第四章　不基于模型的预测

前一章讲解了如何应用动态规划算法对一个已知状态转移概率的 MDP 进行策略评估或通过策略迭代或者直接的价值迭代来寻找最优策略和最有价值函数，同时也指出了动态规划算法的一些缺点。从本章开始的连续两章内容将讲解如何解决一个可以被认为是 MDP、但却不掌握 MDP 具体细节的问题，也就是讲述个体如何在没有对环境动力学认识的模型的条件下如何直接通过个体与环境的实际交互来评估一个策略的好坏或者寻找到最优价值函数和最优策略。其中本章将聚焦于策略评估，也就是预测问题；下一章将利用本讲的主要观念来进行控制进而找出最优策略以及最有价值函数。

本章分为三个部分，将分别从理论上阐述基于完整采样的蒙特卡罗强化学习、基于不完整采样的时序差分强化学习以及介于两者之间的 λ 时序差分强化学习。这部分内容比较抽象，在讲解理论的同时会通过一些精彩的实例来加深对概念和算法的理解。

4.1　蒙特卡罗强化学习

蒙特卡罗强化学习 (Monte-Carlo reinforcement learning, MC **学习**)：指在不清楚 MDP 状态转移概率的情况下，直接从经历完整的状态序列 (episode) 来估计状态的真实价值，并认为某状态的价值等于在多个状态序列中以该状态算得到的所有收获的平均。

完整的状态序列 (complete episode)：指从某一个状态开始，个体与环境交互直到终止状态，环境给出终止状态的奖励为止。完整的状态序列不要求起始状态一定是某一个特定的状态，但是要求个体最终进入环境认可的某一个终止状态。

蒙特卡罗强化学习有如下特点：不依赖状态转移概率，直接从经历过的完整的状态序列中学习，使用的思想就是用平均收获值代替价值。理论上完整的状态序列越多，结果越准确。

我们可以使用蒙特卡罗强化学习来评估一个给定的策略。基于特定策略 π 的一个 Episode 信息可以表示为如下的一个序列：

$$S_1, A_1, R_2, S_2, A_2, ..., S_t, A_t, R_{t+1}, ..., S_k \sim \pi$$

t 时刻状态 S_t 的收获可以表述为：

$$G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$$

其中 T 为终止时刻。该策略下某一状态 s 的价值：

$$v_\pi(s) = E_\pi[G_t | S_t = s]$$

不难发现，在蒙特卡罗算法评估策略时要针对多个包含同一状态的完整状态序列求收获继而再取收获的平均值。如果一个完整的状态序列中某一需要计算的状态出现在序列的多个位置，也就是说个体在与环境交互的过程中从某状态出发后又一次或多次返回过该状态，这种现象在第二章介绍收获的计算时遇到过：一位学生从上“第一节课”开始因“浏览手机”以及在“第三节课”选择泡吧后多次重新回到“第一节课”。在这种情况下，根据收获的定义，在一个状态序列下，不同时刻的同一状态其计算得到的收获值是不一样的。很明显，在蒙特卡罗强化学习算法中，计算收获时也会碰到这种情况。我们有两种方法可以选择，一是仅把状态序列中第一次出现该状态时的收获值纳入到收获平均值的计算中；另一种是针对一个状态序列中每次出现的该状态，都计算对应的收获值并纳入到收获平均值的计算中。两种方法对应的蒙特卡罗评估分别称为：首次访问 (first visit) 和每次访问 (every visit) 蒙特卡罗评估。

在求解状态收获的平均值的过程中，我们介绍一种非常实用的不需要存储所有历史收获的计算方法：累进更新平均值 (incremental mean)。而且这种计算平均值的思想也是强化学习的一个核心思想之一。具体公式如下：

$$\begin{aligned}
\mu_k &= \frac{1}{k}\sum_{j=1}^{k} x_j \\
&= \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) \\
&= \frac{1}{k}\left(x_k + (k-1)\mu_{k-1}\right) \\
&= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})
\end{aligned}$$

累进更新平均值利用前一次的平均值和当前数据以及数据总个数来计算新的平均值：当每产生一个需要平均的新数据 x_k 时，先计算 x_k 与先前平均值 μ_{k-1} 的差，再将这个差值乘以一定的系数 $\frac{1}{k}$ 后作为误差对旧平均值进行修正。如果把该式中平均值和新数据分别看成是状态的价

值和该状态的收获，那么该公式就变成了递增式的蒙特卡罗法更新状态价值。其公式如下：

$$\begin{aligned} N(S_t) &\leftarrow N(S_t) + 1 \\ V(S_t) &\leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t)) \end{aligned} \tag{4.1}$$

在一些实时或者无法统计准确状态被访问次数时，可以用一个系数 α 来代替状态计数的倒数，此时公式变为：

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)) \tag{4.2}$$

以上就是蒙特卡罗学习方法的主要思想和描述，下文将介绍另一种强化学习方法：时序差分学习。

4.2 时序差分强化学习

时序差分强化学习 (temporal-difference reinforcement learning, TD 学习)：指从采样得到的不完整的状态序列学习，该方法通过合理的引导（bootstrapping），先估计某状态在该状态序列完整后可能得到的收获，并在此基础上利用前文所属的累进更新平均值的方法得到该状态的价值，再通过不断的采样来持续更新这个价值。

具体地说，在 TD 学习中，算法在估计某一个状态的收获时，用的是离开该状态的即刻奖励 R_{t+1} 与下一时刻状态 S_{t+1} 的预估状态价值乘以衰减系数 γ 组成：

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{4.3}$$

其中：$R_{t+1} + \gamma V(S_{t+1})$ 称为 TD **目标值**。$R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ 称为 TD **误差**。

引导 (bootstrapping)：指的是用 TD 目标值代替收获 G_t 的过程。

可以看出，不管是 MC 学习还是 TD 学习，它们都不再需要清楚某一状态的所有可能的后续状态以及对应的状态转移概率，因此也不再像动态规划算法那样进行全宽度的回溯来更新状态的价值。MC 和 TD 学习使用的都是通过个体与环境实际交互生成的一系列状态序列来更新状态的价值。这在解决大规模问题或者不清楚环境动力学特征的问题时十分有效。不过 MC 学习和 TD 学习两者也是有着很明显的差别的。下文将通过一个例子来详细阐述这两种学习方法格子的特点。

想象一下作为个体的你如何预测下班后开车回家这个行程所花费的时间。在回家的路上你会依次经过一段高速公路、普通公路、和你家附近街区三段路程。由于你经常开车上下班，在下班的路上多次碰到过各种情形，比如取车的时候发现下雨，高速路况的好坏、普通公路是否堵车

等等。在每一种状态下时，你对还需要多久才能到家都有一个经验性的估计。表 1 的“既往经验预计（仍需耗时）”列给出了这个经验估计，这个经验估计基本反映了各个状态对应的价值，通常你对下班回家总耗时的预估是 30 分钟。

表 4.1: 驾车返家数据

单位：分钟

状态	已耗时	既往经验预计		MC 更新 ($\alpha = 1$)		TD 更新 ($\alpha = 1$)	
		仍需耗时	总耗时	仍需耗时	总耗时	仍需耗时	总耗时
离开办公室	0	30	30	43	43	40	40
取车时下雨	5	35	40	38	43	30	35
驶离高速	20	15	35	23	43	20	40
跟在卡车后	30	10	40	13	43	13	43
家附近街区	40	3	43	3	43	3	43
返回家中	43	0	43	0	43	0	43

假设你现在又下班准备回家了，当花费了 5 分钟从办公室到车旁时，发现下雨了。此时根据既往经验，估计还需要 35 分钟才能到家，因此整个行程将耗费 40 分钟。随后你进入了高速公路，高速公路路况非常好，你一共仅用了 20 分钟就离开了高速公路，通常根据经验你只再需要 15 分钟就能到家，加上已经过去的 20 分钟，你将这次返家预计总耗时修正为 35 分钟，比先前的估计少了 5 分钟。但是当你进入普通公路时，发现交通流量较大，你不得不跟在一辆卡车后面龟速行驶，这个时候距离出发已经过去 30 分钟了，根据以往你路径此段的经验，你还需要 10 分钟才能到家，那么现在你对于回家总耗时的预估又回到了 40 分钟。最后你在出发 40 分钟后到达了家附近的街区，根据经验，还需要 3 分钟就能到家，此后没有再出现新的情况，最终你在 43 分钟的时候到达家中。经历过这一次的下班回家，你对于处在途中各种状态下返家的还需耗时（对应于各状态的价值）有了新的估计，但分别使用 MC 算法和 TD 算法得到的对于各状态返家还需耗时的更新结果和更新时机都是不一样的。

如果使用 MC 算法，在整个驾车返家的过程中，你对于所处的每一个状态，例如“取车时下雨”，“离开高速公路”，“被迫跟在卡车后”、“进入街区”等时，都不会立即更新这些状态对应的返家还需耗时的估计，这些状态的返家仍需耗时仍然分别是先前的 35 分钟、15 分钟、10 分钟和 3 分钟。但是当你到家发现整个行程耗时 43 分钟后，通过用实际总耗时减去到达某状态的已耗时，你发现在本次返家过程中在实际到达上述各状态时，仍需时间则分别变成了：38 分钟、23 分钟、13 分钟和 3 分钟。如果选择修正系数为 1，那么这些新的耗时将成为今后你在各状态

时的预估返家仍需耗时，相应的整个行程的预估耗时被更新为 43 分钟。

如果使用 TD 算法，则又是另外一回事，当取车发现下雨时，同样根据经验你会认为还需要 35 分钟才能返家，此时，你将立刻更新对于返家总耗时的估计，为仍需的 35 分钟加上你离开办公室到取车现场花费的 5 分钟，即 40 分钟。同样道理，当驶离高速公路，根据经验，你对到家还需时间的预计为 15 分钟，但由于之前你在高速上较为顺利，节省了不少时间，在第 20 分钟时已经驶离高速，实际从取车到驶离高速只花费了 15 分钟，则此时你又立刻更新了从取车时下雨到到家所需的时间为 30 分钟，而整个回家所需时间更新为 35 分钟。当你在驶离高速在普通公路上又行驶了 10 分钟被堵，你预计还需 10 分钟才能返家时，你对于刚才驶离高速公路返家还需耗时又做了更新，将不再是根据既往经验预估的 15 分钟，而是现在的 20 分钟，加上从出发到驶离高速已花费的 20 分钟，整个行程耗时预估因此被更新为 40 分钟。直到你花费了 40 分钟只到达家附近的街区还预计有 3 分钟才能到家时，你更新了在普通公路上对于返家还需耗时的预计为 13 分钟。最终你按预计 3 分钟后进入家门，不再更新剩下的仍需耗时。

通过比较可以看出，MC 算法只在整个行程结束后才更新各个状态的仍需耗时，而 TD 算法则每经过一个状态就会根据在这个状态与前一个状态间实际所花时间来更新前一个状态的仍需耗时。下图则用折线图直观地显示了分别使用 MC 和 TD 算法时预测的驾车回家总耗时的区别。需要注意的是，在这个例子中，与各状态价值相对应的指标并不是图中显示的驾车返家总耗时，而是处于某个状态时驾车返家的仍需耗时。

TD 学习能比 MC 学习更快速灵活的更新状态的价值估计，这在某些情况下有着非常重要的实际意义。回到驾车返家这个例子中来，我们给驾车返家制定一个新的目标，不再以耗时多少来评估状态价值，而是要求安全平稳的返回家中。假如有一次你在驾车回家的路上突然碰到险情：对面开过来一辆车感觉要和你迎面相撞，严重的话甚至会威胁生命，不过由于最后双方驾驶员都采取了紧急措施没有让险情实际发生，最后平安到家。如果是使用蒙特卡罗学习，路上发生的这一险情可能引发的极大负值奖励将不会被考虑，你不会更新在碰到此类险情时的状态的价值；但是在 TD 学习时，碰到这样的险情过后，你会立即大幅调低这个状态的价值，并在今后再次碰到类似情况时采取其它行为，例如降低速度等来让自身处在一个价值较高的状态中，尽可能避免发生意外事件的发生。

通过驾车返家这个例子，我们应该能够认识到：TD 学习在知道结果之前就可以学习，也可以在没有结果时学习，还可以在持续进行的环境中学习，而 MC 学习则要等到最后结果才能学习。TD 学习在更新状态价值时使用的是 TD 目标值，即基于即时奖励和下一状态的预估价值来替代当前状态在状态序列结束时可能得到的收获，它是当前状态价值的有偏估计，而 MC 学习则使用实际的收获来更新状态价值，是某一策略下状态价值的无偏估计。TD 学习存在偏倚 (bias) 的原因是在于其更新价值时使用的也是后续状态预估的价值，如果能使用后续状态基于某策略的真实 TD 目标值 (true TD target) 来更新当前状态价值的话，那么此时的 TD 学习得到的价值

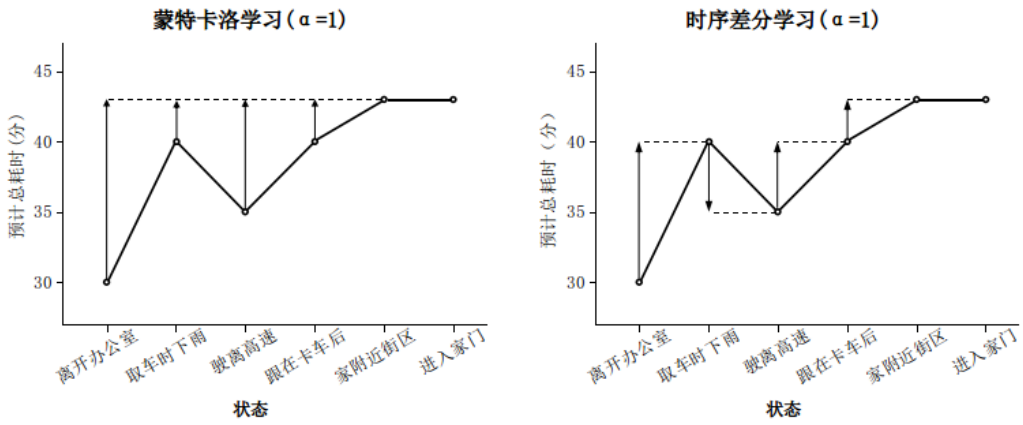


图 4.1: MC 和 TD 学习在驾车返家示例中的比较

也是实际价值的无偏估计。虽然绝大多数情况下 TD 学习得到的价值是有偏估计的，但是其方差 (Variance) 却较 MC 学习得到的方差要低，且对初始值敏感，通常比 MC 学习更加高效，这也主要得益于 TD 学习价值更新灵活，对初始状态价值的依赖较大。我们将继续通过一个示例来剖析 TD 学习和 MC 学习的特点。

假设在一个强化学习问题中有 A 和 B 两个状态，模型未知，不涉及策略和行为，只涉及状态转换和即时奖励，衰减系数为 1。现有如下表所示 8 个完整状态序列的经历，其中除了第 1 个状态序列发生了状态转移外，其余 7 个完整的状态序列均只有一个状态构成。现要求根据现有信息计算状态 A、B 的价值分别是多少?

表 4.2: AB 状态转移经历

序号	状态转移及即时奖励
1	A:0, B:0
2	B:1
3	B:1
4	B:1
5	B:1
6	B:1
7	B:1
8	B:0

我们考虑分别使用 MC 算法和 TD 算法来计算状态 A、B 的价值。首先考虑 MC 算法，在

8 个完整的状态序列中，只有第一个序列中包含状态 A，因此 A 价值仅能通过第一个序列来计算，也就等同于计算该序列中状态 A 的收获：

$$V(A) = G(A) = R_A + \gamma R_B = 0$$

状态 B 的价值，则需要通过状态 B 在 8 个序列中的收获值来平均，其结果是 6/8。因此在使用 MC 算法时，状态 A、B 的价值分别为 6/8 和 0。

再来考虑应用 TD 算法，TD 算法在计算状态序列中某状态价值时是应用其后续状态的预估价值来计算的，在 8 个状态序列中，状态 B 总是出现在终止状态中，因而直接使用终止状态时获得的奖励来计算价值再针对状态序列数做平均，这样得到的状态 B 的价值依然是 6/8。状态 A 由于只存在于第一个状态序列中，因此直接使用包含状态 B 价值的 TD 目标值来得到状态 A 的价值，由于状态 A 的即时奖励为 0，因而计算得到的状态 A 的价值与 B 的价值相同，均为 6/8。

TD 算法在计算状态价值时利用了状态序列中前后状态之间的关系，由于已知信息仅有 8 个完整的状态序列，而且状态 A 的后续状态 100% 是状态 B，而状态 B 式中作为终止状态，有 1/4 的几率获得奖励 0，3/4 的几率获得奖励 1。符合这样状态转移概率的 MDP 如下图所示。可以看出，TD 算法试图构建一个 MDP$\langle S, A, \hat{P}, \hat{R}, \gamma\rangle$ 并使得这个 MDP 尽可能的符合已经产生的状态序列，也就是说 TD 算法将首先根据已有经验估计状态间的转移概率：

$$\hat{P}^a_{s,s'} = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} 1(S_t^k, a_t^k, s_{t+1}^k = s, a, s')$$

同时估计某一个状态的即时奖励：

$$\hat{R}^a_s = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} 1(s_t^k, a_t^k = s, a) r_t^k$$

最后计算该 MDP 的状态函数，如图 4.2 所示。

而 MC 算法则直接依靠完整状态序列的奖励得到的各状态对应的收获来计算状态价值，因而这种算法是以最小化收获与状态价值之间均方差为目标的：

$$\sum_{k=1}^{K} \sum_{t=1}^{T_k} (G_t^k - V(s_t^k))^2$$

通过上面的示例，我们能体会到 TD 算法与 MC 算法之间的另一个差别：TD 算法使用了 MDP 问题的马儿可夫属性，在具有马尔科夫性的环境下更有效；但是 MC 算法并不利用马儿可

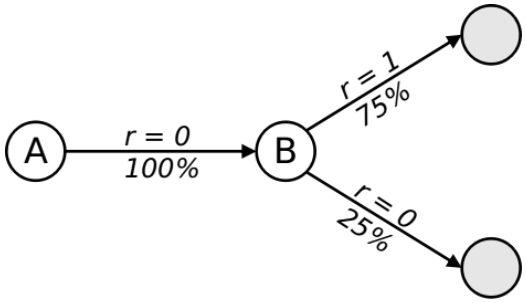


图 4.2: AB 状态 TD 算法构建的 MDP

夫属性，适用范围不限于具有马尔科夫性的环境。

本章阐述的蒙特卡罗 (MC) 学习算法、时序差分 (TD) 学习算法和上一章讲述的动态规划 (DP) 算法都可以用来计算状态价值。他们它们的特点也是十分鲜明的，前两种是在不依赖模型的情况下的常用方法，这其中又以 MC 学习需要完整的状态序列来更新状态价值，TD 学习则不需要完整的状态序列；DP 算法则是基于模型的计算状态价值的方法，它通过计算一个状态 S 所有可能的转移状态 S' 及其转移概率以及对应的即时奖励来计算这个状态 S 的价值。

在是否使用引导数据上，MC 学习并不使用引导数据，它使用实际产生的奖励值来计算状态价值；TD 和 DP 则都是用后续状态的预估价值作为引导数据来计算当前状态的价值。

在是否采样的问题上，MC 和 TD 不依赖模型，使用的都是个体与环境实际交互产生的采样状态序列来计算状态价值的，而 DP 则依赖状态转移概率矩阵和奖励函数，全宽度计算状态价值，没有采样之说。

图 4.3，图 4.4 和图 4.5 非常直观的体现了三种算法的区别，其中：

MC 算法：深度采样学习。一次学习完整经历，使用实际收获更新状态预估价值，如图 4.3 所示。

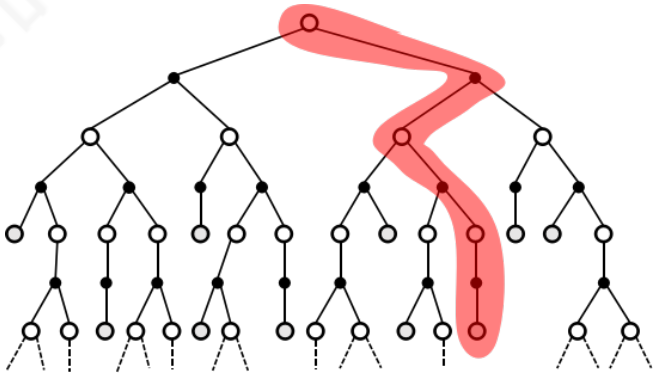


图 4.3: MC 学习深度采样回溯

TD 算法：浅层采样学习。经历可不完整，使用后续状态的预估状态价值预估收获再更新当前状态价值，如图 4.4 所示。

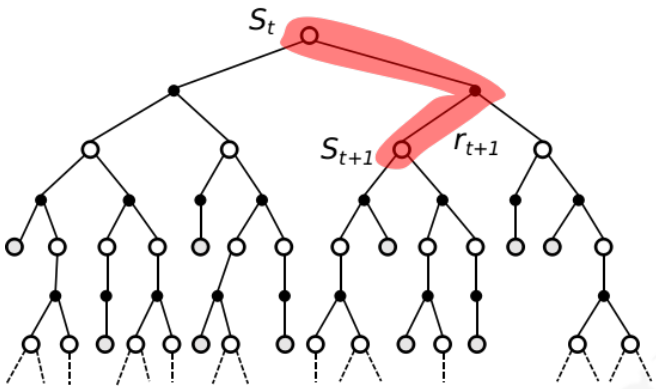


图 4.4: TD 学习浅层采样回溯

DP 算法：浅层全宽度 (采样) 学习。依据模型，全宽度地使用后续状态预估价值来更新当前状态价值，如图 4.5 所示。

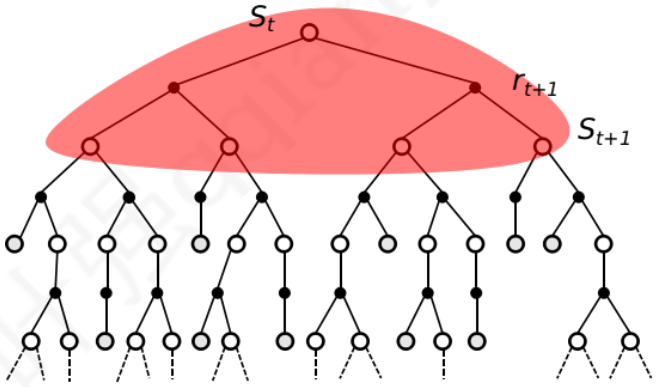


图 4.5: DP 学习浅层全宽度 (采样) 回溯

综合上述三种学习方法的特点，可以小结如下：当使用单个采样，同时不经历完整的状态序列更新价值的算法是 TD 学习；当使用单个采样，但依赖完整状态序列的算法是 MC 学习；当考虑全宽度采样，但对每一个采样经历只考虑后续一个状态时的算法是 DP 学习；如果既考虑所有状态转移的可能性，同时又依赖完整状态序列的，那么这种算法是穷举 (exhausive search) 法。需要说明的是：DP 利用的是整个 MDP 问题的模型，也就是状态转移概率，虽然它并不实际利用采样经历，但它利用了整个模型的规律，因此也被认为是全宽度 (full width) 采样的。

4.3 n 步时序差分学习简介

第二节所介绍的 TD 算法实际上都是 TD(0) 算法，括号内的数字 0 表示的是在当前状态下往前多看 1 步，要是往前多看 2 步更新状态价值会怎样？这就引入了 n-步预测的概念。

n-**步预测**指从状态序列的当前状态 (S_t) 开始往序列终止状态方向观察至状态 S_{t+n-1}，使用这 n 个状态产生的即时奖励 ($R_{t+1}, R_{t+2}, ..., R_{t+n}$) 以及状态 S_{t+n} 的预估价值来计算当前第状态 S_t 的价值。4.6

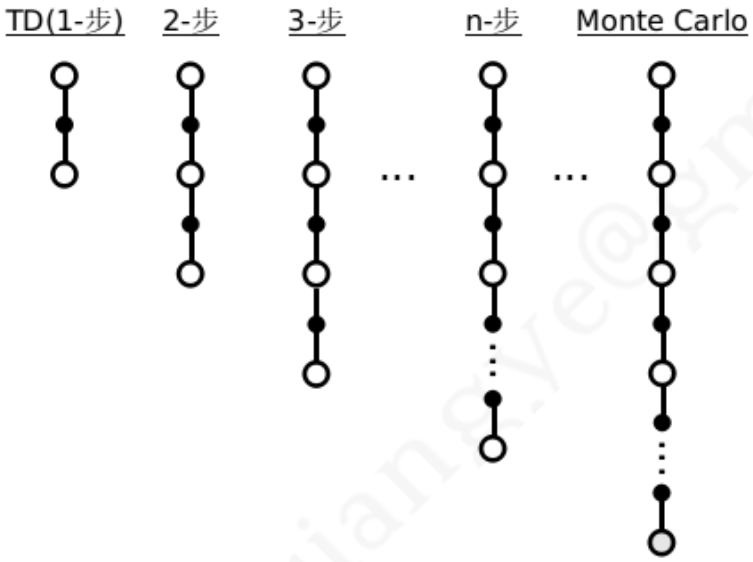


图 4.6: n-步预测

TD 是 TD(0) 的简写，是基于 1-步预测的。根据 n-步预测的定义，可以推出当 n=1,2 和 ∞ 时对应的预测值如表 4.3 所示。从该表可以看出，MC 学习是基于 ∞-步预测的。

表 4.3: n-步收获

n=1	TD 或 TD(0)	$G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1})$
n=2	TD 或 TD(0)	$G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})$
...	...	...
n=∞	MC	$G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \vdots + \gamma^{T-1} R_T$

定义 n-**步收获**为：

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) \tag{4.4}$$

由此可以得到 n-步 TD 学习对应的状态价值函数的更新公式为:

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t^{(n)} - V(S_t) \right) \tag{4.5}$$

从公式 4.4，4.5 可以得到，当 n=1 时等同于 TD(0) 学习，n 取无穷大时等同于 MC 学习。由于 TD 学习和 MC 学习又各有优劣，那么会不会存在一个 n 值使得预测能够充分利用两种学习的优点或者得到一个更好的预测效果呢? 研究认为不同的问题其对应的比较高效的步数不是一成不变的。选择多少步数作为一个较优的计算参数是需要尝试的超参数调优问题。

为了能在不增加计算复杂度的情况下综合考虑所有步数的预测，我们引入了一个新的参数 λ，并定义：λ-**收获**为:

从 n=1 到 ∞ 的所有步收获的权重之和。其中，任意一个 n-步收获的权重被设计为 $(1-\lambda)\lambda^{n-1}$，如图 4.7 所示。通过这样的权重设计，可以得到 λ-收获的计算公式为:

$$G_t^{\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \tag{4.6}$$

对应的 TD(λ) 被描述为:

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t^{(\lambda)} - V(S_t) \right) \tag{4.7}$$

图 4.8 显示了 TD(λ) 中对于 n-收获的权重分配，左侧阴影部分是 3-步收获的权重值，随着 n 的增大，其 n-收获的权重呈几何级数的衰减。当在 T 时刻到达终止状态时，未分配的权重 (右侧阴影部分) 全部给予终止状态的实际收获值。如此设计可以使一个完整的状态序列中所有的 n-步收获的权重加起来为 1，离当前状态越远的收获其权重越小。

前向认识 TD(λ)

TD(λ) 的设计使得在状态序列中，一个状态的价值 $V(S_t)$ 由 $G_t^{(\lambda)}$ 得到，而后者又间接由所有后续状态价值计算得到，因此可以认为更新一个状态的价值需要知道所有后续状态的价值。也就是说，必须要经历完整的状态序列获得包括终止状态的每一个状态的即时奖励才能更新当前状态的价值。这和 MC 算法的要求一样，因此 TD(λ) 算法有着和 MC 方法一样的劣势。λ 取值区间为 [0,1]，当 $\lambda = 1$ 时对应的就是 MC 算法。这个实际计算带来了不便。

反向认识 TD(λ)

反向认识 TD(λ) 为 TD(λ) 算法进行在线实时单步更新学习提供了理论依据。为了解释这一点，需要先引入 “效用迹” 这个概念。我们通过一个之前的一个例子来解释这个问题 (图 4.9)。老鼠在依次连续接受了 3 次响铃和 1 次亮灯信号后遭到了电击，那么在分析遭电击的原因时，到底是响铃的因素较重要还是亮灯的因素更重要呢?

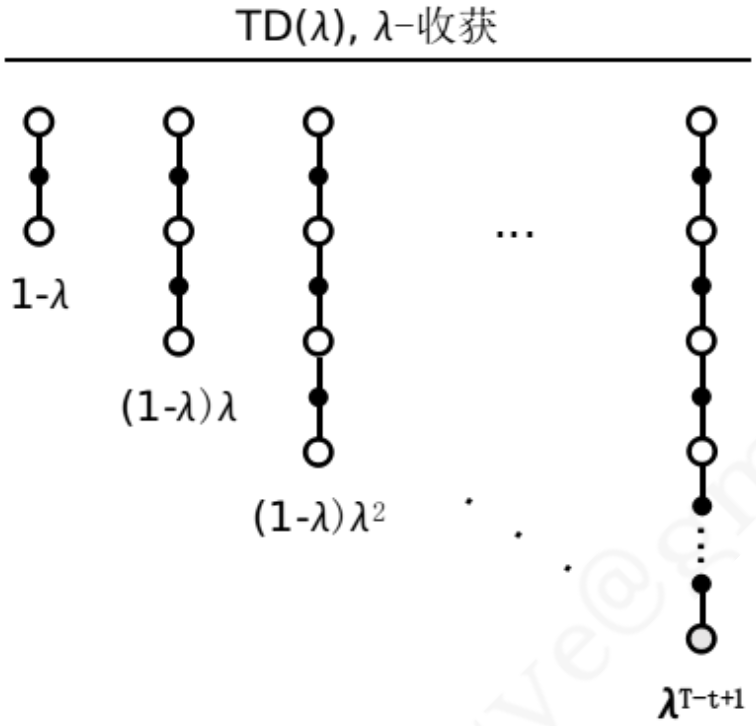


图 4.7: λ-收获权重权重分配

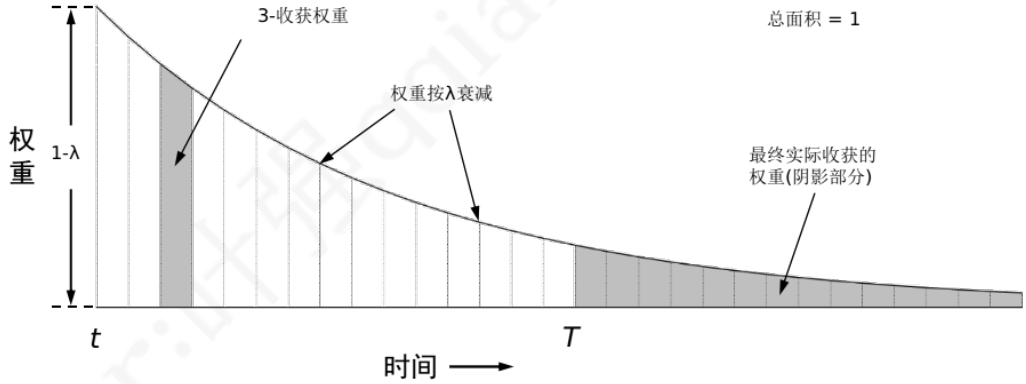


图 4.8: TD(λ) 对于权重分配的图解

如果把老鼠遭到电击的原因认为是之前接受了**较多次数**的响铃，则称这种归因为频率启发 (frequency heuristic) 式；而把电击归因于**最近**少数几次状态的影响，则称为就近启发 (recncy heuristic) 式。如果给每一个状态引入一个数值：**效用** (eligibility, E) 来表示该状态对后续状态的影响，就可以同时利用到上述两个启发。而所有状态的效用值总称为**效用迹** (eligibility traces, ES)。

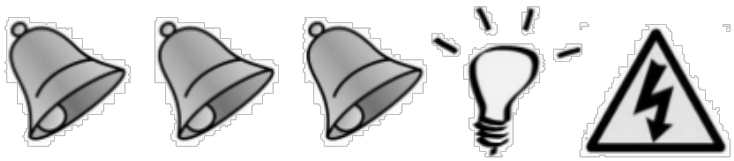


图 4.9: 是响铃还是亮灯引起了老鼠遭电击

定义：

$$\begin{aligned} E_0(s) &= 0 \\ E_t(s) &= \gamma\lambda E_{t-1}(s) + 1(S_t = s),\ \gamma, \lambda \in [0, 1] \end{aligned} \tag{4.8}$$

公式 4.8中的 $1(S_t = s)$ 是一个真判断表达式，表示当 $S_t = s$ 时取值为 1，其余条件下取值为 0。

图 4.10 给出了效用 E 对于时间 t 的一个可能的曲线：

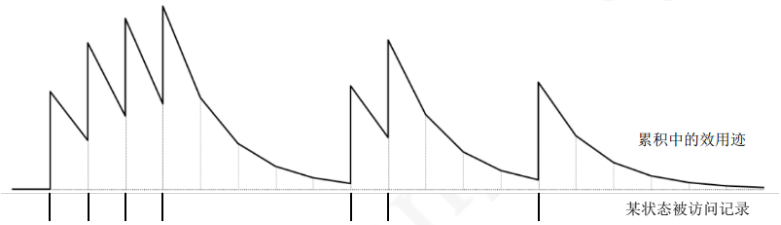


图 4.10: 状态的一个可能的效用时间曲线

该图横坐标是时间，横坐标下有竖线的位置代表当前时刻的状态为 s，纵坐标是效用的值。可以看出当某一状态连续出现，E 值会在一定衰减的基础上有一个单位数值的提高，此时认为该状态将对后续状态的影响较大，如果该状态很长时间没有经历，那么该状态的的 E 值将逐渐趋于 0，表明该状态对于较远的后续状态价值的影响越来越少。

需要指出的是，针对每一个状态存在一个 E 值，且 E 值并不需要等到状态序列到达终止状态才能计算出来，它是根据已经经过的状态序列来计算得到，并且在每一个时刻都对每一个状态进行一次更新。E 值存在饱和现象，有一个瞬时最高上限：

$$E_{sat} = \frac{1}{1 - \gamma\lambda}$$

E 值是一个非常符合神经科学相关理论的、非常精巧的设计。可以把它看成是神经元的一个参数，它反映了神经元对某一刺激的敏感性和适应性。神经元在接受刺激时会有反馈，在持续刺激时反馈一般也比较强，当间歇一段时间不刺激时，神经元又逐渐趋于静息状态；同时不论如何增加刺激的频率，神经元有一个最大饱和反馈。

如果我们在更新状态价值时把该状态的效用同时考虑进来，那么价值更新可以表示为：

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \\ V(s) &\leftarrow V(s) + \alpha\delta_t E_t(s)\end{aligned} \tag{4.9}$$

当 $\lambda = 0$ 时，$S_t = s$ 一直成立，此时价值更新等同于 TD(0) 算法：

$$V(S_t) \leftarrow V(S_t) + \alpha\delta_t$$

当 $\lambda = 1$ 时，在每完成一个状态序列后更新状态价值时，其完全等同于 MC 学习；但在引入了效用迹后，可以每经历一个状态就更新状态的价值，这种实时更新的方法并不完全等同于 MC。

当 $\lambda \in (0, 1)$ 时，在每完成一个状态序列后更新价值时，基于前向认识的 TD(λ) 与基于反向认识的 TD(λ) 完全等效；不过在进行在线实时学习时，两者存在一些差别。这里就不在详细展开了。

4.4 编程实践：蒙特卡罗学习评估 21 点游戏的玩家策略

本章的编程实践将使用 MC 学习来评估二十一点游戏中一个玩家的策略。为了完成这个任务，我们需要先了解二十一点游戏的规则，并构建一个游戏场景让庄家和玩家在一个给定的策略下进行博弈生成对局数据。这里的对局数据在强化学习看来就是一个个完整的状态序列组成的集合。然后我们使用本章介绍的蒙特卡罗算法来评估其中玩家的策略。本节的难点不在于蒙特卡罗学习算法的实现，而是对游戏场景的实现并生成让蒙特卡罗学算法学习的多个状态序列。

4.4.1 二十一点游戏规则

二十一点游戏是一个比较经典的对弈游戏，其规则也有各种不同的版本，为了简化，本文仅介绍由一个庄家 (dealer) 和一个普通玩家 (player，下文简称玩家) 共 2 位游戏者参与的一个比较基本的规则版本。游戏使用一副除大小王以外的 52 张扑克牌，游戏者的目标是使手中的牌的点数之和不超过 21 点且尽量大。其中 2-10 的数字牌点数就是牌面的数字，J,Q,K 三类牌均记为 10 点，A 既可以记为 1 也可以记为 11，由游戏者根据目标自己决定。牌的花色对于计算点数没有影响。

开局时，庄家将依次连续发 2 张牌给玩家和庄家，其中庄家的第一张牌是明牌，其牌面信息对玩家是开放的，庄家从第二张牌开始的其它牌的信息不对玩家开放。玩家可以根据自己手中牌

的点数决定是否继续叫牌 (twist) 或停止叫牌 (stick), 玩家可以持续叫牌，但一旦手中牌点数超过 21 点则停止叫牌。当玩家停止叫牌后，庄家可以决定是否继续叫牌。如果庄家停止叫牌，对局结束，双方亮牌计算输赢。

计算输赢的规则如下：如果双方点数均超过 21 点或双方点数相同，则和局；一方 21 点另一方不是 21 点，则点数为 21 点的游戏者赢；如果双方点数均不到 21 点，则点数离 21 点近的玩家赢。

4.4.2 将二十一点游戏建模为强化学习问题

为了讲解基于完整状态序列的蒙特卡罗学习算法，我们把二十一点游戏建模成强化学习问题，设定由下面三个参数来集体描述一个状态：庄家的明牌 (第一张牌) 点数；玩家手中所有牌点数之和；玩家手中是否还有“可用 (useable)”的 A(ace)。前两个比较好理解，第三个参数是与玩家策略相关的，玩家是否有 A 这个比较好理解，可用的 A 指的是玩家手中的 A 按照目标最大化原则是否没有被计作 1 点，如果这个 A 没有被记为 1 点而是计为了 11 点，则成这个 A 为可用的 A，否则认为没有可用的 A，当然如果玩家手中没有 A，那么也被认为是没有可用的 A。例如玩家手中的牌为“A,3,6”，那么此时根据目标最大化原则，A 将被计为 11 点，总点数为 20 点，此时玩家手中的 A 称为可用的 A。加入玩家手中的牌为“A，5,7”，那么此时的 A 不能被计为 11 点只能按 1 计，相应总点数被计为 13 点，否则总点数将为 23 点，这时的 A 就不能称为可用的 A。

根据我们对状态的设定，我们使用由三个元素组成的元组来描述一个状态。例如使用 (10,15, 0) 表示的状态是庄家的明牌是 10，玩家手中的牌加起来点数是 15，并且玩家手中没有可用的 A，(A,17,1) 表述的状态是庄家第一张牌为 A，玩家手中牌总点数为 17, 玩家手中有可用的 A。这样的状态设定不考虑玩家手中的具体牌面信息，也不记录庄家除第一张牌外的其它牌信息。所有可能的状态构成了状态空间。

该问题的行为空间比较简单，玩家只有两种选择：“继续叫牌”或“停止叫牌”。

该问题中的状态如何转换取决于游戏者的行为以及后续发给游戏者的牌，状态间的转移概率很难计算。

可以设定奖励如下：当棋局未结束时，任何状态对应的奖励为 0；当棋局结束时，如果玩家赢得对局，奖励值为 1，玩家输掉对局，奖励值为-1，和局是奖励为 0。

本问题中衰减因子 $\gamma = 1$。

游戏者在选择行为时都会遵循一个策略。在本例中，庄家遵循的策略是只要其手中的牌点数达到或超过 17 点就停止叫牌。我们设定玩家遵循的策略是只要手中的牌点数不到 20 点就会继续叫牌，点数达到或超过 20 点就停止叫牌。

我们的任务是评估玩家的这个策略，即计算在该策略下的状态价值函数，也就是计算状态空间中的每一个状态其对应的价值。

4.4.3 游戏场景的搭建

首先来搭建这个游戏场景，实现生成对局数据的功能，我们要实现的功能包括：统计游戏者手中牌的总点数、判断当前牌局信息对应的奖励、实现庄家与玩家的策略、模拟对局的过程生成对局数据等。为了能尽可能生成较符合实际的对局数据，我们将循环使用一副牌，对局过程中发牌、洗牌、收集已使用牌等过程都将得到较为真实的模拟。我们使用面向对象的编程思想，通过构建游戏者类和游戏场景类来实现上述功能。

首先我们导入一些必要的库：

```
from random import shuffle
from queue import Queue
from tqdm import tqdm
import math
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from utils import str_key, set_dict, get_dict
```

经过初步的分析和整理，我们认为一个单纯的二十一点游戏者应该至少能记住对局过程中手中牌的信息，知道自己的行为空间，还应该能辨认单张牌的点数以及手中牌的总点数，此外游戏者能够接受发给他的牌以及一局结束后将手中的牌扔掉等。为此我们编写了一个名称为 Gamer 的游戏者类。代码如下：

```
class Gamer():
    '''游戏者
    '''
    def __init__(self, name = "", A = None, display = False):
        self.name = name
        self.cards = [] # 手中的牌
        self.display = display # 是否显示对局文字信息
        self.policy = None # 策略
        self.learning_method = None # 学习方法
        self.A = A # 行为空间

```

```
    def __str__(self):
        return self.name

    def _value_of(self, card):
        '''根据牌的字符判断牌的数值大小，A被输出为1，JQK均为10，其余按牌字符对应
           的数字取值
        Args:
            card: 牌面信息 str
        Return:
            牌的大小数值 int, A 返回 1
        '''
        try:
            v = int(card)
        except:
            if card == 'A':
                v = 1
            elif card in ['J','Q','K']:
                v = 10
            else:
                v = 0
        finally:
            return v

    def get_points(self):
        '''统计一手牌分值，如果使用了A的1点，同时返回True
        Args:
            cards 庄家或玩家手中的牌 list ['A','10','3']
        Return
            tuple (返回牌总点数,是否使用了可复用Ace)
            例如['A','10','3'] 返回 (14, False)
               ['A','10'] 返回  (21, True)
        '''
        num_of_useable_ace = 0 # 默认没有拿到Ace
        total_point = 0 # 总值
        cards = self.cards
        if cards is None:
            return 0, False
        for card in cards:
            v = self._value_of(card)
            if v == 1:
```

```
                num_of_useable_ace += 1
                v = 11
            total_point += v
        while total_point > 21 and num_of_useable_ace > 0:
            total_point -= 10
            num_of_useable_ace -= 1
        return total_point, bool(num_of_useable_ace)

    def receive(self, cards = []): # 玩家获得一张或多张牌
        cards = list(cards)
        for card in cards:
            self.cards.append(card)

    def discharge_cards(self): # 玩家把手中的牌清空，扔牌
        '''扔牌
        '''
        self.cards.clear()


    def cards_info(self): # 玩家手中牌的信息
        '''
        显示牌面具体信息
        '''
        self._info("{}{}现在的牌:{}\n".format(self.role, self,self.cards))

    def _info(self, msg):
        if self.display:
            print(msg, end="")
```

在上面的代码中，构造一个游戏者可以提供三个参数，分别是该游戏者的姓名 (name)，行为空间 (A) 和是否在终端显示具体信息 (display)。其中设置第三个参数主要是由于调试和展示的需要，我们希望一方面游戏在生成大量对局信息时不要输出每一局的细节，另一方面在观察细节时希望能在终端给出某时刻庄家和玩家手中具体牌的信息以及他们的行为等。我们还给游戏者增加了一些辅助属性，比如游戏者姓名、策略、学习方法等，还设置了一个 display 以及一些显示信息的方法用来在对局中在终端输出对局信息。在计算单张牌面点数的时候，借用了异常处理。在统计一手牌的点数时，要考虑到可能出现多张 A 的情况。读者可以输入一些测试牌的信息观察这两个方法的输出。

在二十一点游戏中，庄家和玩家都是一个游戏者，我们可以从 Gamer 类继承出 Dealer 类和 Player 类分别表示庄家和普通玩家。庄家和普通玩家的区别在于两者的角色不同、使用的策略不同。其中庄家使用固定的策略，他还能显示第一张明牌给其他玩家。在本章编程实践中，玩家则使用最基本的策略，由于我们的玩家还要进行基于蒙特卡罗算法的策略评估，他还需要具备构建一个状态的能力。我们扩展的庄家类如下：

```
class Dealer(Gamer):
    '''庄家
    '''
    def __init__(self, name = "", A = None, display = False):
        super(Dealer,self).__init__(name, A, display)
        self.role = "庄家" # 角色
        self.policy = self.dealer_policy # 庄家的策略

    def first_card_value(self): # 显示第一张明牌
        if self.cards is None or len(self.cards) == 0:
            return 0
        return self._value_of(self.cards[0])

    def dealer_policy(self, Dealer = None): # 庄家策略的细节
        action = ""
        dealer_points, _ = self.get_points()
        if dealer_points >= 17:
            action = self.A[1] # "停止叫牌"
        else:
            action = self.A[0] # "继续叫牌"
        return action
```

在庄家类的构造方法中声明其基类是游戏者 (Gamer)，这样他就具备了游戏者的所有属性和方法了。我们给庄家贴了个“庄家”的角色标签，同时指定了其策略，在具体的策略方法中，规定庄家的牌只要达到或超过 17 点就不再继续叫牌。玩家类的代码如下：

```
class Player(Gamer):
    '''玩家
    '''
    def __init__(self, name = "", A = None, display = False):
        super(Player, self).__init__(name, A, display)
        self.policy = self.naive_policy
```

```
        self.role = "玩家"

    def get_state(self, dealer):
        dealer_first_card_value = dealer.first_card_value()
        player_points, useable_ace = self.get_points()
        return dealer_first_card_value, player_points, useable_ace

    def get_state_name(self, dealer):
        return str_key(self.get_state(dealer))

    def naive_policy(self, dealer=None):
        player_points, _ = self.get_points()
        if player_points < 20:
            action = self.A[0]
        else:
            action = self.A[1]
        return action
```

类似的,我们的玩家类也继承子游戏者 (Gamer),指定其策略为最原始的策略 (naive_policy),规定玩家只要点数小于 20 点就会继续叫牌。玩家同时还会根据当前局面信息得到当前局面的状态，为策略评估做准备。

至此游戏者这部分的建模工作就完成了，接下来将准备游戏桌、游戏牌、组织游戏对局、判定输赢等功能。我们把所有的这些功能包装在一个名称为 Arena 的类中。Arena 类的构造方法如下:

```
class Arena():
    '''负责游戏管理
    '''
    def __init__(self, display = None, A = None):
        self.cards = ['A','2','3','4','5','6','7','8','9','10','J','Q',"K"]*4
        self.card_q = Queue(maxsize = 52) # 洗好的牌
        self.cards_in_pool = [] # 已经用过的公开的牌
        self.display = display
        self.episodes = [] # 产生的对局信息列表
        self.load_cards(self.cards)# 把初始状态的52张牌装入发牌器
        self.A = A # 获得行为空间
```

Arena 类接受两个参数，这两个参数与构建游戏者的参数一样。Arena 包含的属性有：一副不包括大小王、花色信息的牌 (cards)、一个装载洗好了的牌的发牌器 (cards_q)，一个负责收集已经使用过的废牌的池子 (cards_in_pool)，一个记录了对局信息的列表 (episodes)，还包括是否显示具体信息以及游戏的行为空间等。在构造一个 Arena 对象时，我们同时把一副新牌洗好并装进了发牌器，这个工作在 load_cards 方法里完成。我们来看看这个方法的细节。

```
    def load_cards(self, cards):
        '''把收集的牌洗一洗，重新装到发牌器中
        Args:
            cards 要装入发牌器的多张牌 list
        Return:
            None
        '''
        shuffle(cards) # 洗牌
        for card in cards:# deque数据结构只能一个一个添加
            self.card_q.put(card)
        cards.clear() # 原来的牌清空
        return
```

这个方法接受一个参数 (cards)，多数时候我们将 cards_in_pool 传给这个方法，也就是把桌面上已使用的废牌收集起来传给这个方法，该方法将首先把这些牌的次序打乱，模拟洗牌操作。随后将洗好的牌放入发牌器。完成洗牌装牌功能。Arena 应具备根据庄家和玩家手中的牌的信息判断当前谁赢谁输的能力，该能力通过如下的方法 (reward_of) 来实现：

```
def reward_of(self, dealer, player):
        '''判断玩家奖励值，附带玩家、庄家的牌点信息
        '''
        dealer_points, _ = dealer.get_points()
        player_points, useable_ace = player.get_points()
        if player_points > 21:
            reward = -1
        else:
            if player_points > dealer_points or dealer_points > 21:
                reward = 1
            elif player_points == dealer_points:
                reward = 0
            else:
                reward = -1
```

```
        return reward, player_points, dealer_points, useable_ace
```

该方法接受庄家和玩家为参数，计算对局过程中以及对局结束时牌局的输赢信息 (reward) 后，同时还返回当前玩家、庄家具体的总点数以及玩家是否有可用的 A 等信息。

下面的方法实现了 Arena 对象如何向庄家或玩家发牌的功能：

```
    def serve_card_to(self, player, n = 1):
            '''给庄家或玩家发牌，如果牌不够则将公开牌池的牌洗一洗重新发牌
            Args:
                player 一个庄家或玩家
                n 一次连续发牌的数量
            Return:
                None
            '''
            cards = []  #将要发出的牌
            for _ in range(n):
                # 要考虑发牌器没有牌的情况
                if self.card_q.empty():
                    self._info("\n发牌器没牌了，整理废牌，重新洗牌;")
                    shuffle(self.cards_in_pool)
                    self._info("一共整理了{}张已用牌，重新放入发牌器\n".format(
                        len(self.cards_in_pool)))
                    assert(len(self.cards_in_pool) > 20) # 确保一次能收集较多的
                        牌
                    #代码编写不合理时，可能会出现即使某一玩家爆点了也还持续的叫
                        牌，会导致玩家手中牌变多而发牌器和已使用的牌都很少，需避
                        免这种情况。
                    self.load_cards(self.cards_in_pool) # 将收集来的用过的牌洗好
                        送入发牌器重新使用
                cards.append(self.card_q.get()) # 从发牌器发出一章牌
            self._info("发了{}张牌({})给{}{};".format(n, cards, player.role,
                player))
            #self._info(msg)
            player.receive(cards) # 某玩家接受发出的牌
            player.cards_info()

    def _info(self, message):
        if self.display:
```

```
            print(message, end="")
```

这个方法 (serve_card_to) 接受一个玩家 (player) 和一个整数 (n) 作为参数，表示向该玩家一次发出一定数量的牌，在发牌时如果遇到发牌器里没有牌的情况时会将已使用的牌收集起来洗好后送入发牌器，随后在把需要数量的牌发给某一玩家。代码中的方法 (_info) 负责根据条件在终端输出对局信息。

当一局结束时，Arena 对象还负责把玩家手中的牌回收至已使用的废牌区，这个功能由下面这个方法来完成：

```
    def recycle_cards(self, *players):
        '''回收玩家手中的牌到公开使用过的牌池中
        '''
        if len(players) == 0:
            return
        for player in players:
            for card in player.cards:
                self.cards_in_pool.append(card)
            player.discharge_cards() # 玩家手中不再留有这些牌
```

剩下一个最关键的功能就是，如何让庄家和玩家进行一次对局，编写下面的方法来实现这个功能：

```
    def play_game(self, dealer, player):
        '''玩一局21点，生成一个状态序列以及最终奖励（中间奖励为0）
        Args:
            dealer/player 庄家和玩家
        Returns:
            tuple: episode, reward
        '''
        self._info("========= 开始新一局 =========\n")
        self.serve_card_to(player, n=2) # 发两张牌给玩家
        self.serve_card_to(dealer, n=2) # 发两张牌给庄家
        episode = [] # 记录一个对局信息
        if player.policy is None:
            self._info("玩家需要一个策略")
            return
```

```
        if dealer.policy is None:
            self._info("庄家需要一个策略")
            return
        while True:
            action = player.policy(dealer)
            # 玩家的策略产生一个行为
            self._info("{}{}选择:{};".format(player.role, player, action))
            episode.append((player.get_state_name(dealer), action)) # 记录一个(s
                ,a)
            if action == self.A[0]: # 继续叫牌
                self.serve_card_to(player) # 发一张牌给玩家
            else: # 停止叫牌
                break
        # 玩家停止叫牌后要计算下玩家手中的点数，玩家如果爆了，庄家就不用继续了
        reward, player_points, dealer_points, useable_ace = self.reward_of(
            dealer, player)

        if player_points > 21:
            self._info("玩家爆点{}输了，得分:{}\n".format(player_points, reward)
                )
            self.recycle_cards(player, dealer)
            self.episodes.append((episode, reward)) # 预测的时候需要形成episode
                list后集中学习V
            # 在蒙特卡罗控制的时候，可以不需要episodes list,生成一个episode学习
                一个，下同
            self._info("========== 本局结束 ==========\n")
            return episode, reward
        # 玩家并没有超过21点
        self._info("\n")
        while True:
            action = dealer.policy() # 庄家从其策略中获取一个行为
            self._info("{}{}选择:{};".format(dealer.role, dealer, action))
            # 状态只记录庄家第一章牌信息，此时玩家不再叫牌，(s,a)不必重复记录
            if action == self.A[0]: # 庄家"继续叫牌":
                self.serve_card_to(dealer)
            else:
                break
        # 双方均停止叫牌了
        self._info("\n双方均停止叫牌;\n")
        reward, player_points, dealer_points, useable_ace = self.reward_of(
```

```
            dealer, player)
        player.cards_info()
        dealer.cards_info()
        if reward == +1:
            self._info("玩家赢了!")
        elif reward == -1:
            self._info("玩家输了!")
        else:
            self._info("双方和局!")
        self._info("玩家{}点,庄家{}点\n".format(player_points, dealer_points))
        self._info("========== 本局结束 ==========\n")
        self.recycle_cards(player, dealer) # 回收玩家和庄家手中的牌至公开牌池
        self.episodes.append((episode, reward)) # 将刚才产生的完整对局添加值状态
            序列列表，蒙特卡罗控制不需要
        return episode, reward
```

这段代码虽然比较长，但里面包含许多反映对局过程的信息，使得代码也比较容易理解。该方法接受一个庄家一个玩家为参数，产生一次对局，并返回该对局的详细信息。需要指出的是玩家的策略要做到在玩家手中的牌超过 21 点时强制停止叫牌。其次在玩家停止叫牌后，Arena 对局面进行一次判断，如果玩家超过 21 点则本局结束，否则提示庄家选择行为。当庄家停止叫牌后后，Arena 对局面再次进行以此判断，结束对局并将该对局产生的详细信息记录一个 episode 对象，并附加地把包含了该局信息的 episode 对象联合该局的最终输赢 (奖励) 登记至 Arena 的成员属性 episodes 中。

有了生成一次对局的方法，我们编写下面的代码来一次性生成多个对局：

```
    def play_games(self, dealer, player, num = 2, show_statistic = True):
        '''一次性玩多局游戏
        '''
        results = [0, 0, 0]# 玩家负、和、胜局数
        self.episodes.clear()
        for i in tqdm(range(num)):
            episode, reward = self.play_game(dealer, player)
            results[1+reward] += 1
            if player.learning_method is not None:
                player.learning_method(episode ,reward)
        if show_statistic:
            print("共玩了{}局，玩家赢{}局，和{}局，输{}局，胜率：{:.2f},不输率
```

```
                :{:.2f}"\
13              .format(num, results[2],results[1],results[0],results[2]/num,(
                    results[2]+results[1])/num))
14          return
15
16      def _info(self, message):
17          if self.display:
18              print(message, end = "")
```

该方法接受一个庄家、一个玩家、需要产生的对局数量、以及是否显示多个对局的统计信息这四个参数，生成指定数量的对局信息，这些信息都保存在 Arena 的 episodes 对象中。为了兼容具备学习能力的玩家，我们设置了在每一个对局结束后，如果玩家能够从中学习，则提供玩家一次学习的机会，在本章中的玩家不具备从对局中学习改善策略的能力，这部分内容将在下一章详细讲解。如果参数设置为显示统计信息，则会在指定数量的对局结束后显示一共对局多少，玩家的胜率等。

4.4.4 生成对局数据

至此，我们所有的准备工作就完成了。下面的代码将生成一个庄家、一个玩家，一个 Arena 对象，并进行 20 万次的对局：

```
A=["继续叫牌","停止叫牌"]
display = False
# 创建一个玩家一个庄家，玩家使用原始策略，庄家使用其固定的策略
player = Player(A = A, display = display)
dealer = Dealer(A = A, display = display)
# 创建一个场景
arena = Arena(A = A, display = display)
# 生成num个完整的对局

arena.play_games(dealer, player, num = 200000)

# 将输出类似如下的结果
# 100%|██████████| 200000/200000 [00:18<00:00, 11014.64it/s]
# 共玩了200000局，玩家赢58647局，和11250局，输130103局，胜率：0.29,不输率:0.35
```

4.4.5 策略评估

对局生成的数据均保存在对象 arena.episodes 中，接下来的工作就是使用这些数据来对 player 的策略进行评估，下面的代码完成这部分功能：

```
# 统计个状态的价值，衰减因子为1，中间状态的即时奖励为0，递增式蒙特卡罗评估
def policy_evaluate(episodes, V, Ns):
    for episode, r in episodes:
        for s, a in episode:
            ns = get_dict(Ns, s)
            v = get_dict(V, s)
            set_dict(Ns, ns+1, s)
            set_dict(V, v+(r-v)/(ns+1), s)

V = {} # 状态价值字典
Ns = {} # 状态被访问的次数节点
policy_evaluate(arena.episodes, V, Ns) # 学习V值
```

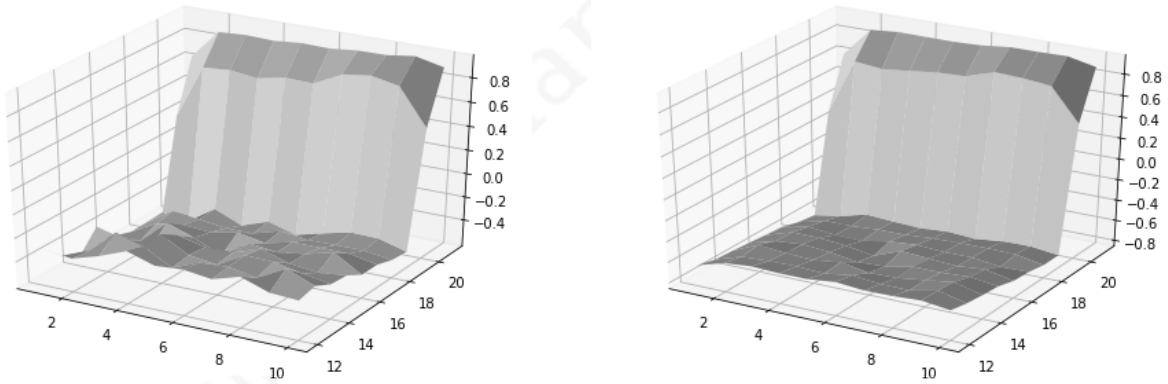
其中，V 和 Ns 保存着蒙特卡罗策略评估进程中的价值和统计次数数据，我们使用的是每次访问计数的方法。我们还可以编写如下的方法将价值函数绘制出来：

```
def draw_value(value_dict, useable_ace = True, is_q_dict = False, A = None):
    # 定义figure
    fig = plt.figure()
    # 将figure变为3d
    ax = Axes3D(fig)
    # 定义x, y
    x = np.arange(1, 11, 1) # 庄家第一张牌
    y = np.arange(12, 22, 1) # 玩家总分数
    # 生成网格数据
    X, Y = np.meshgrid(x, y)
    # 从V字典检索Z轴的高度
    row, col = X.shape
    Z = np.zeros((row,col))
    if is_q_dict:
        n = len(A)
    for i in range(row):
        for j in range(col):
            state_name = str(X[i,j])+"_"+str(Y[i,j])+"_"+str(useable_ace)
```

```
            if not is_q_dict:
                Z[i,j] = get_dict(value_dict, state_name)
            else:
                assert(A is not None)
                for a in A:
                    new_state_name = state_name + "_" + str(a)
                    q = get_dict(value_dict, new_state_name)
                    if q >= Z[i,j]:
                        Z[i,j] = q
    # 绘制3D曲面
    ax.plot_surface(X, Y, Z, rstride = 1, cstride = 1, color="lightgray")
    plt.show()

draw_value(V, useable_ace = True, A = A) # 绘制有可用的A时状态价值图
draw_value(V, useable_ace = False, A = A) # 绘制无可用的A时状态价值图
```

结果如下：



(a) 有可用的 Ace　　(b) 没有可用的 Ace

图 4.11: 二十一点游戏玩家原始策略的价值函数 (20 万次迭代)

我们可以设置各对象 display 的值为 True，来生成少量对局并输出对局的详细信息：

```
# 观察几局对局信息
display = True
player.display, dealer.display, arena.display = display, display, display
arena.play_games(dealer, player, num = 2)

```

```
# 将输出类似如下的结果：
# ========= 开始新一局 =========
# 发了2张牌(['4', '8'])给玩家;玩家现在的牌:['4', '8']
# 发了2张牌(['10', 'K'])给庄家;庄家现在的牌:['10', 'K']
# 玩家选择:继续叫牌;发了1张牌(['K'])给玩家;玩家现在的牌:['4', '8', 'K']
# 玩家选择:停止叫牌;玩家爆点22输了，得分:-1
# ========= 本局结束 ==========
# ========= 开始新一局 =========
# 发了2张牌(['9', 'A'])给玩家;玩家现在的牌:['9', 'A']
# 发了2张牌(['5', '7'])给庄家;庄家现在的牌:['5', '7']
# 玩家选择:停止叫牌;
# 庄家选择:继续叫牌;发了1张牌(['7'])给庄家;庄家现在的牌:['5', '7', '7']
# 庄家选择:停止叫牌;
# 双方均了停止叫牌;
# 玩家现在的牌:['9', 'A']
# 庄家现在的牌:['5', '7', '7']
# 玩家赢了!玩家20点,庄家19点
# ========= 本局结束 ==========
# 共玩了2局，玩家赢1局，和0局，输1局，胜率：0.50,不输率:0.50
```

本节编程实践中，我们构建了游戏者基类并扩展形成了庄家类和玩家类来模拟玩家的行为，同时构建了游戏场景类来负责进行对局管理。在此基础上使用蒙特卡罗算法对游戏中玩家的原始策略进行了评估。在策略评估环节，我们并没有把价值函数 (字典)、计数函数 (字典) 以及策略评估方法设计为玩家类的成员对象和成员方法，这只是为了讲解的方便，读者完全可以将它们设计为玩家的成员变量和方法。下一章的编程实践中，我们将继续通过二十一点游戏介绍如何使用蒙特卡罗控制寻找最优策略，本节建立的 Dealer, Player 和 Arena 类将得到复用和扩展。