要 13 步智能体才能从起点到终点，因此最佳算法收敛的情况下，每回合的总奖励应该是 -13 ，这样人工分析出期望的奖励也便于我们判断算法的收敛情况从而做出相应调整。现在我们可以在代码中定义环境，如下：

```
import gym # 导入gym模块
from envs.gridworld_env import CliffWalkingWapper # 导入自定义装饰器
env = gym.make('CliffWalking-v0') # 定义环境
env = CliffWalkingWapper(env) # 装饰环境
```

我们在程序中使用了一个装饰器重新定义环境，但不影响对环境的理解，感兴趣的同学具体看相关代码。由于 gym 环境封装得比较好，因此我们想要使用这个环境只需要使用 gym.make 命令输入函数名即可，然后我们就可以查看环境的状态和动作数：

```
n_states = env.observation_space.n # 状态数
n_actions = env.action_space.n # 动作数
print(f"状态数: {n_states}, 动作数: {n_actions}")
```

输出结果如下：

```
状态数: 48, 动作数: 4
```

我们的状态数是 48，这里我们设置的是智能体当前所在网格的编号，而动作数是 4，这表示有 0 1 2 3 这四个对应上下左右 4 个动作。另外我们也可以初始化环境并输出当前的状态：

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

结果显示为：

```
初始状态: 36
```

也就是当前智能体的状态即当前所在的网格编号 36，正好对应我们前面讲到的起点。

3.5.2 强化学习基本接口

这里所说的接口是指一般强化学习的训练模式，也是大多数算法伪代码遵循的规则，步骤如下：

- (1) 初始化环境和智能体；
- (2) 对于每个回合，智能体选择动作；
- (3) 环境接收动作并反馈下一个状态和奖励；
- (4) 智能体进行策略更新（学习）；
- (5) 多个回合之后算法收敛保存模型以及做后续的分析、画图等。

代码如下：

```
env = gym.make('CliffWalking-v0') # 定义环境
env = CliffWalkingWapper(env) # 装饰环境
env.seed(1) # 设置随机种子
n_states = env.observation_space.n # 状态数
n_actions = env.action_space.n # 动作数
agent = QLearning(n_states,n_actions,cfg) # cfg存储算法相关参数
for i_ep in range(cfg.train_eps): # cfg.train_eps表示最大的训练回合数
    ep_reward = 0 # 记录每个回合的奖励
    state = env.reset() # 重置环境
    while True:
        action = agent.choose_action(state) # 算法选择一个动作
        next_state, reward, done, _ = env.step(action) # 环境根据动作反馈奖励和下一个状态
        agent.update(state, action, reward, next_state, done) # 算法更新
        state = next_state # 更新状态
        ep_reward += reward
        if done: # 终止状态
            break
```

通常我们会记录并分析奖励的变化，所以在接口基础上加一些变量以记录每回合的奖励。此外，由于强化学习训练过程中得到的奖励可能会产生振荡，因此我们也使用一个滑动平均的量来反映奖励变化的趋势，如下：

```
rewards = []
ma_rewards = [] # 滑动平均奖励
for i_ep in range(cfg.train_eps):
    ep_reward = 0 # 记录每个回合的奖励
    state = env.reset() # 重置环境，重新开始（开始一个新的回合）
    while True:
        action = agent.choose_action(state) # 根据算法选择一个动作
        next_state, reward, done, _ = env.step(action) # 与环境进行一次动作交互
        agent.update(state, action, reward, next_state, done) # Q学习算法更新
        state = next_state # 存储上一个观察值
        ep_reward += reward
    rewards.append(ep_reward)
    ma_rewards.append(np.mean(rewards[-cfg.window_size:]))
```

```

if done:
    break
rewards.append(ep_reward)
if ma_rewards:
    ma_rewards.append(ma_rewards[-1]*0.9+ep_reward*0.1)
else:
    ma_rewards.append(ep_reward)

```

3.5.3 Q 学习算法

了解基本接口之后，现在我们看看 Q 学习算法具体是怎么实现的。前文讲到智能体在整个训练中只做两件事，一是选择动作，一是更新策略，所以我们可以定义一个 Qlearning 类，里面主要包含两个函数，即 choose_action() 和 update()。我们先看看 choose_action() 函数是怎么定义的，如下：

```

def choose_action(self, state):
    self.sample_count += 1
    self.epsilon = self.epsilon_end + (self.epsilon_start - self.epsilon_end) * \
        math.exp(-1. * self.sample_count / self.epsilon_decay) # epsilon是会递减的，这里选择指数递减
    # 带有探索的贪心策略
    if np.random.uniform(0, 1) > self.epsilon:
        action = np.argmax(self.Q_table[ str(state)]]) # 选择Q(s,a)最大值对应的动作
    else:
        action = np.random.choice(self.action_dim) # 随机选择动作
    return action

```

一般我们使用 ϵ -贪心策略选择动作。我们的输入就是当前的状态，随机选取一个值，当这个值大于我们设置的 epsilon 时，我们选取最大 Q 值对应的动作，否则随机选择动作，这样就能在训练中让智能体保持一定的探索率，这也是平衡探索与利用的技巧之一。

下面是我们要实现的策略更新函数：

```

def update(self, state, action, reward, next_state, done):
    Q_predict = self.Q_table[ str(state)][action]
    if done: # 终止状态
        Q_target = reward
    else:
        Q_target = reward + self.gamma * np. max(self.Q_table[ str(next_state)])
    self.Q_table[ str(state)][action] += self.lr * (Q_target - Q_predict)

```

这里面实现的逻辑就是伪代码中的更新公式，如式式 (3.33) 所示。

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \max_a Q(S', a) - Q(S, A) \right) \quad (3.33)$$

注意终止状态下，我们是获取不到下一个动作的，我们直接将 Q_target 更新为对应的奖励即可。

3.5.4 结果分析

到现在我们就基本完成了 Q 学习算法的代码实现，具体可以查看 GitHub 上的源码，代码运行结果如图 3.37 所示。

由于这个环境比较简单，因此可以看到算法很快达到收敛，然后我们再测试训练好的模型，一般测试模型只需要 20~50 左右的回合。

如图 3.38 所示，这里我们测试的回合数为 30，可以看到每个回合智能体都得到了最优的奖励，说明我们算法的训练效果很不错！

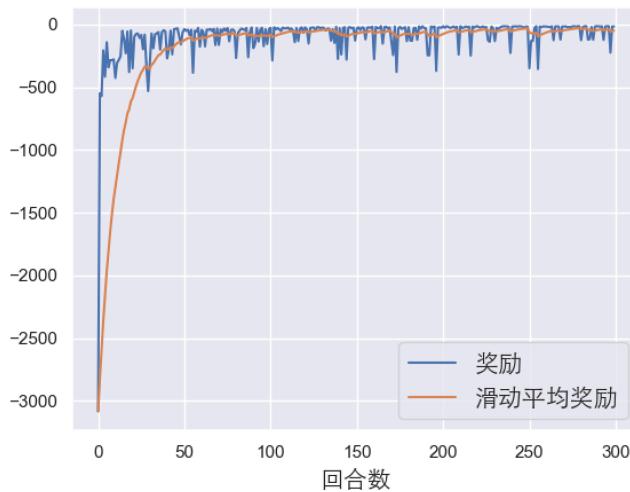


图 3.37 CliffWalking-v0 环境下 Q 学习算法的训练学习曲线

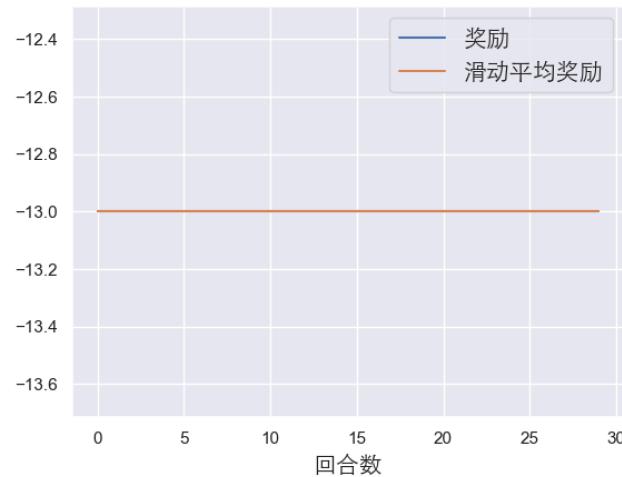


图 3.38 CliffWalking-v0 环境下 Q 学习算法的测试学习曲线

3.6 关键词

概率函数和奖励函数：概率函数定量地表达状态转移的概率，其可以表现环境的随机性。但是实际上，我们经常处于一个未知的环境中，即概率函数和奖励函数是未知的。

Q 表格：其表示形式是表格，其中表格的横轴为动作（智能体的动作），纵轴为环境的状态，每一个坐标点对应某时刻智能体和环境的状态，并通过对应的奖励反馈选择被执行的动作。一般情况下，Q 表格是一个已经训练好的表格，不过我们也可以每执行一步，就对 Q 表格进行更新，然后用下一个状态的 Q 值来更新当前状态的 Q 值（即时序差分方法）。

时序差分（temporal difference, TD）方法：一种 Q 函数（Q 值）的更新方式，流程是使用下一步的 Q 值 $Q(s_{t+1}, a_{t+1})$ 来更新当前步的 Q 值 $Q(s_t, a_t)$ 。完整的计算公式如下： $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$ 。

Sarsa 算法：一种更新前一时刻状态的单步更新的强化学习算法，也是一种同策略学习算法。该算法由于每次更新 Q 函数时需要知道前一步的状态、动作、奖励以及当前时刻的状态、将要执行的动作，即 $s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}$ 这几个值，因此被称为 Sarsa 算法。智能体每进行一次循环，都会用 $s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}$ 对前一步的 Q 值（函数）进行一次更新。

3.7 习题

3-1 构成强化学习的马尔可夫决策过程的四元组有哪些变量？

3-2 请通俗地描述强化学习的“学习”流程。

3-3 请描述基于 Sarsa 算法的智能体的学习过程。

3-4 Q 学习算法和 Sarsa 算法的区别是什么？

3-5 同策略和异策略的区别是什么？

3.8 面试题

3-1 友善的面试官：同学，你能否简述同策略和异策略的区别呢？

3-2 友善的面试官：能否细致地讲一下 Q 学习算法，最好可以写出其 $Q(s, a)$ 的更新公式。另外，它是同策略还是异策略，原因是什么呢？

3-3 友善的面试官：好的，看来你对于 Q 学习算法很了解，那么能否讲一下与 Q 学习算法类似的 Sarsa 算法呢，最好也可以写出其对应的 $Q(s, a)$ 更新公式。另外，它是同策略还是异策略，为什么？

3-4 友善的面试官：请问基于价值的方法和基于策略的方法的区别是什么？

3-5 友善的面试官：请简述一下时序差分方法。

3-6 友善的面试官：请问蒙特卡洛方法和时序差分方法是无偏估计吗？另外谁的方差更大呢？为什么？

3-7 友善的面试官：能否简单说一下动态规划方法、蒙特卡洛方法和时序差分方法的异同点？

参考文献

[1] 诸葛越, 江云胜, 葫芦娃. 百面深度学习[M]. 北京: 人民邮电出版社, 2020.

[2] 邱锡鹏. 神经网络与深度学习[M]. 北京: 机械工业出版社, 2020.

第 4 章 策略梯度

4.1 策略梯度算法

如图 4.1 所示，强化学习有 3 个组成部分：演员（actor）、环境和奖励函数。智能体玩视频游戏时，演员负责操控游戏的摇杆，比如向左、向右、开火等操作；环境就是游戏的主机，负责控制游戏的画面、负责控制怪兽的移动等；奖励函数就是当我们做什么事情、发生什么状况的时候，可以得到多少分数，比如打败一只怪兽得到 20 分等。同样的概念用在围棋上也是一样的，演员就是 Alpha Go，它要决定棋子落在哪一个位置；环境就是对手；奖励函数就是围棋的规则，赢就是得一分，输就是负一分。在强化学习里，环境与奖励函数不是我们可以控制的，它们是在开始学习之前给定的。我们唯一需要做的就是调整演员里面的策略，使得演员可以得到最大的奖励。演员里面的策略决定了演员的动作，即给定一个输入，它会输出演员现在应该要执行的动作。



图 4.1 强化学习的组成部分

策略一般记作 π 。假设我们使用深度学习来做强化学习，策略就是一个网络。网络里面有一些参数，我们用 θ 来代表 π 的参数。网络的输入是智能体看到的东西，如果让智能体玩视频游戏，智能体看到的东西就是游戏的画面。智能体看到的东西会影响我们训练的效果。例如，在玩游戏的时候，也许我们认为游戏的画面是前后相关的，所以应该让策略去看从游戏开始到当前这个时间点之间所有画面的总和。因此我们可能会觉得要用到循环神经网络（recurrent neural network, RNN）来处理它，不过这样会比较难处理。我们可以用向量或矩阵来表示智能体的观测，并将观测输入策略网络，策略网络就会输出智能体要采取的动作。图 4.2 就是具体的例子，策略是一个网络；输入是游戏的画面，它通常是由像素组成的；输出是我们可以执行的动作，有几个动作，输出层就有几个神经元。假设我们现在可以执行的动作有 3 个，输出层就有 3 个神经元，每个神经元对应一个可以采取的动作。输入一个东西后，网络会给每一个可以采取的动作一个分数。我们可以把这个分数当作概率，演员根据概率的分布来决定它要采取的动作，比如 0.7 的概率向左走、0.2 的概率向右走、0.1 的概率开火等。概率分布不同，演员采取的动作就会不一样。

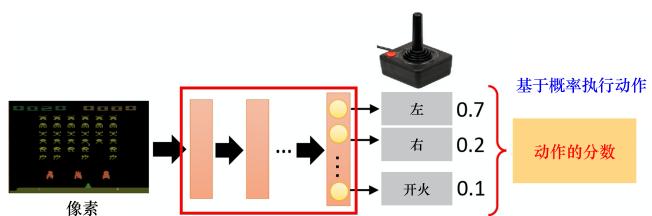


图 4.2 演员的策略

接下来我们用一个例子来说明演员与环境交互的过程。如图 4.3 所示，首先演员会看到一个视频游戏的初始画面，接下来它会根据内部的网络（内部的策略）来决定一个动作。假设演员现在决定的动作是向右，决定完动作以后，它就会得到一个奖励，奖励代表它采取这个动作以后得到的分数。

我们把游戏初始的画面记作 s_1 ，把第一次执行的动作记作 a_1 ，把第一次执行动作以后得到的奖励记作 r_1 。不同的人有不同的记法，有人觉得在 s_1 执行 a_1 得到的奖励应该记为 r_2 ，这两种记法都可以。演员决定一个动作以后，就会看到一个新的游戏画面 s_2 。把 s_2 输入给演员，演员决定要开火，它可能打败了一只怪兽，就得到五分。这个过程反复地持续下去，直到在某一个时间点执行某一个动作，得到奖励之后，环境决定这个游戏结束。例如，如果在这个游戏里面，我们控制宇宙飞船去击杀怪兽，如果宇宙飞船被毁或是把所有的怪兽都清空，游戏就结束了。

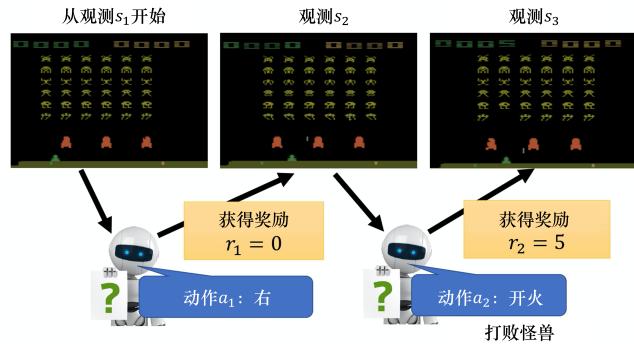


图 4.3 玩视频游戏的例子

如图 4.4 所示，一场游戏称为一个回合。将这场游戏里面得到的所有奖励都加起来，就是**总奖励 (total reward)**，也就是**回报**，我们用 R 来表示它。演员要想办法来最大化它可以得到的奖励。

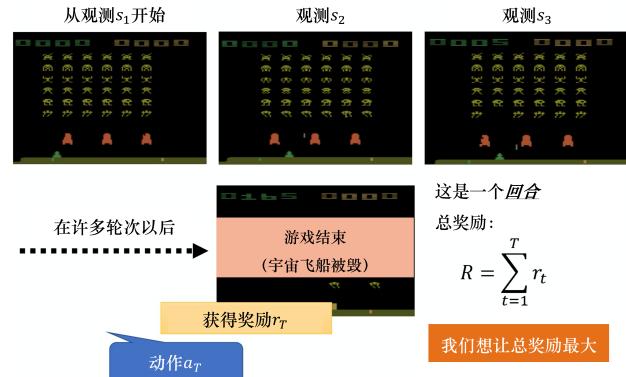


图 4.4 回报的例子

如图 4.5 所示，首先，环境是一个函数，我们可以把游戏的主机看成一个函数，虽然它不一定是神经网络，可能是基于规则的 (rule-based) 模型，但我们可以把它看作一个函数。这个函数一开始先“吐”出一个状态 (游戏画面 s_1)，接下来演员看到游戏画面 s_1 以后，它“吐”出动作 a_1 。环境把动作 a_1 当作它的输入，再“吐”出新的游戏画面 s_2 。演员看到新的游戏画面 s_2 ，再采取新的动作 a_2 。环境看到 a_2 ，再“吐”出 s_3 这个过程会一直持续下去，直到环境觉得应该要停止为止。

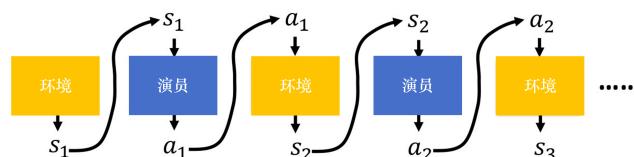


图 4.5 演员和环境

在一场游戏里面，我们把环境输出的 s 与演员输出的动作 a 全部组合起来，就是一个轨迹，即

$$\tau = \{s_1, a_1, s_2, a_2, \dots, s_t, a_t\} \quad (4.1)$$

给定演员的参数 θ ，我们可以计算某个轨迹 τ 发生的概率为

$$\begin{aligned} p_\theta(\tau) &= p(s_1)p_\theta(a_1|s_1)p(s_2|s_1, a_1)p_\theta(a_2|s_2)p(s_3|s_2, a_2)\dots \\ &= p(s_1)\prod_{t=1}^T p_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t) \end{aligned} \quad (4.2)$$

我们先计算环境输出 s_1 的概率 $p(s_1)$ ，再计算根据 s_1 执行 a_1 的概率 $p_\theta(a_1|s_1)$ ， $p_\theta(a_1|s_1)$ 是由策略里面的网络参数 θ 所决定的。策略网络的输出是一个分布，演员根据这个分布进行采样，决定实际要采取的动作。接下来环境根据 a_1 与 s_1 产生 s_2 ，因为 s_2 与 s_1 是有关系的（游戏画面是连续的，下一个游戏画面与上一个游戏画面通常是有关系的），所以给定上一个游戏画面 s_1 和演员采取的动作 a_1 ，就会产生 s_2 。主机在决定输出游戏画面的时候，可能有概率，也可能没有概率，这取决于环境（主机内部设定）。如果主机输出游戏画面的时候没有概率，游戏的每次的画面都一样，我们只要找到一条路径就可以过关了，这样的游戏没有意义。所以输出游戏画面时通常有一定概率，给定同样的前一个画面，我们采取同样的动作，下次产生的画面不一定是一样的。反复执行下去，我们就可以计算一个轨迹 τ 出现的概率有多大。某个轨迹出现的概率取决于环境的动作和智能体的动作。环境的动作是指环境根据其函数内部的参数或内部的规则采取的动作。 $p(s_{t+1}|s_t, a_t)$ 代表的是环境，通常我们无法控制环境，因为环境是设定好的。我们能控制的是 $p_\theta(a_t|s_t)$ 。给定一个 s_t ，演员要采取的 a_t 取决于演员的参数 θ ，所以智能体的动作是演员可以控制的。演员的动作不同，每个同样的轨迹就有不同的出现的概率。

在强化学习里面，除了环境与演员以外，还有奖励函数。如图 4.6 所示，奖励函数根据在某一个状态采取的某一个动作决定这个动作可以得到的分数。对奖励函数输入 s_1, a_1 ，它会输出 r_1 ；输入 s_2, a_2 ，奖励函数会输出 r_2 。我们把轨迹所有的奖励 r 都加起来，就得到了 $R(\tau)$ ，其代表某一个轨迹 τ 的奖励。

在某一场游戏的某一个回合里面，我们会得到 $R(\tau)$ 。我们要做的就是调整演员内部的参数 θ ，使得 $R(\tau)$ 的值越大越好。但实际上 $R(\tau)$ 并不只是一个标量（scalar），它是一个随机变量，因为演员在给定同样的状态下会采取什么样的动作，这是有随机性的。环境在给定同样的观测时要采取什么样的动作，要产生什么样的观测，本身也是有随机性的，所以 $R(\tau)$ 是一个随机变量。我们能够计算的是 $R(\tau)$ 的期望值。给定某一组参数 θ ，我们可计算 r_θ 的期望值为

$$\bar{R}_\theta = \sum_{\tau} R(\tau)p_\theta(\tau) \quad (4.3)$$

我们要穷举所有可能的轨迹 τ ，每一个轨迹 τ 都有一个概率。比如 θ 对应的模型很强，如果有一个回合 θ

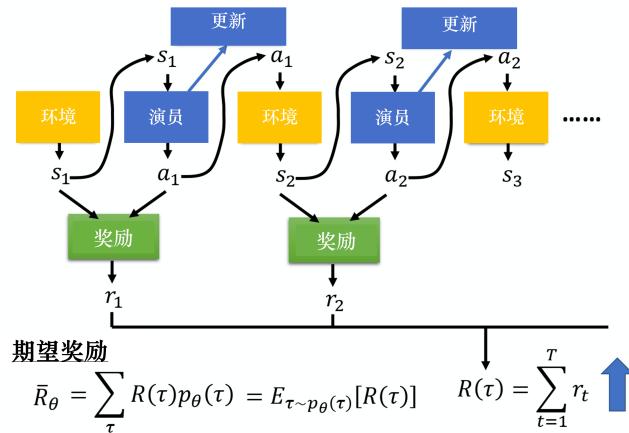


图 4.6 期望的奖励

很快就死掉了，因为这种情况很少会发生，所以该回合对应的轨迹 τ 的概率就很小；如果有一个回合 θ —

直没死，因为这种情况很可能发生，所以该回合对应的轨迹 τ 的概率就很大。我们可以根据 θ 算出某一个轨迹 τ 出现的概率，接下来计算 τ 的总奖励。总奖励使用 τ 出现的概率进行加权，对所有的 τ 进行求和，就是期望值。给定一个参数，我们可以计算期望值为

$$\bar{R}_\theta = \sum_{\tau} R(\tau) p_\theta(\tau) = \mathbb{E}_{\tau \sim p_\theta(\tau)} [R(\tau)] \quad (4.4)$$

从分布 $p_\theta(\tau)$ 采样一个轨迹 τ ，计算 $R(\tau)$ 的期望值，就是期望奖励 (expected reward)。我们要最大化期望奖励。因为我们要让奖励越大越好，所以可以使用梯度上升 (gradient ascent) 来最大化期望奖励。要进行梯度上升，我们先要计算期望奖励 \bar{R}_θ 的梯度。我们对 \bar{R}_θ 做梯度运算

$$\nabla \bar{R}_\theta = \sum_{\tau} R(\tau) \nabla p_\theta(\tau) \quad (4.5)$$

其中，只有 $p_\theta(\tau)$ 与 θ 有关。奖励函数 $R(\tau)$ 不需要是可微分的 (differentiable)，这不影响我们解决接下来的问题。例如，如果在生成对抗网络 (generative adversarial network, GAN) 里面， $R(\tau)$ 是一个判别器 (discriminator)，它就算无法微分，我们还是可以做接下来的运算。

我们可以对 $\nabla p_\theta(\tau)$ 使用式 (4.6)，得到 $\nabla p_\theta(\tau) = p_\theta(\tau) \nabla \log p_\theta(\tau)$ 。

$$\nabla f(x) = f(x) \nabla \log f(x) \quad (4.6)$$

接下来，我们可得

$$\frac{\nabla p_\theta(\tau)}{p_\theta(\tau)} = \nabla \log p_\theta(\tau) \quad (4.7)$$

如式 (4.8) 所示，我们对 τ 进行求和，把 $R(\tau)$ 和 $\log p_\theta(\tau)$ 这两项使用 $p_\theta(\tau)$ 进行加权，既然使用 $p_\theta(\tau)$ 进行加权，它们就可以被写成期望的形式。也就是我们从 $p_\theta(\tau)$ 这个分布里面采样 τ ，去计算 $R(\tau)$ 乘 $\nabla \log p_\theta(\tau)$ ，对所有可能的 τ 进行求和，就是期望的值 (expected value)。

$$\begin{aligned} \nabla \bar{R}_\theta &= \sum_{\tau} R(\tau) \nabla p_\theta(\tau) \\ &= \sum_{\tau} R(\tau) p_\theta(\tau) \frac{\nabla p_\theta(\tau)}{p_\theta(\tau)} \\ &= \sum_{\tau} R(\tau) p_\theta(\tau) \nabla \log p_\theta(\tau) \\ &= \mathbb{E}_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)] \end{aligned} \quad (4.8)$$

实际上期望值 $\mathbb{E}_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)]$ 无法计算，所以我们用采样的方式采样 N 个 τ 并计算每一个的值，把每一个的值加起来，就可以得到梯度，即

$$\begin{aligned} \mathbb{E}_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)] &\approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla \log p_\theta(\tau^n) \\ &= \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n) \end{aligned} \quad (4.9)$$

$\nabla \log p_\theta(\tau)$ 的具体计算过程可写为

$$\begin{aligned} \nabla \log p_\theta(\tau) &= \nabla \left(\log p(s_1) + \sum_{t=1}^T \log p_\theta(a_t | s_t) + \sum_{t=1}^T \log p(s_{t+1} | s_t, a_t) \right) \\ &= \nabla \log p(s_1) + \nabla \sum_{t=1}^T \log p_\theta(a_t | s_t) + \nabla \sum_{t=1}^T \log p(s_{t+1} | s_t, a_t) \\ &= \nabla \sum_{t=1}^T \log p_\theta(a_t | s_t) \\ &= \sum_{t=1}^T \nabla \log p_\theta(a_t | s_t) \end{aligned} \quad (4.10)$$

注意, $p(s_1)$ 和 $p(s_{t+1}|s_t, a_t)$ 来自环境, $p_\theta(a_t|s_t)$ 来自智能体。 $p(s_1)$ 和 $p(s_{t+1}|s_t, a_t)$ 由环境决定, 与 θ 无关, 因此 $\nabla \log p(s_1) = 0$, $\nabla \sum_{t=1}^T \log p(s_{t+1}|s_t, a_t) = 0$ 。

$$\begin{aligned}
\nabla \bar{R}_\theta &= \sum_{\tau} R(\tau) \nabla p_\theta(\tau) \\
&= \sum_{\tau} R(\tau) p_\theta(\tau) \frac{\nabla p_\theta(\tau)}{p_\theta(\tau)} \\
&= \sum_{\tau} R(\tau) p_\theta(\tau) \nabla \log p_\theta(\tau) \\
&= \mathbb{E}_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)] \\
&\approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla \log p_\theta(\tau^n) \\
&= \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n)
\end{aligned} \tag{4.11}$$

我们可以直观地理解式 (4.11), 也就是在我们采样到的数据里面, 采样到在某一个状态 s_t 要执行某一个动作 a_t , (s_t, a_t) 是在整个轨迹 τ 的里面的某一个状态和动作的对。假设我们在 s_t 执行 a_t , 最后发现 τ 的奖励是正的, 我们就要增加在 s_t 执行 a_t 的概率。反之, 如果在 s_t 执行 a_t 会导致 τ 的奖励变成负的, 我们就要减少在 s_t 执行 a_t 的概率。这怎么实现呢? 我们用梯度上升来更新参数, 原来有一个参数 θ , 把 θ 加上梯度 $\nabla \bar{R}_\theta$, 当然我们要有一个学习率 η , 学习率也是要调整的, 可用 Adam、RMSPProp 等方法来调整学习率, 即

$$\theta \leftarrow \theta + \eta \nabla \bar{R}_\theta \tag{4.12}$$

我们可以使用式 (4.13) 来计算梯度。实际上要计算梯度, 如图 4.7 所示, 首先我们要收集很多 s 与 a 的对 (pair), 还要知道这些 s 与 a 在与环境交互的时候, 会得到多少奖励。这些数据怎么收集呢? 我们要用参数为 θ 的智能体与环境交互, 也就是拿已经训练好的智能体先与环境交互, 交互完以后, 就可以得到大量游戏的数据, 我们会记录在第一场游戏里面, 我们在状态 s_1 采取动作 a_1 , 在状态 s_2 采取动作 a_2 。智能体本身是有随机性的, 在同样的状态 s_1 下, 不是每次都会采取动作 a_1 的, 所以我们要记录, 在状态 s_1^1 采取 a_1^1 、在状态 s_2^1 采取 a_2^1 等, 整场比赛结束以后, 得到的奖励是 $R(\tau^1)$ 。我们会采样到另外一些数据, 也就是另外一场游戏。在另外一场游戏里面, 在状态 s_1^2 采取 a_1^2 , 在状态 s_2^2 采取 a_2^2 , 我们采样到的就是 τ^2 , 得到的奖励是 $R(\tau^2)$ 。

这时我们就可以把采样到的数据代入式 (4.13) 里面, 把梯度算出来。也就是把每一个 s 与 a 的对拿进来, 计算在某一个状态下采取某一个动作的对数概率 (log probability) $\log p_\theta(a_t^n | s_t^n)$ 。对这个概率取梯度, 在梯度前面乘一个权重, 权重就是这场比赛的奖励。我们计算出梯度后, 就可以更新模型。

$$\nabla \bar{R}_\theta = \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n) \tag{4.13}$$

更新完模型以后, 我们要重新采样数据再更新模型。注意, 一般策略梯度 (policy gradient, PG) 采样的数据只会用一次。我们采样这些数据, 然后用这些数据更新参数, 再丢掉这些数据。接着重新采样数据, 才能去更新参数。

接下来我们讲一些实现细节。如图 4.8 所示, 我们可以把强化学习想成一个分类问题, 这个分类问题就是输入一个图像, 输出某个类。在解决分类问题时, 我们要收集一些训练数据, 数据中要有输入与输出的对。在实现的时候, 我们把状态当作分类器的输入, 就像在解决图像分类的问题, 只是现在的类不是图像里面的东西, 而是看到这张图像我们要采取什么样的动作, 每一个动作就是一个类。比如第一个类是向左, 第二个类是向右, 第三个类是开火。

在解决分类问题时, 我们要有输入和正确的输出, 要有训练数据。但在强化学习中, 我们通过采样来获得训练数据。假设在采样的过程中, 在某个状态下, 我们采样到要采取动作 a , 那么就把动作 a 当作标

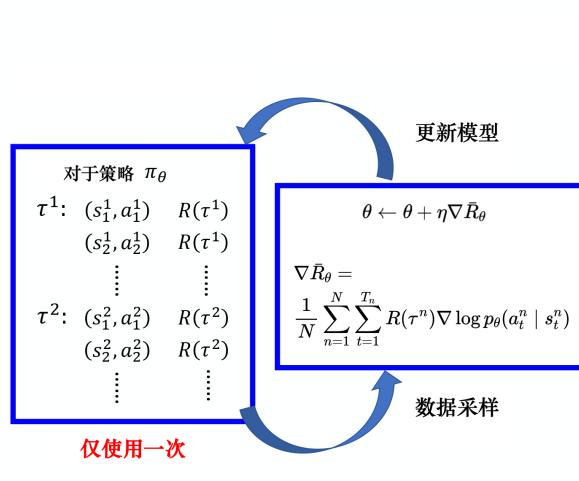


图 4.7 策略梯度

准答案 (ground truth)。比如，我们在某个状态下，采样到要向左。因为是采样，所以向左这个动作不一定概率最高。假设我们采样到向左，在训练的时候，让智能体调整网络的参数，如果看到某个状态，我们就向左。在一般的分类问题里面，我们在实现分类的时候，目标函数都会写成最小化交叉熵 (cross entropy)，最小化交叉熵就是最大化对数似然 (log likelihood)。

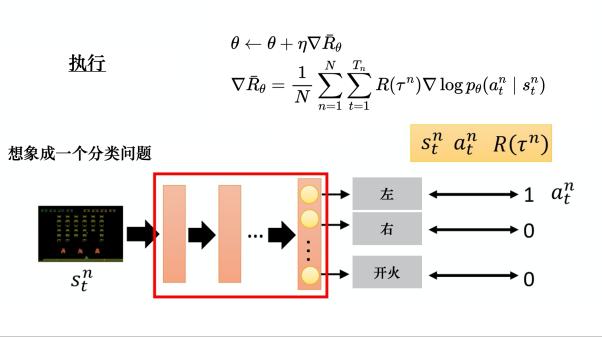


图 4.8 策略梯度实现细节

我们在解决分类问题的时候，目标函数就是最大化或最小化的对象，因为我们现在是最大化似然 (likelihood)，所以其实是最大化，我们要最大化

$$\frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \log p_\theta(a_t^n | s_t^n) \quad (4.14)$$

我们可在 PyTorch 里调用现成的函数来自动计算损失函数，并且把梯度计算出来。这是一般的分类问题，强化学习与分类问题唯一不同的地方是损失前面乘一个权重——整场游戏得到的总奖励 $R(\tau)$ ，而不是在状态 s 采取动作 a 的时候得到的奖励，即

$$\frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \log p_\theta(a_t^n | s_t^n) \quad (4.15)$$

我们要把每一笔训练数据，都使用 $R(\tau)$ 进行加权。如图 4.9 所示，我们使用 PyTorch 或 TensorFlow 之类的深度学习框架计算梯度就结束了，与一般分类问题差不多。

4.2 策略梯度实现技巧

下面我们介绍一些在实现策略梯度时可以使用的技巧。

$$\begin{aligned} \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \log p_\theta(a_t^n | s_t^n) &\xrightarrow{\text{PyTorch / TensorFlow.....}} \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \nabla \log p_\theta(a_t^n | s_t^n) \\ \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \log p_\theta(a_t^n | s_t^n) &\xrightarrow{} \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n) \end{aligned}$$

图 4.9 自动求梯度

4.2.1 技巧 1：添加基线

第一个技巧：添加基线（baseline）。如果给定状态 s 采取动作 a ，整场游戏得到正的奖励，就要增加 (s, a) 的概率。如果给定状态 s 执行动作 a ，整场游戏得到负的奖励，就要减小 (s, a) 的概率。但在很多游戏里面，奖励总是正的，最低都是 0。比如打乒乓球游戏，分为 0 ~ 21 分，所以 $R(\tau)$ 总是正的。假设我们直接使用式 (4.15)，在训练的时候告诉模型，不管是什么动作，都应该要把它概率提升。虽然 $R(\tau)$ 总是正的，但它的值是有大有小的，比如我们在玩乒乓球游戏时，得到的奖励总是正的，但采取某些动作可能得到 0 分，采取某些动作可能得到 20 分。

如图 4.10 所示，假设我们在某一个状态有 3 个动作 a 、 b 、 c 可以执行。根据式 (4.16)，我们要把这 3 个动作的概率，对数概率都提高。但是它们前面的权重 $R(\tau)$ 是不一样的。权重是有大有小的，权重小的，该动作的概率提高的就少；权重大的，该动作的概率提高的就多。因为对数概率是一个概率，所以动作 a 、 b 、 c 的对数概率的和是 0。所以提高少的，在做完归一化（normalize）以后，动作 b 的概率就是下降的；提高多的，该动作的概率才会上升。

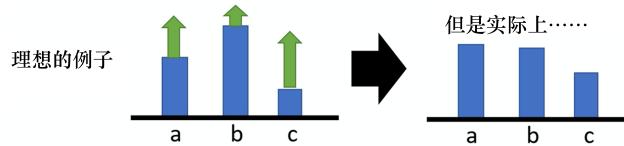


图 4.10 动作的概率的例子

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n) \quad (4.16)$$

这是一个理想的情况，但是实际上，我们是在做采样本来这边应该是一个期望（expectation），对所有可能的 s 与 a 的对进行求和。但我们真正在学习的时候，只是采样了少量的 s 与 a 的对。因为我们做的是采样，所以有一些动作可能从来都没有被采样到。如图 4.11 所示，在某一个状态，虽然可以执行的动作有 a 、 b 、 c ，但我们可能只采样到动作 b 或者只采样到动作 c ，没有采样到动作 a 。但现在所有动作的奖励都是正的，所以根据式 (4.16)，在这个状态采取 a 、 b 、 c 的概率都应该要提高。我们会遇到的问题是，因为 a 没有被采样到，所以其他动作的概率如果都要提高， a 的概率就要下降。所以 a 不一定是一个不好的动作，它只是没有被采样到。但因为 a 没有被采样到，它的概率就会下降，这显然是有问题的。要怎么解决这个问题呢？我们会希望奖励不总是正的。

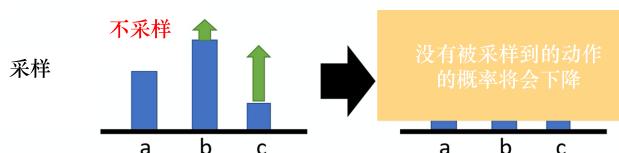


图 4.11 采样动作的问题

为了解决奖励总是正的的问题，我们可以把奖励减 b ，即

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (R(\tau^n) - b) \nabla \log p_\theta(a_t^n | s_t^n) \quad (4.17)$$

其中， b 称为基线。通过这种方法，我们就可以让 $R(\tau) - b$ 这一项有正有负。如果我们得到的总奖励 $R(\tau) > b$ ，就让 (s, a) 的概率上升。如果 $R(\tau) < b$ ，就算 $R(\tau)$ 是正的，值很小也是不好的，我们就让 (s, a) 的概率下降，让这个状态采取这个动作的分数下降。 b 怎么设置呢？我们可以对 τ 的值取期望，计算 τ 的平均值，令 $b \approx E[R(\tau)]$ 。所以在训练的时候，我们会不断地把 $R(\tau)$ 的值记录下来，会不断地计算 $R(\tau)$ 的平均值，把这个平均值当作 b 来使用。这样就可以让我们在训练的时候， $R(\tau) - b$ 是有正有负的，这是第一个技巧。

4.2.2 技巧 2：分配合适的分数

第二个技巧：给每一个动作分配合适的分数（credit）。如式 (4.18) 所示，只要在同一个回合里面，在同一场游戏里面，所有的状态-动作对就使用同样的奖励项进行加权。

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (R(\tau^n) - b) \nabla \log p_\theta(a_t^n | s_t^n) \quad (4.18)$$

这显然是不公平的，因为在同一场游戏里面，也许有些动作是好的，有些动作是不好的。假设整场游戏的结果是好的，但并不代表这场游戏里面每一个动作都是好的。若是整场游戏结果不好，但并不代表游戏里面的每一个动作都是不好的。所以我们希望可以给每一个不同的动作前面都乘上不同的权重。每一个动作的不同权重反映了每一个动作到底是好的还是不好的。例如，如图 4.12a 所示，假设游戏都很短，只有 3 ~ 4 个交互，在 s_a 执行 a_1 得到 5 分，在 s_b 执行 a_2 得到 0 分，在 s_c 执行 a_3 得到 -2 分。整场游戏下来，我们得到 +3 分，那我们得到 +3 分代表在 s_b 执行 a_2 是好的吗？这并不一定代表在 s_b 执行 a_2 是好的。因为这个正的分数，主要来自在 s_a 执行了 a_1 ，与在 s_b 执行 a_2 是没有关系的，也许在 s_b 执行 a_2 反而是不好的，因为它导致我们接下来会进入 s_c ，执行 a_3 被扣分。所以整场游戏得到的结果是好的，并不代表每一个动作都是好的。

如果按照我们刚才的说法，整场游戏得到的分数是 +3 分，因此在训练的时候，每一个状态-动作对都会被乘上 +3。在理想的状况下，如果我们的采样数据够多，就可以解决这个问题。因为假设我们的采样数据够多， (s_b, a_2) 被采样到很多。某一场游戏里，在 s_b 执行 a_2 ，我们会得到 +3 分。但在另外一场游戏里，如图 4.12b 所示，在 s_b 执行 a_2 ，我们却得到了 -7 分，为什么会得到 -7 分呢？因为我们在 s_b 执行 a_2 之前，在 s_a 执行 a_2 得到 -5 分，-5 分也不是在 s_b 执行 a_2 导致的。因为 (s_a, a_2) 先发生，所以 (s_a, a_2) 与 (s_b, a_2) 是没有关系的。在 s_b 执行 a_2 可能造成的问题只有会在接下来得到 -2 分，而与前面的 -5 分没有关系。但是假设我们采样状态-动作对的次数够多，把所有产生这种情况的分数通通都集合起来，这可能不是一个问题。但现在的问题是，我们采样的次数是不够多的。在采样的次数不够多的情况下，我们要给每一个状态-动作对分配合理的分数，要让大家知道它合理的贡献。

一个做法是计算某个状态-动作对的奖励的时候，不把整场游戏得到的奖励全部加起来，只计算从这个动作执行以后得到的奖励。因为这场游戏在执行这个动作之前发生的事情是与执行这个动作是没有关系的，所以在执行这个动作之前得到的奖励都不能算是这个动作的贡献。我们把执行这个动作以后发生的所有奖励加起来，才是这个动作真正的贡献。所以图 4.12a 中，在 s_b 执行 a_2 这件事情，也许它真正会导致我们得到的分数应该是 -2 分而不是 +3 分，因为前面的 +5 分并不是执行 a_2 的功劳。实际上执行 a_2 以后，到游戏结束前，我们只被扣了 2 分，所以分数应该是 -2。同理，图 4.12b 中，执行 a_2 实际上不应该是扣 7 分，因为前面扣 5 分，与在 s_b 执行 a_2 是没有关系的。在 s_b 执行 a_2 ，只会让我们被扣两分而已。

分配合适的分数这一技巧可以表达为

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \left(\sum_{t'=t}^{T_n} r_{t'}^n - b \right) \nabla \log p_\theta(a_t^n | s_t^n) \quad (4.19)$$

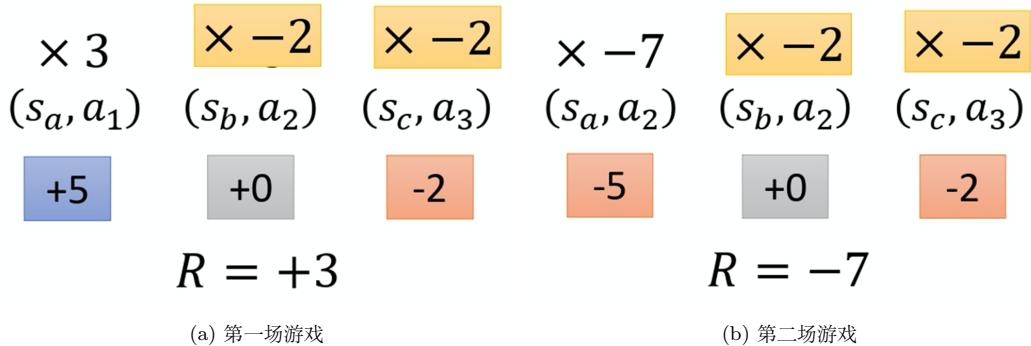


图 4.12 分配合适的分数

原来的权重是整场游戏的奖励的总和，现在改成从某个时刻 t 开始，假设这个动作是在 t 开始执行的，从 t 一直到游戏结束所有奖励的总和才能代表这个动作的好坏。

接下来更进一步，我们把未来的奖励做一个折扣，即

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \left(\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n - b \right) \nabla \log p_\theta(a_t^n | s_t^n) \quad (4.20)$$

为什么要把未来的奖励做一个折扣呢？因为虽然在某一时刻，执行某一个动作，会影响接下来所有的结果（有可能在某一时刻执行的动作，接下来得到的奖励都是这个动作的功劳），但在一般的情况下，时间拖得越长，该动作的影响力就越小。比如在第 2 个时刻执行某一个动作，那在第 3 个时刻得到的奖励可能是在第 2 个时刻执行某个动作的功劳，但是在第 100 个时刻之后又得到奖励，那可能就不是在第 2 个时刻执行某一个动作的功劳。实际上，我们会在 R 前面乘一个折扣因子 γ ($\gamma \in [0, 1]$ ，一般会设为 0.9 或 0.99)，如果 $\gamma = 0$ ，这表示我们只关心即时奖励；如果 $\gamma = 1$ ，这表示未来奖励等同于即时奖励。时刻 t' 越大，它前面就多次乘 γ ，就代表现在在某一个状态 s_t ，执行某一个动作 a_t 的时候，它真正的分数是执行这个动作之后所有奖励的总和，而且还要乘 γ 。例如，假设游戏有两个回合，我们在游戏的第二回合的某一个 s_t 执行 a_t 得到 +1 分，在 s_{t+1} 执行 a_{t+1} 得到 +3 分，在 s_{t+2} 执行 a_{t+2} 得到 -5 分，第二回合结束。 a_t 的分数应该是

$$1 + \gamma \times 3 + \gamma^2 \times (-5)$$

实际上就是这么实现的。 b 可以是依赖状态 (state-dependent) 的，事实上 b 通常是一个网络估计出来的，它是一个网络的输出。我们把 $R - b$ 这一项称为**优势函数 (advantage function)**，用 $A^\theta(s_t, a_t)$ 来代表优势函数。优势函数取决于 s 和 a ，我们就是要计算在某个状态 s 采取某个动作 a 的时候，优势函数的值。在计算优势函数值时，我们要计算 $\sum_{t'=t}^{T_n} r_{t'}^n$ ，需要有一个模型与环境交互，才能知道接下来得到的奖励。优势函数 $A^\theta(s_t, a_t)$ 的上标是 θ ， θ 代表用模型 θ 与环境交互。从时刻 t 开始到游戏结束为止，所有 r 的加和减去 b ，这就是优势函数。优势函数的意义是，假设我们在某一个状态 s_t 执行某一个动作 a_t ，相较于其他可能的动作， a_t 有多好。优势函数在意的不是绝对的好，而是相对的好，即**相对优势 (relative advantage)**。因为在优势函数中，我们会减去一个基线 b ，所以这个动作是相对的好，不是绝对的好。 $A^\theta(s_t, a_t)$ 通常可以由一个网络估计出来，这个网络称为评论员 (critic)。

4.3 REINFORCE: 蒙特卡洛策略梯度

如图 4.13 所示，蒙特卡洛方法可以理解为算法完成一个回合之后，再利用这个回合的数据去学习，做一次更新。因为我们已经获得了整个回合的数据，所以也能够获得每一个步骤的奖励，我们可以很方便地计算每个步骤的未来总奖励，即回报 G_t 。 G_t 是未来总奖励，代表从这个步骤开始，我们能获得的奖励之和。 G_1 代表我们从第一步开始，往后能够获得的总奖励。 G_2 代表从第二步开始，往后能够获得的总奖励。

相比蒙特卡洛方法一个回合更新一次，时序差分方法是每个步骤更新一次，即每走一步，更新一次，时序差分方法的更新频率更高。时序差分方法使用 Q 函数来近似地表示未来总奖励 G_t 。

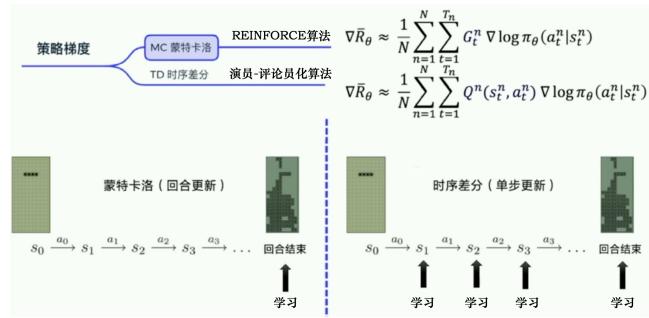


图 4.13 蒙特卡洛方法与时序差分方法

我们介绍一下策略梯度中最简单的也是最经典的一个算法 **REINFORCE**。REINFORCE 用的是回合更新的方式，它在代码上的处理上是先获取每个步骤的奖励，然后计算每个步骤的未来总奖励 G_t ，将每个 G_t 代入

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} G_t^n \nabla \log \pi_\theta(a_t^n | s_t^n) \quad (4.21)$$

优化每一个动作的输出。所以我们在编写代码时会设计一个函数，这个函数的输入是每个步骤获取的奖励，输出是每一个步骤的未来总奖励。因为未来总奖励可写为

$$\begin{aligned} G_t &= \sum_{k=t+1}^T \gamma^{k-t-1} r_k \\ &= r_{t+1} + \gamma G_{t+1} \end{aligned} \quad (4.22)$$

即上一个步骤和下一个步骤的未来总奖励的关系如式 (4.22) 所示，所以在代码的计算上，我们是从后往前推，一步一步地往前推，先算 G_T ，然后往前推，一直算到 G_1 。

如图 4.14 所示，REINFORCE 的伪代码主要看最后 4 行，先产生一个回合的数据，比如

$$(s_1, a_1, G_1), (s_2, a_2, G_2), \dots, (s_T, a_T, G_T)$$

然后针对每个动作计算梯度 $\nabla \ln \pi(a_t | s_t, \theta)$ 。在代码上计算时，我们要获取神经网络的输出。神经网络会输出每个动作对应的概率值（比如 0.2、0.5、0.3），然后我们还可以获取实际的动作 a_t ，把动作转成独热 (one-hot) 向量（比如 [0,1,0]）与 $\log[0.2, 0.5, 0.3]$ 相乘就可以得到 $\ln \pi(a_t | s_t, \theta)$ 。

独热编码 (one-hot encoding) 通常用于处理类别间不具有大小关系的特征。例如血型，一共有 4 个取值 (A 型、B 型、AB 型、O 型)，独热编码会把血型变成一个 4 维稀疏向量，A 型血表示为 (1,0,0,0)，B 型血表示为 (0,1,0,0)，AB 型血表示为 (0,0,1,0)，O 型血表示为 (0,0,0,1)^[1]。

如图 4.15 所示，手写数字识别是一个经典的多分类问题，输入是一张手写数字的图片，经过神经网络处理后，输出的是各个类别的概率。我们希望输出的概率分布尽可能地贴近真实值的概率分布。因为真实值只有一个数字 9，所以如果我们用独热向量的形式给它编码，也可以把真实值理解为一个概率分布，9 的概率就是 1，其他数字的概率就是 0。神经网络的输出一开始可能会比较平均，通过不断地迭代、训练优化之后，我们会希望输出 9 的概率可以远高于输出其他数字的概率。

如图 4.16 所示，我们所做的就是提高输出 9 的概率，降低输出其他数字的概率，让神经网络输出的概率分布能够更贴近真实值的概率分布。我们可以用交叉熵来表示两个概率分布之间的差距。

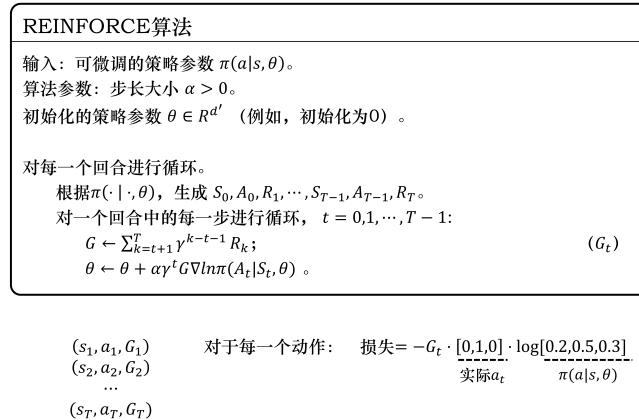


图 4.14 REINFORCE 算法

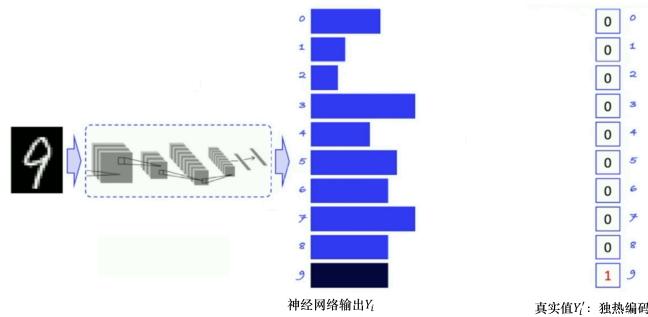


图 4.15 监督学习例子：手写数字识别

我们看一下监督学习的优化流程，即怎么让输出逼近真实值。如图 4.17 所示，监督学习的优化流程就是将图片作为输入传给神经网络，神经网络会判断图片中的数字属于哪一类数字，输出所有数字可能的概率，再计算交叉熵，即神经网络的输出 Y_i 和真实的标签值 Y'_i 之间的距离 $-\sum Y'_i \cdot \log(Y_i)$ 。我们希望尽可能地缩小这两个概率分布之间的差距，计算出的交叉熵可以作为损失函数传给神经网络里面的优化器进行优化，以自动进行神经网络的参数更新。

类似地，如图 4.18 所示，策略梯度预测每一个状态下应该要输出的动作的概率，即输入状态 s_t ，输出动作 a_t 的概率，比如 0.02、0.08、0.9。实际上输出给环境的动作是随机选择一个动作，比如我们选择向右这个动作，它的独热向量就是 $(0, 0, 1)$ 。我们把神经网络的输出和实际动作代入交叉熵的公式就可以求出输出动作的概率和实际动作的概率之间的差距。但实际的动作 a_t 只是我们输出的真实的动作，它不一定是正确的动作，它不能像手写数字识别一样作为一个正确的标签来指导神经网络朝着正确的方向更新，所以我们需要乘一个奖励回报 G_t 。 G_t 相当于对真实动作的评价。如果 G_t 越大，未来总奖励越大，那就说明当前输出的真实的动作就越好，损失就越需要重视。如果 G_t 越小，那就说明动作 a_t 不是很好，损失的

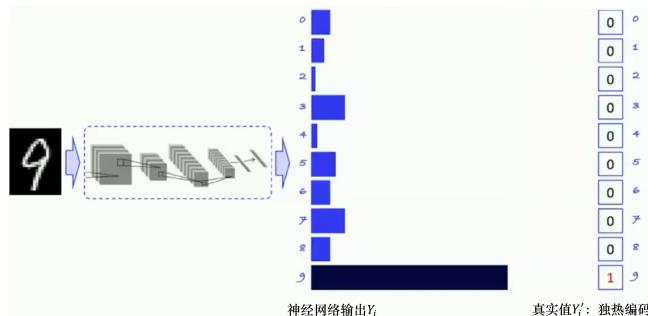


图 4.16 提高数字 9 的概率

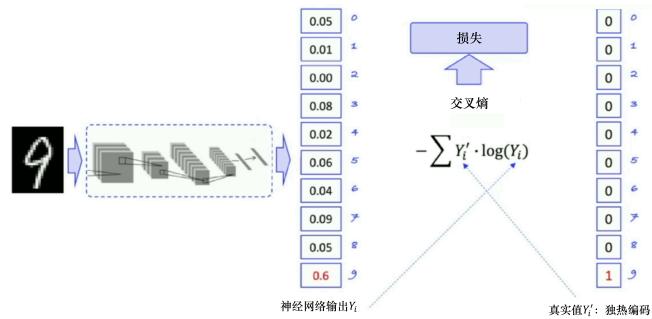


图 4.17 优化流程

权重就要小一点儿，优化力度也要小一点儿。通过与手写数字识别的一个对比，我们就知道为什么策略梯度损失会构造成这样。

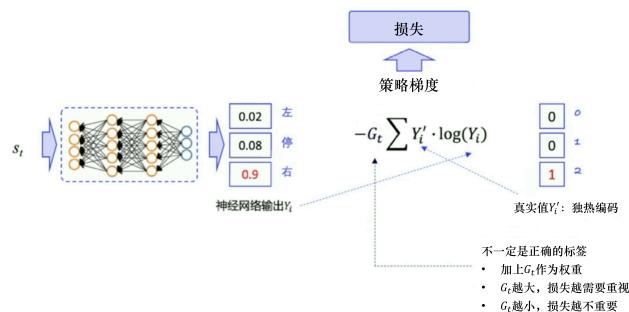


图 4.18 策略梯度损失

如图 4.19 所示，实际上我们在计算策略梯度损失的时候，要先对实际执行的动作取独热向量，再获取神经网络预测的动作概率，将它们相乘，我们就可以得到 $\ln \pi(a_t | s_t, \theta)$ ，这就是我们要构造的损失。因为我们可以获取整个回合的所有轨迹，所以我们可以对这一条轨迹里面的每个动作都去计算一个损失。把所有的损失加起来，我们再将其“扔”给 Adam 的优化器去自动更新参数就好了。

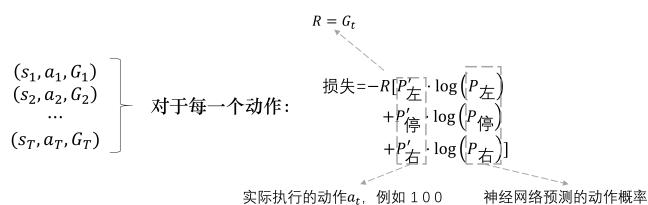


图 4.19 损失计算

图 4.20 所示为 REINFORCE 算法示意，首先我们需要一个策略模型来输出动作概率，输出动作概率后，通过 `sample()` 函数得到一个具体的动作，与环境交互后，我们可以得到整个回合的数据。得到回合数据之后，我们再去执行 `learn()` 函数，在 `learn()` 函数里面，我们就可以用这些数据去构造损失函数，“扔”给优化器优化，更新我们的策略模型。

4.4 关键词

策略 (policy): 在每一个演员中会有对应的策略，这个策略决定了演员的后续动作。具体来说，策略就是对于外界的输入，输出演员现在应该要执行的动作。一般地，我们将策略写成 π 。

回报 (return): 一个回合 (episode) 或者试验 (trial) 得到的所有奖励的总和，也被人们称为总奖励 (total reward)。一般地，我们用 R 来表示它。

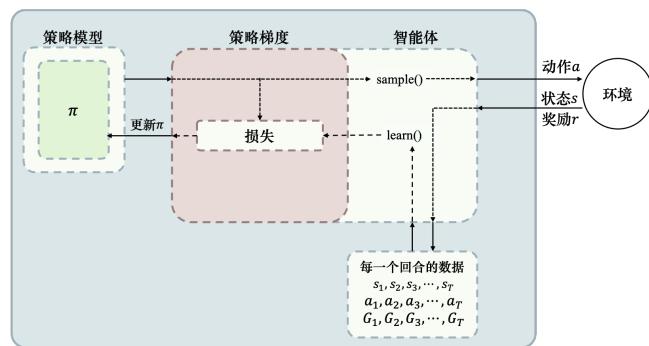


图 4.20 REINFORCE 算法示意

轨迹 (trajectory): 一个试验中我们将环境输出的状态 s 与演员输出的动作 a 全部组合起来形成的集合称为轨迹，即 $\tau = \{s_1, a_1, s_2, a_2, \dots, s_t, a_t\}$ 。

奖励函数 (reward function): 用于反映在某一个状态采取某一个动作可以得到的奖励分数，这是一个函数。即给定一个状态-动作对 (s_1, a_1) ，奖励函数可以输出 r_1 。给定 (s_2, a_2) ，它可以输出 r_2 。把所有的 r 都加起来，我们就得到了 $R(\tau)$ ，它代表某一个轨迹 τ 的奖励。

期望奖励 (expected reward): $\bar{R}_\theta = \sum_\tau R(\tau)p_\theta(\tau) = E_{\tau \sim p_\theta(\tau)}[R(\tau)]$ 。

REINFORCE: 基于策略梯度的强化学习的经典算法，其采用回合更新的模式。

4.5 习题

4-1 如果我们想让机器人自己玩视频游戏，那么强化学习中的 3 个组成部分（演员、环境、奖励函数）具体分别代表什么？

4-2 在一个过程中，一个具体的轨迹 s_1, a_1, s_2, a_2 出现的概率取决于什么？

4-3 当我们最大化期望奖励时，应该使用什么方法？

4-4 我们应该如何理解策略梯度的公式呢？

4-5 我们可以使用哪些方法来进行梯度提升的计算？

4-6 进行基于策略梯度的优化的技巧有哪些？

4-7 对于策略梯度的两种方法，蒙特卡洛强化学习和时序差分强化学习两种方法有什么联系和区别？

4-8 请详细描述 REINFORCE 算法的计算过程。

4.6 面试题

4-1 友善的面试官：同学来吧，给我手动推导一下策略梯度公式的计算过程。

4-2 友善的面试官：可以说一下你所了解的基于策略梯度优化的技巧吗？

参考文献

- [1] 茅葛越, 江云胜, 葫芦娃. 百面深度学习[M]. 北京: 人民邮电出版社, 2020.

第 5 章 近端策略优化

5.1 从同策略到异策略

在介绍近端策略优化 (proximal policy optimization, PPO) 之前，我们先回顾同策略和异策略这两种训练方法的区别。在强化学习里面，要学习的是一个智能体。如果要学习的智能体和与环境交互的智能体是相同的，我们称之为同策略。如果要学习的智能体和与环境交互的智能体不是相同的，我们称之为异策略。

为什么我们会想要考虑异策略？让我们回忆一下策略梯度。策略梯度是同策略的算法，因为在策略梯度中，我们需要一个智能体、一个策略和一个演员。演员去与环境交互搜集数据，搜集很多的轨迹 τ ，根据搜集到的数据按照策略梯度的公式更新策略的参数，所以策略梯度是一个同策略的算法。PPO 是策略梯度的变形，它是现在 OpenAI 默认的强化学习算法。

$$\nabla \bar{R}_\theta = \mathbb{E}_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)] \quad (5.1)$$

问题在于式 (5.1) 的 $\mathbb{E}_{\tau \sim p_\theta(\tau)}$ 是对策略 π_θ 采样的轨迹 τ 求期望。一旦更新了参数，从 θ 变成 θ' ，概率 $p_\theta(\tau)$ 就不对了，之前采样的数据也不能用了。所以策略梯度是一个会花很多时间来采样数据的算法，其大多数时间都在采样数据。智能体与环境交互以后，接下来就要更新参数。我们只能更新参数一次，然后就要重新采样数据，才能再次更新参数。这显然是非常耗时的，所以我们想要从同策略变成异策略，这样就可以用另外一个策略 $\pi_{\theta'}$ 、另外一个演员 θ' 与环境交互 (θ' 被固定了)，用 θ' 采样到的数据去训练 θ 。假设我们可以用 θ' 采样到的数据去训练 θ ，我们可以多次使用 θ' 采样到的数据，可以多次执行梯度上升 (gradient ascent)，可以多次更新参数，都只需要用同一批数据。因为假设 θ 有能力学习另外一个演员 θ' 所采样的数据，所以 θ' 只需采样一次，并采样多一点的数据，让 θ 去更新很多次，这样就会比较有效率。

具体怎么做呢？这就需要介绍重要性采样 (importance sampling) 的概念。

对于一个随机变量，我们通常用概率密度函数来刻画该变量的概率分布特性。具体来说，给定随机变量的一个取值，可以根据概率密度函数来计算该值对应的概率（密度）。反过来，也可以根据概率密度函数提供的概率分布信息来生成随机变量的一个取值，这就是采样。因此，从某种意义上来说，采样是概率密度函数的逆向应用。与根据概率密度函数计算样本点对应的概率值不同，采样过程往往没有那么直接，通常需要根据待采样分布的具体特点来选择合适的采样策略。^[1]

假设我们有一个函数 $f(x)$ ，要计算从分布 p 采样 x ，再把 x 代入 f ，得到 $f(x)$ 。我们该怎么计算 $f(x)$ 的期望值呢？假设我们不能对分布 p 做积分，但可以从分布 p 采样一些数据 x^i 。把 x^i 代入 $f(x)$ ，取它的平均值，就可以近似 $f(x)$ 的期望值。

现在有另外一个问题，假设我们不能从分布 p 采样数据，只能从另外一个分布 q 采样数据 x ， q 可以是任何分布。如果我们从 q 采样 x^i ，就不能使用式 (5.2)。因为式 (5.2) 是假设 x 都是从 p 采样出来的。

$$\mathbb{E}_{x \sim p}[f(x)] \approx \frac{1}{N} \sum_{i=1}^N f(x^i) \quad (5.2)$$

所以我们做一个修正，期望值 $\mathbb{E}_{x \sim p}[f(x)]$ 就是 $\int f(x)p(x)dx$ ，我们对其做如下的变换：

$$\int f(x)p(x)dx = \int f(x) \frac{p(x)}{q(x)} q(x)dx = \mathbb{E}_{x \sim q}[f(x) \frac{p(x)}{q(x)}] \quad (5.3)$$

就可得

$$\mathbb{E}_{x \sim p}[f(x)] = \mathbb{E}_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right] \quad (5.4)$$

我们就可以写成对 q 里面所采样出来的 x 取期望值。我们从 q 里面采样 x , 再计算 $f(x)\frac{p(x)}{q(x)}$, 再取期望值。所以就算我们不能从 p 里面采样数据, 但只要能从 q 里面采样数据, 就可以计算从 p 采样 x 代入 f 以后的期望值。

因为是从 q 采样数据, 所以我们从 q 采样出来的每一笔数据, 都需要乘一个**重要性权重 (importance weight)** $\frac{p(x)}{q(x)}$ 来修正这两个分布的差异。 $q(x)$ 可以是任何分布, 唯一的限制就是 $q(x)$ 的概率是 0 的时候, $p(x)$ 的概率不为 0, 不然会没有定义。假设 $q(x)$ 的概率是 0 的时候, $p(x)$ 的概率也都是 0, $p(x)$ 除以 $q(x)$ 是有定义的。所以这个时候我们就可以使用重要性采样, 把从 p 采样换成从 q 采样。

重要性采样有一些问题。虽然我们可以把 p 换成任何的 q 。但是在实现上, p 和 q 的差距不能太大。差距太大, 会有一些问题。比如, 虽然式 (5.4) 成立 (式 (5.4) 左边是 $f(x)$ 的期望值, 它的分布是 p , 式 (5.4) 右边是 $f(x)\frac{p(x)}{q(x)}$ 的期望值, 它的分布是 q), 但如果不是计算期望值, 而是计算方差, $\text{Var}_{x \sim p}[f(x)]$ 和 $\text{Var}_{x \sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$ 是不一样的。两个随机变量的平均值相同, 并不代表它们的方差相同。

我们可以将 $f(x)$ 和 $f(x)\frac{p(x)}{q(x)}$ 代入方差的公式 $\text{Var}[X] = E[X^2] - (E[X])^2$, 可得

$$\text{Var}_{x \sim p}[f(x)] = \mathbb{E}_{x \sim p} [f(x)^2] - (\mathbb{E}_{x \sim p}[f(x)])^2 \quad (5.5)$$

$$\begin{aligned} \text{Var}_{x \sim q}\left[f(x)\frac{p(x)}{q(x)}\right] &= \mathbb{E}_{x \sim q}\left[\left(f(x)\frac{p(x)}{q(x)}\right)^2\right] - \left(\mathbb{E}_{x \sim q}\left[f(x)\frac{p(x)}{q(x)}\right]\right)^2 \\ &= \mathbb{E}_{x \sim p}\left[f(x)^2\frac{p(x)}{q(x)}\right] - (\mathbb{E}_{x \sim p}[f(x)])^2 \end{aligned} \quad (5.6)$$

$\text{Var}_{x \sim p}[f(x)]$ 和 $\text{Var}_{x \sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$ 的差别在于第一项是不同的, $\text{Var}_{x \sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$ 的第一项多乘了 $\frac{p(x)}{q(x)}$, 如果 $\frac{p(x)}{q(x)}$ 差距很大, $f(x)\frac{p(x)}{q(x)}$ 的方差就会很大。所以理论上它们的期望值一样, 也就是, 我们只要对分布 p 采样足够多次, 对分布 q 采样足够多次, 得到的结果会是一样的。但是如果我们采样的次数不够多, 因为它们的方差差距是很大的, 所以我们就有可能得到差别非常大的结果。

例如, 当 $p(x)$ 和 $q(x)$ 差距很大时, 就会有问题。如图 5.1 所示, 假设蓝线是 $p(x)$ 的分布, 绿线是 $q(x)$ 的分布, 红线是 $f(x)$ 。如果我们要计算 $f(x)$ 的期望值, 从分布 $p(x)$ 做采样, 显然 $\mathbb{E}_{x \sim p}[f(x)]$ 是负的。这是因为左边区域 $p(x)$ 的概率很高, 所以采样会到这个区域, 而 $f(x)$ 在这个区域是负的, 所以理论上这一项算出来会是负的。

接下来我们改成从 $q(x)$ 采样, 因为 $q(x)$ 在右边区域的概率比较高, 所以如果我们采样的点不够多, 可能只会采样到右侧。如果我们只采样到右侧, 可能 $\mathbb{E}_{x \sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$ 是正的。我们这边采样到这些点, 去计算它们的 $f(x)\frac{p(x)}{q(x)}$ 都是正的。我们采样到这些点都是正的, 取期望值以后也都是正的, 这是因为采样的次数不够多。假设我们采样次数很少, 只能采样到右边。左边虽然概率很低, 但也有可能被采样到。假设我们好不容易采样到左边的点, 因为左边的点的 $p(x)$ 和 $q(x)$ 是差很多的, 这边 $p(x)$ 很大, $q(x)$ 很小。 $f(x)$ 好不容易终于采样到一个负的, 这个负的就会被乘上一个非常大的权重, 这样就可以平衡刚才那边一直采样到正的值的情况。最终我们算出这一项的期望值, 终究还是负的。但前提是我们要采样足够多次, 这件事情才会发生。但有可能采样次数不够多, $\mathbb{E}_{x \sim p}[f(x)]$ 与 $\mathbb{E}_{x \sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$ 可能就有很大的差距。这就是重要性采样的问题。

现在要做的就是把重要性采样用在异策略的情况下, 把同策略训练的算法改成异策略训练的算法。怎么改呢? 如式 (5.7) 所示, 之前我们用策略 π_θ 与环境交互, 采样出轨迹 τ , 计算 $R(\tau)\nabla \log p_\theta(\tau)$ 。现在我们不用 θ 与环境交互, 假设有另外一个策略 π'_θ , 它就是另外一个演员, 它的工作是做示范 (demonstration)。

$$\nabla \bar{R}_\theta = \mathbb{E}_{\tau \sim p_{\theta'(\tau)}} \left[\frac{p_\theta(\tau)}{p_{\theta'}(\tau)} R(\tau) \nabla \log p_\theta(\tau) \right] \quad (5.7)$$

θ' 的工作是为 θ 做示范。它与环境交互, 告诉 θ 它与环境交互会发生什么事, 借此来训练 θ 。我们要训练的是 θ , θ' 只负责做示范, 负责与环境交互。我们现在的 τ 是从 θ' 采样出来的, 是用 θ' 与环境交互。所以采样出来的 τ 是从 θ' 采样出来的, 这两个分布不一样。但没有关系, 假设我们本来是从 p 采样,

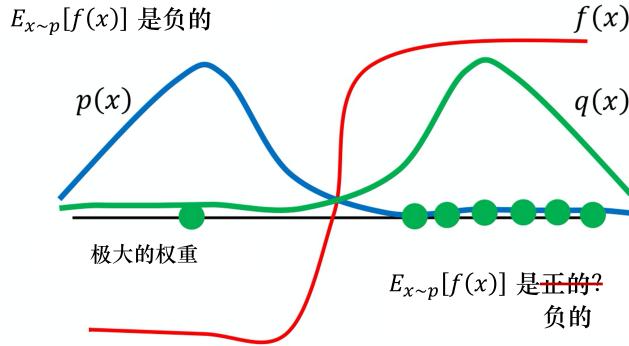


图 5.1 重要性采样的问题

但发现不能从 p 采样，所以我们不用 θ 与环境交互，可以把 p 换成 q ，在后面补上一个重要性权重。同理，我们把 θ 换成 θ' 后，要补上一个重要性权重 $\frac{p_\theta(\tau)}{p_{\theta'}(\tau)}$ 。这个重要性权重就是某一个轨迹 τ 用 θ 算出来的概率除以这个轨迹 τ 用 θ' 算出来的概率。这一项是很重要的，因为我们要学习的是演员 θ ，而 θ 和 θ' 是不太一样的， θ' 见到的情形与 θ 见到的情形可能不是一样的，所以中间要有一个修正的项。

Q：现在的数据是从 θ' 采样出来的，从 θ 换成 θ' 有什么好处呢？

A：因为现在与环境交互的是 θ' 而不是 θ ，所以采样的数据与 θ 本身是没有关系的。因此我们就可以让 θ' 与环境交互采样大量的数据， θ 可以多次更新参数，一直到 θ 训练到一定的程度。更新多次以后， θ' 再重新做采样，这就是同策略换成异策略的妙处。

实际在做策略梯度的时候，我们并不是给整个轨迹 τ 一样的分数，而是将每一个状态-动作对分开计算。实际更新梯度的过程可写为

$$\mathbb{E}_{(s_t, a_t) \sim \pi_\theta} [A^\theta(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n)] \quad (5.8)$$

我们用演员 θ 采样出 s_t 与 a_t ，采样出状态-动作的对，我们会计算这个状态-动作对的优势 (advantage) $A^\theta(s_t, a_t)$ ，就是它有多好。 $A^\theta(s_t, a_t)$ 即用累积奖励减去基线，这一项就是估测出来的。它要估测的是，在状态 s_t 采取动作 a_t 是好的还是不好的。接下来在后面乘 $\nabla \log p_\theta(a_t^n | s_t^n)$ ，也就是如果 $A^\theta(s_t, a_t)$ 是正的，就要增大概率；如果是负的，就要减小概率。

我们可以通过重要性采样把同策略变成异策略，从 θ 变成 θ' 。所以现在 s_t, a_t 是 θ' 与环境交互以后所采样到的数据。但是训练时，要调整的参数是模型 θ 。因为 θ' 与 θ 是不同的模型，所以我们要有一个修正的项。这个修正的项，就是用重要性采样的技术，把 s_t, a_t 用 θ 采样出来的概率除以 s_t, a_t 用 θ' 采样出来的概率。

$$\mathbb{E}_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(s_t, a_t)}{p_{\theta'}(s_t, a_t)} A^\theta(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n) \right] \quad (5.9)$$

其中， $A^\theta(s_t, a_t)$ 有一个上标 θ ， θ 代表 $A^\theta(s_t, a_t)$ 是演员 θ 与环境交互的时候计算出来的。但是实际上从 θ 换到 θ' 的时候， $A^\theta(s_t, a_t)$ 应该改成 $A^{\theta'}(s_t, a_t)$ ，为什么呢？ $A(s_t, a_t)$ 这一项是想要估测在某一个状态采取某一个动作，接下来会得到累积奖励的值减去基线的值。我们怎么估计 $A(s_t, a_t)$ ？我们在状态 s_t 采取动作 a_t ，接下来会得到的奖励的总和，再减去基线就是 $A(s_t, a_t)$ 。之前是 θ 与环境交互，所以我们观察到的是 θ 可以得到的奖励。但现在是 θ' 与环境交互，所以我们得到的这个优势是根据 θ' 所估计出来的优势。但我们现在先不要管那么多，就假设 $A^\theta(s_t, a_t)$ 和 $A^{\theta'}(s_t, a_t)$ 可能是差不多的。

接下来，我们可以拆解 $p_\theta(s_t, a_t)$ 和 $p_{\theta'}(s_t, a_t)$ ，即

$$\begin{aligned} p_\theta(s_t, a_t) &= p_\theta(a_t | s_t) p_\theta(s_t) \\ p_{\theta'}(s_t, a_t) &= p_{\theta'}(a_t | s_t) p_{\theta'}(s_t) \end{aligned} \quad (5.10)$$

于是我们可得

$$\mathbb{E}_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} \frac{p_\theta(s_t)}{p_{\theta'}(s_t)} A^{\theta'}(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n) \right] \quad (5.11)$$

这里需要做的一件事情是，假设模型是 θ 的时候，我们看到 s_t 的概率，与模型是 θ' 的时候，我们看到 s_t 的概率是一样的，即 $p_\theta(s_t) = p_{\theta'}(s_t)$ 。因为 $p_\theta(s_t)$ 和 $p_{\theta'}(s_t)$ 是一样的，所以我们可得

$$\mathbb{E}_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} A^{\theta'}(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n) \right] \quad (5.12)$$

Q：为什么我们可以假设 $p_\theta(s_t)$ 和 $p_{\theta'}(s_t)$ 是一样的？

A：因为我们会看到状态往往与采取的动作是没有太大的关系的。比如我们玩不同的雅达利游戏，其实看到的游戏画面都是差不多的，所以也许不同的 θ 对 s_t 是没有影响的。但更直接的理由就是 $p_\theta(s_t)$ 很难算， $p_\theta(s_t)$ 有一个参数 θ ，它表示的是我们用 θ 去与环境交互，计算 s_t 出现的概率，而这个概率很难算。尤其是如果输入的是图片，同样的 s_t 可能根本就不会出现第二次。我们根本没有办法估计 $p_\theta(s_t)$ ，所以干脆就无视这个问题。

但是 $p_\theta(a_t | s_t)$ 很好算，我们有参数 θ ，它就是一个策略网络。我们输入状态 s_t 到策略网络中，它会输出每一个 a_t 的概率。所以我们只要知道 θ 和 θ' 的参数就可以计算 $\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)}$ 。

式 (5.12) 是梯度，我们可以从梯度反推原来的目标函数：

$$\nabla f(x) = f(x) \nabla \log f(x) \quad (5.13)$$

注意，对 θ 求梯度时， $p_{\theta'}(a_t | s_t)$ 和 $A^{\theta'}(s_t, a_t)$ 都是常数。

所以实际上，当我们使用重要性采样的时候，要去优化的目标函数为

$$J^{\theta'}(\theta) = \mathbb{E}_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} A^{\theta'}(s_t, a_t) \right] \quad (5.14)$$

我们将其记为 $J^{\theta'}(\theta)$ ，因为 $J^{\theta'}(\theta)$ 括号里面的 θ 代表我们要去优化的参数。 θ' 是指我们用 θ' 做示范，就是现在真正在与环境交互的是 θ' 。因为 θ 不与环境交互，是 θ' 在与环境交互。然后我们用 θ' 与环境交互，采样出 s_t, a_t 以后，要去计算 s_t 与 a_t 的优势 $A^{\theta'}(s_t, a_t)$ ，再用它乘 $\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)}$ 。 $\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)}$ 是容易计算的，我们可以从采样的结果来估测 $A^{\theta'}(s_t, a_t)$ ，所以 $J^{\theta'}(\theta)$ 是可以计算的。实际上在更新参数的时候，我们就是按照式 (5.12) 来更新参数的。

5.2 近端策略优化

我们可以通过重要性采样把同策略换成异策略，但重要性采样有一个问题：如果 $p_\theta(a_t | s_t)$ 与 $p_{\theta'}(a_t | s_t)$ 相差太多，即这两个分布相差太多，重要性采样的结果就会不好。怎么避免它们相差太多呢？这就是 PPO 要做的事情。

注意，由于在 PPO 中 θ' 是 θ_{old} ，即行为策略也是 π_θ ，因此 PPO 是同策略的算法。如式 (5.15) 所示，PPO 实际上做的事情就是这样，在异策略的方法里优化目标函数 $J^{\theta'}(\theta)$ 。但是这个目标函数又牵涉到重要性采样。在做重要性采样的时候， $p_\theta(a_t | s_t)$ 不能与 $p_{\theta'}(a_t | s_t)$ 相差太多。做示范的模型不能与真正的模型相差太多，相差太多，重要性采样的结果就会不好。我们在训练的时候，应多加一个约束 (constraint)。这个约束是 θ 与 θ' 输出的动作的 KL 散度 (KL divergence)，这一项用于衡量 θ 与 θ' 的相似程度。我们希望在训练的过程中，学习出的 θ 与 θ' 越相似越好。因为如果 θ 与 θ' 不相似，最后的结果就会不好。所以在 PPO 里面有两项：一项是优化本来要优化的 $J^{\theta'}(\theta)$ ，另一项是一个约束。这个约束就好像正则化 (regularization) 的项 (term) 一样，它所做的就是希望最后学习出的 θ 与 θ' 相差不大。

$$\begin{aligned} J_{\text{PPO}}^{\theta'}(\theta) &= J^{\theta'}(\theta) - \beta \text{KL}(\theta, \theta') \\ J^{\theta'}(\theta) &= \mathbb{E}_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} A^{\theta'}(s_t, a_t) \right] \end{aligned} \quad (5.15)$$

PPO 有一个前身：**信任区域策略优化 (trust region policy optimization, TRPO)**。TRPO 可表示为

$$J_{\text{TRPO}}^{\theta'}(\theta) = \mathbb{E}_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t, a_t) \right], \text{KL}(\theta, \theta') < \delta \quad (5.16)$$

TRPO 与 PPO 不一样的地方是约束所在的位置不一样，PPO 直接把约束放到要优化的式子里面，我们就可以用梯度上升的方法去最大化式 (5.16)。但 TRPO 是把 KL 散度当作约束，它希望 θ 与 θ' 的 KL 散度小于 δ 。如果我们使用的是基于梯度的优化，有约束是很难处理的。TRPO 是很难处理的，因为它把 KL 散度约束当作一个额外的约束，没有放在目标 (objective) 里面，所以它很难计算。因此我们一般就使用 PPO，而不使用 TRPO。PPO 与 TRPO 的性能差不多，但 PPO 在实现上比 TRPO 容易得多。

KL 散度到底指的是什么？这里我们直接把 KL 散度当作一个函数，输入是 θ 与 θ' ，但并不是把 θ 或 θ' 当作一个分布，计算这两个分布之间的距离。所谓的 θ 与 θ' 的距离并不是参数上的距离，而是行为 (behavior) 上的距离。假设我们有两个演员—— θ 和 θ' ，所谓参数上的距离就是计算这两组参数有多相似。这里讲的不是参数上的距离，而是它们行为上的距离。我们先代入一个状态 s ，它会对动作的空间输出一个分布。假设我们有 3 个动作，3 个可能的动作就输出 3 个值。行为距离 (behavior distance) 就是，给定同样的状态，输出动作之间的差距。这两个动作的分布都是概率分布，所以我们可以计算这两个概率分布的 KL 散度。把不同的状态输出的这两个分布的 KL 散度的平均值就是我们所指的两个演员间的 KL 散度。

Q：为什么不直接计算 θ 和 θ' 之间的距离？计算这个距离甚至不用计算 KL 散度，L1 与 L2 的范数 (norm) 也可以保证 θ 与 θ' 很相似。

A：在做强化学习的时候，之所以我们考虑的不是参数上的距离，而是动作上的距离，是因为很有可能对于演员，参数的变化与动作的变化不一定是完全一致的。有时候参数稍微变了，它可能输出动作的就差很多。或者是参数变很多，但输出的动作可能没有什么改变。所以我们真正在意的是演员的动作上的差距，而不是它们参数上的差距。因此在做 PPO 的时候，所谓的 KL 散度并不是参数的距离，而是动作的距离。

5.2.1 近端策略优化惩罚

PPO 算法有两个主要的变种：**近端策略优化惩罚 (PPO-penalty)** 和 **近端策略优化裁剪 (PPO-clip)**。

我们来看一下 **PPO1** 算法，即近端策略优化惩罚算法。它先初始化一个策略的参数 θ^0 。在每一个迭代里面，我们用前一个训练的迭代得到的演员的参数 θ^k 与环境交互，采样到大量状态-动作对。根据 θ^k 交互的结果，我们估测 $A^{\theta^k}(s_t, a_t)$ 。我们使用 PPO 的优化公式。但与原来的策略梯度不一样，原来的策略梯度只能更新一次参数，更新完以后，我们就要重新采样数据。但是现在不同，我们用 θ^k 与环境交互，采样到这组数据以后，我们可以让 θ 更新很多次，想办法最大化目标函数，如式 (5.17) 所示。这里面的 θ 更新很多次也没有关系，因为我们已经有重要性采样，所以这些经验，这些状态-动作对是从 θ^k 采样出来的也没有关系。 θ 可以更新很多次，它与 θ^k 变得不太一样也没有关系，我们可以照样训练 θ 。

$$J_{\text{PPO}}^{\theta^k}(\theta) = J^{\theta^k}(\theta) - \beta \text{KL}(\theta, \theta^k) \quad (5.17)$$

在 PPO 的论文里面还有一个**自适应 KL 散度 (adaptive KL divergence)**。这里会遇到一个问题就，即 β 要设置为多少。这个问题与正则化一样，正则化前面也要乘一个权重，所以 KL 散度前面也要乘一个权重，但 β 要设置为多少呢？我们有一个动态调整 β 的方法。在这个方法里面，我们先设一个可以接受的 KL 散度的最大值。假设优化完式 (5.17) 以后，KL 散度的值太大，这就代表后面惩罚的项 $\beta \text{KL}(\theta, \theta^k)$ 没有发挥作用，我们就把 β 增大。另外，我们设一个 KL 散度的最小值。如果优化完式 (5.17) 以后，KL 散度比最小值还要小，就代表后面这一项的效果太强了，我们怕他只优化后一项，使 θ 与 θ^k 一样，这不是我们想要的，所以我们要减小 β 。 β 是可以动态调整的，因此我们称之为**自适应 KL 惩罚**

(adaptive KL penalty)。我们可以总结一下自适应 KL 惩罚：如果 $\text{KL}(\theta, \theta^k) > \text{KL}_{\max}$, 增大 β ; 如果 $\text{KL}(\theta, \theta^k) < \text{KL}_{\min}$, 减小 β 。

近端策略优化惩罚可表示为

$$\begin{aligned} J_{\text{PPO}}^{\theta^k}(\theta) &= J^{\theta^k}(\theta) - \beta \text{KL}(\theta, \theta^k) \\ J^{\theta^k}(\theta) &\approx \sum_{(s_t, a_t)} \frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)} A^{\theta^k}(s_t, a_t) \end{aligned} \quad (5.18)$$

5.2.2 近端策略优化裁剪

如果我们觉得计算 KL 散度很复杂，那么还有一个 **PPO2** 算法，PPO2 即近端策略优化裁剪。近端策略优化裁剪的目标函数里面没有 KL 散度，其要最大化的目标函数为

$$\begin{aligned} J_{\text{PPO2}}^{\theta^k}(\theta) &\approx \sum_{(s_t, a_t)} \min \left(\frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)} A^{\theta^k}(s_t, a_t), \right. \\ &\quad \left. \text{clip} \left(\frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)}, 1 - \varepsilon, 1 + \varepsilon \right) A^{\theta^k}(s_t, a_t) \right) \end{aligned} \quad (5.19)$$

其中，

- 操作符 (operator) \min 是在第一项与第二项里面选择比较小的项。
- 第二项前面有一个裁剪 (clip) 函数，裁剪函数是指，在括号里面有 3 项，如果第一项小于第二项，那就输出 $1 - \varepsilon$ ；第一项如果大于第三项，那就输出 $1 + \varepsilon$ 。
- ε 是一个超参数，是我们要调整的，可以设置成 0.1 或 0.2。

假设设置 $\varepsilon = 0.2$ ，我们可得

$$\text{clip} \left(\frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)}, 0.8, 1.2 \right) \quad (5.20)$$

如果 $\frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)}$ 算出来小于 0.8，那就输出 0.8；如果算出来大于 1.2，那就输出 1.2。

我们先要理解

$$\text{clip} \left(\frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)}, 1 - \varepsilon, 1 + \varepsilon \right) \quad (5.21)$$

图 5.2 的横轴代表 $\frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)}$ ，纵轴代表裁剪函数的输出。如果 $\frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)}$ 大于 $1 + \varepsilon$ ，输出就是 $1 + \varepsilon$ ；如果小于 $1 - \varepsilon$ ，输出就是 $1 - \varepsilon$ ；如果介于 $1 + \varepsilon \sim 1 - \varepsilon$ ，输出等于输入。

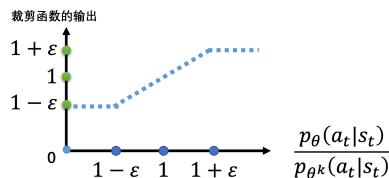
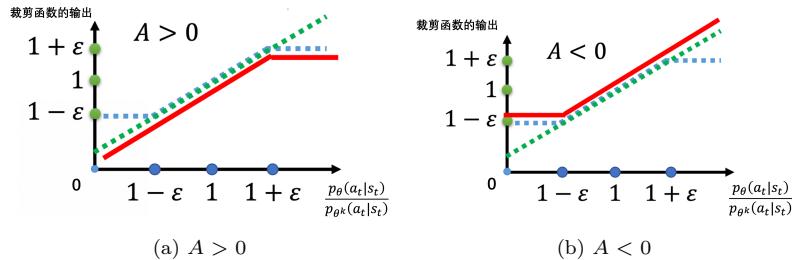


图 5.2 裁剪函数

如图 5.3a 所示， $\frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)}$ 是绿色的线； $\text{clip} \left(\frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)}, 1 - \varepsilon, 1 + \varepsilon \right)$ 是蓝色的线；在绿色的线与蓝色的线中间，我们要取一个最小的结果。假设前面乘上的项 A 大于 0，取最小的结果，就是红色的这条线。如图 5.3b 所示，如果 A 小于 0，取最小结果的以后，就得到红色的这条线。

虽然式 (5.19) 看起来有点儿复杂，但实现起来是比较简单的，因为式 (5.19) 想要做的就是希望 $p_\theta(a_t | s_t)$ 与 $p_{\theta^k}(a_t | s_t)$ 比较接近，也就是做示范的模型与实际上学习的模型在优化以后不要差距太大。

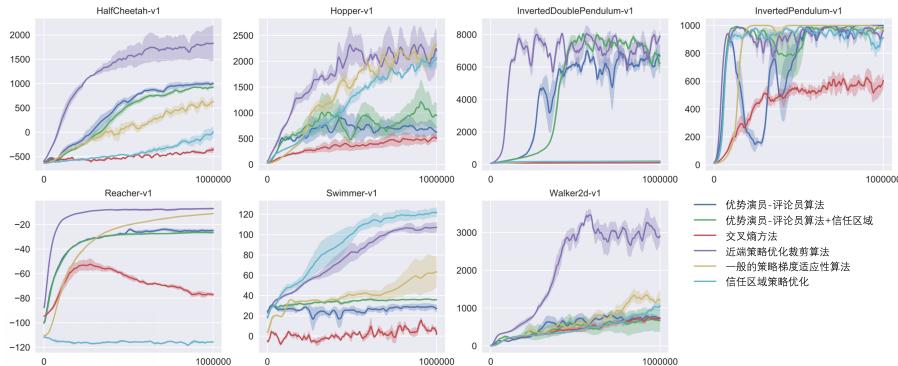
怎么让它做到不要差距太大呢？如果 $A > 0$ ，也就是某一个状态-动作对是好的，我们希望增大这个状态-动作对的概率。也就是，我们想让 $p_\theta(a_t | s_t)$ 越大越好，但它与 $p_{\theta^k}(a_t | s_t)$ 的比值不可以超过 $1 + \varepsilon$ 。如果超过 $1 + \varepsilon$ ，就没有好处了。红色的线就是目标函数，我们希望目标函数值越大越好，我们希望 $p_\theta(a_t | s_t)$ 越大越好。但是 $\frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)}$ 只要大过 $1 + \varepsilon$ ，就没有好处了。所以在训练的时候，当 $p_\theta(a_t | s_t)$ 被训练到

图 5.3 A 对裁剪函数输出的影响

$\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)} > 1 + \varepsilon$ 时，它就会停止。假设 $p_\theta(a_t|s_t)$ 比 $p_{\theta^k}(a_t|s_t)$ 还要小，并且这个优势是正的。因为这个动作是好的，我们希望这个动作被采取的概率越大越好，希望 $p_\theta(a_t|s_t)$ 越大越好。所以假设 $p_\theta(a_t|s_t)$ 还比 $p_{\theta^k}(a_t|s_t)$ 小，那就尽量把它变大，但只要大到 $1 + \varepsilon$ 就好。

如果 $A < 0$ ，也就是某一个状态-动作对是不好的，那么我们希望把 $p_\theta(a_t|s_t)$ 减小。如果 $p_\theta(a_t|s_t)$ 比 $p_{\theta^k}(a_t|s_t)$ 还大，那我们就尽量把它减小，减到 $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}$ 是 $1 - \varepsilon$ 的时候停止，此时不用再减得更小。这样的好处就是，我们不会让 $p_\theta(a_t|s_t)$ 与 $p_{\theta^k}(a_t|s_t)$ 差距太大。要实现这个其实很简单。

图 5.4 所示为 PPO 与其他算法的比较。优势演员-评论员和优势演员-评论员 + 信任区域(trust region)算法是基于演员-评论员的方法。PPO 算法是用紫色线表示，图 5.4 中每张子图表示某一个强化学习的任务，在多数情况下，PPO 都是不错的，即时不是最好的，也是第二好的。

图 5.4 PPO 与其他算法的比较^[2]

5.3 关键词

同策略 (on-policy): 要学习的智能体和与环境交互的智能体是同一个时对应的策略。

异策略 (off-policy): 要学习的智能体和与环境交互的智能体不是同一个时对应的策略。

重要性采样 (important sampling): 使用另外一种分布，来逼近所求分布的一种方法，在强化学习中通常和蒙特卡洛方法结合使用，公式如下：

$$\int f(x)p(x)dx = \int f(x)\frac{p(x)}{q(x)}q(x)dx = E_{x \sim q}[f(x)\frac{p(x)}{q(x)}] = E_{x \sim p}[f(x)]$$

我们在已知 q 的分布后，可以使用上式计算出从 p 这个分布采样 x 代入 f 以后得到的期望值。

近端策略优化 (proximal policy optimization, PPO): 避免在使用重要性采样时由于在 θ 下的 $p_\theta(a_t|s_t)$ 与在 θ' 下的 $p_{\theta'}(a_t|s_t)$ 相差太多，导致重要性采样结果偏差较大而采取的算法。具体来说就是在训练的过程中增加一个限制，这个限制对应 θ 和 θ' 输出的动作的 KL 散度，来衡量 θ 与 θ' 的相似程度。

5.4 习题

5-1 基于同策略的策略梯度有什么可改进之处？或者说其效率较低的原因在于什么？

5-2 使用重要性采样时需要注意的问题有哪些？

5-3 基于异策略的重要性采样中的数据是从 θ' 中采样出来的，从 θ 换成 θ' 有什么优势？

5-4 在本节中近端策略优化中的 KL 散度指的是什么？

5.5 面试题

5-1 友善的面试官：请问什么是重要性采样呀？

5-2 友善的面试官：请问同策略和异策略的区别是什么？

5-3 友善的面试官：请简述一下近端策略优化算法。其与信任区域策略优化算法有何关系呢？

参考文献

- [1] 诸葛越, 葫芦娃. 百面机器学习[M]. 北京: 人民邮电出版社, 2018.
- [2] SCHULMAN J, WOLSKI F, DHARIWAL P, et al. Proximal policy optimization algorithms[J]. arXiv preprint arXiv:1707.06347, 2017.

第 6 章 深度 Q 网络

传统的强化学习算法会使用表格的形式存储状态价值函数 $V(s)$ 或动作价值函数 $Q(s, a)$ ，但是这样的方法存在很大的局限性。例如，现实中的强化学习任务所面临的状态空间往往是连续的，存在无穷多个状态，在这种情况下，就不能再使用表格对价值函数进行存储。价值函数近似利用函数直接拟合状态价值函数或动作价值函数，降低了对存储空间的要求，有效地解决了这个问题。

为了在连续的状态和动作空间中计算值函数 $Q_\pi(s, a)$ ，我们可以用一个函数 $Q_\phi(s, a)$ 来表示近似计算，称为**价值函数近似 (value function approximation)**。

$$Q_\phi(s, a) \approx Q_\pi(s, a) \quad (6.1)$$

其中， s 、 a 分别是状态 s 和动作 a 的向量表示，函数 $Q_\phi(s, a)$ 通常是一个参数为 ϕ 的函数，比如神经网络，其输出为一个实数，称为**Q 网络 (Q-network)**。

深度 Q 网络 (Deep Q-network, DQN) 是指基于深度学习的 Q 学习算法，主要结合了价值函数近似与神经网络技术，并采用目标网络和经历回放的方法进行网络的训练。在 Q 学习中，我们使用表格来存储每个状态 s 下采取动作 a 获得的奖励，即状态-动作值函数 $Q(s, a)$ 。然而，这种方法在状态量巨大甚至是连续的任务中，会遇到维度灾难问题，往往是不可行的。因此，深度 Q 网络采用了价值函数近似的表示方法。

6.1 状态价值函数

深度 Q 网络是基于价值的算法，在基于价值的算法里面，我们学习的不是策略，而是**评论员 (critic)**。评论员的任务是评价现在的动作有多好或有多不好。假设有一个演员，其要学习一个策略来得到尽量高的回报。评论员就是评价演员的策略 π 好还是不好，即策略评估。例如，有一种评论员称为**状态价值函数** V_π 。状态价值函数是指，假设演员的策略是 π ，用 π 与环境交互，假设 π 看到了某一个状态 s ，例如在玩雅达利游戏，状态 s 是某一个画面， π 看到某一个画面，接下来一直到游戏结束，期望的累积奖励有多大。如图 6.1a 所示， V_π 是一个函数，输入一个状态，它会输出一个标量。这个标量代表演员的策略 π 看到状态 s 的时候，预期到游戏结束的时候，它可以获得多大的奖励。例如，假设我们在玩太空侵略者，图 6.1b 所示的状态 s ，这个游戏画面， $V_\pi(s)$ 也许会很大，因为这时还有很多的怪兽可以击杀，所以我们会得到很高的分数。一直到游戏结束的时候，我们仍然有很多的分数可以获得。图 6.1c 所示的情况我们得到的 $V_\pi(s)$ 可能就很小，因为剩下的怪兽也不多，并且红色的防护罩已经消失了，所以我们可能很快就会“死掉”。因此接下来得到预期的奖励，就不会太大。

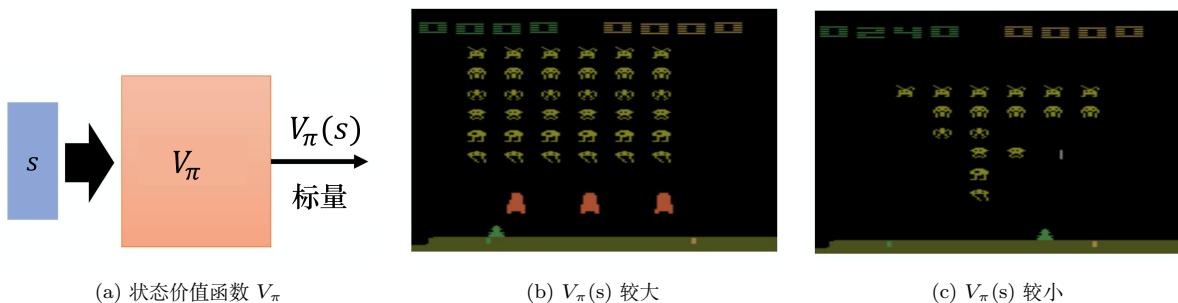


图 6.1 玩太空侵略者

这里需要强调，评论员无法凭空评价一个状态的好坏，它所评价的是在给定某一个状态的时候，如果接下来交互的演员的策略是 π ，我们会得到多少奖励，这个奖励就是我们评价得出的值。因为就算是同样的状态，接下来的 π 不一样，得到的奖励也是不一样的。例如，在左边的情况下，假设是一个正常的 π ，它可以击杀很多怪兽；假设它是一个很弱的 π ，它就站在原地不动，马上就被射死了，我们得到的 $V_\pi(s)$