# 3.9-A 强化学习——表格型求解方法

强化学习（Reinforcement learning，RL）是机器学习的一个领域，关注智能体（intelligent agents）应该如何在一个环境（environment）中采取动作（actions），以最大化累积收益（cumulative reward）的概念。监督学习、无监督学习和强化学习是机器学习三种基本的范式。RL 的重点是在试探（exploration）未知领域和开发（exploitation）现有知识之间找到平衡。环境通常以马尔可夫决策过程（Markov decision process，MDP）的形式表述[1]。对于拥有较小状态空间的问题（简单的 RL 问题），是指其状态和动作空间小到可以用数组（arrays）或者表格（tables）的形式表示的价值函数。通常，这类简单问题可以找到精确解，即最优价值函数和最优策略；对于拥有较大状态空间的问题，表格型的方法不仅需要解决所需内存问题，同时需要更长的计算时间，甚至近乎于无限的时间。因此，面对巨大的状态空间，并不期望找到最优策略或最优价值函数，而是用泛化方法（generalization），通常称为函数逼近（function approximation），从一个预期的函数（例如一个价值函数）中获取实列，并试图对它们进行泛化来逼近整个函数，从而找到一个比较好的近似解。

用表格型求解的简单 RL 问题，可以更方便理解 RL 的概念、一般架构、原理和计算的基本方法。从*Reinforcement Learning - An Introduction（Sutton, R. S., Barto, A. G. ）*[2]经典著作提取关键核心内容，结合 Shangtong Zhang 等对该著作的代码书写[3]阐述表格型求解方法的强化学习。

## 3.9.1 多（）臂赌博机与动作-价值方法（Action-value Methods）

### 3.9.1.1 多（）臂赌博机（Multi-armed Bandits）问题和算法实现

臂赌博机问题，源于想象赌徒在一排老虎机（单臂赌博机）前，决定玩哪台机器，每台机器玩多少次，以什么顺序玩，及是继续使用当前的机器还是尝试不同的机器。每台机器从特定于该机器的概率分布中提供随机收益（这是先验未知的），而赌徒的目标是通过拉动一系列老虎机的控制杆以获得最大的收益。在每次试验中，赌徒面临的关键权衡是“开发（exploitation）”具有当前最高预期收益估计的机器，还是“试探（exploration）”以获得更多关于其它机器预期收益的信息[4]。拉动不同老虎机控制杆的动作（actions），每一个动作在被选择时都有一个期望或平均收益/奖励（reward），称为这个动作的价值（value）。将在t时刻选择的动作记作At​，并将对应的收益记作Rt​。任一动作a对应的真实价值记作q∗​(a)，是给定动作a时收益的期望： q∗​(a)≐E[Rt​∣At​=a]。虽然并不能确切的知道动作的价值，但是可以进行估计，将动作a在时刻t时的价值估计记作Qt​(a)，且希望该值趋近q∗​(a)（真实价值/期望收益）。

开发/试探的权衡中，“开发”是利用当前所知关于动作价值的知识（信息），选择具有最大估计价值的动作，以获得基于已有知识条件下的最大收益；而“试探”则是尝试随机动作，不受已有动作价值的约束，来探索未知环境，选择的动作不再一定是具有最大估计价值的动作。“开发”为最大化当前时刻的期望收益，但长远来看，“试探”可能会带来总体收益的最大化。因为“开发”和“试探”不可能同时进行，因此二者之间存在冲突，需要寻找权衡二者之间关系以最大化预期累积收益的平衡方法。通过价值的估计选择动作的方法统称为动作-价值方法。动作价值的真实值是选择这个动作时的期望收益。因此可以通过计算实际收益的平均值来估计动作的价值，有 sum of rewards when  taken prior to  number of times  taken prior to Qt​(a)≐ number of times a taken prior to t sum of rewards when a taken prior to t​=∑i=1t−1​1Ai​=a​∑i=1t−1​Ri​⋅1Ai​=a​​,时刻前通过执行动作 得到的收益总和时刻前执行动作的次数 Qt​(a)≐t 时刻前执行动作 a 的次数 t 时刻前通过执行动作 a 得到的收益总和 ​=∑i=1t−1​1Ai​=a​∑i=1t−1​Ri​⋅1Ai​=a​​），式中， 1predicate ​表示断言（predicate）为真其值为1，否则为0的随机变量。当分母为0时，将Qt​(a)定义为某个默认值，例如值为0；当分母趋向于无穷大时，根据大数定律（the law of large numbers），Qt​(a)会收敛到q∗​(a)。因为每次对一个动作价值的估计，都为该动作已有和当前收益的均值，所以该种估计动作价值的方法称为采样平均方法（sample-average method）（采样平均方法为估计动作价值的一种方法，且不一定为最好的方法）。使用t时刻前动作价值的估计结果，在t时刻选择具有最高估计值的动作，即为开发/试探权衡中的“开发”，记作At​≐aargmax​Qt​(a)，式中， aargmax​是对应Qt​(a)动作价值中具有最大值的动作a。

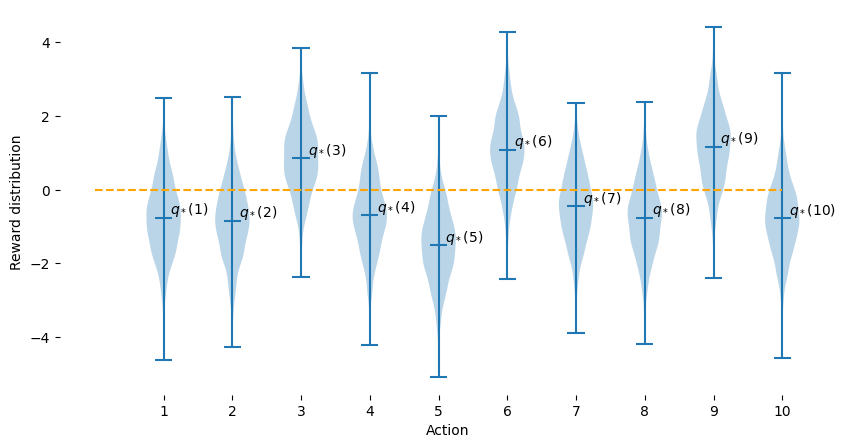
“开发”的权衡策略专注于当前估计的动作价值，为了避免陷入局部最优而采取“试探”的策略，以独立于动作-价值估计的方式偶尔（例如一个很小的概率）从所有动作中等概率的随机选择一个动作。

对于臂赌博机问题，假设k=10（有10个老虎机），每个老虎机都有特定于该机器提供奖励的概率分布，例如下图中假设拉动每个老虎机控制杆动作收益的真实值（期望值）q∗​(a)来自于均值为0和方差为1的标准正态分布，然后其实际的收益从均值为q∗​(a)单位方差的正态分布中生成（实验中，每个老虎机的随机采样样本数配置为2000），从而拉动不同老虎机控制杆后获得的动作真实值为1,…,10q∗​(a),a=1,…,10，对应到时刻t，选择At​（时刻t时的动作）时，实际的收益Rt​（时刻t时的收益）则由一个均值为q∗​(At​)，方差为1的正态分布决定。

%load\_ext autoreload   
%autoreload 2  
from usda.migrated\_project.RL\_an\_introduction import chapter02  
  
import matplotlib  
%matplotlib inline  
matplotlib.style.use('default')

The autoreload extension is already loaded. To reload it, use:  
 %reload\_ext autoreload

chapter02.figure\_2\_1(k=10,num=2000,seed=3467)



基于采样平均方法公式（1），就一个动作而言，令表示这一动作被选择次后获得的收益，次后估计的动作价值，简写为Qn​≐n−1R1​+R2​+⋯+Rn−1​​。上述公式计算时需要维护所有收益的记录， 以便在估计动作价值时进行计算，这无疑增加了内存使用量和计算成本，因此设计增量式公式（incremental formulas）以小而恒定的计算来更新动作价值的平均值，其推演过程如下，

公式（2）对同样有效，对任意Q1​有Q2​=R1​。对于每一新的动作收益，公式（2）计算方法只需要存储Qn​和n，其一般的形式可写为  NewEstimate ← OldEstimate + StepSize [ Target − OldEstimate ],(3)（新估计值←旧估计值+步长×[目标−旧估计值]），式中，[目标−旧估计值][目标−旧估计值]是估计值的误差，误差会随着向目标（target）靠近的每一步而减小。公式（2）中的目标为第n次的收益，步长（stepsize）记作n，更一般的步长表示为α或者αt​(a)。

多臂赌博机算法的伪代码可以写为：

**Algorithm 1**A simple bandit algorithm

**for***a*=1 to *k*(**Initialize**)**do**

*Q*(*a*)←0

*N*(*a*)←0

**end for**

**while**(**Loop forever** )**do**

 with probability 1− (breaking ties randomly)  a random action  with probability

*A*←{argmax*a*​*Q*(*a*) a random action ​ with probability 1−*ε* (breaking ties randomly)  with probability *ε*​

*R*←bandit(*A*)

1*N*(*A*)←*N*(*A*)+1

*Q*(*A*)←*Q*(*A*)+*N*(*A*)1​[*R*−*Q*(*A*)]

**end while**

对应到 Shangtong Zhang 书写的代码为：

class Bandit:  
 # @k\_arm: # of arms  
 # @epsilon: probability for exploration in epsilon-greedy algorithm  
 # @initial: initial estimation for each action  
 # @step\_size: constant step size for updating estimations  
 # @sample\_averages: if True, use sample averages to update estimations instead of constant step size  
 # @UCB\_param: if not None, use UCB algorithm to select action  
 # @gradient: if True, use gradient based bandit algorithm  
 # @gradient\_baseline: if True, use average reward as baseline for gradient based bandit algorithm  
 def \_\_init\_\_(self, k\_arm=10, epsilon=0., initial=0., step\_size=0.1, sample\_averages=False, UCB\_param=None,  
 gradient=False, gradient\_baseline=False, true\_reward=0.):  
 self.k = k\_arm  
 self.step\_size = step\_size  
 self.sample\_averages = sample\_averages  
 self.indices = np.arange(self.k)  
 self.time = 0  
 self.UCB\_param = UCB\_param  
 self.gradient = gradient  
 self.gradient\_baseline = gradient\_baseline  
 self.average\_reward = 0  
 self.true\_reward = true\_reward  
 self.epsilon = epsilon  
 self.initial = initial  
  
 def reset(self):  
 # real reward for each action  
 self.q\_true = np.random.randn(self.k) + self.true\_reward  
  
 # estimation for each action  
 self.q\_estimation = np.zeros(self.k) + self.initial  
  
 # # of chosen times for each action  
 self.action\_count = np.zeros(self.k)  
  
 self.best\_action = np.argmax(self.q\_true)  
  
 self.time = 0  
  
 # get an action for this bandit  
 def act(self):  
 if np.random.rand() < self.epsilon:  
 return np.random.choice(self.indices)  
  
 if self.UCB\_param is not None:  
 UCB\_estimation = self.q\_estimation + \  
 self.UCB\_param \* np.sqrt(np.log(self.time + 1) / (self.action\_count + 1e-5))  
 q\_best = np.max(UCB\_estimation)  
 return np.random.choice(np.where(UCB\_estimation == q\_best)[0])  
  
 if self.gradient:  
 exp\_est = np.exp(self.q\_estimation)  
 self.action\_prob = exp\_est / np.sum(exp\_est)  
 return np.random.choice(self.indices, p=self.action\_prob)  
  
 q\_best = np.max(self.q\_estimation)  
 return np.random.choice(np.where(self.q\_estimation == q\_best)[0])  
  
 # take an action, update estimation for this action  
 def step(self, action):  
 # generate the reward under N(real reward, 1)  
 reward = np.random.randn() + self.q\_true[action]  
 self.time += 1  
 self.action\_count[action] += 1  
 self.average\_reward += (reward - self.average\_reward) / self.time  
  
 if self.sample\_averages:  
 # update estimation using sample averages  
 self.q\_estimation[action] += (reward - self.q\_estimation[action]) / self.action\_count[action]  
 elif self.gradient:  
 one\_hot = np.zeros(self.k)  
 one\_hot[action] = 1  
 if self.gradient\_baseline:  
 baseline = self.average\_reward  
 else:  
 baseline = 0  
 self.q\_estimation += self.step\_size \* (reward - baseline) \* (one\_hot - self.action\_prob)  
 else:  
 # update estimation with constant step size  
 self.q\_estimation[action] += self.step\_size \* (reward - self.q\_estimation[action])  
 return reward  
  
  
def simulate(runs, time, bandits):  
 rewards = np.zeros((len(bandits), runs, time))  
 best\_action\_counts = np.zeros(rewards.shape)  
 for i, bandit in enumerate(bandits):  
 for r in trange(runs):  
 bandit.reset()  
 for t in range(time):  
 action = bandit.act()  
 reward = bandit.step(action)  
 rewards[i, r, t] = reward  
 if action == bandit.best\_action:  
 best\_action\_counts[i, r, t] = 1  
 mean\_best\_action\_counts = best\_action\_counts.mean(axis=1)  
 mean\_rewards = rewards.mean(axis=1)  
 return mean\_best\_action\_counts, mean\_rewards

其中，动作价值估计对应变量q\_estimation；不同动作计数对应变量action\_count；开发/试探权衡中，动作选择对应语句为（act()方法）：

if np.random.rand() < self.epsilon:  
 return np.random.choice(self.indices)  
...  
q\_best = np.max(self.q\_estimation)  
 return np.random.choice(np.where(self.q\_estimation == q\_best)[0])

时刻所选动作的收益（奖励）对应语句reward = np.random.randn() + self.q\_true[action]；计算动作价值（）时，使用采样平均方法的增量式实现（sample\_averages=True），对应语句self.q\_estimation[action] += (reward - self.q\_estimation[action]) / self.action\_count[action]。

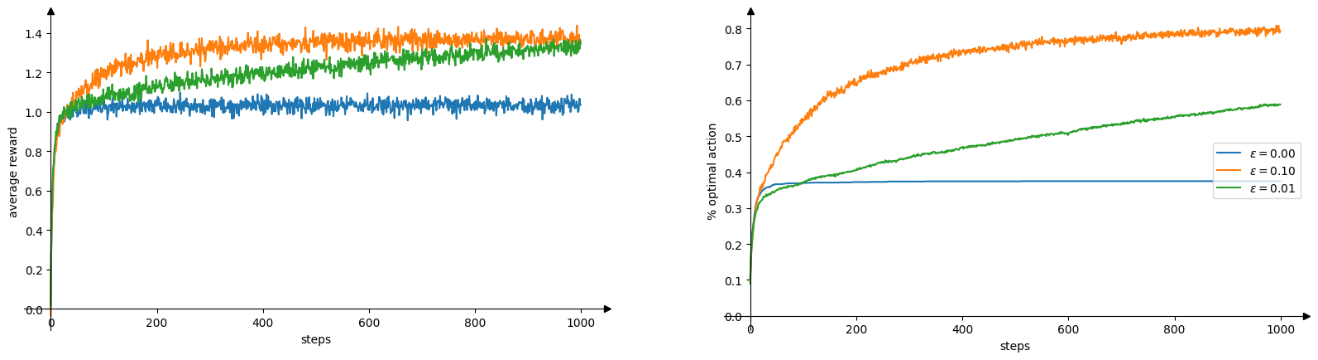
下述使用迁移至USDA库的代码实现下述计算（或从 [Shangtong Zhang 代码实现仓库](https://github.com/ShangtongZhang/reinforcement-learning-an-introduction)①下载执行），比较开发/试探权衡，配置不同的，值为0时，为完全“开发”模式，无“试探”过程；值为0.1时和0.01时，考虑了以较小几率加入”试探“的过程，0.1的值相对0.01具有更多”试探“的机会。配置执行迭代次数为2000次，每一次迭代选择拉动老虎机控制杆的次数（一个时刻选择一台拉动）为1000次，每一新一轮迭代都需要调用reset()方法随机重置每个老虎机的奖励（动作的真实价值）q\_true；时刻动作的价值估计，q\_estimation；已执行的动作计数（已拉动每台老虎机控制杆次数），action\_count；及具有最大价值的动作best\_action，并从头执行1000次动作配置time=0。

runs=2000  
time=1000  
epsilons = [0, 0.1, 0.01]  
  
bandits = [chapter02.Bandit(epsilon=eps, sample\_averages=True) for eps in epsilons]  
best\_action\_counts, rewards = chapter02.simulate(runs, time, bandits)

100%|█████████████████████████████████████████████| 2000/2000 [01:01<00:00, 32.70it/s]  
100%|█████████████████████████████████████████████| 2000/2000 [00:59<00:00, 33.48it/s]  
100%|█████████████████████████████████████████████| 2000/2000 [01:02<00:00, 31.88it/s]

下述左侧图显示了期望的收益随着经验（不断拉动老虎机控制杆至1000次）的增长而增长，尤其在开始阶段3种配置开发/试探权衡的参数试验结果值均快速的爬升；并可以观察到从长远来看加入“试探”的权衡策略表现的更好；且当ϵ=0.1时，具有相对更高期望的收益。这一现象也表现在随着经验的增加，动作的价值随之不断提升的过程。虽然ϵ=0.01时，期望收益和动作价值改善的比较慢，但是从更长远的情况来看，最终性能的表现要好于ϵ=0.1时的权衡状态。

chapter02.figure\_2\_2(best\_action\_counts, rewards,epsilons,figsize=(20, 5))



下面打印了ϵ=0.1，在time=1000次时期望收益q\_true、动作价值q\_estimation、动作计数action\_count等状态值。

bandit=bandits[1]  
print(f'time:\t{bandit.time};\  
 \nq\_true:\t{np.around(bandit.q\_true,3)};\  
 \nq\_estimation:\t{np.around(bandit.q\_estimation,3)};\  
 \naction\_count:\t{bandit.action\_count};\  
 \nbest\_action:\t{bandit.best\_action};\  
 \naverage\_reward:\t{np.around(bandit.average\_reward,3)};')

time: 1000;   
q\_true: [-1.055 -1.079 2.074 -0.71 -0.672 -1.416 -0.564 0.407 -0.659 0.057];   
q\_estimation: [-0.924 -0.749 2.117 -0.444 -0.6 -1.369 -0.625 0.338 -0.879 -0.01 ];   
action\_count: [ 11. 9. 846. 8. 15. 11. 14. 69. 7. 10.];   
best\_action: 2;   
average\_reward: 1.755;

### 3.9.1.2 （指数近因）加权平均（exponential recency-weighted average）和乐观初始值（Optimistic Initial Values）

使用采样平均方法估计动作价值，适用于收益的概率分布不随时间变化的赌博机问题，但是对于非平稳的强化学习问题，给近期的收益赋予比过去很久的收益更高的权值是一种合理的处理方式，将公式（2）用于更新个过去收益的均值的增量更新规则调整为Qn+1​≐Qn​+α[Rn​−Qn​]，式中，步长参数是一个常数，推导有

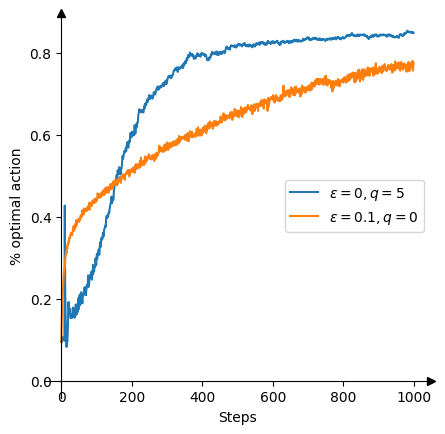
使得Qn+1​成为对过去的收益和初始的估计Q1​的加权平均，可以验证 (1−a)n+∑i=1n​α(1−α)n−i=1。赋予收益Ri​的权值α(1−α)n−i依赖于它被观测到的具体时刻与当前时刻的差，即n−i。因为1−α小于1，因此赋予Ri​的权值随着相隔次数的增加而递减（以指数形式递减），因此也称为指数近因加权平均。

上述试验初始化动作价值时为0，为动作收益真实值的均值（因为采样于均值为0和方差为1的标准正态分布）。如果初始化动作价值为 +5 ，从前文打印的10个老虎机收益分布的小提琴图 (ViolinPlot)可以得知，+5 是一个非常乐观的初始化值。从下述打印的比较结果可以观察到，在平稳问题上，乐观初始动作价值是有效的，虽然开始迭代阶段波动较大，因为需要更多的“试探”而表现糟糕，但是随着时间的推移，最终获得了较高且平稳的动作的价值。但需要注意，任何仅仅关注初始条件的方法都不太可能对一般的非平稳情况有帮助（开始时刻仅出现一次，因此不应该过多的关注）。

runs=2000  
time=1000  
  
bandits = []  
bandits.append(chapter02.Bandit(epsilon=0, initial=5, step\_size=0.1))  
bandits.append(chapter02.Bandit(epsilon=0.1, initial=0, step\_size=0.1))  
best\_action\_counts, \_ = chapter02.simulate(runs, time, bandits)

100%|█████████████████████████████████████████████| 2000/2000 [01:07<00:00, 29.67it/s]  
100%|█████████████████████████████████████████████| 2000/2000 [01:04<00:00, 30.91it/s]

chapter02.figure\_2\_3(best\_action\_counts)



### 3.9.1.3 基于置信度上界（ Upper-Conﬁdence-Bound，UCB）的动作选择

动作-价值估计总是存在不确定性，但是上述试验在估计了动作价值后，开发/试探权衡中的“开发”总是选择具有最大估计价值的动作，因此最好是根据各个动作的潜力来选择可能事实上最优的动作，从而需要考虑各自的价值估计有多接近最大值，及其估计的不确定性，调整“开发”的动作选择公式，有At​≐aargmax​[Qt​(a)+cNt​(a)lnt​​]，式中， lnt表示时刻t的自然对数（即e≈2.71828的多少次方等于t）； Nt​(a)表示在时刻t之前动作a被选择的次数； c是一个大于0的常数，控制变化幅度，决定了置信水平。如果Nt​(a)=0，则认为a是满足最大化条件的动作。

基于 UCB 选择的思想是平方根项是对动作价值估计的不确定性或方差的度量。由于Nt​(a)出现在不确定项的分母上，因此随着Nt​(a)的增加，该项减小，不确定性可能会减小；如果Nt​(a)没有变化，而随着时刻t的增加，不确定性则可能会增加。自然对数的使用意味着随着时间的推移，增加会变得越来越小，但它是无限的。所有动作都会被选中，但是随着时间的流逝，具有较低价值估计的动作或已经被选择了更多次的动作被选择到的频率较低。

UCB 代码实现为，

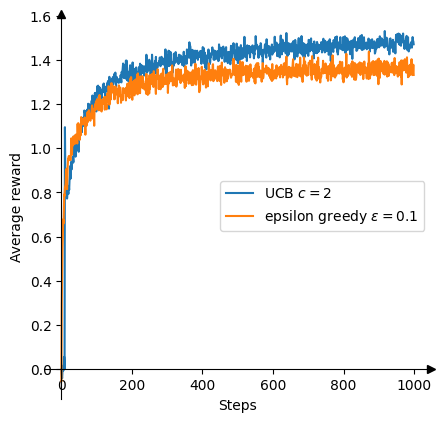
if self.UCB\_param is not None:  
 UCB\_estimation = self.q\_estimation + self.UCB\_param \* np.sqrt(np.log(self.time + 1) / (self.action\_count + 1e-5))   
 q\_best = np.max(UCB\_estimation)  
 return np.random.choice(np.where(UCB\_estimation == q\_best)[0])

下述试验结果也验证了 UCB 算法的可行性，相较的平均收益曲线，UCB 方法相对收敛到更高的平均收益水平上。

runs=2000  
time=1000  
  
bandits = []  
bandits.append(chapter02.Bandit(epsilon=0, UCB\_param=2, sample\_averages=True))  
bandits.append(chapter02.Bandit(epsilon=0.1, sample\_averages=True))  
\_, average\_rewards = chapter02.simulate(runs, time, bandits)

100%|█████████████████████████████████████████████| 2000/2000 [01:28<00:00, 22.64it/s]  
100%|█████████████████████████████████████████████| 2000/2000 [01:00<00:00, 32.82it/s]

chapter02.figure\_2\_4(average\_rewards)



### 3.9.1.4 梯度（Gradient）赌博机算法

针对每个动作被选择的概率。对应的代码实现为，

if self.gradient:  
 exp\_est = np.exp(self.q\_estimation)  
 self.action\_prob = exp\_est / np.sum(exp\_est)  
 return np.random.choice(self.indices, p=self.action\_prob)

基于随机梯度上升（stochastic gradient ascent）的思想，存在一种求解 soft-max 动作偏好（soft-max action preferences）的自然学习算法，每一时刻，在选择动作At​并获得收益Rt​后，动作偏好（动作价值估计）更新为, Ht+1​(At​)≐Ht​(At​)+α(Rt​−Rˉt​)(1−πt​(At​)), and Ht+1​(a)≐Ht​(a)−α(Rt​−Rˉt​)πt​(a), for all a=At​，式中，α是一个大于0的数，表示步长； Rˉt​∈R是在时刻t内所有收益的平均值，作为比较收益的一个基准项。如果收益高于基准项，那么在未来选择动作At​的概率就会增加，反之概率就会降低，而未选择的动作被选中的概率上升。对应的代码实现为，

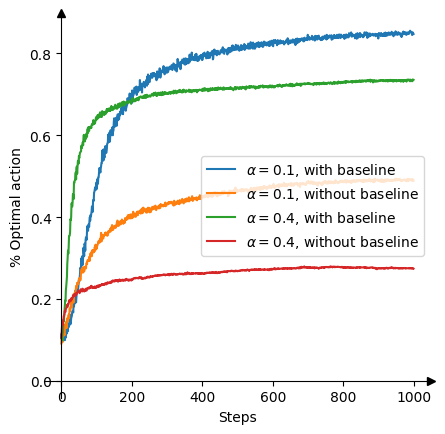
elif self.gradient:  
 one\_hot = np.zeros(self.k)  
 one\_hot[action] = 1  
 if self.gradient\_baseline:  
 baseline = self.average\_reward  
 else:  
 baseline = 0  
 self.q\_estimation += self.step\_size \* (reward - baseline) \* (one\_hot - self.action\_prob)

下述试验梯度赌博机算法，配置真实的期望收益为 +4，在给定收益基准项（reward baseline term）（即平均收益）时，所有动作的价值估计可以马上适应到新的收益水平；如果没有基准项（），性能显著降低。

runs=2000  
time=1000  
  
bandits = []  
bandits.append(chapter02.Bandit(gradient=True, step\_size=0.1, gradient\_baseline=True, true\_reward=4))  
bandits.append(chapter02.Bandit(gradient=True, step\_size=0.1, gradient\_baseline=False, true\_reward=4))  
bandits.append(chapter02.Bandit(gradient=True, step\_size=0.4, gradient\_baseline=True, true\_reward=4))  
bandits.append(chapter02.Bandit(gradient=True, step\_size=0.4, gradient\_baseline=False, true\_reward=4))  
best\_action\_counts, \_ = chapter02.simulate(runs, time, bandits)

100%|█████████████████████████████████████████████| 2000/2000 [01:39<00:00, 20.15it/s]  
100%|█████████████████████████████████████████████| 2000/2000 [01:32<00:00, 21.58it/s]  
100%|█████████████████████████████████████████████| 2000/2000 [01:32<00:00, 21.56it/s]  
100%|█████████████████████████████████████████████| 2000/2000 [01:58<00:00, 16.85it/s]

chapter02.figure\_2\_5(best\_action\_counts,bandits)

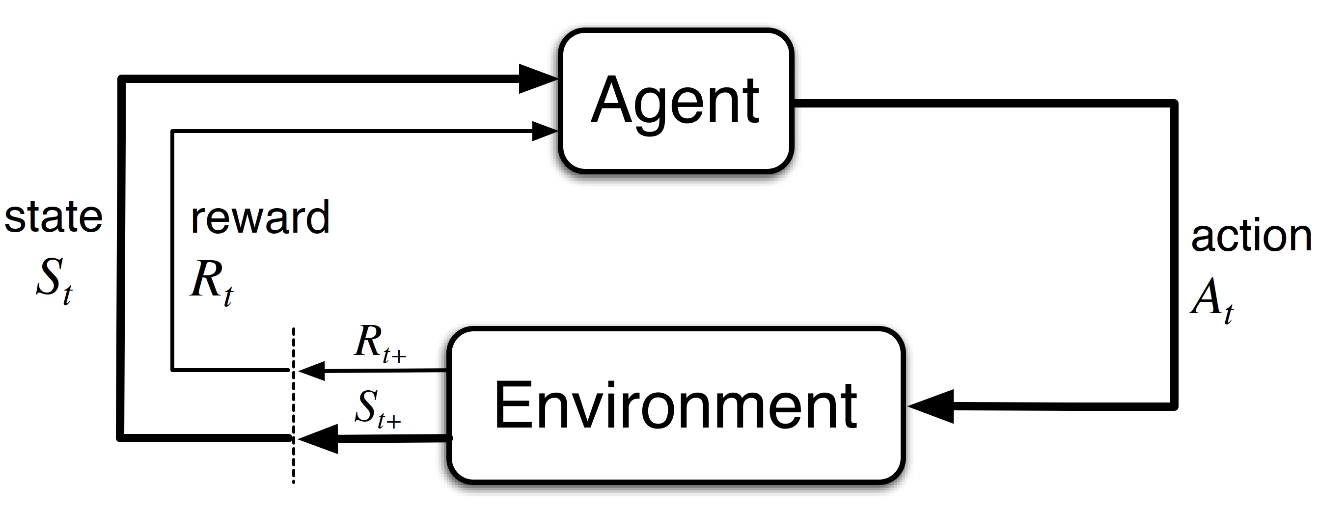


## 3.9.2 有限马尔可夫决策过程与动态规划

### 3.9.2.1 强化学习要素

#### 1) 智能体-环境交互接口（Agent–Environment Interface）

马尔可夫决策过程（Markov Decision Processes，MDP）是一种通过交互式学习实现目标的理论框架。进行学习及实施决策的机器被称为智能体（agent）。智能体之外所有与其相互作用的事物都被称为环境（environment）。这些事物之间持续进行交互，智能体选择动作，环境对这些动作做出相应的响应，并向智能体呈现出新的状态。环境也会产生一个收益，通常是特定的数值，这就是智能体在动作选择的过程中想要最大化的目标，如图[2]48，



更具体的说，在每个离散时刻，智能体和环境都发生了交互。在每个时刻，智能体观察到所在环境状态的某种特征表达，St​∈S，并且在此基础上选择一个动作， At​∈A(s)。下一时刻，作为其动作的结果，智能体接收到一个数值化的收益（reward），Rt+1​∈R⊂R，并进入到一个新的状态St+1​。从而，MDP 和智能体共同给出了一个序列或轨迹（trajectory），如S0​,A0​,R1​,S1​,A1​,R2​,S2​,A2​,R3​,…。

#### 在有限 MDP 中，状态、动作和收益的集合（S、A和R）都只有有限个元素，随机变量Rt​和St​具有定义明确的离散概率分布，并且只依赖于先前的状态和动作，即给定先前的状态和动作时，这些随机变量的特定值， S和rR，在t时刻出现的概率是Pr{St​=s′,Rt​=rSt−1​=s,At−1​=a},(1)，对于任意s′，S， R，及As​。函数p定义了 MDP 的动态特性。上述公式是一个定义，为函数p。动态函数p:S×R×S×A→[0,1]是有4个参数的普通确定性函数（deterministic function）。公式（1）中的"|"为条件概率符号，表示函数p为每个s和a的选择都指定了一个概率分布，即​p(s′,rs,a)=1, for all sS,aA(s)。

对于 MDP，由p给出的概率完全刻画了环境的动态特征， St​和Rt​的每个可能的值出现的概率只取决于前一个状态St−1​和前一个动作At−1​，且与更早之前的状态和动作完全无关。这个限制并不是针对决策过程，而是状态。状态必须包括过去智能体和环境交互的方方面面的信息，这些信息会对未来产生一定影响。这样，状态就被认为具有马尔可夫性。从四参数动态函数p中，可以计算出关于环境的任何其它信息，例如状态转移概率（表示为一个三参数函数p:S×S×A→[0,1])）： p(s′∣s,a)≐Pr{St​=s′∣St−1​=s,At−1​=a}=∑r∈R​p(s′,r∣s,a)；如果定义“状态-动作”二元组的期望收益，将其表示为一个双参数函数r:S×A→R： r(s,a)≐E[Rt​∣St−1​=s,At−1​=a]=∑r∈R​r∑s′∈S​p(s′,r∣s,a)；和“状态-动作-后继状态”三元组的期望收益，表示为一个三参数函数r:S×A×S→R： r(s,a,s′)≐E[Rt​∣St−1​=s,At−1​=a,St​=s′]=∑r∈R​rp(s′∣s,a)p(s′,r∣s,a)​。

#### [2）目标和收益](https://richiebao.github.io/USDA_CH_endup/#/./markdown/3_9_A_%E5%BC%BA%E5%8C%96%E5%AD%A6%E4%B9%A0_%E8%A1%A8%E6%A0%BC%E5%9E%8B%E6%B1%82%E8%A7%A3%E6%96%B9%E6%B3%95?id=_2%ef%bc%89%e7%9b%ae%e6%a0%87%e5%92%8c%e6%94%b6%e7%9b%8a)

在强化学习中，智能体的目标被形式化表征为一种特殊信号，称为收益（rewards），它通过环境传递给智能体。在每个时刻，收益都是一个单一标量数值， Rt​∈R。通俗来说，智能体的目标是最大化其收到的总收益，这意味着需要最大化的不是当前收益，而是长期的累积收益。即所有的目标（goals）都可以归结为：最大化智能体接收到的标量信号（收益）累积和的概率期望值。收益信号并不是传授智能体如何实现目标的先验知识，只能用来传达要实现的目标是什么。

#### 3）回报和分幕

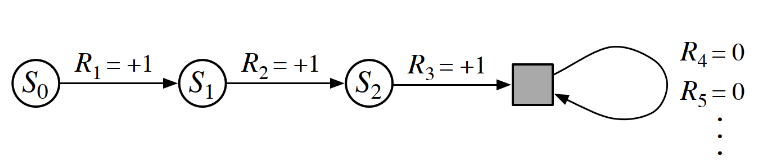
智能体的目标是最大限度的提高长期收益，如果把时刻t后接收到的收益序列表示为Rt+1​,Rt+2​,Rt+3​,…，将寻求其最大化期望回报（expected return）记为Gt​，定义为收益序列的一些特定函数。最简单情况是求收益的总和Gt​≐Rt+1​+Rt+2​+Rt+3​+⋯+RT​,(2)，式中， T是最终时刻。这种方法在有“最终时刻”这种概念的应用中是有意义的。在这类应用中，智能体和环境的交互被自然的分成一系列子序列（每个序列都存在最终时刻），称每个子序列为幕（episodes）。例如，一盘游戏、一次走迷宫的旅程或任何这类重复性的交互过程。每幕都以一种特殊状态结束，称之为终结状态（terminal state）。随后会重新从某个标准的起始状态或起始状态分布中的某个状态样本开始。即使结束方式不同，下一幕的开始状态与上一幕的结束方式完全无关。因此，这些幕可以被认为在同样的终结状态下结束，只是对不同的结果有不同的收益。具有这种分幕重复特性的任务称为分幕式任务（episodic tasks）。在分幕任务中，有时需要区分非终结状态集，记为S，和包含终结和非终结状态的所有状态集，记作S+。终结的时间T是一个随机变量，通常随着幕的不同而不同。

另一方面，诸如长期运行机器人的应用，其智能体-环境交互不一定能被自然的分成单独的幕，而是持续不断的发生，将其称之为持续性任务（continuing tasks）。因为最终时刻T=∞，试图最大化的回报往往也趋于无穷，因此引入折扣（discounting）。智能体尝试选择动作时，使得它在未来收到的经过折扣系数加权后的收益（discounted rewards）总和最大化，选择动作At​来最大化期望折后回报， Gt​≐Rt+1​+γRt+2​+γ2Rt+3​+⋯=∑k=0∞​γkRt+k+1​,(3)，式中，γ是一个参数， 0≤γ≤1，称为折扣率（discount rate）。

折扣率决定了未来收益的现值（present value），未来时刻k的收益只有它当前值的γk−1倍。如果γ<1，那么只要收益序列{Rk​}有界，公式（3）中的无限序列总和就是一个有限值。如果γ=0，则为Gt​≐Rt+1​，其目标只是学习如何选择At​来最大化Rt+1​，即只关心最大化当前收益。随着γ接近1，折后回报将同时考虑更多未来的收益。

连续时间步的回报彼此相关，有Gt​≐Rt+1​+γRt+2​+γ2Rt+3​+γ3Rt+4​+⋯=Rt+1​+γ(Rt+2​+γRt+3​+γ2Rt+4​+⋯)=Rt+1​+γGt+1​，(4)。如果定义GT​=0，公式（4）会适用任意t<T时刻，即使最终时刻出现在t+1也不例外。尽管公式（4）中定义的回报是对无限个收益子项求和，但只要收益是一个非零常数且γ<1，那这个回报仍旧是有限的。例如，如果收益是一个常数+1，那么回报为Gt​=∑k=0∞​γk=1−γ1​。

对于分幕式任务，通常使用St,i​来表示幕i中t时刻的状态，同样有Rt,i​、πt,i​和Ti​等。然而实际上在讨论分幕式任务时，不必区分不同的幕，而直接用St​表示St,i​。因此对于分幕式任务和持续性任务也可以使用一种统一的符号表述，同样包括将回报定义为有限项的总和（式（2））和无限项的总和（式（3）），如下述状态转移图[2]57，



图中灰色填充方块表示把幕的终止当作一个特殊的吸收状态（absorbing state）的入口。从开始，就会得到一个收益序列+1,+1,+1,0,0,0,…+1,+1,+1,0,0,0,…，可以看到，不管是计算有限个时刻的收益（T=3），还是无限序列的全部收益总和，都得到相同的回报。因此也可以将回报表示为Gt​≐∑k=t+1T​γk−t−1Rk​，允许公式包括T=∞和γ=1(二者不同时)的可能性。

*4）策略和价值函数*

价值函数（Value Functions）是状态（states）或状态-动作二元组（state–action pairs）的函数，用来评估当前智能体在给定状态或状态-动作下未来预期的收益有多好，即回报的期望值。智能体期望未来能得到的收益取决于智能体所选择的动作，因此，价值函数与特定行为方式相关，称之为策略（policies），即从状态到每个动作的选择概率之间的映射。如果智能体在时刻t选择了策略π，那么π(a∣s)就是当St​=s时At​=a的概率。同p一样，π就是一个普通的函数，其中“∣∣”可理解为每个s∈S都定义了一个在a∈A上的概率分布。

将策略π下状态s的价值函数记为vπ​(s)，即从状态s开始，智能体按照策略π进行决策所获得回报的概率期望值。对于 MDP 可以正式定义为， vπ​(s)≐Eπ​[Gt​∣St​=s]=Eπ​[∑k=0∞​γkRt+k+1​∣St​=s], for all s∈S，(5)，式中， Eπ​[⋅]表示在给定策略π时，一个随机变量的期望值， t可以是任意时刻，且终止状态的价值始终为0。将函数vπ​称为策略π的状态价值函数。

类似的，策略π下，在状态s时采取动作a的价值记为qπ​(s,a)，是根据策略π，从状态s开始，执行动作a之后，所有可能决策序列的期望回报， qπ​(s,a)≐Eπ​[Gt​∣St​=s,At​=a]=Eπ​[∑k=0∞​γkRt+k+1​∣St​=s,At​=a]，(6)。将函数qπ​称为策略π的动作价值函数。

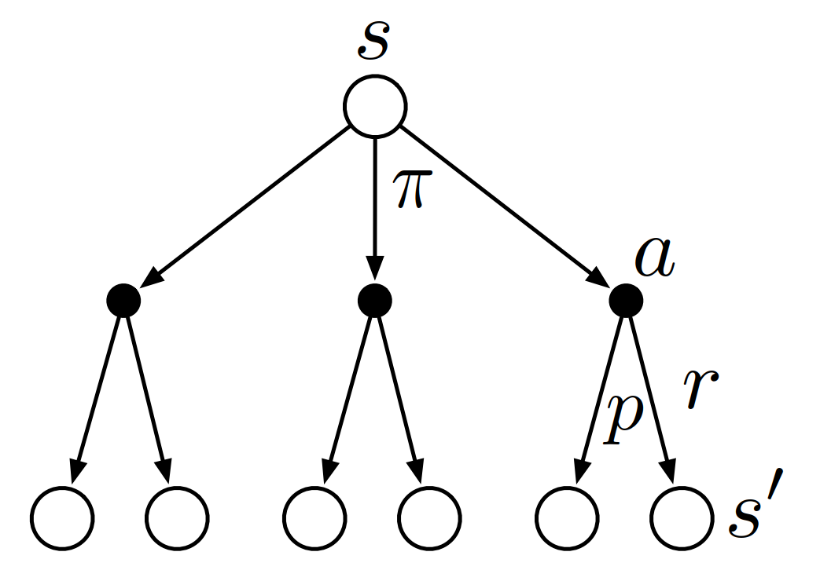
状态和动作的价值函数vπ​和qπ​都能从经验中估算得到。例如，一个智能体遵循策略π，并且对每个遇到的状态都记录该状态后实际回报的平均值，那么，随着状态出现次数接近无穷大，其均值会收敛到状态价值vπ​(s)；同样，如果每个状态的每个动作都保留单独的平均值，这些均值也会收敛到动作价值qπ​(s,a)，这种估算方法称为蒙特卡洛方法（Monte Carlo methods），涉及从真实回报的多个随机样本中求平均值。当环境中有很多状态时，独立估算每个状态的平均值往往不切实际，因此可以将价值函数vπ​和qπ​进行参数化，参数的数量远少于状态的数量，然后通过调整价值函数的参数更好的计算回报值。

同连续时间步的回报彼此相关的公式（4），价值函数也可以建立为一个递归公式，对于任何策略和任何状态，的价值与其可能的后继状态价值之间存在的关系为，

，

式（7）被称为的贝尔曼方程（Bellman equation），用等式表达了状态价值和后继状态价值之间的关系。

以回溯图（Backup diagram）的方式表述状态之间的关系（从一个状态向后观察所有可能到达的后继状态），如图[2]59，



图中，空心圆表示一个状态，实心圆表示一个“状态-动作”二元组。将状态作为根节点，智能体可以根据其策略，采取动作集合中的任意一个动作（图中给出了3个动作）。对于每一个动作，环境会根据其动态特性函数，以一个后继状态（图中给出了两个状态）及其收益作为响应。贝尔曼方程（7）随所有的可能性求均值，按其发生的概率对每种可能性进行加权。回溯操作就是将后继状态（或状态-动作二元组）的价值信息回传给当前时刻的状态（或状态-动作二元组）。

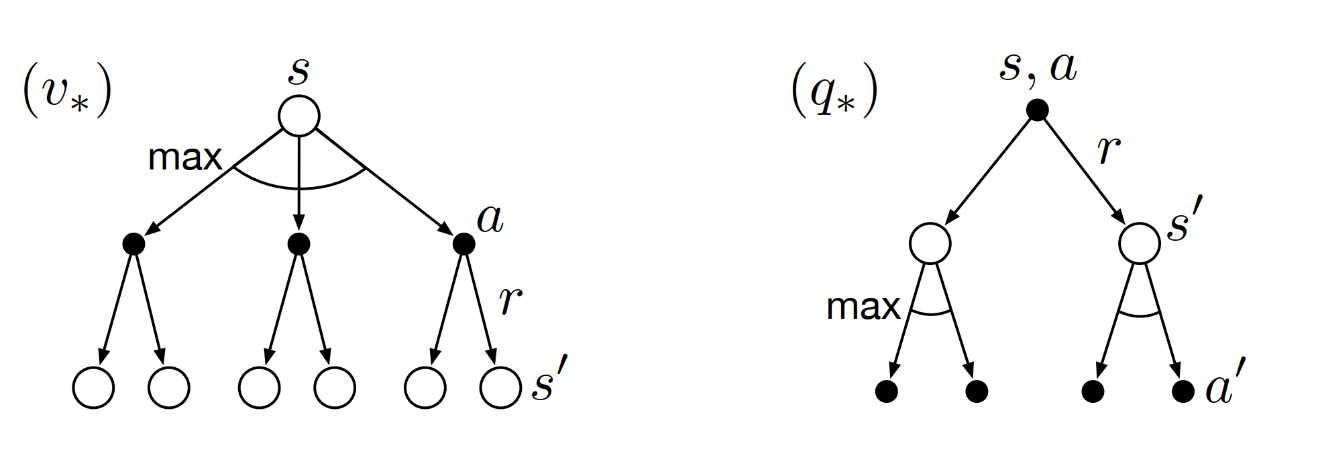
#### 5）最优策略和最优价值函数

强化学习的任务是要找出一个策略，使其能够在长期过程中获得最大收益。对于有限 MDP，可以通过比较价值函数精确定义一个最优策略。本质上，价值函数定义了策略上的一个偏序关系，即如果策略相当或好于策略，那么上所有状态上的期望回报都应该等于或者大于。用公式表述为，若对于所有的，则有。总会存在至少一个策略不略于其它所有的策略，即为最优策略（optimal policy）。尽管最优策略可能不止一个，但通常用来表示。最优策略共享相同的状态价值函数，称之为最优状态价值函数（optimal state-value function），定义为，, for all ；最优策略也共享相同的动作价值函数，称之为最优动作价值函数（optimal action-value function），定义为，, for all and 。对于“状态-动作”二元组，给出了在状态下，先采取动作，之后按照最优策略去决策的期望回报。因此可以用来表示，有，。

最优状态价值函数的贝尔曼最优方程推导为，

公式（8）和（9）就是的贝尔曼最优方程的两种形式。贝尔曼最优方程阐述了，最优策略下各个状态的价值一定等于这个状态下最优动作的期望回报。同样，的贝尔曼最优方程为，

最优状态价值函数（）和最优动作价值函数（）的贝尔曼最优方程中进行的后继状态和动作的扩展过程如图[2]64，



图中在智能体的选择节点处加入了弧线表示应该在给定策略下取最大值而不是期望值。

对于有限 MDP 来说，的贝尔曼最优方程有独立于策略的唯一解。贝尔曼最优方程实际上是一个方程组，每个状态对应一个方程等式。如果有个状态，那么将有个含有个未知量的方程。如果环境的动态变化特征是已知的，那么原则上可以用解非线性方程组的方法求解方程组。类似的求解的一组解。有了，就可以确定一个最优策略。对于每个状态，都存在一个或多个动作，可以在贝尔曼最优方程的条件下产生最大的价值。如果在一个策略中，只有这些动作的概率非零，那么这个策略就是一个最优策略。定义的意义在于，可以将最优的长期（全局）回报期望值转化为每个状态对应的一个当前局部量的计算。在给定的情况下，对于任意状态，智能体只需找到使得最大化的动作就可以了。它将最优长期回报的期望值表达为对应每个“状态-动作”二元组的一个当前局部量。因此，将价值函数表达为“状态-动作”二元组的函数而不仅仅是状态的函数后，最优动作的选取就不再需要知道后继状态及其对应的价值，即，不再需要知道任何环境的动态变化特性。

### 3.9.2.2 动态规划与网格问题

动态规划（dynamic programming，DP）是一类优化方法，在给定一个用马尔可夫决策过程（MDP）描述的完备环境模型情况下，可以计算最优策略。假设环境为有限 MDP，即假设状态集合、动作集合和收益集合是有限的，且整个系统的动态特性（dynamics）由对于任意和（表示在分幕任务下加上一个终止状态）的四参数概率分布给出。对于连续状态和动作问题，可以近似的量化为离散集合，再使用有限状态下的 DP 算法。DP 的核心思想是使用价值函数来结构化的组织对最优策略的搜索，如最优状态价值函数的贝尔曼最优方程（公式9）和最优动作价值函数的贝尔曼最优方程（公式10）。

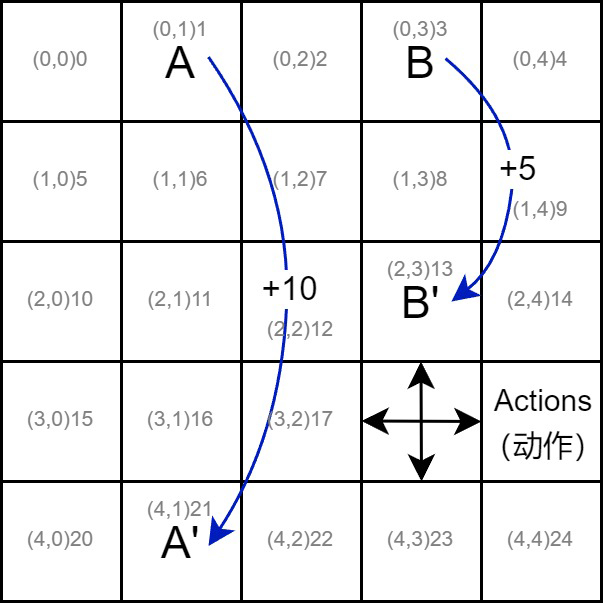
#### 1）策略评估（预测）

对于任意一个策略，如何计算其状态价值函数，在 DP 文献中称为策略评估（policy evaluation），有时也称其为预测问题（prediction problem）。对应到的贝尔曼方程（公式 7），指的是处于环境状态 时，智能体在策略下采取动作的概率。期望的下标表明期望的计算是以遵循策略为条件的。只要或者任何状态在下都能保证最后终止，那么唯一存在。

如果环境的动态特性完全已知，则公式 7 是一个有着 个未知数及 个等式的联立线性方程组。理论上这个方程组的解可以直接被解出，但是通常使用迭代方法求解。考虑一个近似的价值函数序列，，从映射到（实数集）。初始的近似值可以任意选取（但终止状态值必须为0）。然后下一轮迭代使用的贝尔曼方程进行更新，对于任意，有

显然，是这个更新规划的不动点（ﬁxed point），因为的贝尔曼方程已经保证了这种情况下的等式成立，序列在时将会收敛到。这个算法称为迭代策略评估（iterative policy evaluation）。

以网格问题为例，下图网格代表的是一个简单的有限 MDP。网格中的（格子）单元代表的是环境中的状态。每个单元有4个可选的动作：东、南、西和北，每个动作都会使智能体在对应的方向上前进一格。如果采取的动作使智能体跳出了网格，则智能体会在原位置保持不动，且得到一个值为-1的收益（）；如果智能体位于A和B单元中，则智能体直接跳到A'和B'中，而不是邻接的单元中，且分别得到+10和+5的收益；其它情况，智能体跳到邻接单元时，其动作的收益均是0。



将上述过程用代码描述如下，定义网格的大小WORLD\_SIZE为5；的位置对应为A\_POS、B\_POS和A\_PRIME\_POS、B\_PRIME\_POS等4个变量；折扣率为DISCOUNT；四个动作的概率（）相等，为ACTION\_PROB = 0.25。

%load\_ext autoreload   
%autoreload 2  
from usda.migrated\_project.RL\_an\_introduction import chapter03  
  
import matplotlib  
%matplotlib inline  
matplotlib.style.use('default')   
  
import numpy as np  
import matplotlib.pyplot as plt

The autoreload extension is already loaded. To reload it, use:  
 %reload\_ext autoreload

WORLD\_SIZE = 5  
A\_POS = [0, 1]  
A\_PRIME\_POS = [4, 1]  
B\_POS = [0, 3]  
B\_PRIME\_POS = [2, 3]  
DISCOUNT = 0.9  
ACTION\_PROB = 0.25  
  
# left, up, right, down  
ACTIONS=chapter03.grid\_world.ACTIONS  
ACTIONS\_FIGS=chapter03.grid\_world.ACTIONS\_FIGS  
print(f'action={ACTIONS};\naction figs:{ACTIONS\_FIGS}')

action=[array([ 0, -1]), array([-1, 0]), array([0, 1]), array([1, 0])];  
action figs:['←', '↑', '→', '↓']

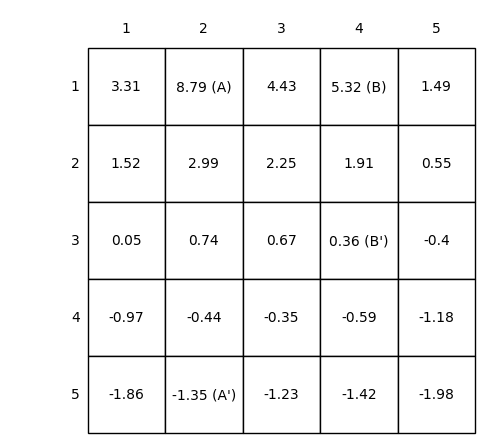
为了从得到下一个近似，迭代策略评估对于每个状态采用相同的操作：根据给定的策略，得到所有可能单步转移之后的即时收益reward和的每个后继状态旧的价值函数value[next\_i, next\_j]，利用这二者的期望值更新新的价值函数，对应代码new\_value[i, j] += ACTION\_PROB \* (reward + DISCOUNT \* value[next\_i, next\_j])（贝尔曼公式（7）），这种方法称期望更新（expected update）。迭代策略评估的每一轮迭代都更新一次所有状态的价值函数，即一次更新是对整个状态空间的一次遍历，以产生新的近似价值函数。期望更新可以有很多种不同的形式，具体取决于使用状态还是“状态-动作”二元组，或者后继状态价值函数的具体组合方式。

step()函数定义了时刻，智能体所在状态（即位于的一个单元格）state，执行等概率的动作（action）进入下一个状态（next\_state），并返回上述定义的收益（reward）和下一个状态（单元索引）。

def step(state, action):  
 if state == A\_POS:  
 return A\_PRIME\_POS, 10  
 if state == B\_POS:  
 return B\_PRIME\_POS, 5  
  
 next\_state = (np.array(state) + action).tolist()  
 x, y = next\_state  
 if x < 0 or x >= WORLD\_SIZE or y < 0 or y >= WORLD\_SIZE:  
 reward = -1.0  
 next\_state = state  
 else:  
 reward = 0  
 return next\_state, reward

计算给定折扣率，等概率的4种动作策略下的价值函数如下。当和的状态价值函数变化和趋近于0（）时，结束循环。

value = np.zeros((WORLD\_SIZE, WORLD\_SIZE))  
k=0  
while True:  
 # keep iteration until convergence  
 new\_value = np.zeros\_like(value)  
 for i in range(WORLD\_SIZE):  
 for j in range(WORLD\_SIZE):  
 for action in ACTIONS:  
 (next\_i, next\_j), reward = chapter03.step([i, j], action)  
 # bellman equation  
 new\_value[i, j] += ACTION\_PROB \* (reward + DISCOUNT \* value[next\_i, next\_j])  
 k+=1  
 if np.sum(np.abs(value - new\_value)) < 1e-4:  
 chapter03.draw\_image(np.round(new\_value, decimals=2))  
 plt.show()  
 break  
 value = new\_value  
print(f'iterations={k}')

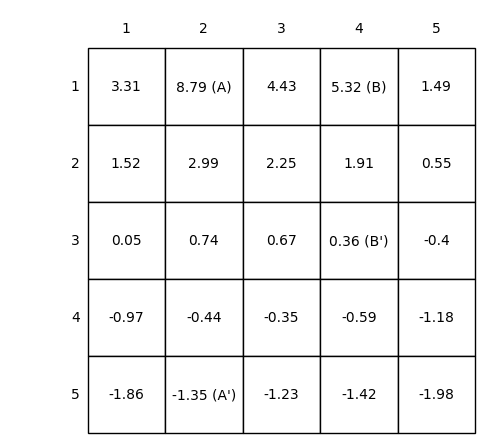


iterations=77

除了使用上述的贝尔曼方程通过迭代法求解，也通过解个状态的联立线性方程组求精确解，如下过程。

A = -1 \* np.eye(WORLD\_SIZE \* WORLD\_SIZE)  
b = np.zeros(WORLD\_SIZE \* WORLD\_SIZE)  
for i in range(WORLD\_SIZE):  
 for j in range(WORLD\_SIZE):  
 s = [i, j] # current state  
 index\_s = np.ravel\_multi\_index(s, (WORLD\_SIZE, WORLD\_SIZE))  
 for a in ACTIONS:  
 s\_, r = chapter03.step(s, a)  
 index\_s\_ = np.ravel\_multi\_index(s\_, (WORLD\_SIZE, WORLD\_SIZE))  
  
 A[index\_s, index\_s\_] += ACTION\_PROB \* DISCOUNT  
 b[index\_s] -= ACTION\_PROB \* r  
  
x = np.linalg.solve(A, b)  
print(f'{A.shape},{b.shape},{x.shape}')  
chapter03.draw\_image(np.round(x.reshape(WORLD\_SIZE, WORLD\_SIZE), decimals=2))

(25, 25),(25,),(25,)



#### 2）策略改进和迭代

计算一个指定策略下的价值函数，是为了寻找更好的策略，如果和是任意两个确定的策略，对任意，，称策略相比于一样好或者更好，则肯定能得到一样好或者更好的期望回报。一旦一个策略根据产生了一个更好的策略，那么通过计算可以进一步得到一个更优的策略。这样一个链式的方法可以得到一个不断改进的策略和价值函数序列，式中，代表策略评估，表示策略改进。每一个策略都能保证比前一个更优（除非前一个已经是最优）。由于有限 MDP 必然只有有限种策略，所以在有限次的迭代后，这种方法一定收敛到一个最优策略和最优价值函数。这种寻找最优策略的方法称为策略迭代（policy iteration）。每一次策略评估都是一个迭代计算的过程，需要基于前一个策略的价值函数开始计算。

策略迭代的伪代码如下：

**Algorithm 2**Policy Iteration (using iterative policy evaluation) for estimating

**1. Initialization**

*V*(*s*)∈R and *π*(*s*)∈A(*s*) arbitrarily for all *s*∈S; *V*(*terminal*)≐0

**2. Policy Evaluation**

**while**(**Loop**)**do**

Δ←0Δ←0

**for**Loop for each *s*∈S**do**

*v*←*V*(*s*)

*V*(*s*)←∑*s*′,*r*​*p*(*s*′,*r*∣*s*,*π*(*s*))[*r*+*γV*(*s*′)]

Δ←max(Δ,∣*v*−*V*(*s*)∣)

**end for**

until Δ<*θ* (a small positive number determining the accuracy of estimation)

**end while**

**3. Policy Improvement**

*policy*−*stable*←*true*

**for**each *s*∈S**do**

*old*−*action*←*π*(*s*)

*π*(*s*)←argmax*a*​∑*s*′,*r*​*p*(*s*′,*r*∣*s*,*a*)[*r*+*γV*(*s*′)]

**if***old*−*action*=*π*(*s*)**then**

*policy*−*stable*←*false*

**end if**

**end for**

**if***policy*−*stable***then**

stop and return *V*≈*v*∗​ and *π*≈*π*∗​

**else**

got to 2

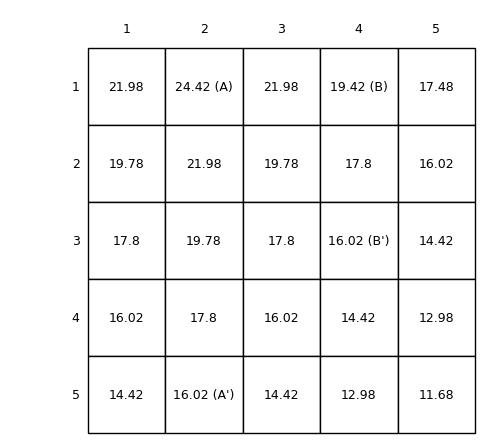
**end if**

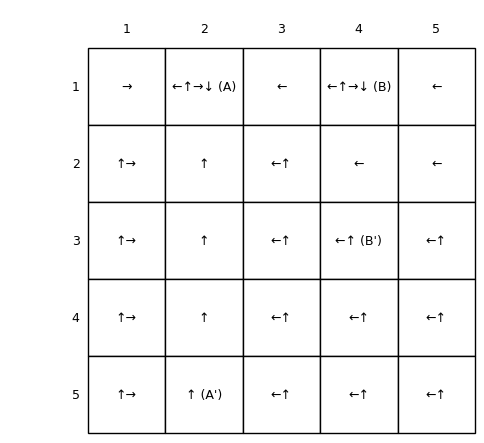
计算最优状态价值函数和最优策略如下，其中

for action in ACTIONS:  
 (next\_i, next\_j), reward = chapter03.step([i, j], action)  
 values.append(reward + DISCOUNT \* value[next\_i, next\_j])  
new\_value[i, j] = np.max(values)

对应到贝尔曼最优方程（9）。从最优策略结果可以观察到状态的4个方向动作均为最优策略，而其它大部分状态的动作都潜在指向状态单元格（其具有最大状态-动作收益）。最优策略图中有些状态有多个箭头，表示有多个动作其价值函数都可以达到最大值。

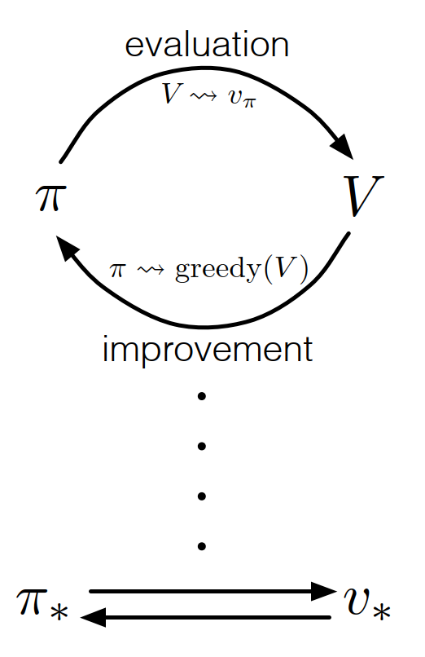
value = np.zeros((WORLD\_SIZE, WORLD\_SIZE))  
while True:  
 # keep iteration until convergence  
 new\_value = np.zeros\_like(value)  
 for i in range(WORLD\_SIZE):  
 for j in range(WORLD\_SIZE):  
 values = []  
 for action in ACTIONS:  
 (next\_i, next\_j), reward = chapter03.step([i, j], action)  
 # value iteration  
 values.append(reward + DISCOUNT \* value[next\_i, next\_j])  
 new\_value[i, j] = np.max(values)  
 if np.sum(np.abs(new\_value - value)) < 1e-4:  
 chapter03.draw\_image(np.round(new\_value, decimals=2))  
 chapter03.draw\_policy(new\_value)  
 plt.show()  
 break  
 value = new\_value





#### 3）广义策略迭代

策略迭代包括两个同时进行的相互作用的流程，一个使得价值函数与当前策略一致（策略评估）；另一个根据当前价值函数贪心（greedy）的更新策略（策略改进）。在策略迭代中，这两个流程交替进行，也可以在两次策略改进之间执行一次策略评估或者仅有一个状态在评估流程中得到更新后马上返回到改进流程等不同方式，但只要两个流程持续更新所有状态，那么最后的结果通常都收敛到最优价值函数和一个最优策略。将上述策略评估和策略改进相互作用的一般思路称为广义策略迭代（generalized policy iteration，GPI），其流程如下图[2]86，策略总是基于特定的价值函数进行改进，价值函数也始终向对应特定策略的真实价值函数收敛。如果评估流程和改进流程都很稳定，即不再产生变化，那么价值函数和策略必定是最优的。价值函数只有在与当前策略一致时稳定，且策略只有在对当前价值函数是贪心策略时才稳定。



## 3.9.3 蒙特卡洛方法和21点

在*策略和价值函数*部分解释了蒙特卡洛方法（Monte Carlo Methods，MC），通过平均样本的回报来解决强化学习问题。对于分幕式任务，价值估计及策略改进在每个幕结束时进行，因此 MC 为逐幕做出改进，而不是每幕下的每步。类似赌博机算法采样并平均每个动作的收益，MC 采样并平均每个“状态-动作”二元组的回报，该回报取决于同一幕中后来状态采取的动作，从 MDP 采样样本的经验回报中学习价值函数。MC 算法对于每个状态的估计是独立的，即对一个状态的估计完全不依赖于对其它状态的估计；且计算一个状态的价值与状态的个数无关，因此 MC 适合在仅仅需要获得一个或者一个子集的状态的价值函数时使用。可以从这个特定的状态开始采样生成一些幕序列，然后获取回报的平均值，而无需考虑其它的状态。

### 3.9.3.1 MC 预测和同轨策略

21点游戏是手中所有扑克牌点数之和在不超过21点的情况下越大越好。对于点数：所有头牌（J、Q、K）的点数为10；A可以当作1，也可以当作11；其它牌点数同牌自身。对于游戏规则：假设每个玩家（player）都独立的与庄家（dealer）进行比赛。游戏开始时给玩家和庄家各发两张牌。庄家的牌一张正面朝上，一张背面朝上。如果玩家的两张牌一张为 A，一张为10，则为21点（A当作点数11），此种情况玩家直接胜出，称为天和（natural）。如果庄家也为天和，则平局。如果玩家不是天和，则可以一张一张继续叫牌（hits），直到主动停止（停牌）(sticks/stand)或者牌的点数和超过21点（爆牌，goes bust）。如果爆牌则玩家输。如果玩家选择停牌，就轮到庄家行动。庄家根据一个固定的策略进行游戏：庄家一直要牌，直到点数等于或超过17时停牌。如果庄家爆牌则玩家胜出，否则根据谁的点数更靠近21决定胜负或平局。

MC 预测的伪代码如下：

**Algorithm 3**First-visit MC prediction, for estimating *V*≈*vπ*​

**Input:** a policy *π* to be evaluated

**Initialize:**

*V*(*s*)∈R, arbitrarily, for all *s*∈S

Returns (s) ←← an empty list, for all *s*∈S

**while**(**Loop forever** ,for each episode)**do**

Generate an episode following *π*:*S*0​,*A*0​,*R*1​,*S*1​,*A*1​,*R*2​,…,*ST*−1​,*AT*−1​,*RT*​

*G*←0

**for**(Loop for each step of episode,  *t*=*T*−1,*T*−2,…,0 )**do**

*G*←*γG*+*Rt*+1​

Unless *St*​ appears in *S*0​,*S*1​,…,*St*−1​:

Append *G* to Returns (*St*​)

*V*(*St*​)← average (Returns(*St*​))

**end for**

**end while**

根据游戏规则书写代码，并假设所有的牌来自无穷多的一组牌（每次取出的牌再放回牌堆中）。动作（）有两个，要（叫）牌（hit）和停牌（stand），对应代码为：

%load\_ext autoreload   
%autoreload 2  
from usda.migrated\_project.RL\_an\_introduction import chapter05  
  
import matplotlib  
%matplotlib inline  
matplotlib.style.use('default')  
  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
from tqdm import tqdm

The autoreload extension is already loaded. To reload it, use:  
 %reload\_ext autoreload

# actions: hit or stand  
ACTION\_HIT = 0  
ACTION\_STAND = 1 # "strike" in the book  
ACTIONS = [ACTION\_HIT, ACTION\_STAND]  
print(ACTIONS)

[0, 1]

在第一个试验中，玩家的策略只在点数和为20或21时停牌；且，如果玩家手上有一张牌A，可以视作11点且不爆牌，则称这张牌A为可用的（usable），此时这张牌总会被视为11，因为如果视作1的化，点数总和必定小于等于11，玩家无需进行动作选择而直接继续叫牌；同理，手中牌点数总和小于等于11时，动作为继续叫牌，因此玩家的策略可写为（手牌点数和0~21对应采取的动作策略）：

# policy for player  
POLICY\_PLAYER = np.zeros(22, dtype=int)  
for i in range(12, 20):  
 POLICY\_PLAYER[i] = ACTION\_HIT  
POLICY\_PLAYER[20] = ACTION\_STAND  
POLICY\_PLAYER[21] = ACTION\_STAND  
  
print(POLICY\_PLAYER)

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1]

庄家的策略假设为直到点数等于或超过17时停牌，因此庄家的策略可写为：

# policy for dealer  
POLICY\_DEALER = np.zeros(22)  
for i in range(12, 17):  
 POLICY\_DEALER[i] = ACTION\_HIT  
for i in range(17, 22):  
 POLICY\_DEALER[i] = ACTION\_STAND  
print(POLICY\_DEALER)

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1.]

将每一局看作一幕，收益对应游戏结果的胜、负和平局，得到的收益分别为+1、-1和0。根据21点的游戏规则，模拟一局（一幕）的函数定义为play(policy\_player, initial\_state=None, initial\_action=None)（查看USDA库的迁移代码或者 Shangtong Zhang 的源码），因为玩家做出选择的动作只会依赖于3个变量，手牌总和（12 ~ 21），庄家显示的牌（A ~ 10），及是否有可用的牌A，因此一幕的返回值包括游戏初始状态state = [usable\_ace\_player, player\_sum, dealer\_card1]，包括玩家是否将牌 A 视作点数11的变量usable\_ace\_player，手牌点数和player\_sum和庄家的所显示的牌点数dealer\_card1。根据这3个变量组合可呈现的状态总共有200（）个；同时返回一幕的收益；及用于重要度采样而追踪玩家的轨迹（trajectory）player\_trajectory.append([(usable\_ace\_player, player\_sum, dealer\_card1), action])，包括状态state的3个变量和动作action。

为了通过 MC 计算上述策略的状态价值函数（usable\_ace\_player、player\_sum和dealer\_card1各个组合状态对应的回报/所有幕收益总和的均值），需要根据策略模拟尽可能多的21点游戏。在此次试验中，模拟次数（幕数）包括10,000和500,000次，以便比较不同次数的模拟结果情况。为了智能体可以持续不断的选择所有可能的动作，有两种方法可以保证这一点，分别称为同轨策略（on-policy）和离轨策略（off-policy）方法。在同轨策略中，用于生成采样数据序列的策略和用于实际决策的待评估和改进的策略相同；而在离轨策略中，用于评估或者改进的策略与生成采样数据的策略不同。

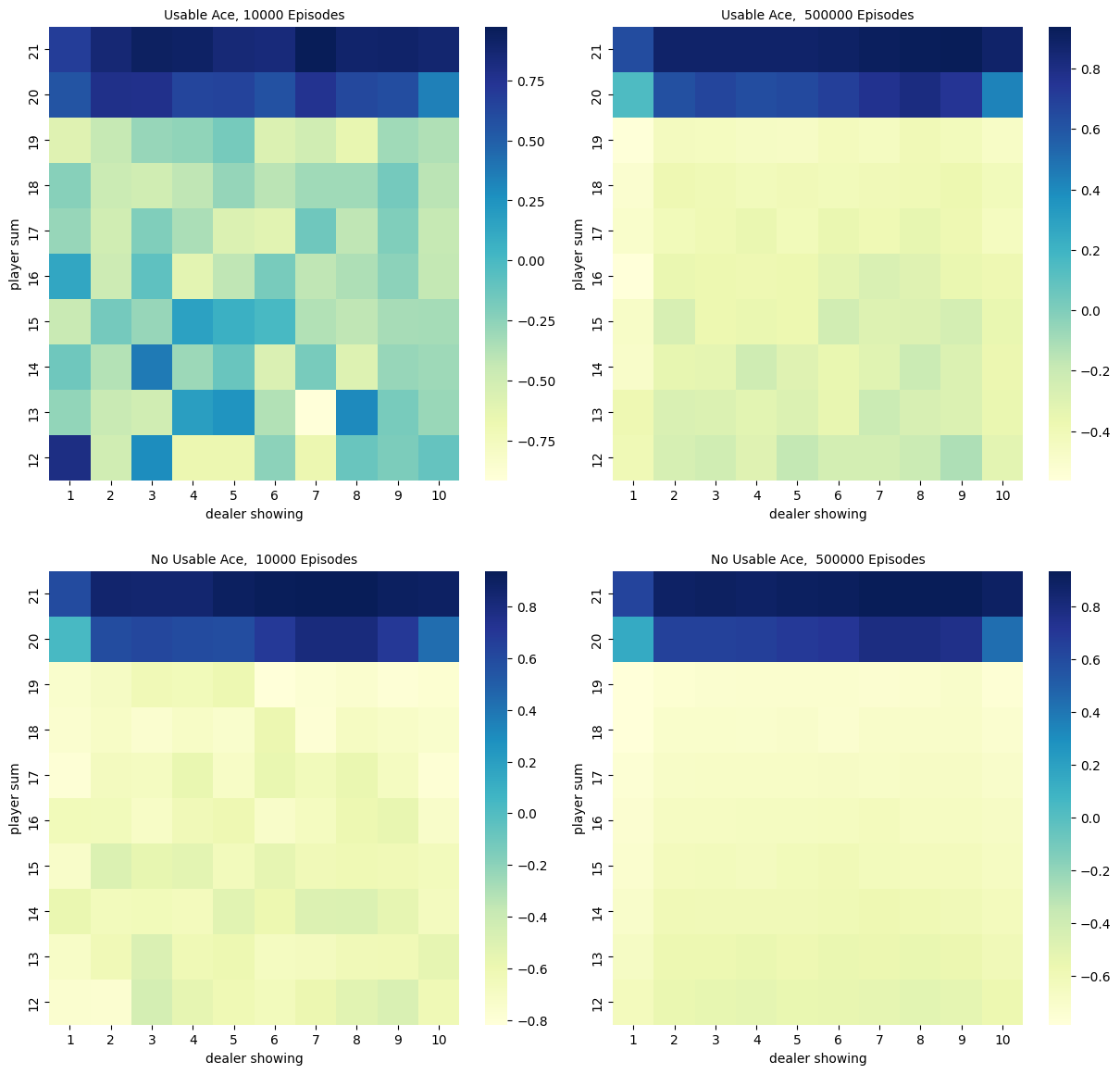
在 MC 状态价值函数预测上，使用同轨策略，动作的选择从玩家（庄家）的策略列表中根据手牌点数和对应选择待执行的动作，定义的函数为：

# function form of target policy of player  
def target\_policy\_player(usable\_ace\_player, player\_sum, dealer\_card):  
 return POLICY\_PLAYER[player\_sum]  
  
# Monte Carlo Sample with On-Policy  
def monte\_carlo\_on\_policy(episodes):  
 states\_usable\_ace = np.zeros((10, 10))  
 # initialze counts to 1 to avoid 0 being divided  
 states\_usable\_ace\_count = np.ones((10, 10))  
 states\_no\_usable\_ace = np.zeros((10, 10))  
 # initialze counts to 1 to avoid 0 being divided  
 states\_no\_usable\_ace\_count = np.ones((10, 10))  
 for i in tqdm(range(0, episodes)):  
 \_, reward, player\_trajectory = play(target\_policy\_player)  
 for (usable\_ace, player\_sum, dealer\_card), \_ in player\_trajectory:  
 player\_sum -= 12  
 dealer\_card -= 1  
 if usable\_ace:  
 states\_usable\_ace\_count[player\_sum, dealer\_card] += 1  
 states\_usable\_ace[player\_sum, dealer\_card] += reward  
 else:  
 states\_no\_usable\_ace\_count[player\_sum, dealer\_card] += 1  
 states\_no\_usable\_ace[player\_sum, dealer\_card] += reward  
 return states\_usable\_ace / states\_usable\_ace\_count, states\_no\_usable\_ace / states\_no\_usable\_ace\_count

从下述计算结果可以很容易的观察到，有可用牌A的状态的估计会更不确定、不规律，因为这样的状态相对罕见；但随着模拟观测次数的增加，如500,000次后，价值函数（value function）趋近真实价值/期望收益。从图中也可以观察到，不管庄家显示的牌为多少点数和玩家是否有可用的牌A，如果玩家初始的手牌点数总和为20或21时，具有最高的回报，这是显而易见的事实；同时，在有可用牌A时，玩家初始手牌点数和如果在12 ~ 14（15）时，具有比手牌点数在16 ~ 19区间相对高一些的回报。

chapter05.figure\_5\_1()

100%|████████████████████████████████████████| 10000/10000 [00:00<00:00, 34682.39it/s]  
100%|██████████████████████████████████████| 500000/500000 [00:12<00:00, 40959.10it/s]



### 3.9.3.2 基于试探性出发的 MC

MC 策略迭代可以逐幕交替进行评估与改进。每一幕结束后，使用观测到的回报进行测量评估，然后在该幕序列访问到的每一个状态上进行策略的改进。使用这个思路的一个简单算法称为基于试探性出发的蒙特卡洛（Monte Carlo Exploring Starts，MC ES/Monte Carlo ES），其伪代码如下，

下，

**Algorithm 4**Monte Carlo ES (Exploring Starts), for estimating *π*≈*π*∗​

**Initialize:**

*π*(*s*)∈A(*s*) (arbitrarily), for all *s*∈S

*Q*(*s*,*a*)∈R (arbitrarily), for all *s*∈S,*a*∈A(*s*)

 Returns (*s*,*a*)← empty list, for all *s*∈S,*a*∈A(*s*)

**while**(**Loop forever** ,for each episode)**do**

Choose *S*0​∈S,*A*0​∈A(*S*0​) randomly such that all pairs have probability >0>0

Generate an episode from

*S*0​,*A*0​, following *π*:*S*0​,*A*0​,*R*1​,…,*ST*−1​,*AT*−1​,*RT*​

*G*←0

**for**(Loop for each step of episode,  *t*=*T*−1,*T*−2,…,0 )**do**

*G*←*γG*+*Rt*+1​

Unless the pair *St*​,*At*​ appears in *S*0​,*A*0​,*S*1​,*A*1​…,*St*−1​,*At*−1​:

Append *G* to Returns (*St*​,*At*​)

*Q*(*St*​,*At*​)←average(Returns(*St*​,*At*​))

*π*(*St*​)←argmax*a*​*Q*(*St*​,*a*)

**end for**

**end while**

使用MC ES 解决21点问题。对于每一幕随机初始化状态和动作，包括随机等概率的选择庄家的扑克牌、玩家手牌的点数和确定是否有可用的牌A，其对应的代码为：

initial\_state = [bool(np.random.choice([0, 1])),  
 np.random.choice(range(12, 22)),  
 np.random.choice(range(1, 11))]

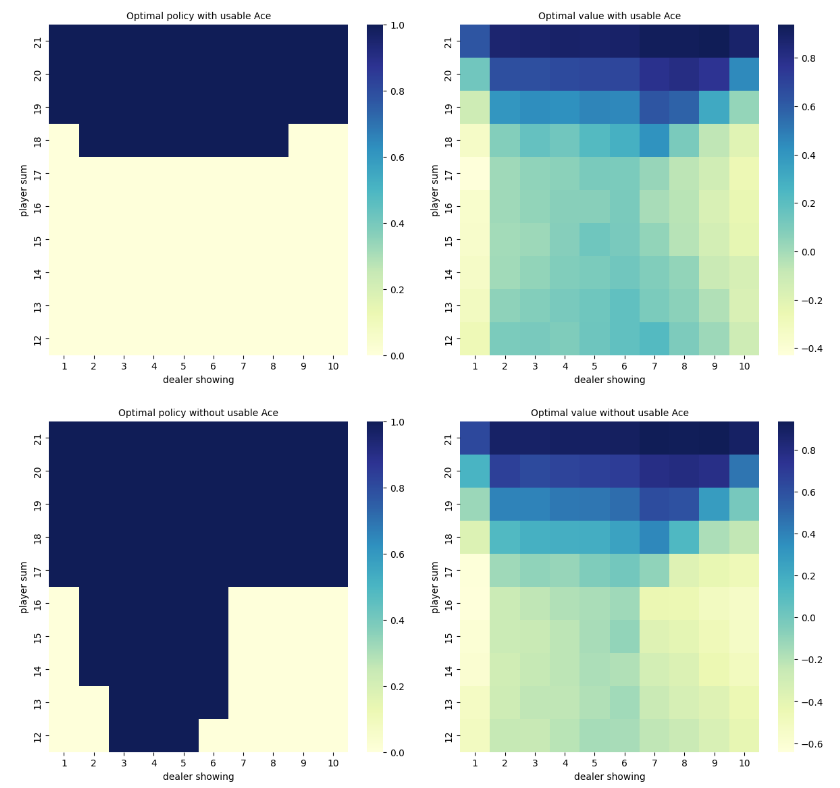
初始策略同前一试验定义的target\_policy\_player()方式，之后选择对应动作价值函数最大的动作，即策略改进的方法是当前价值函数上贪心的选择动作，对任意一个状态，有，对应的代码为下述代码中behavior\_policy()函数。定义的monte\_carlo\_es()函数实现了Monte Carlo ES算法，如下：

# Monte Carlo with Exploring Starts  
def monte\_carlo\_es(episodes):  
 # (playerSum, dealerCard, usableAce, action)  
 state\_action\_values = np.zeros((10, 10, 2, 2))  
 # initialze counts to 1 to avoid division by 0  
 state\_action\_pair\_count = np.ones((10, 10, 2, 2))  
  
 # behavior policy is greedy  
 def behavior\_policy(usable\_ace, player\_sum, dealer\_card):  
 usable\_ace = int(usable\_ace)  
 player\_sum -= 12  
 dealer\_card -= 1  
 # get argmax of the average returns(s, a)  
 values\_ = state\_action\_values[player\_sum, dealer\_card, usable\_ace, :] / \  
 state\_action\_pair\_count[player\_sum, dealer\_card, usable\_ace, :]  
 return np.random.choice([action\_ for action\_, value\_ in enumerate(values\_) if value\_ == np.max(values\_)])  
  
 # play for several episodes  
 for episode in tqdm(range(episodes)):  
 # for each episode, use a randomly initialized state and action  
 initial\_state = [bool(np.random.choice([0, 1])),  
 np.random.choice(range(12, 22)),  
 np.random.choice(range(1, 11))]  
 initial\_action = np.random.choice(ACTIONS)  
 current\_policy = behavior\_policy if episode else target\_policy\_player  
 \_, reward, trajectory = play(current\_policy, initial\_state, initial\_action)  
 first\_visit\_check = set()  
 for (usable\_ace, player\_sum, dealer\_card), action in trajectory:  
 usable\_ace = int(usable\_ace)  
 player\_sum -= 12  
 dealer\_card -= 1  
 state\_action = (usable\_ace, player\_sum, dealer\_card, action)  
 if state\_action in first\_visit\_check:  
 continue  
 first\_visit\_check.add(state\_action)  
 # update values of state-action pairs  
 state\_action\_values[player\_sum, dealer\_card, usable\_ace, action] += reward  
 state\_action\_pair\_count[player\_sum, dealer\_card, usable\_ace, action] += 1  
  
 return state\_action\_values / state\_action\_pair\_count

计算结果如下，图左列为有无可用的牌A下的最优策略，右侧为各自对应的状态价值函数。从最优策略中可以对应任意状态选择最优的策略以获取最好的回报。

chapter05.figure\_5\_2()

100%|███████████████████████████████████████| 500000/500000 [00:51<00:00, 9731.88it/s]



### 3.9.3.3 基于重要度采样的离轨策略估计

#### 1）离轨策略

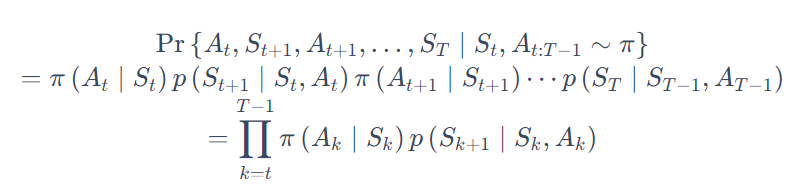
所有的学习控制方法都面临一个困境：希望学到的动作可以使随后的智能体行为是最优的，但是为了保证找到最优动作却需要搜索所有动作，采取非最优的行动。为了在遵循试探策略采取行动同时学习到最优策略，一个比较直接的方法是采用两个策略，一个用来学习并最终成为最优策略，称为目标策略（target policy）；另一更具有试探性，用来产生智能体的行动样本，称为行动策略（behavior policy）。因为行动策略学习所用的数据“离开（off）”了带学习的目标策略，因此整个过程被称为离轨策略学习（o↵-policy learning）。在预测任务中，假设希望预测或，但需要遵循另一个策略所获得的若干幕样本，其中即为目标策略，而为行动策略，两个策略都固定且已知。

为了使用从得到的多幕样本序列去预测，要求在下发生的每个动作都至少偶尔能在下发生，即，对任意，要求，称其为覆盖假设（assumption of coverage）。根据这个假设，在与不同的状态中，必须是随机的；另一方面，目标策略则可能是确定性的。对应代码中目标策略即为前文定义的target\_policy\_player()；行动策略定义为behavior\_policy\_player()函数，如下，

# function form of behavior policy of player  
def behavior\_policy\_player(usable\_ace\_player, player\_sum, dealer\_card):  
 if np.random.binomial(1, 0.5) == 1:  
 return ACTION\_STAND  
 return ACTION\_HIT

#### 2）重要度采样（importance sampling）

几乎所有的离轨策略方法都采用了重要度采样，其是一种在给定来自其它分布样本条件下，估计某种分布期望值的通用方法。将重要度采样应用于离轨策略学习，对回报值根据其轨迹在目标策略与行动策略中出现的相对概率进行加权，这个相对概率也被称为重要度采样比（importance-sampling ratio）。给定起始状态，后续状态-动作轨迹$A\_t, S\_{t+1}, A\_{t+1}, , S\_T *π*下发生的概率是，



式中， 是在*智能体-环境交互接口*公式（1）中定义的状态转移概率函数。因此，在目标函数和行动策略轨迹下的相对概率，即重要度采样比（importance-sampling ratio）是，。尽管整体轨迹概率值与 MDP 的状态转移概率（未知的）有关，但是分子和分母中完全相同，可以被约分。因此，重要度采样比只与两个策略和样本序列数据相关，而与 MDP 的动态特性（状态转移概率）无关。估计目标策略下的期望回报（价值）时，只从行动策略中估计并不准确，因此应用比例系数可以调整回报使其具有正确的期望值，。

对各幕中的时刻跨幕连续编号，从而使用唯一时刻编号指代特定幕中的特定时刻。并定义所有访问过状态s的时刻为T(s)，用T(t)表示在时刻t后的首次终止，用Gt​表示在t之后到达T(t)时的回报值。那么{Gt​}t∈T(s)​就是状态s对应的回报值， {ρt:T(t)−1​}t∈T(s)​是相应的重要度采样比。为了预测vπ​(s)，根据重要度采样比调整回报值并对结果进行平均，有，V(s)≐∣T(s)∣∑t∈T(s)​ρt:T(t)−1​Gt​​，这种通过简单平均实现的重要度采样被称为普通重要度采样（ordinary importance sampling）；如果采用加权平均的方法，有， V(s)≐∑t∈T(s)​ρt:T(t)−1​∑t∈T(s)​ρt:T(t)−1​Gt​​，称为加权重要度采样（weighted importance sampling）。

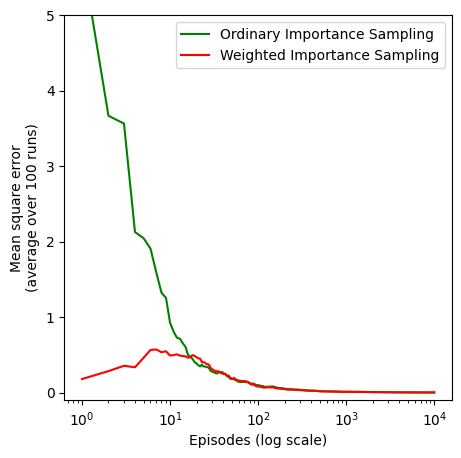
对应的代码定义于monte\_carlo\_off\_policy()中，如下：

# Monte Carlo Sample with Off-Policy  
def monte\_carlo\_off\_policy(episodes,initial\_state = [True, 13, 2]):  
 rhos = []  
 returns = []  
  
 for i in range(0, episodes):  
 \_, reward, player\_trajectory = play(behavior\_policy\_player, initial\_state=initial\_state)  
  
 # get the importance ratio  
 numerator = 1.0  
 denominator = 1.0  
 for (usable\_ace, player\_sum, dealer\_card), action in player\_trajectory:  
 if action == target\_policy\_player(usable\_ace, player\_sum, dealer\_card):  
 denominator \*= 0.5  
 else:  
 numerator = 0.0  
 break  
 rho = numerator / denominator  
 rhos.append(rho)  
 returns.append(reward)  
  
 rhos = np.asarray(rhos)  
 returns = np.asarray(returns)  
 weighted\_returns = rhos \* returns  
  
 weighted\_returns = np.add.accumulate(weighted\_returns)  
 rhos = np.add.accumulate(rhos)  
  
 ordinary\_sampling = weighted\_returns / np.arange(1, episodes + 1)  
  
 with np.errstate(divide='ignore',invalid='ignore'):  
 weighted\_sampling = np.where(rhos != 0, weighted\_returns / rhos, 0)  
  
 return ordinary\_sampling, weighted\_sampling

试验21点游戏中状态值的离轨策略估计，并分别采用了普通重要度采样和加权重要度采样。假设初始状态配置为initial\_state = [True, 13, 2]，即玩家有一张可用的牌A，手牌点数总和为13，而庄家露出的牌为2。通过从这个初始状态开始等概率选择要牌或停牌（行动策略）可以得到采样数据。目标策略只有在手牌点数和达到20或21时停牌。利用目标策略独立生成1亿幕数据后对回报进行平均可以得到状态值为 -0.27726。两种离轨策略方法分别独立运行100次，每次都从估计为0开始学习10,000幕，并将运行100次后的结果平均。从下述结果可以观察到，两种算法的错误率最终都趋近于0，但是加权重要度采样在开始时错误率明显较低。

chapter05.figure\_5\_3(true\_value = -0.27726,episodes = 10000,runs = 100)

100%|███████████████████████████████████████████████| 100/100 [00:17<00:00, 5.57it/s]



## 3.9.4 时序差分学习

时序差分（temporal-difference，TD）、动态规划（dynamic programming，DP）和蒙特卡洛（Monte Carlo Methods，MC）这三种方法之间的关系时强化学习（reinforcement learning，RL）领域经常出现的话题。TD 学习结合了 MC 和 DP 方法思想，可以从环境互动的经验中学习策略，而不需要构建关于环境动态特性（状态转移概率）的模型；且无需等待交互的结果（获得一幕结束后的回报），而是基于已获得其它状态的估计值来更新当前状态的价值函数。

### 3.9.4.1 时序差分预测和随机游走

#### 1）时序差分预测

适用于非平稳环境简单的每次访问型 MC 方法可以表示为，，式中，是时刻真实的回报，是常量步长参数（对比*多臂赌博机*增量式公式（2））。该方法称为常量 MC。MC 方法必须等到一幕结束后才能确定的增量，而 TD 方法只需每幕下的一个时刻结束，在时刻，使用观察到的和估计值进行一次有效更新，有，。实际上 MC 更新的目标是，而 TD 更新的目标是。这种 TD 方法被称为 TD(0)，或单步 TD（one-step TD）。

在 TD(0)的更新中，括号里的数值是一种误差，衡量的是的估计值与更好的估计之间的差异，，被称为 TD 误差。每个时刻 TD 误差是当前时刻估计的误差，并取决于下一个状态和下一个收益。如果价值函数数组在一幕内没有改变，则 MC 误差可以写为 TD 误差之和，

TD(0) 伪代码如下：

**Algorithm 5**Tabular TD(0)TD(0) for estimating *vπ*​

**Input:** the policy *π* to be evaluated

**Algorithm parameter:** step size *α*∈(0,1]

**Initialize***V*(*s*), for all *s*∈S+, arbitrarily except that  (*V*( terminal )=0)=0

**for**(Loop for each episode)**do**

**Initialize** *S*

**for**(Loop for each step of episode)**do**

*A*← action given by *π* for *S*

Take action *A*, observe *R*,*S*′

*V*(*S*)←*V*(*S*)+*α*[*R*+*γV*(*S*′)−*V*(*S*)]

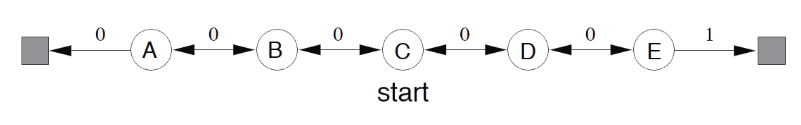
*S*←*S*′

**end for**

**end for**

#### 2）随机游走

马尔可夫收益过程（Markov reward process，MRP）是不包含动作的马尔可夫决策过程，这个过程并不区分价值变化是由环境还是由智能体引起的变化。如图[Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.]125，所有阶段都从中心状态开始，在每个时刻以相同的概率向左或向右移动一个状态。幕终止于最左侧或最右侧。终止于最右侧时，有 +1 的收益；除此之外的收益均为0。例如，一个典型的幕可能包含一下“状态-收益”序列：。由于这个任务没有折扣，所以每个状态的真实价值是从这个状态开始并终止于最右侧的概率。因此中心状态的真实价值为，状态的真实价值分别为：。



%load\_ext autoreload   
%autoreload 2  
from usda.migrated\_project.RL\_an\_introduction import chapter06  
from usda.migrated\_project.RL\_an\_introduction import chapter07  
  
import numpy as np

The autoreload extension is already loaded. To reload it, use:  
 %reload\_ext autoreload

根据上文对 MRP 过程的描述，定义真实价值TRUE\_VALUE，和一个初始状态价值VALUES。

print(f'{chapter06.random\_walk.TRUE\_VALUE}\n{chapter06.random\_walk.VALUES}')

[0. 0.16666667 0.33333333 0.5 0.66666667 0.83333333  
 1. ]  
[0. 0.5 0.5 0.5 0.5 0.5 1. ]

动作有两个（实际只需定义一个变量），为向左ACTION\_LEFT = 0和向右ACTION\_RIGHT = 1。

定义 TD 函数temporal\_difference()，其中values[old\_state] += alpha \* (reward + values[state] - values[old\_state])对应 TD(0)公式（2）.

# @values: current states value, will be updated if @batch is False  
# @alpha: step size  
# @batch: whether to update @values  
def temporal\_difference(values, alpha=0.1, batch=False):  
 state = 3  
 trajectory = [state]  
 rewards = [0]  
 while True:  
 old\_state = state  
 if np.random.binomial(1, 0.5) == ACTION\_LEFT:  
 state -= 1  
 else:  
 state += 1  
 # Assume all rewards are 0  
 reward = 0  
 trajectory.append(state)  
 # TD update  
 if not batch:  
 values[old\_state] += alpha \* (reward + values[state] - values[old\_state])  
 if state == 6 or state == 0:  
 break  
 rewards.append(reward)  
 return trajectory, rewards

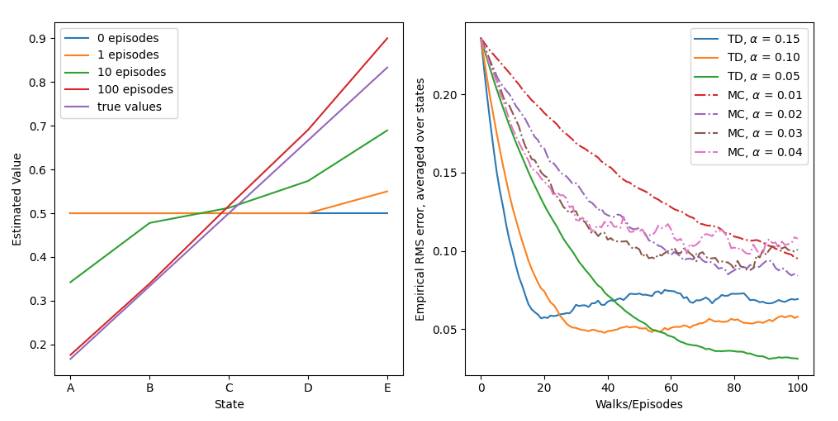
案例比较了 TD(0) 和 MC 两种方法，因此定义 MC方法的函数monte\_carlo()，其中values[state\_] += alpha \* (returns - values[state\_])对应 MC公式（1）。

# @values: current states value, will be updated if @batch is False  
# @alpha: step size  
# @batch: whether to update @values  
def monte\_carlo(values, alpha=0.1, batch=False):  
 state = 3  
 trajectory = [state]  
  
 # if end up with left terminal state, all returns are 0  
 # if end up with right terminal state, all returns are 1  
 while True:  
 if np.random.binomial(1, 0.5) == ACTION\_LEFT:  
 state -= 1  
 else:  
 state += 1  
 trajectory.append(state)  
 if state == 6:  
 returns = 1.0  
 break  
 elif state == 0:  
 returns = 0.0  
 break  
  
 if not batch:  
 for state\_ in trajectory[:-1]:  
 # MC update  
 values[state\_] += alpha \* (returns - values[state\_])  
 return trajectory, [returns] \* (len(trajectory) - 1)

下图左显示经历不同数量的幕采样序列之后，运行一次 TD(0) 所得到的价值估计值。在 100 幕后，估计值已经非常接近真实值。由于使用了常数步长参数（配置），估计值会反映较近若干幕的结果，不规律的上下波动。右图则给出了不同的取值，及 TD(0)和 MC 两种方法的计算结果。从结果可以观察到 TD(0) 要好于 MC。

chapter06.example\_6\_2()

100%|██████████████████████████████████████████████| 100/100 [00:00<00:00, 280.16it/s]  
100%|██████████████████████████████████████████████| 100/100 [00:00<00:00, 300.48it/s]  
100%|██████████████████████████████████████████████| 100/100 [00:00<00:00, 302.06it/s]  
100%|██████████████████████████████████████████████| 100/100 [00:00<00:00, 362.10it/s]  
100%|██████████████████████████████████████████████| 100/100 [00:00<00:00, 326.72it/s]  
100%|██████████████████████████████████████████████| 100/100 [00:00<00:00, 342.83it/s]  
100%|██████████████████████████████████████████████| 100/100 [00:00<00:00, 337.79it/s]



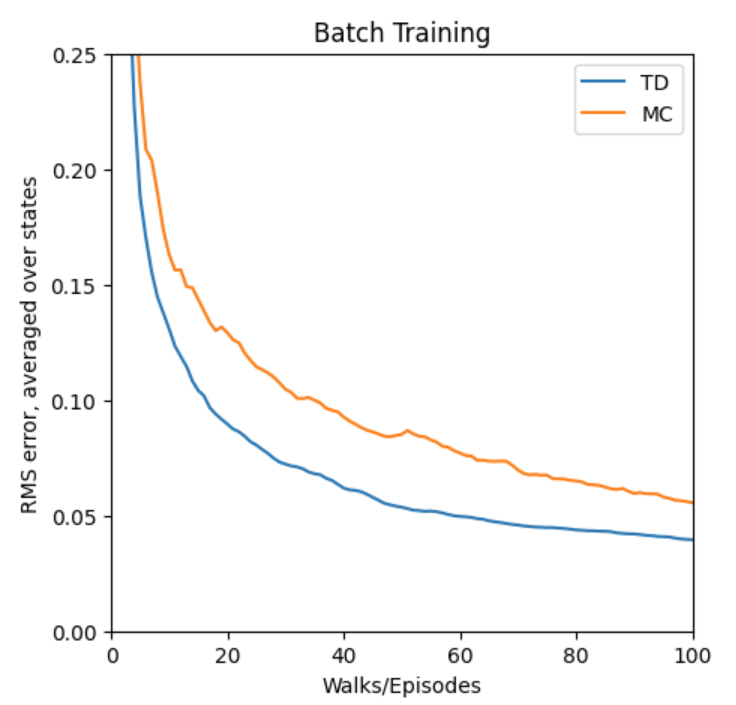
#### 3）批量更新的随机游走

如果价值函数仅根据所有增量的和更新一次，然后利用新的值函数再次处理所有可用的经验，产生新的总增量，依此类推，直至价值函数收敛，称为批量更新（batch updating）（因为只有处理了整批的训练数据后才进行更新）。在批量更新下，只要选择足够小的步长参数， TD(0)就能确定收敛到与无关的唯一结果。对于批量更新的随机游走，每经过新的一幕序列之后，之前所有幕的数据就被视为一个批次。TD(0)和 MC 不断使用这些批次进行逐次更新。 最后将所得的价值函数与进行比较，绘制在5个状态下的平均均方根误差（以整个试验的100次独立重复为基础）的学习曲线。对应的代码为batch\_updating()函数。

# Figure 6.2  
# @method: 'TD' or 'MC'  
def batch\_updating(method, episodes, alpha=0.001,runs = 100):  
 # perform 100 independent runs   
 total\_errors = np.zeros(episodes)  
 for r in tqdm(range(0, runs)):  
 current\_values = np.copy(VALUES)  
 current\_values[1:6] = -1  
 errors = []  
 # track shown trajectories and reward/return sequences  
 trajectories = []  
 rewards = []  
 for ep in range(episodes):  
 if method == 'TD':  
 trajectory\_, rewards\_ = temporal\_difference(current\_values, batch=True)  
 else:  
 trajectory\_, rewards\_ = monte\_carlo(current\_values, batch=True)  
 trajectories.append(trajectory\_)  
 rewards.append(rewards\_)  
 while True:  
 # keep feeding our algorithm with trajectories seen so far until state value function converges  
 updates = np.zeros(7)  
 for trajectory\_, rewards\_ in zip(trajectories, rewards):  
 for i in range(0, len(trajectory\_) - 1):  
 if method == 'TD':  
 updates[trajectory\_[i]] += rewards\_[i] + current\_values[trajectory\_[i + 1]] - current\_values[trajectory\_[i]]  
 else:  
 updates[trajectory\_[i]] += rewards\_[i] - current\_values[trajectory\_[i]]  
 updates \*= alpha  
 if np.sum(np.abs(updates)) < 1e-3:  
 break  
 # perform batch updating  
 current\_values += updates  
 # calculate rms error  
 errors.append(np.sqrt(np.sum(np.power(current\_values - TRUE\_VALUE, 2)) / 5.0))  
 total\_errors += np.asarray(errors)  
 total\_errors /= runs  
 return total\_errors

chapter06.figure\_6\_2(episodes = 100 + 1,alpha=0.001,runs = 100)

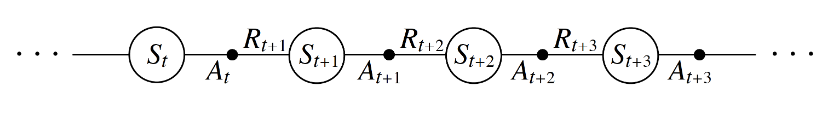
100%|███████████████████████████████████████████████| 100/100 [00:42<00:00, 2.35it/s]  
100%|███████████████████████████████████████████████| 100/100 [00:35<00:00, 2.79it/s]



### 3.9.4.2 Sarsa：同轨策略下的时序差分控制

#### 1）Sarsa

同轨策略下的时序差分（on-policy TD）控制方法，先学习动作价值函数，对所有状态和动作，估计出当前行动策略下对应的。状态-动作二元组的“一幕数据”是状态和动作交替出现的序列，如图[2]129，



前文随机游走的试验讨论了状态之间的转移并学习状态价值函数，而对于“状态-动作”二元组之间的转移并学习“状态-动作”二元组的价值，在数学形式上两者相同：都是带有收益过程的马尔科夫链。确保状态值在 TD(0) 下收敛的定理同样适用于动作价值函数，有，。每当从非终止状态的出现一次转移之后，就进行一次公式（3）的更新。如果是终止状态，则定义为0。这个更新规则用到了描述这个事件的五元组中的所有元素，因此称为 Sarsa。

采用-贪心（-greedy）策略，使得“状态-动作”二元组可以被无限多次的访问到，Sarsa算法收敛到最优策略和动作价值函数，伪代码如下，

**Algorithm 6**Sarsa (on-policy TD control) for estimating *Q*≈*q*∗​

**Algorithm parameter:** step size *α*∈(0,1], small *ε*>0

**Initialize***Q*(*s*,*a*), for all *s*∈S+,*a*∈A(*s*), arbitrarily except that  (*Q*( terminal,⋅)=0⋅)=0

**for**(Loop for each episode)**do**

**Initialize***S*

Choose *A* from *S* using policy derived from *Q* (e.g., *ε*-greedy)

**for**(Loop for each step of episode)**do**

Take action *A*, observe *R*,*S*′

Choose*A*′ from*S*′ using policy derived from *Q* (e.g., *ε*-greedy)

*Q*(*S*,*A*)←*Q*(*S*,*A*)+*α*[*R*+*γQ*(*S*′,*A*′)−*Q*(*S*,*A*)]

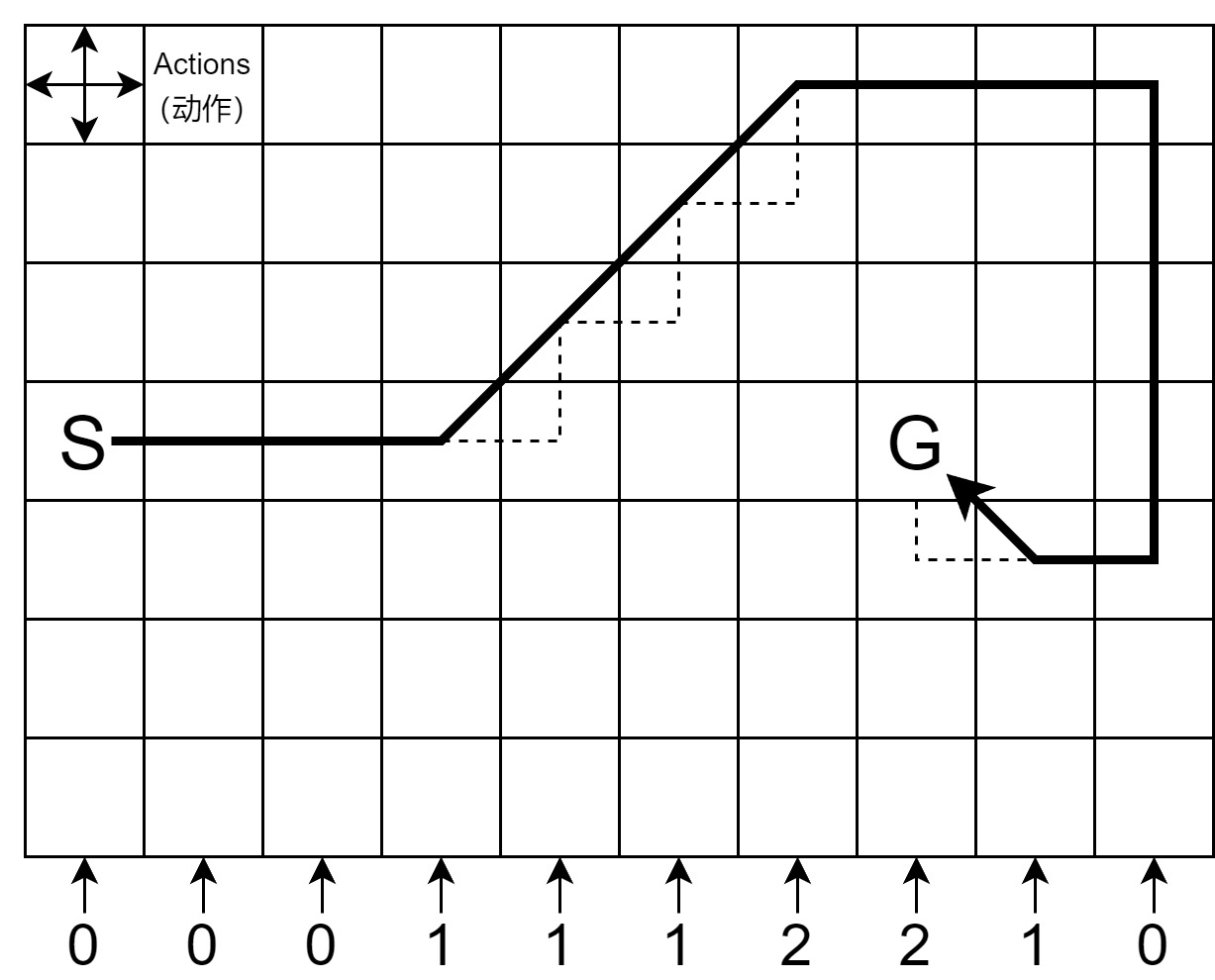
*S*←*S*′;*A*←*A*′;

**end for**

until *S* is terminal

**end for**

#### 2）有风的网格世界



在一个的网格中，设置起始状态 S 为START = [3, 0]，目标状态 G 为GOAL = [3, 7]；智能体有4个动作，上下左右，定义为ACTION\_UP = 0; ACTION\_DOWN = 1; ACTION\_LEFT = 2; ACTION\_RIGHT = 3，置于一个列表为ACTIONS = [ACTION\_UP, ACTION\_DOWN, ACTION\_LEFT, ACTION\_RIGHT]；每列网格都存在一个向上的风向，但风力的大小（向上移动的格数）可能存在差异，定义为WIND = [0, 0, 0, 1, 1, 1, 2, 2, 1, 0]。如果智能体有一个向右的动作，但是向上为1的风力使得该智能体在向右移动一个单元时，也向上移动一个单元。如果智能体处于上下左右边界，则将其推往网格之外对应的动作及风力的作用对其无效。定义step()函数描述状态-动作策略，代码如下，

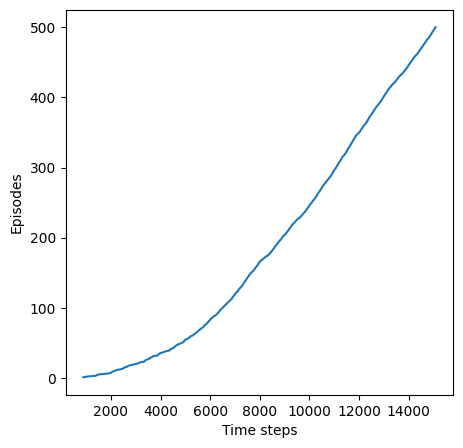
def step(state, action):  
 i, j = state  
 if action == ACTION\_UP:  
 return [max(i - 1 - WIND[j], 0), j]  
 elif action == ACTION\_DOWN:  
 return [max(min(i + 1 - WIND[j], WORLD\_HEIGHT - 1), 0), j]  
 elif action == ACTION\_LEFT:  
 return [max(i - WIND[j], 0), max(j - 1, 0)]  
 elif action == ACTION\_RIGHT:  
 return [max(i - WIND[j], 0), min(j + 1, WORLD\_WIDTH - 1)]  
 else:  
 assert False

这个任务中，在到达目标 G 之前，每步会收到恒定为 -1 的收益，定义REWARD = -1.0；同时配置-贪心（开发/试探的权衡中）策略的，即EPSILON = 0.1；配置 Sarsa 的步长大小，即ALPHA = 0.5；对所有初始化为，即q\_value = np.zeros((WORLD\_HEIGHT, WORLD\_WIDTH, 4))。定义 Sarsa 同轨策略下的 TD 控制函数episode()如下：

# play for an episode  
def episode(q\_value):  
 # track the total time steps in this episode  
 time = 0  
  
 # initialize state  
 state = START  
  
 # choose an action based on epsilon-greedy algorithm  
 if np.random.binomial(1, EPSILON) == 1:  
 action = np.random.choice(ACTIONS)  
 else:  
 values\_ = q\_value[state[0], state[1], :]  
 action = np.random.choice([action\_ for action\_, value\_ in enumerate(values\_) if value\_ == np.max(values\_)])  
  
 # keep going until get to the goal state  
 while state != GOAL:  
 next\_state = step(state, action)  
 if np.random.binomial(1, EPSILON) == 1:  
 next\_action = np.random.choice(ACTIONS)  
 else:  
 values\_ = q\_value[next\_state[0], next\_state[1], :]  
 next\_action = np.random.choice([action\_ for action\_, value\_ in enumerate(values\_) if value\_ == np.max(values\_)])  
  
 # Sarsa update  
 q\_value[state[0], state[1], action] += \  
 ALPHA \* (REWARD + q\_value[next\_state[0], next\_state[1], next\_action] -  
 q\_value[state[0], state[1], action])  
 state = next\_state  
 action = next\_action  
 time += 1  
 return time

打印结果为每一幕的时刻数，曲线越趋于平缓，每一幕的时刻数就越多，从起始状态到目标状态的步数就越多，即 Sarsa 算法还没收敛到最优策略和动作价值函数；而曲线越陡，每一幕的时刻数就越少，从起始状态到目标状态的步数就越少，即 Sarsa 算法逐步收敛到最优策略和动作价值函数。同时打印迭代 500 幕之后的最优动作策略。

chapter06.figure\_6\_3()



Optimal policy is:  
['L', 'R', 'R', 'R', 'R', 'R', 'U', 'R', 'R', 'D']  
['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'L', 'D']  
['R', 'U', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'D']  
['R', 'R', 'R', 'R', 'R', 'R', 'R', 'G', 'D', 'D']  
['D', 'D', 'R', 'R', 'R', 'R', 'U', 'D', 'L', 'L']  
['D', 'R', 'D', 'R', 'R', 'U', 'U', 'D', 'L', 'L']  
['R', 'U', 'R', 'R', 'U', 'U', 'U', 'U', 'U', 'L']  
Wind strength for each column:  
['0', '0', '0', '1', '1', '1', '2', '2', '1', '0']

### 3.9.4.3 Q 学习：离轨策略下的时序差分控制

#### 1）Q 学习

离轨策略下的时序差分控制算法被称为 Q 学习（Q-learning，QL），定义为。QL 中待学习的动作价值函数采用了对最优动作价值函数的直接近似作为学习目标，而与用于生成智能体决策序列轨迹的行动策略是什么无关；而 Sarsa 的学习目标中使用的是待学习的动作价值本身，需要知道下一时刻的动作，因此与生成数据的行动策略相关。QL 的伪代码如下，

下，

**Algorithm 7**Q-learning (off-policy TD control) for estimating  *π*≈*π*∗​

**Algorithm parameter:** step size *α*∈(0,1], small *ε*>0

**Initialize** *Q*(*s*,*a*), for all *s*∈S+,*a*∈A(*s*), arbitrarily except that (*Q*( terminal,⋅)=0⋅)=0

**for**(Loop for each episode)**do**

**Initialize***S*

**for**(Loop for each step of episode)**do**

Choose *A* from *S* using policy derived from *Q* (e.g., *ε*-greedy)

Take action*A*, observe *R*,*S*′

*Q*(*S*,*A*)←*Q*(*S*,*A*)+*α*[*R*+*γ*max*a*​*Q*(*S*′,*a*)−*Q*(*S*,*A*)]

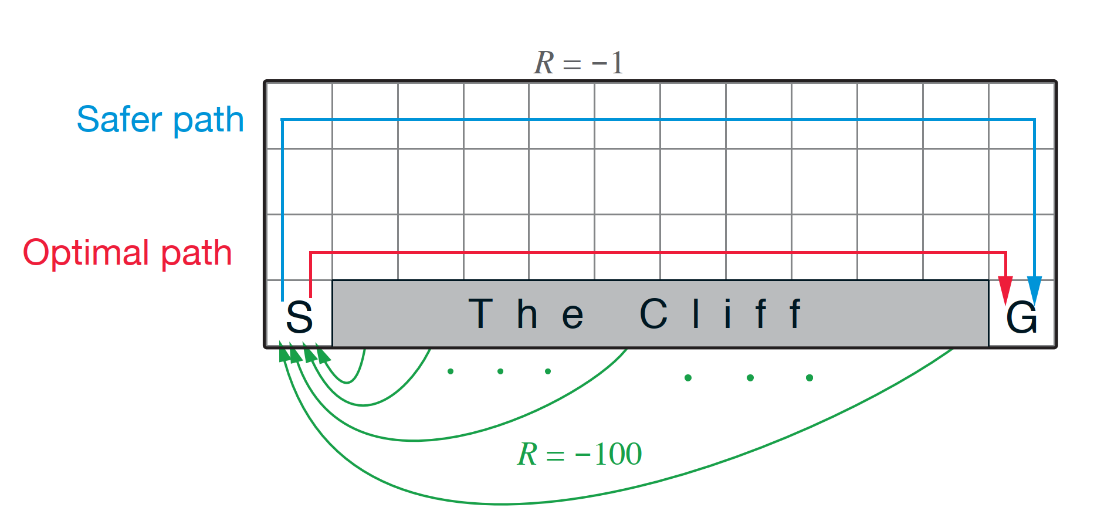
*S*←*S*′

**end for**

until *S* is terminal

**end for**

#### 2）在悬崖边上行走



上图[2]132构建了一个网格，包含起始状态 S 和目标状态 G ，环境中包含一行悬崖，如果动作导致掉入该一行悬崖则收益为-100，并同时将智能体送回起点；而其它的转移得到的收益都是-1。计算比较同轨策略（Sarsa）和离轨策略（QL）两类方法间的区别，其中 QL 对应q\_learning()函数如下，

# an episode with Q-Learning  
# @q\_value: values for state action pair, will be updated  
# @step\_size: step size for updating  
# @return: total rewards within this episode  
def q\_learning(q\_value, step\_size=ALPHA):  
 state = START  
 rewards = 0.0  
 while state != GOAL:  
 action = choose\_action(state, q\_value)  
 next\_state, reward = step(state, action)  
 rewards += reward  
 # Q-Learning update  
 q\_value[state[0], state[1], action] += step\_size \* (  
 reward + GAMMA \* np.max(q\_value[next\_state[0], next\_state[1], :]) -  
 q\_value[state[0], state[1], action])  
 state = next\_state  
 return rewards

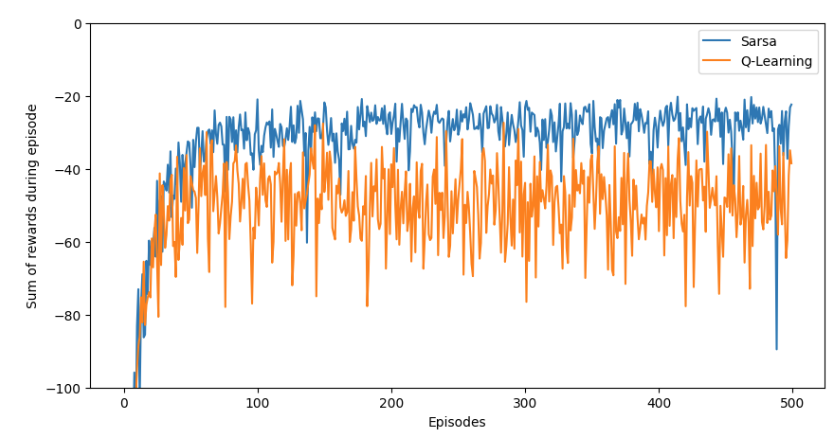
Sarsa 对应sarsa()函数如下代码，并通过配置expected=True参数，包含了期望 Sarsa 算法。

# an episode with Sarsa  
# @q\_value: values for state action pair, will be updated  
# @expected: if True, will use expected Sarsa algorithm  
# @step\_size: step size for updating  
# @return: total rewards within this episode  
def sarsa(q\_value, expected=False, step\_size=ALPHA):  
 state = START  
 action = choose\_action(state, q\_value)  
 rewards = 0.0  
 while state != GOAL:  
 next\_state, reward = step(state, action)  
 next\_action = choose\_action(next\_state, q\_value)  
 rewards += reward  
 if not expected:  
 target = q\_value[next\_state[0], next\_state[1], next\_action]  
 else:  
 # calculate the expected value of new state  
 target = 0.0  
 q\_next = q\_value[next\_state[0], next\_state[1], :]  
 best\_actions = np.argwhere(q\_next == np.max(q\_next))  
 for action\_ in ACTIONS:  
 if action\_ in best\_actions:  
 target += ((1.0 - EPSILON) / len(best\_actions) + EPSILON / len(ACTIONS)) \* q\_value[next\_state[0], next\_state[1], action\_]  
 else:  
 target += EPSILON / len(ACTIONS) \* q\_value[next\_state[0], next\_state[1], action\_]  
 target \*= GAMMA  
 q\_value[state[0], state[1], action] += step\_size \* (  
 reward + target - q\_value[state[0], state[1], action])  
 state = next\_state  
 action = next\_action  
 return rewards

从计算结果来看，QL 学习到了最优策略，沿着悬崖边上走的策略。但是，由于动作是通过-贪心的方式选择，因此执行这个策略时，智能体会偶尔掉入悬崖；而 Sarsa 则考虑了动作被选取的方式，学习到了一条通过网格上半部分的路径，虽然路径更长但更安全。

chapter06.figure\_6\_4(figsize=(10,5))

100%|█████████████████████████████████████████████████| 50/50 [00:53<00:00, 1.07s/it]



Sarsa Optimal Policy:  
['R', 'R', 'R', 'R', 'R', 'R', 'R', 'D', 'L', 'L', 'R', 'D']  
['U', 'R', 'U', 'U', 'U', 'L', 'U', 'R', 'R', 'R', 'R', 'D']  
['U', 'R', 'U', 'U', 'R', 'R', 'U', 'U', 'L', 'U', 'R', 'D']  
['U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'G']  
Q-Learning Optimal Policy:  
['R', 'D', 'U', 'R', 'D', 'R', 'R', 'R', 'R', 'R', 'D', 'D']  
['D', 'R', 'R', 'D', 'R', 'R', 'R', 'R', 'D', 'R', 'R', 'D']  
['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'D']  
['U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'G']

### 3.9.4.4 期望 Sarsa

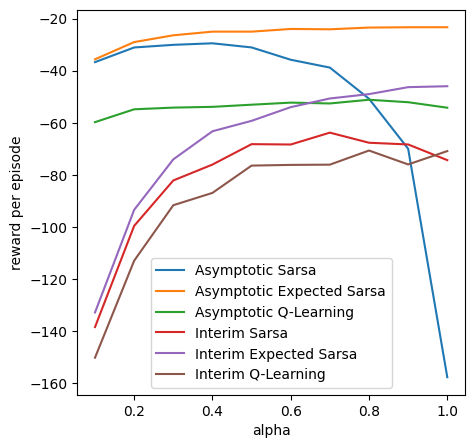
将 QL 中对下一个“状态-动作”二元组取最大值这一步换成取期望的学习算法，更新规则为，

给定下一个状态，该算法确定向期望意义上的 Sarsa 算法所决定的方向上移动，因此该算法称为期望 Sarsa（Expected Sarsa）。

期望 Sarsa 计算上较之 Sarsa 复杂，但是消除了因为随机选择而产生的方差。继续使用悬崖边上行走的任务，比较期望 Sarsa、Sarsa 和 QL 结果如下。期望 Sarsa 保持了 Sarsa 优于 QL 的显著优势。在步长参数大量不同的取值下，期望 Sarsa 都显著的优于 Sarsa。同时，Sarsa 仅仅在值较小时能够有良好的长期表现，而启动期瞬态表现则不如意。因此期望 Sarsa 相比 Sarsa 在试验中表现出明显优势。

chapter06.figure\_6\_6()

100%|█████████████████████████████████████████████████| 10/10 [00:44<00:00, 4.48s/it]  
100%|█████████████████████████████████████████████████| 10/10 [00:42<00:00, 4.21s/it]  
100%|█████████████████████████████████████████████████| 10/10 [00:44<00:00, 4.44s/it]  
100%|█████████████████████████████████████████████████| 10/10 [00:45<00:00, 4.57s/it]  
100%|█████████████████████████████████████████████████| 10/10 [00:44<00:00, 4.50s/it]  
100%|█████████████████████████████████████████████████| 10/10 [00:43<00:00, 4.36s/it]  
100%|█████████████████████████████████████████████████| 10/10 [00:43<00:00, 4.31s/it]  
100%|█████████████████████████████████████████████████| 10/10 [00:43<00:00, 4.31s/it]  
100%|█████████████████████████████████████████████████| 10/10 [00:44<00:00, 4.48s/it]  
100%|█████████████████████████████████████████████████| 10/10 [00:41<00:00, 4.17s/it]



### 3.9.4.5 步自举法（-step Bootstrapping）

#### 1）步时序差分方法

步时序差分方法（-step TD）是 MC 和 TD(0) 更一般的推广。MC 方法根据从某一状态开始到终止状态的收益序列，对这个状态的价值进行更新；而 TD(0) 则只根据后面的单个即使收益，在下一个后继状态价值估计值的基础上进行自举更新。步自举法为一种介于 MC 和 TC(0)之间的方法，根据多个中间时刻的收益进行更新（多于一个时刻的收益，但又不是终止状态的所有收益）。步更新的方法依然属于 TD 方法，其前面状态的估计值会根据它与后继状态的估计值的差异进行更新。时序差分量被扩展成步的方法称为步时序差分（-step TD）方法。

考虑根据“状态-收益”序列（省略了动作）进行状态的更新。在基于 MC 的更新中，$ v\_(S\_t)G\_t R\_{t+1}+R\_{t+2}+^2 R\_{t+3}++^{T-t-1} R\_TTG\_tG\_{t: t+1} R\_{t+1}+V\_t(S\_{t+1})V\_t: tv\_G\_{t: t+1}tt+1V\_t(S\_{t+1})V\_t(S\_{t+1})R\_{t+2}+^2 R\_{t+3}++^{T-t-1} R\_TG\_{t: t+2} R\_{t+1}+R\_{t+2}+^2 V\_{t+1}(S\_{t+2})^2 V\_{t+1}(S\_{t+2})R\_{t+3}+^2 R\_{t+4}++^{T-t-1} R\_TnnG\_{t: t+n} R\_{t+1}+R\_{t+2}++^{n-1} R\_{t+n}+^n V\_{t+n-1}(S\_{t+n})n ,0 t<T-nnnnV\_{t+n-1}(S\_{t+n})t+n Tnnt+n T.G\_{t: t+n}=G\_t)）。

计算步回报（）涉及若干将来时刻的收益和状态，在时刻时获取到和 后才计算，一个比较自然的基于步回报的状态价值函数更新算法是，而对于任意其它状态的价值估计保持不变：。这个算法被称为步时序差分（步 TD）算法。因为在最开始的个时刻，价值函数不会被更新，因此在终止时刻后还将执行对应次数的更新，其伪代码如下，

下，

**Algorithm 8***n*-step TD for estimating *V*≈*vπ*​

**Input:** a policy *π*

**Algorithm parameter:** step size *α*∈(0,1], a positive integer *n*

**Initialize** *V*(*s*) arbitrarily, for all *s*∈S

All store and access operations (for *St*​ and *Rt*​ ) can take their index  mod+1mod*n*+1

**for**(Loop for each episode)**do**

**Initialize and store**0≠*S*0​= terminal

*T*←∞

**for**(Loop for 0,1,2,…*t*=0,1,2,… )**do**

**if***t*<*T***then**

Take an action according to*π*(⋅∣*St*​)

Observe and store the next reward as 1*Rt*+1​ and the next state as 1*St*+1​

**if***St*+1​ is terminal**then**

*T*←*t*+1

**end if**

**end if**

*τ*←*t*−*n*+1 ( *τ* is the time whose state's estimate is being updated)

**if***τ*≥0**then**

*G*←∑*i*=*τ*+1min(*τ*+*n*,*T*)​*γi*−*τ*−1*Ri*​

**if***τ*+*n*<*T***then**

*G*←*G*+*γnV*(*Sτ*+*n*​)

**end if**

*V*(*Sτ*​)←*V*(*Sτ*​)+*α*[*G*−*V*(*Sτ*​)]

**end if**

**end for**

Until *τ*=*T*−1

**end for**

#### 2）步时序差分方法在随机游走上的应用

temporal\_difference()函数实现了步时序差分方法，如下代码。

# n-steps TD method  
# @value: values for each state, will be updated  
# @n: # of steps  
# @alpha: # step size  
def temporal\_difference(value, n, alpha):  
 # initial starting state  
 state = START\_STATE  
  
 # arrays to store states and rewards for an episode  
 # space isn't a major consideration, so I didn't use the mod trick  
 states = [state]  
 rewards = [0]  
  
 # track the time  
 time = 0  
  
 # the length of this episode  
 T = float('inf')  
 while True:  
 # go to next time step  
 time += 1  
  
 if time < T:  
 # choose an action randomly  
 if np.random.binomial(1, 0.5) == 1:  
 next\_state = state + 1  
 else:  
 next\_state = state - 1  
  
 if next\_state == 0:  
 reward = -1  
 elif next\_state == 20:  
 reward = 1  
 else:  
 reward = 0  
  
 # store new state and new reward  
 states.append(next\_state)  
 rewards.append(reward)  
  
 if next\_state in END\_STATES:  
 T = time  
  
 # get the time of the state to update  
 update\_time = time - n  
 if update\_time >= 0:  
 returns = 0.0  
 # calculate corresponding rewards  
 for t in range(update\_time + 1, min(T, update\_time + n) + 1):  
 returns += pow(GAMMA, t - update\_time - 1) \* rewards[t]  
 # add state value to the return  
 if update\_time + n <= T:  
 returns += pow(GAMMA, n) \* value[states[(update\_time + n)]]  
 state\_to\_update = states[update\_time]  
 # update the state value  
 if not state\_to\_update in END\_STATES:  
 value[state\_to\_update] += alpha \* (returns - value[state\_to\_update])  
 if update\_time == T - 1:  
 break  
 state = next\_state

下图展示了一个有19个状态（N\_STATES = 19）随机游走过程的试验结果，初始状态为START\_STATE = 10，其中在左边的收益为-1，所有初始状态价值为0。真实价值为TRUE\_VALUE。

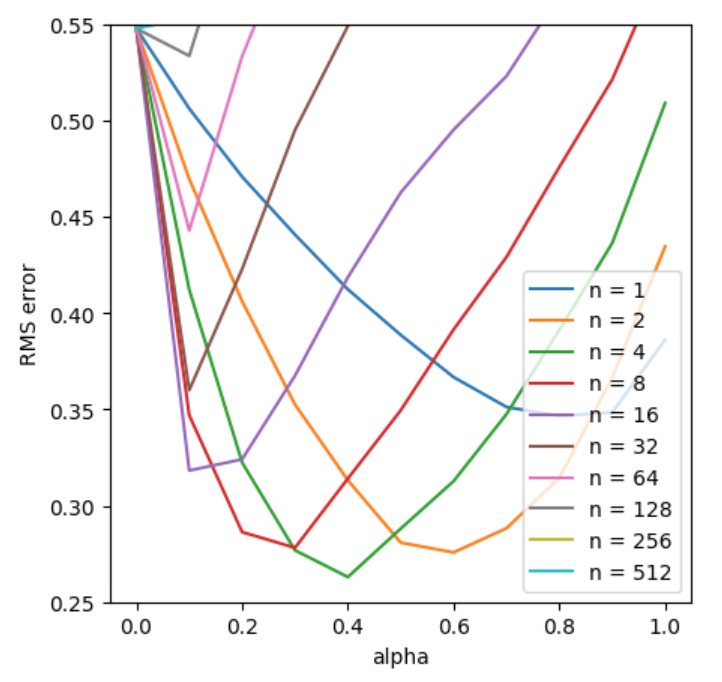
# true state value from bellman equation  
TRUE\_VALUE = np.arange(-20, 22, 2) / 20.0  
TRUE\_VALUE[0] = TRUE\_VALUE[-1] = 0  
print(TRUE\_VALUE)

[ 0. -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.1 0. 0.1 0.2 0.3  
 0.4 0.5 0.6 0.7 0.8 0.9 0. ]

图中展示的为最开始的10幕，并重复100次的平均结果。从结果可以观察到，取中间大小的值时效果最好，例如当步长，时，每幕终止时的价值函数的估计值和真实值的均方根误差的均值开方最小。

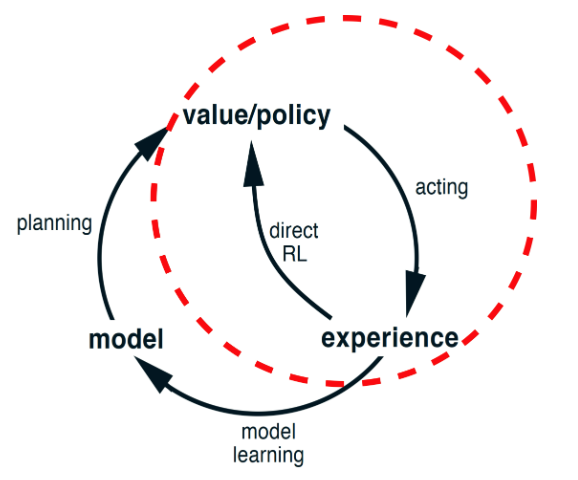
chapter07.figure7\_2()

100%|███████████████████████████████████████████████| 100/100 [04:21<00:00, 2.61s/it]



## 3.9.5 基于表格型方法的规划和学习

### 3.9.5.1 Dyna-Q 和迷宫



上图[2]162中的经验（experience）对应着学习到的（状态、动作、收益和下一个状态）的一个序列。红色虚线圈部分为前文中阐述强化学习（reinforcement learning，RL）的一个结构，使用经验直接更新（改善）价值函数和策略。因为没有使用红色虚线圈外部的模型（model），为无模型（model-free）的强化学习，称为直接强化学习（direct RL）。外部新加入（环境）的模型，指一个智能体可以用来预测环境对其动作的反应的任何事物。给定一个状态和一个动作，作为环境的反应结果，模型就会产生后继状态和下一个收益的预测。如果模型时随机的，那么后继状态和下一个收益有多种可能，每个都有一定的发生概率。一些模型生成对所有可能结果的描述及其对应的概率分布，称为分布模型；如果从所有可能性中通过概率分布采样得到一个确定的结果，则称为样本模型。模型是用来模拟环境和产生模拟经验的。使用经验来改进模型（使模型与现实环境更精确的匹配）称为模型学习（model-learning）。而将模型作为输入生成或改进与它进行交互的策略计算过程，即更新价值函数和策略的过程称为规划（planning）。用模型和规划改善价值函数和策略的方式称为间接强化学习（indirect RL）。上图包含有直接强化学习和基于模型与规划的间接强化学习的简单架构称为 Dyna-Q 算法。Dyna-Q 算法伪代码如下：

下：

**Algorithm 9**Tabular Dyna-Q

**Initialize** *Q*(*s*,*a*) and Model(*s*,*a*) for all *s*∈S and *a*∈A(*s*)

**while**(**Loop forever** )**do**

(a) ←*S*← current (nonterminal) state

(b)  *A*←*ε*-greedy  (*S*,*Q*)

(c) Take action *A*; observe resultant reward,  *R*, and state, ′*S*′

(d)  *Q*(*S*,*A*)←*Q*(*S*,*A*)+*α*[*R*+*γ*max*a*​*Q*(*S*′,*a*)−*Q*(*S*,*A*)]

(e) Model(*S*,*A*)←*R*,*S*′ (assuming deterministic environment)

**for**(Loop repeat*n* times)**do**

*S*← random previously observed state

*A*← random action previously taken in *S*

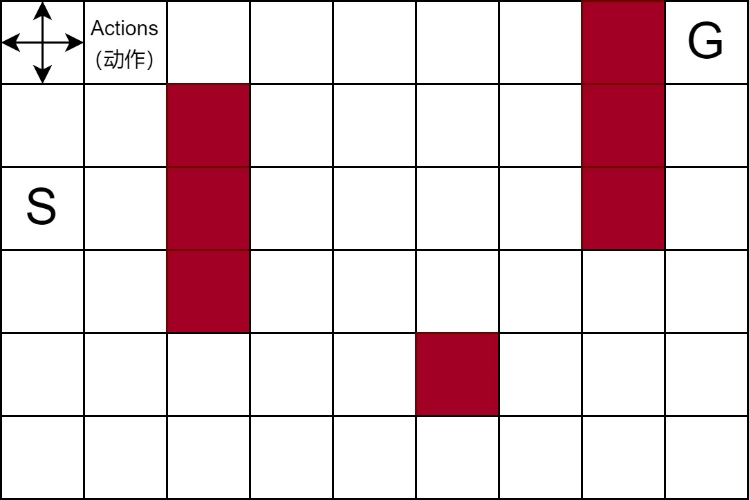
*R*,*S*′←Model(*S*,*A*)

*Q*(*S*,*A*)←*Q*(*S*,*A*)+*α*[*R*+*γ*max*a*​*Q*(*S*′,*a*)−*Q*(*S*,*A*)]

**end for**

**end while**

为了更清晰的理解模型和规划，及与直接强化学习的结合方式，以一个迷宫游戏的 Dyan-Q 实现为例阐述。



上图为一个的网格，其中红色单元为屏障（障碍物），总共占据7个单元，因此总共状态有47个。初始状态为含 S 单元，目标状态为含 G 的单元。每个状态含4个动作，上下左右，这使得智能体一定走到边邻接的单元中的某个状态，除非被屏障或迷宫的边缘阻挡。当遇到屏障时，智能体仍保持它原来的位置。对于收益的配置，除了目标状态为 +1 外，其它转移收益均为0。智能体到达目标状态后，返回到初始状态以开始新的一幕。这是一个带折扣的分幕式任务，配置。定义类Maze（查看USDA库的迁移代码或者 Shangtong Zhang 的源码），包括迷宫的所有信息。环境的模型定义为TrivialModel类，其中feed()方法不断存储和更新可获得的每一时刻的经验；sample()方法从以前的经验中随机采样。

%load\_ext autoreload   
%autoreload 2  
from usda.migrated\_project.RL\_an\_introduction import chapter08

# Trivial model for planning in Dyna-Q  
class TrivialModel:  
 # @rand: an instance of np.random.RandomState for sampling  
 def \_\_init\_\_(self, rand=np.random):  
 self.model = dict()  
 self.rand = rand  
  
 # feed the model with previous experience  
 def feed(self, state, action, next\_state, reward):  
 state = deepcopy(state)  
 next\_state = deepcopy(next\_state)  
 if tuple(state) not in self.model.keys():  
 self.model[tuple(state)] = dict()  
 self.model[tuple(state)][action] = [list(next\_state), reward]  
  
 # randomly sample from previous experience  
 def sample(self):  
 state\_index = self.rand.choice(range(len(self.model.keys())))  
 state = list(self.model)[state\_index]  
 action\_index = self.rand.choice(range(len(self.model[state].keys())))  
 action = list(self.model[state])[action\_index]  
 next\_state, reward = self.model[state][action]  
 state = deepcopy(state)  
 next\_state = deepcopy(next\_state)  
 return list(state), action, list(next\_state), reward

Dyna-Q 算法的基本架构定义为dyna\_q()函数，其中注释# Q-Learning update部分标识了直接强化学习部分；# sample experience from the model部分标识了基于模型和规划的间接强化学习部分。

# play for an episode for Dyna-Q algorithm  
# @q\_value: state action pair values, will be updated  
# @model: model instance for planning  
# @maze: a maze instance containing all information about the environment  
# @dyna\_params: several params for the algorithm  
def dyna\_q(q\_value, model, maze, dyna\_params):  
 state = maze.START\_STATE  
 steps = 0  
 while state not in maze.GOAL\_STATES:  
 # track the steps  
 steps += 1  
  
 # get action  
 action = choose\_action(state, q\_value, maze, dyna\_params)  
  
 # take action  
 next\_state, reward = maze.step(state, action)  
  
 # Q-Learning update  
 q\_value[state[0], state[1], action] += \  
 dyna\_params.alpha \* (reward + dyna\_params.gamma \* np.max(q\_value[next\_state[0], next\_state[1], :]) -  
 q\_value[state[0], state[1], action])  
  
 # feed the model with experience  
 model.feed(state, action, next\_state, reward)  
  
 # sample experience from the model  
 for t in range(0, dyna\_params.planning\_steps):  
 state\_, action\_, next\_state\_, reward\_ = model.sample()  
 q\_value[state\_[0], state\_[1], action\_] += \  
 dyna\_params.alpha \* (reward\_ + dyna\_params.gamma \* np.max(q\_value[next\_state\_[0], next\_state\_[1], :]) -  
 q\_value[state\_[0], state\_[1], action\_])  
  
 state = next\_state  
  
 # check whether it has exceeded the step limit  
 if steps > maze.max\_steps:  
 break  
  
 return steps

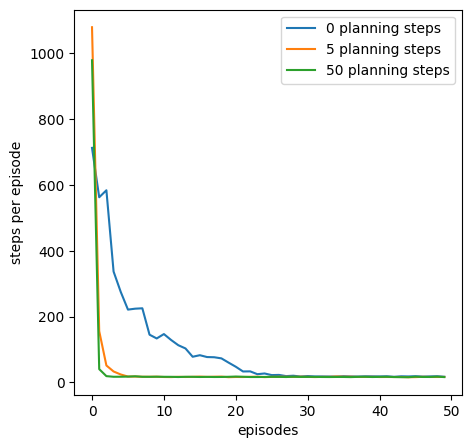
配置步长参数，和“试探”参数；比较规划幕数为0（无规划）、5和50时的结果，图中曲线表示在每幕到达目标状态时智能体所花费的步数，这个数字是 30 次（runs=30）重复试验的平均结果。从结果中可以观察到，无规划智能体用了大约25幕才取得最优性能，而的智能体只用了5幕；的智能体只用了3幕。

dynaParams=chapter08.DynaParams()  
print(f'gamma={dynaParams.gamma}\nepsilon={dynaParams.epsilon}\nalpha={dynaParams.alpha}')

gamma=0.95  
epsilon=0.1  
alpha=0.1

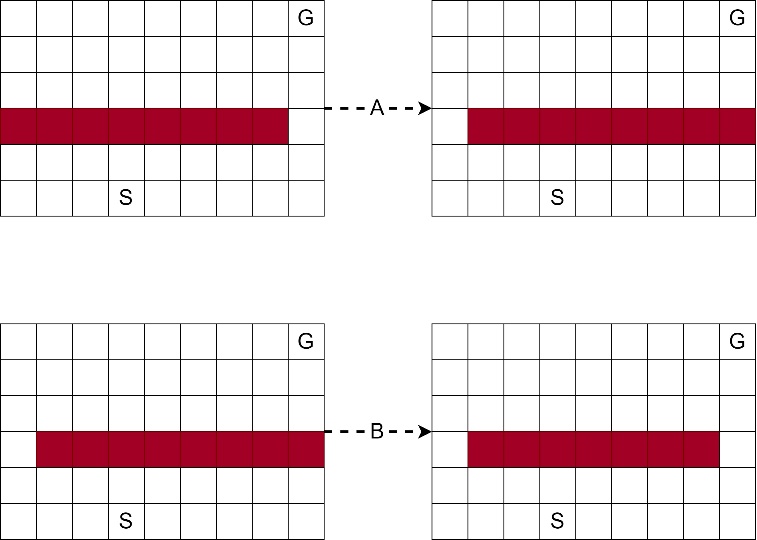
chapter08.figure\_8\_2(runs = 30,episodes = 50,planning\_steps = [0, 5, 50])

100%|█████████████████████████████████████████████████| 30/30 [03:30<00:00, 7.00s/it]



### 3.9.5.2 Dyna-Q+ 和环境的变化

当迷宫的环境发生改变，例如下图 A 中的屏障开始是右侧打开（图左），而后调整为左侧打开而右侧关闭（图右）；图 B 中，开始仅左侧打开（图左）， 而后右侧同时打开（图右），对于 Dyna-Q 算法，以前正确的策略并没有反应出这些改变时，智能体的学习则会遇到困难，这些环境的改变对于智能体而言可能在很长时间，甚至一直都不会被检测到。



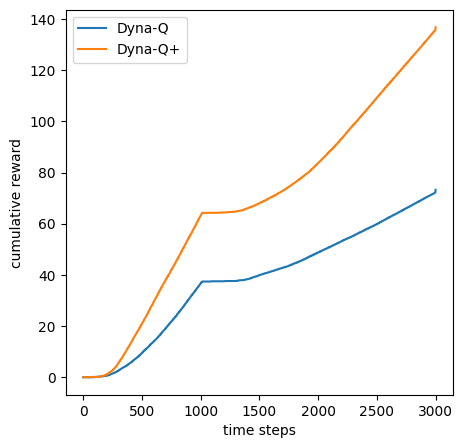
类似开发/试探的权衡，Dyna-Q+借助启发式算法，使智能体跟踪每个“状态-动作”二元组，记录它自上一次与环境进行真实交互出现以来，已经经过了多少时刻。时间越长就越有理由推测这个二元组相关的环境动态特性会产生变化，即关于它的模型不是正确的。为了鼓励测试长期未出现过的动作，在模型中为该动作相关的模拟经验设置“额外收益”提供给智能体，该“额外收益”为，式中，是模型对单步转移的收益，是距上次交互的时长（多少时刻），是一个比较小的数字，对应的代码reward += self.time\_weight \* np.sqrt(self.time - time)。定义Dyna-Q+算法实现的函数TimeModel()。

# Time-based model for planning in Dyna-Q+  
class TimeModel:  
 # @maze: the maze instance. Indeed it's not very reasonable to give access to maze to the model.  
 # @timeWeight: also called kappa, the weight for elapsed time in sampling reward, it need to be small  
 # @rand: an instance of np.random.RandomState for sampling  
 def \_\_init\_\_(self, maze, time\_weight=1e-4, rand=np.random):  
 self.rand = rand  
 self.model = dict()  
  
 # track the total time  
 self.time = 0  
  
 self.time\_weight = time\_weight  
 self.maze = maze  
  
 # feed the model with previous experience  
 def feed(self, state, action, next\_state, reward):  
 state = deepcopy(state)  
 next\_state = deepcopy(next\_state)  
 self.time += 1  
 if tuple(state) not in self.model.keys():  
 self.model[tuple(state)] = dict()  
  
 # Actions that had never been tried before from a state were allowed to be considered in the planning step  
 for action\_ in self.maze.actions:  
 if action\_ != action:  
 # Such actions would lead back to the same state with a reward of zero  
 # Notice that the minimum time stamp is 1 instead of 0  
 self.model[tuple(state)][action\_] = [list(state), 0, 1]  
  
 self.model[tuple(state)][action] = [list(next\_state), reward, self.time]  
  
 # randomly sample from previous experience  
 def sample(self):  
 state\_index = self.rand.choice(range(len(self.model.keys())))  
 state = list(self.model)[state\_index]  
 action\_index = self.rand.choice(range(len(self.model[state].keys())))  
 action = list(self.model[state])[action\_index]  
 next\_state, reward, time = self.model[state][action]  
  
 # adjust reward with elapsed time since last vist  
 reward += self.time\_weight \* np.sqrt(self.time - time)  
  
 state = deepcopy(state)  
 next\_state = deepcopy(next\_state)  
  
 return list(state), action, list(next\_state), reward

对于 A 类型迷宫环境的改变，可以从下述计算结果观察到，经过 1000 次学习后，曲线变平，表明在这一段时刻内智能体没有得到收益，它们在屏障（被堵住的开口）后面徘徊，但过了一段时间，它们依然能够学习到新的最优动作。Dyna-Q+ 要优于Dyna-Q。

chapter08.figure\_8\_4()

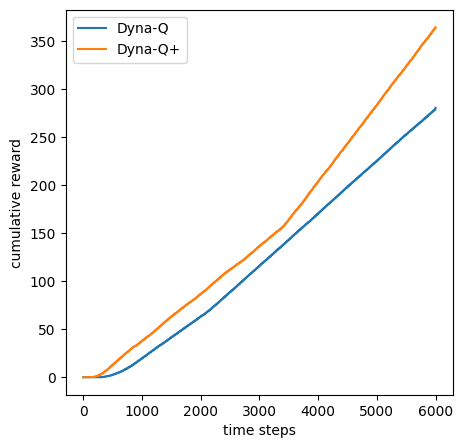
100%|█████████████████████████████████████████████████| 20/20 [01:40<00:00, 5.02s/it]



对于 B 类型迷宫环境的改变， 3000 步以后，打开了一个新开口，但 Dyna-Q 并未切换到捷径，而Dyna-Q+ 累积回报进一步增加，表明发现了这一捷径。

chapter08.figure\_8\_5()

100%|███████████████████████████████████████████████████| 5/5 [03:29<00:00, 41.96s/it]



### 3.9.5.3 优先遍历

对于迷宫的案例，如果智能体从一个零值状态转移到另一个零值状态（只有目标状态的收益为 +1，否则均为 0），价值更新没有变化。只有跳入目标状态之前的状态或者从这个状态跳出的转移过程才会导致价值估计的改变。使用均匀的模拟转移，在到达一个有用的更新之前将进行很多没用的更新，降低了有效更新的效率。因此，可以从目标状态和使得价值发生变化的任何状态进行反向计算，称为规划计算的反向聚焦（backward focusing）。当有用更新的边界不断反向推演传播时，它的范围常常迅速扩大，产生许多可以有效更新的“状态-动作”二元组，但并不是所有这些二元组同样有用，因此可以按照价值改变的大小进行优先级排序。队列头部的“状态-动作”二元组被更新时，也会计算它对其前导“状态-动作”二元组的影响。如果这些影响大于某个较小的阈值，就把相应的前导“状态-动作”二元组也插入到优先队列中（如果已经在队列，则保留优先级高的一个）。通过这种方法，价值变化的影响被有效反向传播直至影响消失。优先遍历的伪代码如下，

**Algorithm 10**Prioritized sweeping for a deterministic environment

**Initialize** *Q*(*s*,*a*) and  Model(*s*,*a*) for all *s*∈S and *a*∈A(*s*)

**while**(**Loop forever** )**do**

(a) *S*← current (nonterminal) state

(b)  policy *A*←policy(*S*,*Q*)

(c) Take action *A*; observe resultant reward,  *R*, and state, ′*S*′

(d)  Model(*S*,*A*)←*R*,*S*′

(e) *P*←∣*R*+*γ*max*a*​*Q*(*S*′,*a*)−*Q*(*S*,*A*)∣.

(f) if *P*>*θ*, then insert *S*,*A* into *PQueue* with priority *P*

**for**(Loop repeat *n* times, while *PQueue* is not empty)**do**

*A*←first(*PQueue*)

*R*,*S*′←Model(*S*,*A*)

*R*,*S*′←Model(*S*,*A*)

*Q*(*S*,*A*)←*Q*(*S*,*A*)+*α*[*R*+*γ*max*a*​*Q*(*S*′,*a*)−*Q*(*S*,*A*)]

**for**(Loop for all ,*A*ˉ predicted to lead to*S*)**do**

 predicted reward for ,*A*ˉ,*S*

*P*←​*R*ˉ+*γ*max*a*​*Q*(*S*,*a*)−*Q*(*S*ˉ,*A*ˉ)​.

if*P*>*θ* then insert ,*A*ˉ into *PQueue* with priority *P*

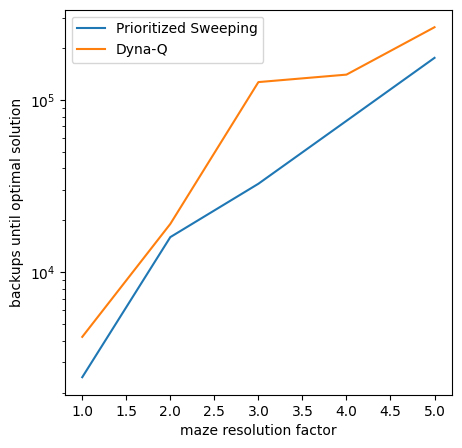
**end for**

**end for**

**end while**

从下述有无优先遍历的 Dyna-Q 对比结果中可以发现，在不同网格单元数量（状态数）不同的情况下，达到最优解所需更新数，优先遍历相对较少，具有更明显的优势。

chapter08.example\_8\_4()



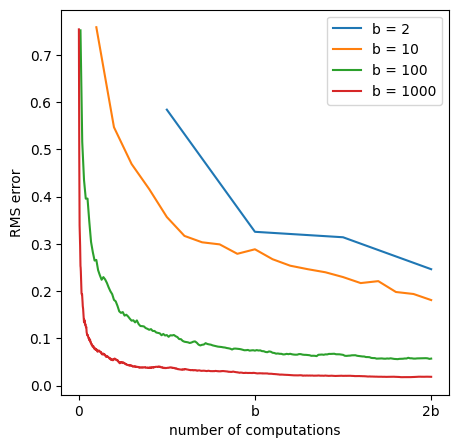
### 3.9.5.4 期望更新与采样更新的对比

讨论不同类型的价值函数更新主要沿着三个维度产生变化，一是，是更新状态价值还是动作价值；二是，估计的价值对应的是最优策略还是给定的任意策略。上述两个维度组合出四类更新，用于近似四个价值函数：；最后一个维度是采用期望更新还是采样更新。因为没有受到采样误差的影响，期望更新肯定会产生更好的估计，但是需要更多的计算。讨论一种将期望更新和采样更新用于近似的特殊例子，其中状态和动作都是离散的，近似价值函数采用表格来表示，环境模型的估计采用。则“动态-动作”二元组的期望更新是；与之对应的的采样更新则是一种类似学习的更新。给定后继状态和收益（来自模型）的采样更新是，式中，是一个通常为正数的步长参数。对于一个特定的初始二元组，设是分支因子（对于所有可能的后继状态，其中的数目），则这个二元组期望更新需要的计算量大约是采样更新的倍。

下述试验显示了在不同分支因子下，期望更新和采样更新的估计误差对计算次数的函数变化。其中所有个后继状态都是等概率发生的，且初始估计的误差是1。后继状态的价值假定是正确的，所以期望更新在完成时会将误差缩小到0。在这种情况下，采样更新会使得误差以比例系数减小，式中，是已经进行过的采样更新的次数（假设采样平均值，）。从计算结果来看，对于中等大小的来说，估计误差的大幅度下降只需要消耗很少一部分计算次数。

chapter08.figure\_8\_7(runs = 100,branch = [2, 10, 100, 1000])

100%|█████████████████████████████████████████████| 100/100 [00:00<00:00, 2927.59it/s]  
100%|██████████████████████████████████████████████| 100/100 [00:00<00:00, 859.20it/s]  
100%|██████████████████████████████████████████████| 100/100 [00:00<00:00, 130.03it/s]  
100%|███████████████████████████████████████████████| 100/100 [00:17<00:00, 5.79it/s]



## 附：符号列表

随机变量用大写字母表示，随机变量的值和标量函数用小写字母表示。实向量用粗体小写字母表示（含随机变量）。矩阵用粗体大写字母表示。

含 *Reinforcement Learning - An Introduction（Sutton, R. S., Barto, A. G. ）*第2部分符号，本章未涉及

注释（Notes）：

① Shangtong Zhang 代码实现仓库，（<https://github.com/ShangtongZhang/reinforcement-learning-an-introduction>）。

参考文献（References）:

[1] Reinforcement learning (Wikipedia), <https://en.wikipedia.org/wiki/Reinforcement_learning>

[2] Sutton, R. S., Barto, A. G. (2018 ). Reinforcement Learning: An Introduction (2nd edition). The MIT Press.（[加]Richard S. Sutton（理查德·桑顿）, [美]Andrew G. Barto（安德鲁·巴图） 著, 俞凯 等译.强化学习（第2版）.电子工业出版社, 2019.9）

[3] reinforcement-learning-an-introduction, <https://github.com/ShangtongZhang/reinforcement-learning-an-introduction>

[4] Multi-armed bandit, <https://en.wikipedia.org/wiki/Multi-armed_bandit#:~:text=In%20probability%20theory%20and%20machine,when%20each%20choice%27s%20properties%20are>