第五章　不基于模型的控制

前一章内容讲解了个体在不依赖模型的情况下如何进行预测，也就是求解在给定策略下的状态价值或行为价值函数。本章则主要讲解在不基于模型的条件下如何通过个体的学习优化价值函数，同时改善自身行为的策略以最大化获得累积奖励的过程，这一过程也称作不基于模型的控制。

生活中有很多关于优化控制的问题，比如控制一个大厦内的多个电梯使得效率最高；控制直升机的特技飞行，机器人足球世界杯上控制机器人球员，围棋游戏等等。所有的这些问题要么我们对其环境动力学的特点无法掌握，但是我们可以去经历、去尝试构建理解环境的模型；要么虽然问题的环境动力学特征是已知的，但由问题的规模太大以至于计算机根据一般算法无法高效的求解，除非使用采样的办法。无论问题是属于两种情况中的哪一个，强化学习都能较好的解决。

在学习动态规划进行策略评估、优化时，我们能体会到：个体在与环境进行交互时，其实际交互的行为需要基于一个策略产生。在评估一个状态或行为的价值时，也需要基于一个策略，因为不同的策略下同一个状态或状态行为对的价值是不同的。我们把用来指导个体产生与环境进行实际交互行为的策略称为行为策略，把用来评价状态或行为价值的策略或者待优化的策略称为目标策略。如果个体在学习过程中优化的策略与自己的行为策略是同一个策略时，这种学习方式称为**现时策略学习** (on-policy learning)，如果个体在学习过程中优化的策略与自己的行为策略是不同的策略时，这种学习方式称为**借鉴策略学习** (off-policy learning)。

了从已知模型的、基于全宽度采样的动态规划学习转至模型未知的、基于采样的蒙特卡洛或时序差分学习进行控制是朝着高效解决中等规模实际问题的一个突破。基于这些思想产生了一些经典的理论和算法，如不完全贪婪搜索策略、现时蒙特卡洛控制，现时时序差分学习、属于借鉴学习算法的 Q 学习等。下文将详细论述。

5.1 行为价值函数的重要性

在不基于模型的控制时，我们将无法通过分析、比较基于状态的价值来改善贪婪策略，这是因为基于状态价值的贪婪策略的改善需要知晓状态间转移概率：

$$\pi'(s) = \underset{a \in A}{argmax} \left(R_s^a + P_{ss'}^a V(s')\right)$$

这不难理解，拿第二章提到的学生马尔科夫决策过程的例子来说，假如需要在贪婪策略下确定学生处在第三节课时的价值，你需要比较学生在第三节课后所能采取的全部两个行为“学习”和“泡吧”后状态的价值。选择继续“学习”比较简单，在获得一个价值为 10 的即时奖励后进入价值恒为 0 的“退出休息”状态，此时得到在“第三节课”后选择继续“学习”的价值为 +10；而选择“泡吧”时，计算就没那么简单了，因为在“泡吧”过后，学生自己并不确定将回到哪个状态，因此无法直接用某一个状态的价值来计算“泡吧”行为的价值。环境按照一定的概率（分别为 0.2, 0.4, 0.4）把学生重新分配至“第一节课”、“第二节课”或“第三节课”。也只有再知道这三个概率值后，我们才能根据后续这三个状态的价值计算得到“泡吧”行为的价值为 +9.4，根据贪婪策略，学生在“第三节课”的价值为 +10。在基于采样的强化学习时，我们无法事先知道这些状态之间在不同行为下的转移概率，因而无法基于状态价值来改善我们的贪婪策略。

生活中也是如此，有时候一个人给自己制定了一个价值很高的目标，却发现不知采取如何的行为来达到这个目标。与其花时间比较目标与现实的差距，倒不如立足于当下，在所有可用的行为中选择一个最高价值的行为。因此如果能够确定某状态下所有状态行为对的价值，那么自然就比较容易从中选出一个最优价值对应的行为了。实践证明，在不基于模型的强化学习问题中，确定状态行为对的价值要容易很多。

生活中有些人喜欢做事但不善于总结，这类人一般要比那些勤于总结的人进步得慢，从策略迭代的角度看，这类人策略更新迭代的周期较长；有些人在总结经验上过于勤快，甚至在一件事情还没有完全定论时就急于总结并推理过程之间的关系，这种总结得到的经验有可能是错误的。强化学习中的个体也是如此，为了让个体的尽早地找到最优策略，可以适当加快策略迭代的速度，但是从一个不完整的状态序列学习则要注意不能过多地依赖状态序列中相邻状态行为对的关系。由于基于蒙特卡洛的学习利用的是完整的状态序列，为了加快学习速度可以在只经历一个完整状态序列后就进行策略迭代；而在进行基于时序差分的学习时，虽然学习速度可以更快，但要注意减少对事件估计的偏差。

5.2 $\epsilon-$ 贪婪策略

在前文讲解动态规划进行策略迭代时，初始阶段我们选择的是均一随机策略 (uniform random policy)，而进行过一次迭代后，我们选择了贪婪策略 (greedy policy)，即每一次只选择能到达的具有最大价值的状态的行为，在随后的每一次迭代中都使用这个贪婪策略。实验发现，这样能够明显加快找到最优策略的速度。贪婪搜索策略在基于模型的动态规划算法中能收敛至最优策略 (价值)，但这在不基于模型、基于采样的蒙特卡罗或时序差分学习中却通常不能收敛至最优策略。虽然这三种算法都采用通过后续状态价值回溯的办法确定当前状态价值，但动态规划算法是考虑了一个状态的后续所有状态价值的。而后两者则仅能考虑到有限次数的、已经采样经历过的状态，那些事实存在但还没经历过的状态对于后两者算法来说都是未探索的不被考虑的状态，有些状态虽然经历过，但由于经历次数不多对其价值的估计也不一定准确。如果存在一些价值更高的未被探索的状态使用贪婪算法将式中无法探索到这些状态，而已经经历过但价值较低的状态也很难再次被经历，如此将无法得到最优策略。

举个例子：假设你刚搬到一个街区，街上有两家餐馆，你决定去两家都尝试一下并给自己的就餐体验打个分，分值在 0-10 分之间。你先体验了第一家，觉得一般，给了 5 分；过了几天又去了第二家，觉得不错，给了 8 分。此时，如果你选择贪婪策略，每次只去评分高的餐馆就餐，那么下一次你将继续选择去第二家餐馆。假设这次体验差了点，给了 6 分。经过了三次体验后，你对第一家餐馆的评分为 5 分，第二家的评分平均下来是 7 分。之后你仍然选择贪婪策略，下一次体验还去了第二家，假设体验为 7 分，那么经过这 4 次体验之后，你能确认对你来说第二家餐馆就一定比第一家好吗？答案是否定的，原因在于你只尝试了一次去第一家就餐，仅靠这一次的体验是不可靠的。贪婪策略并不意味着你今后就一定无法选择第一家就餐，当你每次依据贪婪算法在第二家餐馆就餐时，如果体验分降低导致平均分低于第一家的评分 5 分，那么下一次你将选择去第一家餐馆。不过如果你对第二家餐馆的平均体验分一直在第一家之上，那么依据贪婪策略将无法再去第一家参观体验了，也许你第一次去第一家餐馆就餐时恰好碰到他们刚开张管理还不完善的情况，而现在已经做得很好了。贪婪策略将使你错失第一家餐馆的许多美味了。采取贪婪策略还有一个问题，就是如果这条街上新开了一家餐馆，如果你对没有去过的餐馆评分为 0 的话，你将永远不会去尝试这家餐馆。

贪婪策略产生问题的根源是无法保证持续的探索，为了解决这个问题，一种不完全的贪婪 (ϵ-greedy 搜索策略被提出，其基本思想就是保证能做到持续的探索，具体通过设置一个较小的 ϵ 值，使用 $1-\epsilon$ 的概率贪婪地选择目前认为是最大行为价值的行为，而用 ϵ 的概率随机的从**所有**

m 个可选行为中选择行为，即：

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{如果 } a^* = \underset{a \in A}{argmax}\ Q(s,a) \\ \epsilon/m & \text{其它情况} \end{cases} \tag{5.1}$$

5.3 现时策略蒙特卡罗控制

现时策略蒙特卡罗控制通过 ϵ-贪婪策略采样一个或多个完整的状态序列后，平均得出某一状态行为对的价值，并持续进行策略的评估和改善。通常可以在仅得到一个完整状态序列后就进行一次策略迭代以加速迭代过程。

使用 ϵ-贪婪策略进行现时蒙特卡罗控制仍然只能得到基于该策略的近似行为价值函数，这是因为该策略一直在进行探索，没有一个终止条件。因此我们必须关注以下两个方面：一方面我们不想丢掉任何更好信息和状态，另一方面随着我们策略的改善我们最终希望能终止于某一个最优策略。为此引入了另一个理论概念：GLIE(greedy in the Limit with Infinite Exploration)。它包含两层意思，一是所有的状态行为对会被无限次探索：

$$\lim_{k \to \infty} N_k(s,a) = \infty$$

二是另外随着采样趋向无穷多，策略收敛至一个贪婪策略：

$$\lim_{k \to \infty} \pi_k(a|s) = 1 \left(a = \underset{a' \in A}{argmax}\ Q_k(s,a') \right)$$

存在如下的定理：GLIE 蒙特卡洛控制能收敛至最优的状态行为价值函数:

$$Q(s,a) \to q^*(s,a)$$

如果在使用 ϵ-贪婪策略时，能令 ϵ 随采样次数的无限增加而趋向于 0 就符合 GLIE。这样基于 GLIE 的蒙特卡洛控制流程如下：

1. 基于给定策略 π，采样第 k 个完整的状态序列：$\{S_1, A_1, R_2, \cdots, S_T\}$

2. 对于该状态序列里出现的每一状态行为对 (S_t, A_t)，更新其计数 N 和行为价值函数 Q：

$$\begin{aligned} N(S_t, A_t) &\leftarrow N(S_t, A_t) + 1 \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t)) \end{aligned} \tag{5.2}$$

3. 基于新的行为价值函数 Q 以如下方式改善策略：

$$\begin{aligned} \epsilon &\leftarrow 1/k \\ \pi &\leftarrow \epsilon - greedy(Q) \end{aligned} \tag{5.3}$$

在实际应用中，ϵ 的取值可不局限于取 $1/k$，只要符合 GLIE 特性的设计均可以收敛至最优策略(价值)。

5.4 现时策略时序差分控制

通过上一章关于预测的学习，我们体会到时序差分 (TD) 学习相比蒙特卡罗 (MC) 学习有很多优点：低变异性，可以在线实时学习，可以学习不完整状态序列等。在控制问题上使用 TD 学习同样具备上述的一些优点。本节的现时策略 TD 学习中，我们将介绍 Sarsa 算法和 Sarsa(λ) 算法，在下一节的借鉴策略 TD 学习中将详细介绍 Q 学习算法。

5.4.1 Sarsa 算法

Sarsa 的名称来源于下图所示的序列描述：针对一个状态 S，个体通过行为策略产生一个行为 A，执行该行为进而产生一个状态行为对 (S,A)，环境收到个体的行为后会告诉个体即时奖励 R 以及后续进入的状态 S'；个体在状态 S' 时遵循当前的行为策略产生一个新行为 A'，个体此时并不执行该行为，而是通过行为价值函数得到后一个状态行为对 (S',A') 的价值，利用这个新的价值和即时奖励 R 来更新前一个状态行为对 (S,A) 的价值。

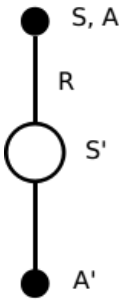


图 5.1: Sarsa 算法示意图

与 MC 算法不同的是，Sarsa 算法在单个状态序列内的每一个时间步，在状态 S 下采取一个行为 A 到达状态 S' 后都要更新状态行为对 (S,A) 的价值 Q(S,A)。这一过程同样使用 ϵ-贪婪策

略进行策略迭代：

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A)) \tag{5.4}$$

Sarsa 的算法流程如算法 1 所述。

算法 1: Sarsa 算法

输入: $episodes, \alpha, \gamma$
输出: Q
initialize: set $Q(s,a)$ arbitrarily, for each s in $\mathbb{S}$ and a in $\mathbb{A}(s)$; set $Q(\text{terminal state}, \cdot) = 0$
repeat for each episode in episodes
  initialize: $S \leftarrow$ first state of episode
  $A = \text{policy}(Q, S)$ (e.g. ϵ-greedy policy)
  repeat for each step of episode
    $R, S' = \text{perform\_action}(S, A)$
    $A' = \text{policy}(Q, S')$ (e.g. ϵ-greedy policy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$
    $S \leftarrow S'; A \leftarrow A'$;
  until S is terminal state;
until all episodes are visited;

在 Sarsa 算法中，Q(S,A) 的值使用一张大表来存储的，这不是很适合解决规模很大的问题；对于每一个状态序列，在 S 状态时采取的行为 A 是基于当前行为策略的，也就是该行为是与环境进行交互实际使用的行为。在更新状态行为对 (S,A) 的价值的循环里，个体状态 S' 下也依据该行为策略产生了一个行为 A', 该行为在当前循环周期里用来得到状态行为对 (S'A') 的价值，并借此来更新状态行为对 (SA) 的价值，在下一个循环周期（时间步）内，状态 S' 和行为 A' 将转换身份为当前状态和当前行为，该行为将被执行。

在更新行为价值时，参数 α 是学习速率参数，γ 是衰减因子。当行为策略满足前文所述的 GLIE 特性同时学习速率参数 α 满足：

$$\sum_{t=1}^{\infty} \alpha_t = \infty, \text{且} \sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

时，Sarsa 算法将收敛至最优策略和最优价值函数。

我们使用一个经典环境有风格子世界来解释 Sarsa 算法的学习过程。如图 5.2 所示一个 10×7 的长方形格子世界，标记有一个起始位置 S 和一个终止目标位置 G，格子下方的数字表示对应的列中一定强度的风。当个体进入该列的某个格子时，会按图中箭头所示的方向自动移动数字表示的格数，借此来模拟世界中风的作用。同样格子世界是有边界的，个体任意时刻只能处在世界内部的一个格子中。个体并不清楚这个世界的构造以及有风，也就是说它不知道格子是长方形

的，也不知道边界在哪里，也不知道自己在里面移动移步后下一个格子与之前格子的相对位置关系，当然它也不清楚起始位置、终止目标的具体位置。但是个体会记住曾经经过的格子，下次在进入这个格子时，它能准确的辨认出这个格子曾经什么时候来过。格子可以执行的行为是朝上、下、左、右移动一步。现在要求解的问题是个体应该遵循怎样的策略才能尽快的从起始位置到达目标位置。

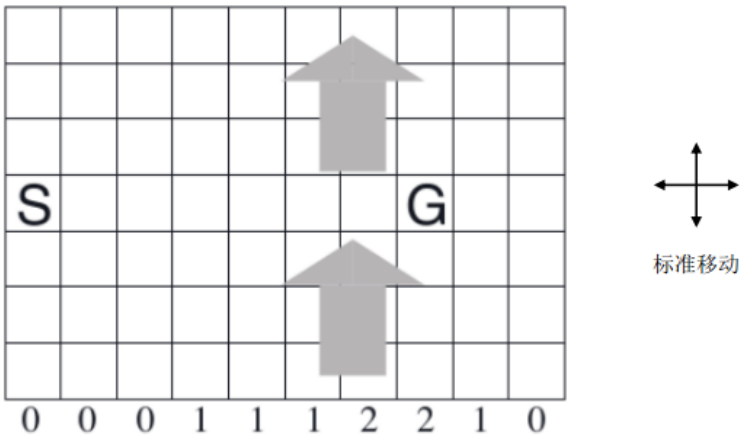


图 5.2: 有风格子世界环境

为了使用计算机程序解决这个问题，我们首先将这个问题用强化学习的语言再描述一遍。这是一个不基于模型的控制问题，也就是要在不掌握马尔科夫决策过程的情况下寻找最优策略。环境世界中每一个格子可以用水平和垂直坐标来描述，如此构成拥有 70 个状态的状态空间 S。行为空间 A 具有四个基本行为。环境的的动力学特征不被个体掌握，但个体每执行一个行为，会进入一个新的状态，该状态由环境告知个体，但环境不会直接告诉个体该状态的坐标位置。即时奖励是根据任务目标来设定，现要求尽快从起始位置移动到目标位置，我们可以设定每移动一步只要不是进入目标位置都给予一个 -1 的惩罚，直至进入目标位置后获得奖励 0 同时永久停留在该位置。

我们将在编程实践环节给出用 Sarsa 算法解决有风格子世界问题的完整代码，这里先给出最优策略为依次采取如下的行为序列：

右、右、右、右、右、右、右、右、右、下、下、下、下、左、左

个体找到该最优策略的进度以及最优策略下个体从起始状态到目标状态的行为轨迹如图 5.3 所示。

可以看出个体在一开始的几百甚至上千步都在尝试各种操作而没有完成一次从起始位置到目标位置的经历。不过一旦个体找到一次目标位置后，它的学习过程将明显加速，最终找到了一条只需要 15 步的最短路径。由于世界的构造以及其内部风的影响，个体两次利用风的影响，先

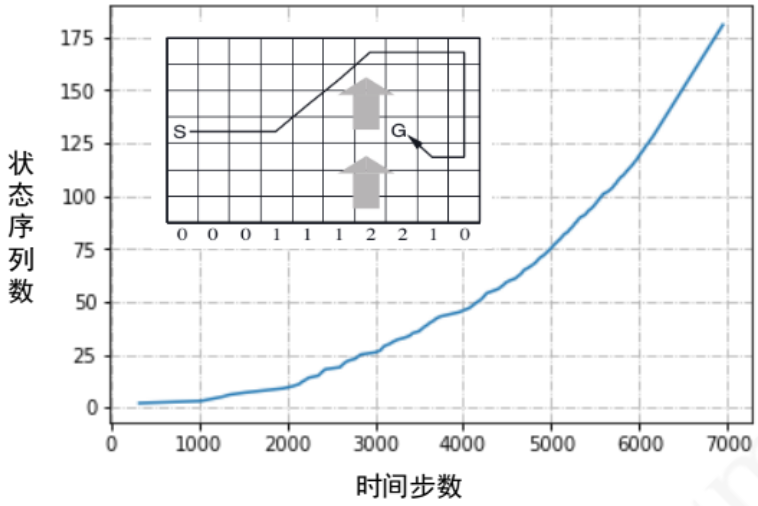


图 5.3: 有风格子世界最优路径和 Sarsa 算法学习曲线

向右并北漂到达最右上角后折返南下再左移才找到这条最短路径。其它路径均比该路径所花费的步数要多。

5.4.2 Sarsa(λ) 算法

在前一章，我们学习了 n-步收获，这里类似的引出一个 n-步 Sarsa 的概念。观察下面一些列的式子：

表 5.1: n-步 Q 收获

n=1	Sarsa	$q_t^{(1)} = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$
n=2		$q_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}, A_{t+2})$
...	...	...
n=∞	MC	$q_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \vdots + \gamma^{T-1} R_T$

这里的 q_t 对应的是一个状态行为对 (s_t, a_t)，表示的是在某个状态下采取某个行为的价值大小。如果 $n = 1$，则表示状态行为对 (s_t, a_t) 的 Q 价值可以用两部分表示，一部分是离开状态 s_t 得到的即时奖励 R_{t+1}，即时奖励只与状态有关，与该状态下采取的行为无关；另一部分是考虑了衰减因子的状态行为对 (s_{t+1}, a_{t+1}) 的价值：环境给了个体一个后续状态 s_{t+1}，观察在该状态基于当前策略得到的行为 a_{t+1} 时的价值 $Q(s_{t+1}, a_{t+1})$。当 $n = 2$ 时，就向前用 2 步的即时奖励，然后再用后续的 $Q(s_{t+2}, a_{t+2})$ 代替；如果 n 趋向于无穷大，则表示一直用带衰减因子的即时奖

励计算 Q 值，直至状态序列结束。

定义 n-**步** Q **收获** (Q-return) 为

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \vdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n}) \tag{5.5}$$

有了如上定义，可以把 n-步 Sarsa 用 n-步 Q 收获来表示，如下式：

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(q_t^{(n)} - Q(S_t, A_t) \right) \tag{5.6}$$

类似于 TD(λ), 可以给 n-步 Q 收获中的每一步收获分配一个权重，并按权重对每一步 Q 收获求和，那么将得到 q^λ 收获，它结合了所有 n-步 Q 收获：

$$q_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)} \tag{5.7}$$

如果使用用某一状态的 q_t^λ 收获来更新状态行为对的 Q 值，那么可以表示成如下的形式：

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(q_t^{(\lambda)} - Q(S_t, A_t) \right) \tag{5.8}$$

公式 (5.8) 即是 Sarsa(λ) 的前向认识，使用它更新 Q 价值需要遍历完整的状态序列。与 TD(λ) 类似，我们也可以反向理解 Sarsa(λ)。同样引入效用追迹 (eligibility traces, ET)，不同的是这次的 E 值针对的不是一个状态，而是一个状态行为对：

$$\begin{aligned} E_0(s,a) &= 0 \\ E_t(s,a) &= \gamma\lambda E_{t-1}(s,a) + 1(S_t = s, A_t = a), \ \gamma, \lambda \in [0,1] \end{aligned} \tag{5.9}$$

它体现的是一个结果与某一个状态行为对的因果关系，与得到该结果最近的状态行为对，以及那些在此之前频繁发生的状态行为对得到这个结果的影响最大。

下式是引入 ET 概念的之后的 Sarsa(λ) 算法中对 Q 值更新的描述：

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \\ Q(s,a) &\leftarrow Q(s,a) + \alpha \delta_t E_t(s,a) \end{aligned} \tag{5.10}$$

公式 (5.10) 便是反向认识的 Sarsa(λ)，基于反向认识的 Sarsa(λ) 算法将可以有效地在线学习，数据学习完即可丢弃。

Sarsa(λ) 的算法流程如算法 2 所述。

算法 2: Sarsa(λ) 算法

输入: episodes, α, γ
输出: Q
initialize: set $Q(s,a)$ arbitrarily, for each s in $\mathbb{S}$ and a in $\mathbb{A}(s)$; set $Q(\text{terminal state},\cdot)=0$
repeat for each episode in episodes
  $E(s,a)=0$ for each s in $\mathbb{S}$ and a in $\mathbb{A}(s)$
  initialize:
  $S \leftarrow$ first state of episode
  $A=\text{policy}(Q,S)$ (e.g. ϵ-greedy policy)
  repeat for each step of episode
    $R, S' = \text{perform\_action}(S,A)$
    $A'=\text{policy}(Q,S')$ (e.g. ϵ-greedy policy)
    $\delta \leftarrow R+\gamma Q(S',A')-Q(S,A)$
    $E(S,A) \leftarrow E(S,A)+1$
    for all $s\in\mathbb{S}, a\in\mathbb{A}(s)$ do
      $Q(S,A) \leftarrow Q(S,A)+\alpha\delta E(s,a)$
      $E(s,a) \leftarrow \gamma\lambda E(s,a)$
    end for
    $S \leftarrow S'; A \leftarrow A'$
  until S is terminal state;
until all episodes are visited;

这里要提及一下的是 $E(s,a)$ 在每浏览完一个状态序列后需要重新置 0，这体现了效用迹仅在一个状态序列中发挥作用；其次要提及的是算法更新 Q 和 E 的时候针对的不是某个状态序列里的 Q 或 E，而是针对个体掌握的整个状态空间和行为空间产生的 Q 和 E。算法为什么这么做，留给读者思考。我们将在编程实践部分实现 Sarsa(λ) 算法。

5.4.3 比较 Sarsa 和 Sarsa(λ)

图 5.4 用格子世界的例子具体解释了 Sarsa 和 Sarsa(λ) 算法的区别：假设图最左侧描述的路线是个体采取两种算法中的一个得到的一个完整状态序列的路径。为了下文更方便描述、解释两个算法之间的区别，先做几个合理的小约定：

1). 认定每一步的即时奖励为 0，直到终点处即时奖励为 1；

2). 根据算法，除了终点以外的任何状态行为对的 Q 值可以在初始时设为任意的，但我们设定所有的 Q 值均为 0；

3). 该路线是个体第一次找到终点的路线。

Sarsa 算法：由于是现时策略学习，一开始个体对环境一无所知，即所有的 Q 值均为 0，它将随机选取移步行为。在到达终点前的每一个位置 S，个体依据当前策略，产生一个移步行为，

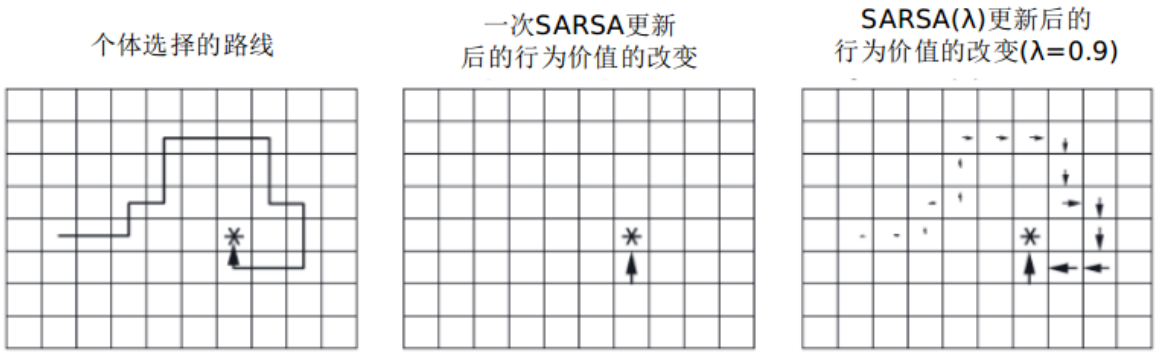


图 5.4: 图解 Sarsa 和 Sarsa(λ) 算法的区别

执行该行为，环境会将其放置到一个新位置 S'，同时给以即时奖励 0，在这个新位置上，根据当前的策略它会产生新位置下的一个行为，个体不执行该行为，仅仅在表中查找新状态下新行为的 Q 值，由于 $Q=0$，依据更新公式，它将把刚才离开的位置以及对应的行为的状态行为对价值 Q 更新为 0。如此直到个体最到达终点位置 S_G，它获得一个即时奖励 1，此时个体会依据公式更新其到达终点位置前所在那个位置（暂用 S_H 表示，也就是终点位置下方，向上的箭头所在的位置）时采取向上移步的那个状态行为对价值 $Q(S_H, A_{up})$，它将不再是 0，这是个体在这个状态序列中唯一一次用非 0 数值来更新 Q 值。这样完成一个状态序列，此时个体已经并只进行了一次有意义的行为价值函数的更新；同时依据新的价值函数产生了新的策略。这个策略绝大多数与之前的相同，只是当个体处在特殊位置 S_H 时将会有一个近乎确定的向上的行为。这里请不要误认为 Sarsa 算法只在经历一个完整的状态序列之后才更新，在这个例子中，由于我们的设定，它每走一步都会更新，只是多数时候更新的数据和原来一样罢了。

此时如果要求个体继续学习，则环境将其放入起点。个体的第二次寻路过程一开始与首次一样都是盲目随机的，直到其进入终点位置下方的位置 S_H，在这个位置，个体更新的策略将使其有非常大的几率选择向上的行为直接进入终点位置 S_G。

同样，经过第二次的寻路，个体了解到到达终点下方的位置 S_H 价值比较大，因为在这个位置直接采取向上移步的行为就可以拿到到达终点的即时奖励。因此它会将那些通过移动一步就可以到达 S_H 位置的其它位置以及相应的到达该位置位置所要采取的行为对所对应的价值进行提升。如此反复，如果采用贪婪策略更新，个体最终将得到一条到达终点的路径，不过这条路径的倒数第二步永远是在终点位置的下方。如果采用 ϵ-贪婪策略更新，那么个体还会尝试到终点位置的左上右等其它方向的相邻位置价值也比较大，此时个体每次完成的路径可能都不一样。通过重复多次搜索，这种 Q 值的实质有意义的更新将覆盖越来越多的状态行为对，个体在早期采取的随机行为的步数将越来越少，直至最终实质性的更新覆盖到起始位置。此时个体将能直接给出一条确定的从起点到终点的路径。

Sarsa(λ) 算法：该算法同时还针对每一次状态序列维护一个关于状态行为对 (S, A) 的 E 表，初始时 E 表值均为 0。当个体首次在起点 S_0 决定移动一步 A_0(假设向右) 时，它被环境告知新位置为 S_1，此时发生如下事情：首先个体会做一个标记，使 $E(S_0, A_0)$ 的值增加 1，表明个体刚刚经历过这个事件 (S_0, A_0)；其次它要估计这个事件的对于解决整个问题的价值，也就是估计 TD 误差，此时依据公式结果为 0，说明个体认为在起点处向右走没什么价值，这个“没有什么价值”有两层含义：不仅说明在 S_0 处往右目前对解决问题没有积极帮助，同时表明个体认为所有能够到达 S_0 状态的状态行为对的价值没有任何积极或消极的变化。随后个体将要更新该状态序列中所有已经经历的 $Q(S, A)$ 值，由于存在 E 值，那些在 (S_0, A_0) 之前近期发生或频繁发生的 (S, A) 的 Q 值将改变得比其它 Q 值明显些，此外个体还要更新其 E 值，以备下次使用。对于刚从起点出发的个体，这次更新没有使得任何 Q 值发生变化，仅仅在 $E(S_0, A_0)$ 处有了一个实质的变化。随后的过程类似，个体有意义的发现就是对路径有一个记忆，体现在 E 里，具体的 Q 值没发生变化。这一情况直到个体到达终点位置时发生改变。此时个体得到了一个即时奖励 1，它会发现这一次变化（从 S_H 采取向上行为 A_{up} 到达 S_G）价值明显，它会计算这个 TD 误差为 1，同时告诉整个经历过程中所有 (S, A)，根据其与 (S_H, A_{up}) 的密切关系更新这些状态行为对的价值 Q，个体在这个状态序列中经历的所有状态行为对的 Q 值都将得到一个非 0 的更新，但是那些在个体到达 S_H 之前就近发生以及频繁发生的状态行为对的价值提升得更加明显。

在图示的例子中没有显示某一状态行为频发的情况，如果个体在寻路的过程中绕过一些弯，多次到达同一个位置，并在该位置采取的相同的动作，最终个体到达终止状态时，就产生了多次发生的 (S, A)，这时的 (S, A) 的价值也会得到较多提升。也就是说，个体每得到一个即时奖励，同时会对所有历史事件的价值进行依次更新，当然那些与该事件关系紧密的事件价值改变的较为明显。这里的事件指的就是状态行为对。在同一状态采取不同行为是不同的事件。

当个体重新从起点第二次出发时，它会发现起点处向右走的价值不再是 0。如果采用贪婪策略更新，个体将根据上次经验得到的新策略直接选择右走，并且一直按照原路找到终点。如果采用 ϵ-贪婪策略更新，那么个体还会尝试新的路线。由于为了解释方便，做了一些约定，这会导致问题并不要求个体找到最短一条路径，如果需要找最短路径，需要在每一次状态转移时给个体一个负的奖励。

可以看出 Sarsa(λ) 算法在状态每发生一次变化后都对整个状态空间和行为空间的 Q 和 E 值进行更新，而事实上在每一个状态序列里，只有个体经历过的状态行为对的 E 才可能不为 0，为什么不仅仅对该状态序列涉及到的状态行为对进行更新呢？这个问题留给读者思考。

5.5 借鉴策略 Q 学习算法

现时策略学习的特点就是产生实际行为的策略与更新价值 (评价) 所使用的策略是同一个策略，而借鉴策略学习 (off-policy learning) 中产生指导自身行为的策略 $\mu(a|s)$ 与评价策略 $\pi(a|s)$ 是不同的策略，具体地说，个体通过策略 $\mu(a|s)$ 生成行为与环境发生实际交互，但是在更新这个状态行为对的价值时使用的是目标策略 $\pi(a|s)$。目标策略 $\pi(a|s)$ 多数是已经具备一定能力的策略，例如人类已有的经验或其他个体学习到的经验。借鉴策略学习相当于站在目标策略 $\pi(a|s)$ 的“肩膀”上学习。借鉴策略学习根据是否经历完整的状态序列可以将其分为基于蒙特卡洛的和基于 TD 的。基于蒙特卡洛的借鉴策略学习目前认为仅有理论上的研究价值，在实际中用处不大。这里主要讲解常用借鉴策略 TD 学习。

借鉴学习 TD 学习任务就是使用 TD 方法在目标策略 $\pi(a|s)$ 的基础上更新行为价值，进而优化行为策略：

$$V(S_t) \leftarrow V(S_t) + \alpha \left(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} (R_{t+1} + \gamma V(S_{t+1})) - V(S_t) \right)$$

对于上式，我们可以这样理解：个体处在状态 S_t 中，基于行为策略 μ 产生了一个行为 A_t，执行该行为后进入新的状态 S_{t+1}，借鉴策略学习要做的事情就是，比较借鉴策略和行为策略在状态 S_t 下产生同样的行为 A_t 的概率的比值，如果这个比值接近 1，说明两个策略在状态 S_t 下采取的行为 A_t 的概率差不多，此次对于状态 S_t 价值的更新同时得到两个策略的支持。如果这一概率比值很小，则表明借鉴策略 π 在状态 S_t 下选择 A_t 的机会要小一些，此时为了从借鉴策略学习，我们认为这一步状态价值的更新不是很符合借鉴策略，因而在更新时打些折扣。类似的，如果这个概率比值大于 1，说明按照借鉴策略，选择行为 A_t 的几率要大于当前行为策略产生 A_t 的概率，此时应该对该状态的价值更新就可以大胆些。

借鉴策略 TD 学习中一个典型的行为策略 μ 是基于行为价值函数 $Q(s,a)$ ϵ-贪婪策略，借鉴策略 π 则是基于 $Q(s,a)$ 的完全贪婪策略，这种学习方法称为 Q 学习 (Q learning)。

Q 学习的目标是得到最优价值 Q(s,a)，在 Q 学习的过程中，t 时刻的与环境进行实际交互的行为 A_t 由策略 μ 产生：

$$A_t \sim \mu(\cdot|S_t)$$

其中策略 μ 是一个 ϵ-贪婪策略。$t+1$ 时刻用来更新 Q 值的行为 A'_{t+1} 由下式产生：

$$A'_{t+1} \sim \pi(\cdot|S_{t+1})$$

其中策略 π 是一个完全贪婪策略。$Q(S_t, A_t)$ 的按下式更新：

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t))$$

其中红色部分的 TD 目标是基于借鉴策略 π 产生的行为 A' 得到的 Q 值。根据这种价值更新的方式，状态 S_t 依据 ϵ-贪婪策略得到的行为 A_t 的价值将朝着 S_{t+1} 状态下贪婪策略确定的最大行为价值的方向做一定比例的更新。这种算法能够使个体的行为策略策略 μ 更加接近贪婪策略，同时保证保证个体能持续探索并经历足够丰富的新状态。并最终收敛至最优策略和最优行为价值函数。

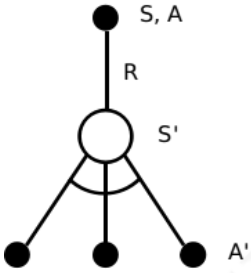


图 5.5: Q 学习算法示意图

下图是 Q 学习具体的行为价值更新公式：

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R + \gamma \underset{a'}{max} Q(S_{t+1}, a') - Q(S_t, A_t) \right) \tag{5.11}$$

Q 学习的算法流程如算法 3所述。

算法 3: Q 学习算法

输入: $episodes, \alpha, \gamma$
输出: Q
initialize: set $Q(s, a)$ arbitrarily, for each s in $\mathbb{S}$ and a in $\mathbb{A}(s)$; set $Q(\text{terminal state}, \cdot) = 0$
repeat for each episode in episodes
  initialize: $S \leftarrow$ first state of episode
  repeat for each step of episode
    $A = \text{policy}(Q, S)$ (e.g. ϵ-greedy policy)
    $R, S' = \text{perform\_action}(S, A)$
    $Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \max_a Q(S', a) - Q(S, A) \right)$
    $S \leftarrow S'$
  until S is terminal state;
until all episodes are visited;

这里通过悬崖行走的例子 (图) 简要讲解 Sarsa 算法与 Q 学习算法在学习过程中的差别。任务要求个体从悬崖的一端以尽可能快的速度行走到悬崖的另一端，每多走一步给以 -1 的奖励。图中悬崖用灰色的长方形表示，在其两端一个是起点，一个是目标终点。个体如果坠入悬崖将得到一个非常大的负向奖励 (-100) 回到起点。可以看出最优路线是贴着悬崖上方行走。Q 学习算法可以较快的学习到这个最优策略，但是 Sarsa 算法学到的是与悬崖保持一定的距离安全路线。在两种学习算法中，由于生成行为的策略依然是 ϵ 贪婪的，因此都会偶尔发生坠入悬崖的情况，如果 ϵ 贪婪策略中的 ϵ 随经历的增加而逐渐趋于 0，则两种算法都将最后收敛至最优策略。

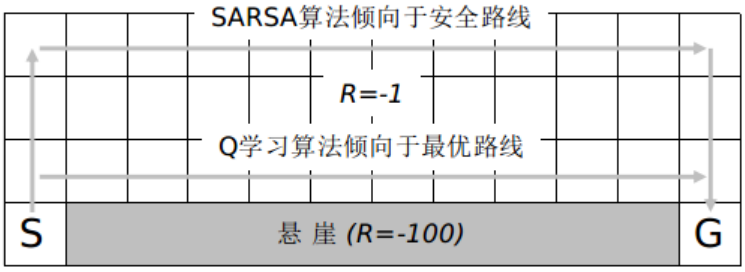


图 5.6: 悬崖行走示例

5.6 编程实践：蒙特卡罗学习求二十一点游戏最优策略

在本节的编程实践中，我们将继续使用前一章二十一点游戏的例子，这次我们要使用基于现时策略蒙特卡罗控制的方法来求解二十一点游戏玩家的最优策略。我们把上一章编写的 Dealer, Player 和 Arena 类保存至文件 blackjack.py 中，并加载这些类和其它一些需要的库和方法：

```
from blackjack import Player, Dealer, Arena
from utils import str_key, set_dict, get_dict
from utils import draw_value, draw_policy
from utils import epsilon_greedy_policy
import math
```

目前的 Player 类不具备策略评估和更新策略的能力，我们基于 Player 类编写一个 MC_Player 类，使其具备使用蒙特卡罗控制算法进行策略更新的能力，代码如下：

```
class MC_Player(Player):
    '''具备蒙特卡罗控制能力的玩家
    '''
```

```
    def __init__(self, name = "", A = None, display = False):
        super(MC_Player, self).__init__(name, A, display)
        self.Q = {}   # 某一状态行为对的价值，策略迭代时使用
        self.Nsa = {} # Nsa的计数：某一状态行为对出现的次数
        self.total_learning_times = 0
        self.policy = self.epsilon_greedy_policy #
        self.learning_method = self.learn_Q # 有了自己的学习方法

    def learn_Q(self, episode, r): # 从状态序列来学习Q值
        '''从一个Episode学习
        '''
        for s, a in episode:
            nsa = get_dict(self.Nsa, s, a)
            set_dict(self.Nsa, nsa+1, s, a)
            q = get_dict(self.Q, s,a)
            set_dict(self.Q, q+(r-q)/(nsa+1), s, a)
        self.total_learning_times += 1

    def reset_memory(self):
        '''忘记既往学习经历
        '''
        self.Q.clear()
        self.Nsa.clear()
        self.total_learning_times = 0

    def epsilon_greedy_policy(self, dealer, epsilon = None):
        '''这里的贪婪策略是带有epsilon参数的
        '''
        player_points, _ = self.get_points()
        if player_points >= 21:
            return self.A[1]
        if player_points < 12:
            return self.A[0]
        else:
            A, Q = self.A, self.Q
            s = self.get_state_name(dealer)
            if epsilon is None:
                #epsilon = 1.0/(self.total_learning_times+1)
                #epsilon = 1.0/(1 + math.sqrt(1 + player.total_learning_times))
                epsilon = 1.0/(1 + 4 * math.log10(1+player.total_learning_times)
```

```
                    )
44              return epsilon_greedy_policy(A, s, Q, epsilon)
```

这样，MC_Player 类就具备了学习 Q 值的方法和一个 ϵ-贪婪策略。接下来我们使用 MC_Player 类来生成对局，庄家的策略仍然不变。

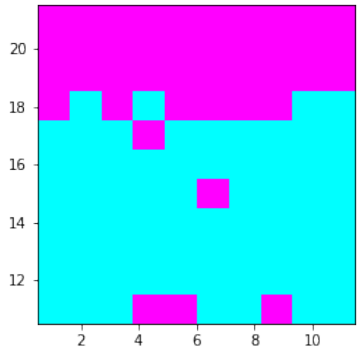
```
1  A=["继续叫牌","停止叫牌"]
2  display = False
3  player = MC_Player(A = A, display = display)
4  dealer = Dealer(A = A, display = display)
5  arena = Arena(A = A, display=display)
6  arena.play_games(dealer = dealer, player = player, num = 200000, show_statistic
       = True)
7
8  # 输出结果类似如下：
9  # 100%|██████████| 200000/200000 [00:25<00:00, 7991.15it/s]
10 # 共玩了200000局，玩家赢85019局，和15790局，输99191局，胜率：0.43,不输率:0.50
```

MC_Player 学习到的行为价值函数和最优策略可以使用下面的代码绘制：

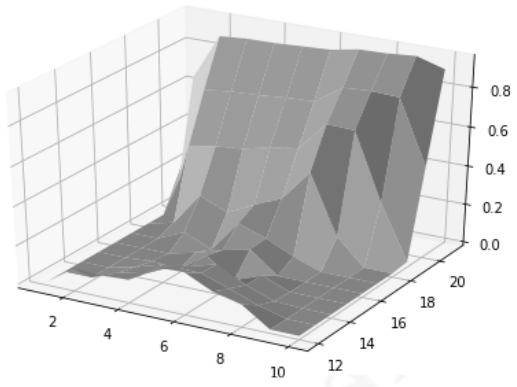
```
1  draw_value(player.Q, useable_ace = True, is_q_dict=True, A = player.A)
2  draw_policy(epsilon_greedy_policy, player.A, player.Q, epsilon = 1e-10,
       useable_ace = True)
3  draw_value(player.Q, useable_ace = False, is_q_dict=True, A = player.A)
4  draw_policy(epsilon_greedy_policy, player.A, player.Q, epsilon = 1e-10,
       useable_ace = False)
```

绘制结果图 5.6 所示。策略图中深色部分 (上半部) 为“停止叫牌”，浅色部分 (下半部) 为“继续交牌”。基于前一章介绍的二十一点游戏规则，迭代 20 万次后得到的贪婪策略为：当玩家手中有可用的 Ace 时，在牌点数达到 17 点，仍可选择叫牌；而当玩家手中没有可用的 Ace 时，当庄家的明牌在 2-7 点间，最好停止叫牌，当庄家的明牌为 A 或者超过 7 点时，可以选择继续交牌至玩家手中的牌点数到达 16 为止。由于训练次数并不多，策略图中还有一些零星的散点。

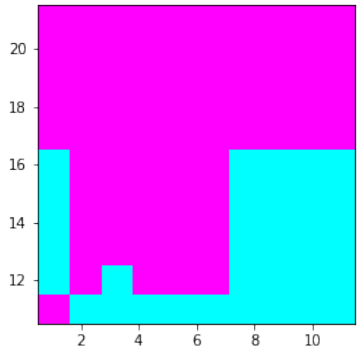
读者可以编写代码生成一些对局的详细数据观察具备 MC 控制能力的玩家的行为策略。



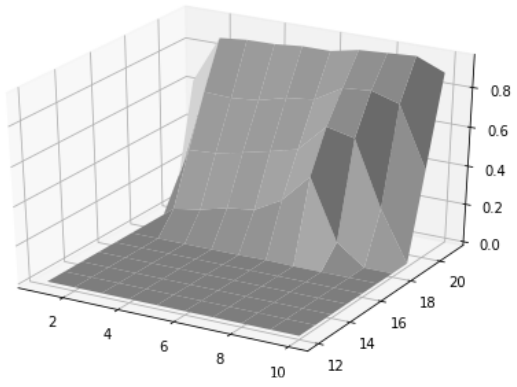
(a) 最优策略 (有可用的 Ace)



(b) 行为价值 (没有可用的 Ace)



(c) 最优策略 (有可用的 Ace)



(d) 行为价值 (没有可用的 Ace)

图 5.7: 二十一点游戏蒙特卡罗控制学习结果 (20 万次迭代)

5.7 编程实践：构建基于 gym 的有风格子世界及个体

强化学习讲究个体与环境的交互，强化学习算法聚焦于如何提高个体在与环境交互中的智能水平，我们在进行编程实践时需要实现这些算法。为了验证这些算法的有效性，我们需要有相应的环境。我们既可以自己编写环境，像前面介绍的二十一点游戏那样，也可以借助一些别人编写的环境，把终点放在个体学习算法的实现上。本节将向大家介绍一个出色基于 python 的强化学习库:gym 库，随后编写一个具备记忆功能的个体基类，为下一节编写个体的各种学习算法做准备。

5.7.1 gym 库简介

gym 库提供了一整套编程接口和丰富的强化学习环境，同时还提供可视化功能，方便观察个体的训练结果。该库的核心在文件 core.py 里，定义了两个最基本的类 Env 和 Space。前者是所有环境类的基类，后者是所有空间类的基类。从 Space 基类衍生出几个常用的空间类，其中最主要的是 Discrete 类和 Box 类。前者对应于一维离散空间，后者对应于多维连续空间。它们既可以应用在行为空间中，也可以用来描述状态空间。例如我要描述第三章提到的 4*4 的格子世界，其一共有 16 个状态，每一个状态只需要用一个数字来描述，这样我可以把这个问题的状态空间用 Discrete(16) 对象来描述就可以了，其对应的行为空间也可使用 Discrete(4) 来描述。

Gym 库的 Env 类包含如下几个关键的变量和方法：

```
class Env(object):

    # Set these in ALL subclasses
    action_space = None
    observation_space = None

    # Override in ALL subclasses
    def step(self, action): raise NotImplementedError
    def reset(self): raise NotImplementedError
    def render(self, mode= 'human', close = False): return
    def seed(self, seed = None): return []
```

Env 类是所有环境类的基类，它只是定义了环境应该具备的属性和功能，具体的环境类需要重写这些方法以完成特定的功能。其中：

step() 方法是最核心的方法，定义环境的动力学，确定个体的下一个状态、奖励信息、个体是否到达终止状态，以及一些额外的信息。其中个体的下一个状态、奖励信息、是否到达终止状态是可以被个体用来进行强化学习训练的。

reset() 方法用于重置环境，这个方法里需要将个体的状态重置为初始状态以及其他可能的一些初始化设置。环境应在个体与其交互前调用此方法。

seed() 设置一些随机数的种子。

render() 负责一些可视化工作。如果需要将个体与环境的交互以动画的形式展示出来的话，需要重写该方法。简单的 UI 设计可以用 gym 包装好了的 pyglet 方法来实现，这些方法在 rendering.py 文件里定义。具体使用这些方法进行 UI 绘制需要了解基本的 OpenGL 编程思想和接口，这里就不展开了。

在知道了 Env 类的主要接口后，我们可以按照其接口规范编写自己的环境类用于个体的训练。要使用 gym 库提供的功能，需要导入 gym 库：

```
import gym # 导入gym库
```

生成一个 gym 库内置的环境对象可以使用下面的代码：

```
env = gym.make("registered_env_name") #参数为注册了的环境名称
```

如果是使用自己编写的环境类，可以像正常生成对象一样：

```
env = MyEnvClassName()
```

个体在于 gym 环境进行交互时，最重要的一句代码是：

```
state, reward, is_done, info = env.step(a)
```

这句代码中，作为环境类的对象 env 执行了 step(a) 方法，方法的参数 a 是个体在当前状态是依据行为策略得到的行为。环境对象的 step(a) 方法返回由四个元素组成的元组，依次代表个体的下一个状态 (state)、获得的即时奖励 (reward)、是否到达终止状态 (is_done)、以及一个信息对象 (info)。给提可以利用前三个元素的信息来进行训练，info 对象则仅提供给编程者调试使用。

我们已经编写了一个符合 Gym 环境基类接口的格子世界环境类，并在此基础上实现了有风格子世界环境、悬崖行走、随机行走等各种环境。为了节约篇幅，这里不再讲解格子世界的详细实现过程，有兴趣的朋友可以参考本书附带的源代码。

5.7.2 状态序列的管理

个体与环境进行交互时会生成一个或多个甚至大量的状态序列，如何管理好这些状态序列是编程实践环节的一个比较重要的任务。状态序列是时间序列，在每一个时间步，个体与环境交互的信息包括个体的状态 (S_t)、采取的行为 (A_t)、上一个行为得到的奖励 R_{t+1} 这三个方面。描述一个完整的状态转换还应包括这两个信息：下一时刻个体的状态 (S_{t+1}) 和下一时刻的状态是否是终止状态 (is_end)。

多个相邻的状态转换构成了一个状态序列。多个完整的状态序列形成了个体的整体记忆，用 Memory 或 Expeirience 表示。通常一个个体的记忆容量不是无限的，在记忆的容量用满的情况下，如果个体需要记录新发生的状态序列，则通常忘记最早的一些状态序列。

在强化学习的个体训练中，如果使用 MC 学习算法，则需要学习完整的序列；如果使用 TD 学习，最小的学习单位则是一个状态转换。特别的，许多常用的 TD 学习算法刻意选择不连续的状态转换来学习来降低 TD 学习在一个序列中的偏差。在这种情况下是否把状态转换按时间次序以状态序列的形式进行管理就显得不那么重要了，本章为了解释一些 MC 学习类算法仍然采取了使用状态序列这一中间形式来管理个体的记忆。

基于上述考虑，我们依次设计了 Transition 类、Episode 类和 Experience 类来综合管理个体与环境交互时产生的多个状态序列。限于篇幅，这里不介绍其具体的实现代码，仅列出这些类一些重要的属性和方法：

```
# 此段代码是不完整的代码，完整代码请参阅本书附带的源代码
class Transition(object):
    def __init__(self, s0, a0, reward:float, is_done:bool, s1):
        self.data = [s0, a0, reward, is_done, s1]

    #...

class Episode(object):
    def __init__(self, e_id:int = 0) -> None:
        self.total_reward = 0   # 总的获得的奖励
        self.trans_list = []    # 状态转移列表
        self.name = str(e_id)   # 可以给Episode起个名字："成功闯关，黯然失败？"

    def push(self, trans:Transition) -> float:
        '''将一个状态转换送入状态序列中，返回该序列当前总的奖励值
        '''

    @property
    def len(self):
        return len(self.trans_list)

    def is_complete(self) -> bool:
        '''判断当前状态序列是否是一个完整的状态序列
        '''

    def sample(self, batch_size = 1):
```

```
        '''从当前状态序列中随机产生一定数量的不连续的状态转换
        '''
    #...

class Experience(object):
    def __init__(self, capacity:int = 20000):
        self.capacity = capacity    # 容量: 指的是trans总数量
        self.episodes = []          # episode列表
        self.total_trans = 0        # 总的状态转换数量

    def _remove_first(self):
        '''删除第一个(最早的)状态序列
        '''

    def push(self, trans):
        '''记住一个状态转换, 根据当前状态序列是否已经完整来将trans加入现有状态序
           列还是开启一个新的状态序列
        '''

    def sample(self, batch_size=1):
        '''随机从经历中产生一定数量的不连续的状态转换
        '''

    def sample_episode(self, episode_num = 1):
        '''从经历中随机获取一定数量完整的状态序列
        '''

    @property
    def last_episode(self):
        '''得到当前最新的一个状态序列
        '''
    #...
```

5.7.3 个体基类的编写

我们把重点放在编写一个个体基类 (Agent) 上，为后续实现各种强化学习算法提供一个基础。这个基类符合 gym 库的接口规范，具备个体最基本的功能，同时我们希望个体具有一定容

量的记忆功能，能够记住曾经经历过的一些状态序列。我们还希望个体在学习时能够记住一些学习过程，便于分析个体的学习效果等。有了个体基类之后，在讲解具体一个强化学习算法时仅需实现特定的方法就可以了。

在第一章讲解强化学习初步概念的时候，我们对个体类进行了一个初步的建模，这次我们要构建的是符合 gym 借口规范的 Agent 基类，其中一个最基本的要求个体类对象在构造时接受环境对象作为参数，内部也有一个成员变量建立对这个环境对象的引用。在我们设计的个体基类中，其成员变量包括对环境对象的应用、状态和行为空间、与环境交互产生的经历、当前状态等。此外对于个体来说，他还具备的能力有：遵循策略产生一个行为、执行一个行为与环境交互、采用什么学习方法、具体如何学习这几个关键功能，其中最关键的是个体执行行为与环境进行交互的方法。下面的代码实现了我们的需求。

```
class Agent(object):
    '''个体基类，没有学习能力
    '''
    def __init__(self, env: Env = None,
                       capacity = 10000):
        # 保存一些Agent可以观测到的环境信息以及已经学到的经验
        self.env = env # 建立对环境对象的引用
        self.obs_space = env.observation_space if env is not None else None
        self.action_space = env.action_space if env is not None else None
        if type(self.obs_space) in [gym.spaces.Discrete]:
            self.S = [str(i) for i in range(self.obs_space.n)]
            self.A = [str(i) for i in range(self.action_space.n)]
        else:
            self.S, self.A = None, None
        self.experience = Experience(capacity = capacity)
        # 有一个变量记录agent当前的state相对来说还是比较方便的。要注意对该变量的
            维护、更新
        self.state = None   # 个体的当前状态

    def policy(self, A, s = None, Q = None, epsilon = None):
        '''均一随机策略
        '''
        return random.sample(self.A, k=1)[0]

    def perform_policy(self, s, Q = None, epsilon = 0.05):
        action = self.policy(self.A, s, Q, epsilon)
        return int(action)
```

```
27
28      def act(self, a0):
29          s0 = self.state
30          s1, r1, is_done, info = self.env.step(a0)
31          # TODO add extra code here
32          trans = Transition(s0, a0, r1, is_done, s1)
33          total_reward = self.experience.push(trans)
34          self.state = s1
35          return s1, r1, is_done, info, total_reward
36
37      def learning_method(self,lambda_ = 0.9, gamma = 0.9, alpha = 0.5, epsilon =
            0.2, display = False):
38          '''这是一个没有学习能力的学习方法
39          具体针对某算法的学习方法，返回值需是一个二维元组：(一个状态序列的时间
                步、该状态序列的总奖励)
40          '''
41          self.state = self.env.reset()
42          s0 = self.state
43          if display:
44              self.env.render()
45          a0 = self.perform_policy(s0, epsilon)
46          time_in_episode, total_reward = 0, 0
47          is_done = False
48          while not is_done:
49              # add code here
50              s1, r1, is_done, info, total_reward = self.act(a0)
51              if display:
52                  self.env.render()
53              a1 = self.perform_policy(s1, epsilon)
54              # add your extra code here
55              s0, a0 = s1, a1
56              time_in_episode += 1
57          if display:
58              print(self.experience.last_episode)
59          return time_in_episode, total_reward
60
61
62      def learning(self, lambda_ = 0.9, epsilon = None, decaying_epsilon = True,
            gamma = 0.9,
63                       alpha = 0.1, max_episode_num = 800, display = False):
```

```
        total_time,  episode_reward, num_episode = 0,0,0
        total_times, episode_rewards, num_episodes = [], [], []
        for i in tqdm(range(max_episode_num)):
            if epsilon is None:
                epsilon = 1e-10
            elif decaying_epsilon:
                epsilon = 1.0 / (1 + num_episode)
            time_in_episode, episode_reward = self.learning_method(lambda_ =
               lambda_, \
                  gamma = gamma, alpha = alpha, epsilon = epsilon, display =
                     display)
            total_time += time_in_episode
            num_episode += 1
            total_times.append(total_time)
            episode_rewards.append(episode_reward)
            num_episodes.append(num_episode)
        #self.experience.last_episode.print_detail()
        return  total_times, episode_rewards, num_episodes

    def sample(self, batch_size = 64):
        '''随机取样
        '''
        return self.experience.sample(batch_size)

    @property
    def total_trans(self):
        '''得到Experience里记录的总的状态转换数量
        '''
        return self.experience.total_trans

    def last_episode_detail(self):
        self.experience.last_episode.print_detail()
```

不难看出 Agent 类的策略是最原始的均一随机策略，不具备学习能力。不过它已经具备了同 gym 环境进行交互的能力了。由于该个体不具备学习能力，可以编写如下的代码来观察均一策略下个体在有风格子世界里的交互情况：

```
# 测试个体基类和有风格子世界环境
```

```
import gym
from gym import Env
from gridworld import WindyGridWorld # 导入有风格子世界环境
from core import Agent # 导入个体基类

env = WindyGridWorld() # 生成有风格子世界环境对象
env.reset() # 重置环境对象
env.render() # 显示环境对象可视化界面

agent = Agent(env, capacity = 10000) # 创建Agent对象
data = agent.learning(max_episode_num = 180, display = False)
# env.close() # 关闭可视化界面
```

运行上述代码将显示如图 5.8 所示的一个有风格子世界交互界面。图中多数格子用粉红色绘制，表示个体在离开该格子时将获得 -1 的即时奖励，白色的格子对应的即时奖励为 0。有蓝色边框的格子是个体的起始状态，金黄色边框对应的格子是终止状态。个体在格子世界中用黄色小圆形来表示。风的效果并未反映在可视化界面上，它将实实在在的影响个体采取一个行为后的后续状态 (位置)。

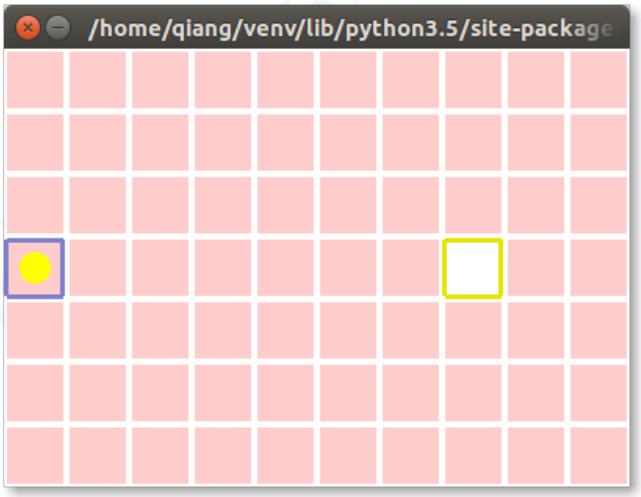


图 5.8: 有风格子世界 gym 环境可视化界面

由于可视化交互在进行多次尝试时浪费计算资源，我们在随后进行 180 次的尝试期间选择不显示个体的动态活动。其 180 次的交互信息存储在对象 data 内。图 5.9 是依据 data 绘制的该个体与环境交互产生的的状态序列时间步数与状态序列次数的关系图，可以看出个体最多曾用

了三万多步才完成一个完整的状态序列。

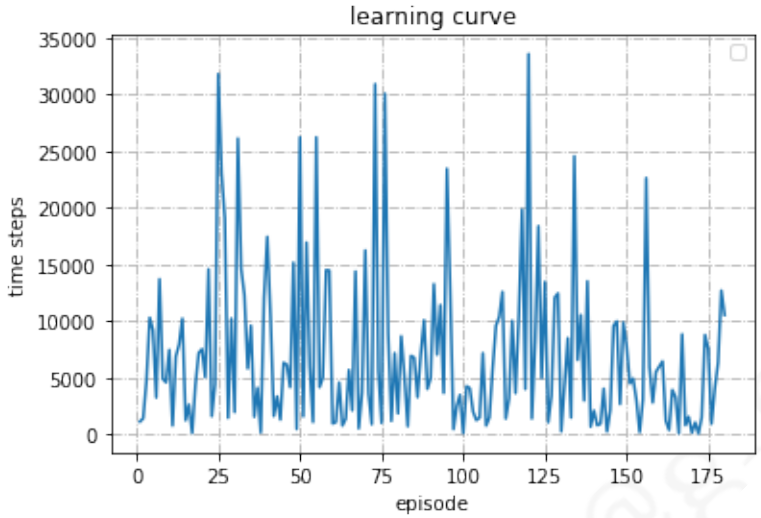


图 5.9: 均一随机策略的个体在有风格子世界环境里的表现

要让个体具备学习能力需要改写策略方法 (policy) 以及学习方法 (learning_method) 方法。下一节将详细介绍不同学习算法的实现并观察它们在有风格子世界中的交互效果。

5.8 编程实践：各类学习算法的实现及与有风格子世界的交互

在本节的编程实践中，我们将使用自己编写的有风格子世界环境类，在 Agent 基类的基础上分别建立具有 Sarsa 学习、Sarsa(λ) 学习和 Q 学习能力的三个个体子类，我们只要分别实现其策略方法以及学习方法就可以了。

对于这三类学习算法要用到贪婪策略或 ϵ-贪婪策略，由于我们计划使用字典来存储行为价值函数数据，还会用到之前编写的根据状态生成键以及读取字典的方法，本节中这些方法都被放在一个名为 utls.py 的文件中，有风格子世界环境类在文件 gridworld.py 文件中实现，个体基类的实现代码存放在 core.py 中。将这三个文件放在当前工作目录下。下面的代码将从这些文件中导入要使用的类和方法：

```
from random import random, choice
from core import Agent
from gym import Env
import gym
from gridworld import WindyGridWorld, SimpleGridWorld
from utils import str_key, set_dict, get_dict
```

```
from utils import epsilon_greedy_pi, epsilon_greedy_policy
from utils import greedy_policy, learning_curve
```

5.8.1 Sarsa 算法

本章正文中的算法 1 给出了 Sarsa 算法的流程，依据流程不难得到如下的实现代码：

```
class SarsaAgent(Agent):
    def __init__(self, env:Env, capacity:int = 20000):
        super(SarsaAgent, self).__init__(env, capacity)
        self.Q = {} # 增加Q字典存储行为价值

    def policy(self, A, s, Q, epsilon):
        '''使用epsilon-贪婪策略
        '''
        return epsilon_greedy_policy(A, s, Q, epsilon)

    def learning_method(self, gamma = 0.9, alpha = 0.1, epsilon = 1e-5, display
        = False, lambda_ = None):
        self.state = self.env.reset()
        s0 = self.state
        if display:
            self.env.render()
        a0 = self.perform_policy(s0, self.Q, epsilon)
        # print(self.action_t.name)
        time_in_episode, total_reward = 0, 0
        is_done = False
        while not is_done:
            s1, r1, is_done, info, total_reward = self.act(a0)
            if display:
                self.env.render()
            a1 = self.perform_policy(s1, self.Q, epsilon)
            #
            old_q = get_dict(self.Q, s0, a0)
            q_prime = get_dict(self.Q, s1, a1)
            td_target = r1 + gamma * q_prime
            new_q = old_q + alpha * (td_target - old_q)
            set_dict(self.Q, new_q, s0, a0)
```

```
            s0, a0 = s1, a1
            time_in_episode += 1
        if display:
            print(self.experience.last_episode)
        return time_in_episode, total_reward
```

5.8.2 Sarsa(λ) 算法

本章正文中的算法 2 给出了 Sarsaλ 算法的流程，依据流程不难得到如下的实现代码：

```
class SarsaLambdaAgent(Agent):
    def __init__(self, env:Env, capacity:int = 20000):
        super(SarsaLambdaAgent, self).__init__(env, capacity)
        self.Q = {}

    def policy(self, A, s, Q, epsilon):
        return epsilon_greedy_policy(A, s, Q, epsilon)

    def learning_method(self, lambda_ = 0.9, gamma = 0.9, alpha = 0.1, epsilon =
        1e-5, display = False):
        self.state = self.env.reset()
        s0 = self.state
        if display:
            self.env.render()
        a0 = self.perform_policy(s0, self.Q, epsilon)
        # print(self.action_t.name)
        time_in_episode, total_reward = 0,0
        is_done = False
        E = {} # 效用值
        while not is_done:
            s1, r1, is_done, info, total_reward = self.act(a0)
            if display:
                self.env.render()
            a1 = self.perform_policy(s1, self.Q, epsilon)

            q = get_dict(self.Q, s0, a0)
            q_prime = get_dict(self.Q, s1, a1)
```

```
            delta = r1 + gamma * q_prime - q

            e = get_dict(E, s0, a0)
            e += 1
            set_dict(E, e, s0, a0)

            for s in self.S: # 对所有可能的Q(s,a)进行更新
                for a in self.A:
                    e_value = get_dict(E, s, a)
                    old_q = get_dict(self.Q, s, a)
                    new_q = old_q + alpha * delta * e_value
                    new_e = gamma * lambda_ * e_value
                    set_dict(self.Q, new_q, s, a)
                    set_dict(E, new_e, s, a)

            s0, a0 = s1, a1
            time_in_episode += 1
        if display:
            print(self.experience.last_episode)
        return time_in_episode, total_reward
```

5.8.3 Q 学习算法

本章正文中的算法 3 给出了 Q 学习算法的流程，依据流程不难得到如下的实现代码：

```
class QAgent(Agent):
    def __init__(self, env:Env, capacity:int = 20000):
        super(QAgent, self).__init__(env, capacity)
        self.Q = {}

    def policy(self, A, s, Q, epsilon):
        return epsilon_greedy_policy(A, s, Q, epsilon)

    def learning_method(self, gamma = 0.9, alpha = 0.1, epsilon = 1e-5, display
        = False, lambda_ = None):
        self.state = self.env.reset()
        s0 = self.state
        if display:
```

```
            self.env.render()
        time_in_episode, total_reward = 0, 0
        is_done = False
        while not is_done:
            self.policy = epsilon_greedy_policy # 行为策略
            a0 = self.perform_policy(s0, self.Q, epsilon)
            s1, r1, is_done, info, total_reward = self.act(a0)
            if display:
                self.env.render()
            self.policy = greedy_policy
            a1 = greedy_policy(self.A, s1, self.Q) # 借鉴策略

            old_q = get_dict(self.Q, s0, a0)
            q_prime = get_dict(self.Q, s1, a1)
            td_target = r1 + gamma * q_prime
            new_q = old_q + alpha * (td_target - old_q)
            set_dict(self.Q, new_q, s0, a0)

            s0 = s1
            time_in_episode += 1
        if display:
            print(self.experience.last_episode)
        return time_in_episode, total_reward
```

读者可借鉴 Agent 基类与环境交互的代码来实现拥有各种不同学习能力的子类与有风格子世界进行交互的代码，体会三种学习算法的区别。以 Sarsa(λ) 算法为例，下面的代码实现其与有风格子世界环境的交互。

```
env = WindyGridWorld()
agent = SarsaLambdaAgent(env, capacity = 100000)
statistics = agent.learning(lambda_ = 0.8, gamma = 1.0, epsilon = 0.2,\
    decaying_epsilon = True, alpha = 0.5, max_episode_num = 800, display = False
        )
```

如果对个体行为的可视化表现感兴趣，可以将 learning 方法内的参数 display 设置为 True。下面的代码可视化展示个体两次完整的交互经历。

```
agent.learning(max_episode_num = 2, display = True)
```

需要指出的这三种学习算法在完成第一个完整状态序列时都可能会花费较长的时间步数，特别是对于 Sarsa(λ) 算法来说，由于在每一个时间步都要做大量的计算工作，因而花费的计算资源更多，该算法的优势是在线实时学习。

我们的 gridworld.py 中提供了悬崖行走环境 (CliffWalk) 类，读者可以直接使用这三个 Agent 类观察比较它们在悬崖行走环境中的表现。