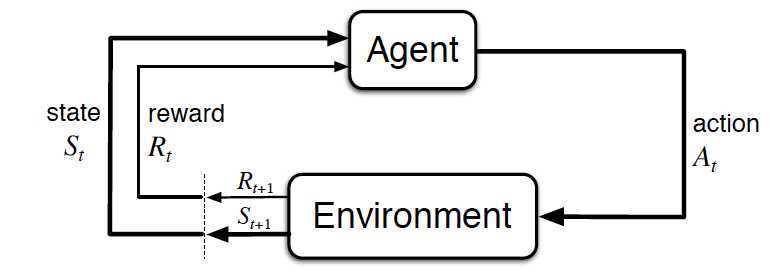
强化学习(Reinforcement Learning, RL)

1. **基本概念**



强化学习中，假如时间是离散的，称为时间步（time step）。

智能体（Agent）：执行强化学习的个体，游戏中为玩家，无人驾驶中为汽车，机器人捡垃圾中为机器人。

状态（State）：（第t个time step环境提供给智能体的状态），游戏中玩家的得分，无人驾驶中汽车是否出车祸了等。State会受到Action的影响，会有一定的概率（转移概率）转换成下一个State。

动作（Action）：（第t个time step中Agent的动作）。Agent根据环境的State和Reward执行的操作。

奖励（Reward）：（第t个time step环境给Agent的奖励）。由Agent获得，可以评价Agent的Action好坏，是否正确。Reward类似一种反馈机制，可以告诉Agent的Action是否正确。Reward有可能会延迟，即第t个time step没有奖励，则为0.Agent的目标就是最大化奖励。

一个典型的强化学习过程：

：

：

：

以此类推

强化学习与监督学习和非监督学习的一个重要区别就是：监督学习和非监督学习不需要与环境互动，只要静态数据就可以。

注意：如果希望通过强化学习来解决现实中的问题，则需要制定环境规则，并且指定状态State，动作Action和奖励Reward。

阶段性任务（Episodic Task）：具有清晰结束点的任务，例如游戏胜利或失败等。

连续性任务（Continous Task）：没有结束时间点，一直持续的任务。例如金融市场上买卖股票的算法。

强化学习的一个重要假设：Agent的目标始终可以描述为最大化期望累积奖励，称之为奖励假设。

回报（Return）：第t个time step中，过去的奖励，不再受Agent控制，只有未来的奖励才受Agent控制。令，称为回报。

通常，Agent无法肯定未来的奖励，只能依赖预测或估计。

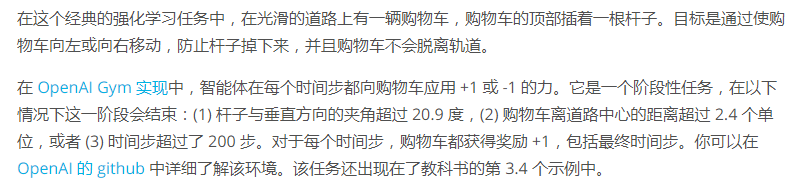
折扣回报（Discount Return）：第t个time step的回报，

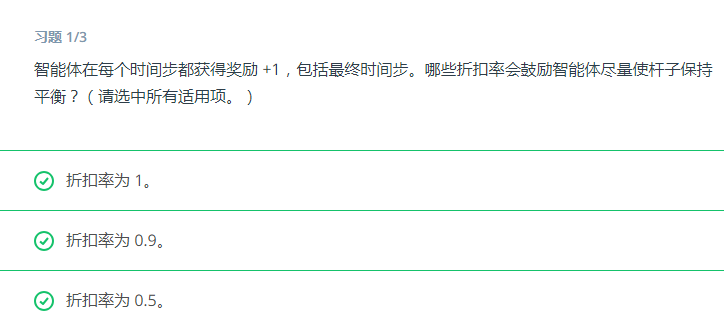


其中，称为折扣率(discount rate)

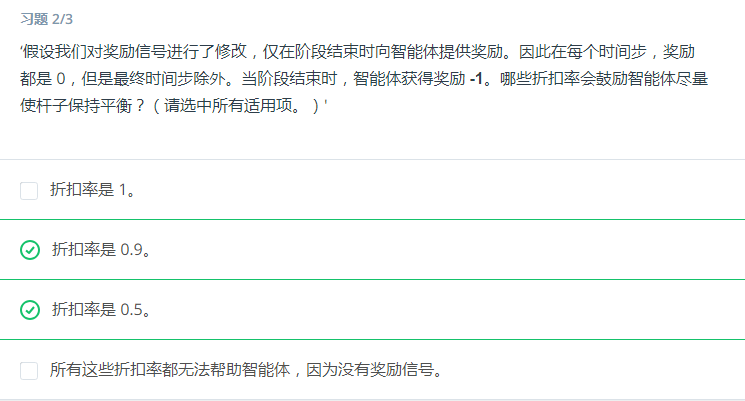
折扣回报让Agent更关心近期奖励，而不是遥远未来的奖励。

Udacity中折扣率与回报的例子：





原因：每个杆子保持平衡的时间步的奖励都是正面的，不管折扣率为多少，Agent都会尽量是杆子保持平衡。



原因：如果折扣率为1，则智能体始终获得负面奖励-1，有了折扣率后，智能体将尽量使杆子保持平衡，这样会有相对小的负面回报。



原因：如果折扣率为1，则智能体始终获得奖励+1，奖励信号将不会像智能体提供任何实用反馈，杆子是否平衡智能体不关心。

如果折扣率为0.5或0.9，最后1步的奖励为，k为time step个数，则智能体为了最大化奖励，就会尽快结束这一阶段（扔杆子等操作）。

所以需要重新设计奖励信号。

* 1. **马尔科夫决策过程(Markov Decision Process, MDP)**

定义：Agent处于环境E中，状态空间为S，其中每个状态是环境展示给Agent的环境描述。Agent能够采取的动作构造动作空间A，若某个动作作用在当前状态s上，则潜在的转移函数P将使得环境从当前状态按某种概率转移到另一个状态，在转移到另一个状态的同时，环境会根据潜在的奖励函数R反馈给Agent。

Udacity中的例子：机器人捡易拉罐

动作空间：机器人有3个动作，search(搜索),wait(等待)和recharge(充电)。

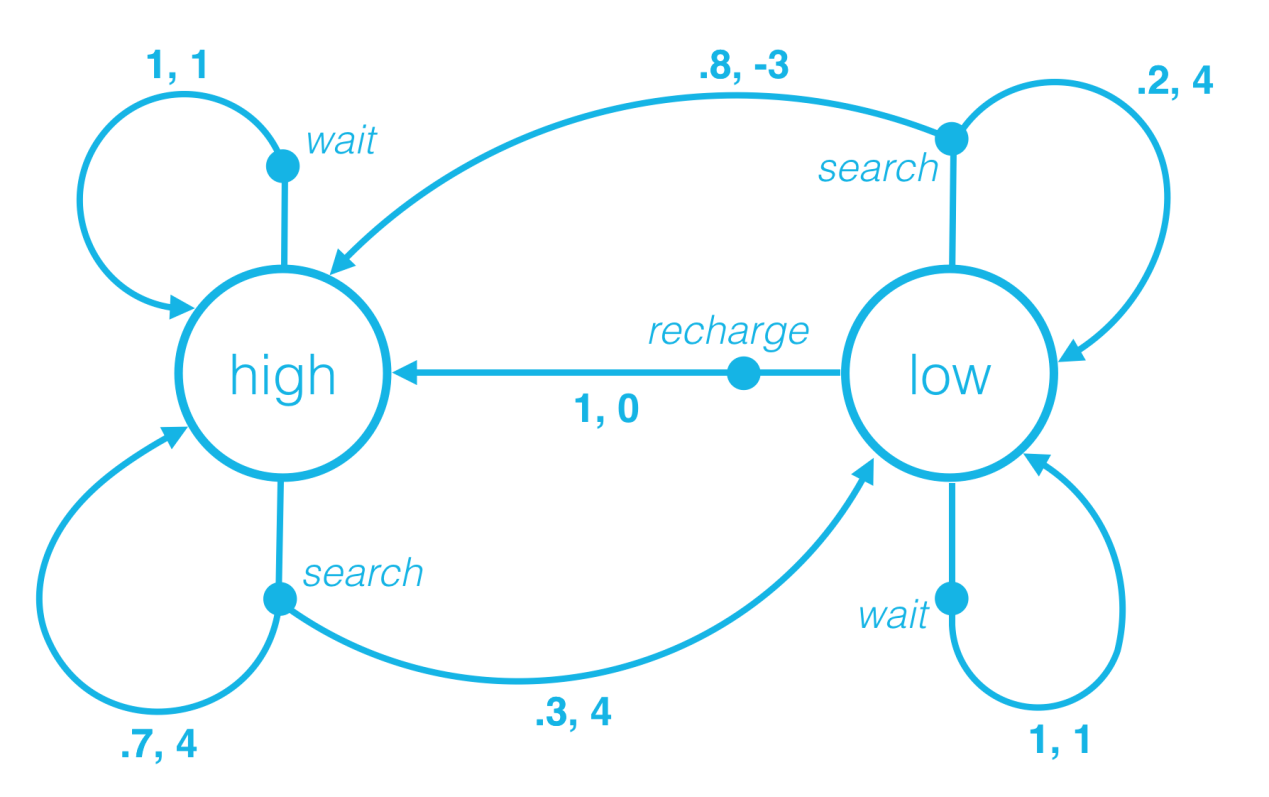


状态空间：机器人电量高high，电量低low



high状态：动作空间中，在high状态下，机器人一般只有两个动作，search和wait，不需要recharge。当机器人执行search动作时：假设其能获取4个Reward，状态有70%的概率保持high，还有30%的概率转变为low。当机器人执行wait动作时：假设其只能获取1个Reward，wait时电量不变，所以下一个状态肯定是high。

low状态：机器人3个动作都有可能执行。当机器人执行search动作时：其有80%的概率停机，这样就需要人工干预对其充电，状态就会变成high，由于人工干预，无论其捡到多少易拉罐，环境对其Reward都是-3，即要惩罚这种情况，尽量减少人工干预，还有20%的概率继续保持low状态，其能获得4个Reward。当机器人执行wait动作时：其能获得环境1个Reward，状态转变肯定还是low。当机器人执行recharge动作时：状态肯定转变为high，环境对其没有Reward。



有限MDP的流程：

1. 有限状态集S，，表示第i个状态
2. 有限动作集A，，表示第i个状态
3. 有限奖励集R
4. 转移概率，表示在当前状态，经过动作，转移到其它状态的概率分布。在一般情况下，环境的状态由之前所有状态决定，即：



但马尔科夫决策中，简化这一步骤，只考虑上一个状态：称为一步动态



策略（policy）：用表示，State到Action的映射。分为确定策略和随机策略。

确定策略：



随机策略：





表示状态s下，Agent选择动作a的概率。这里

一般而言，确定策略可以用随机策略来表示，概率为1，就代表确定策略。

udacity中的机器人捡易拉罐例子：

对于确定策略：



对于随机策略：

状态为low:



状态为high:



**1.2 寻找最佳策略**

价值函数(value function)：从状态s开始，Agent选择策略，然后在所有time step（直到认为结束）都采用此策略选择动作，预期的折扣回报。



价值函数对应特定的策略，策略发生改变，价值函数也随之改变。

贝尔曼方程（Bellman Equation）：



从上述方程可以看出，一个状态的价值和下一个状态满足递推关系，其价值由即时奖励和后续状态的价值按一定衰减组成。

策略的比较：对于策略和，当且仅当任何状态s下，都有：



则称，即优于。

从上述定义可以看出，任何两个策略其实并不一定能过判断哪个要好。但对于特定的环境，肯定至少有一个策略比其它策略更好或效果一样，称为最优策略。MDP的目标就是求出最优策略。



从上述定义可以看出，所有的最优策略，肯定有相同的价值函数，记为。



动作价值函数：价值函数没有考虑动作带来的价值影响，所以定义动作价值函数。从状态s开始，Agent采取动作a，再使用策略选择未来time step的动作，预期的折扣回报。这里的策略不只是针对当前状态s的，每个状态都使用这样的策略。



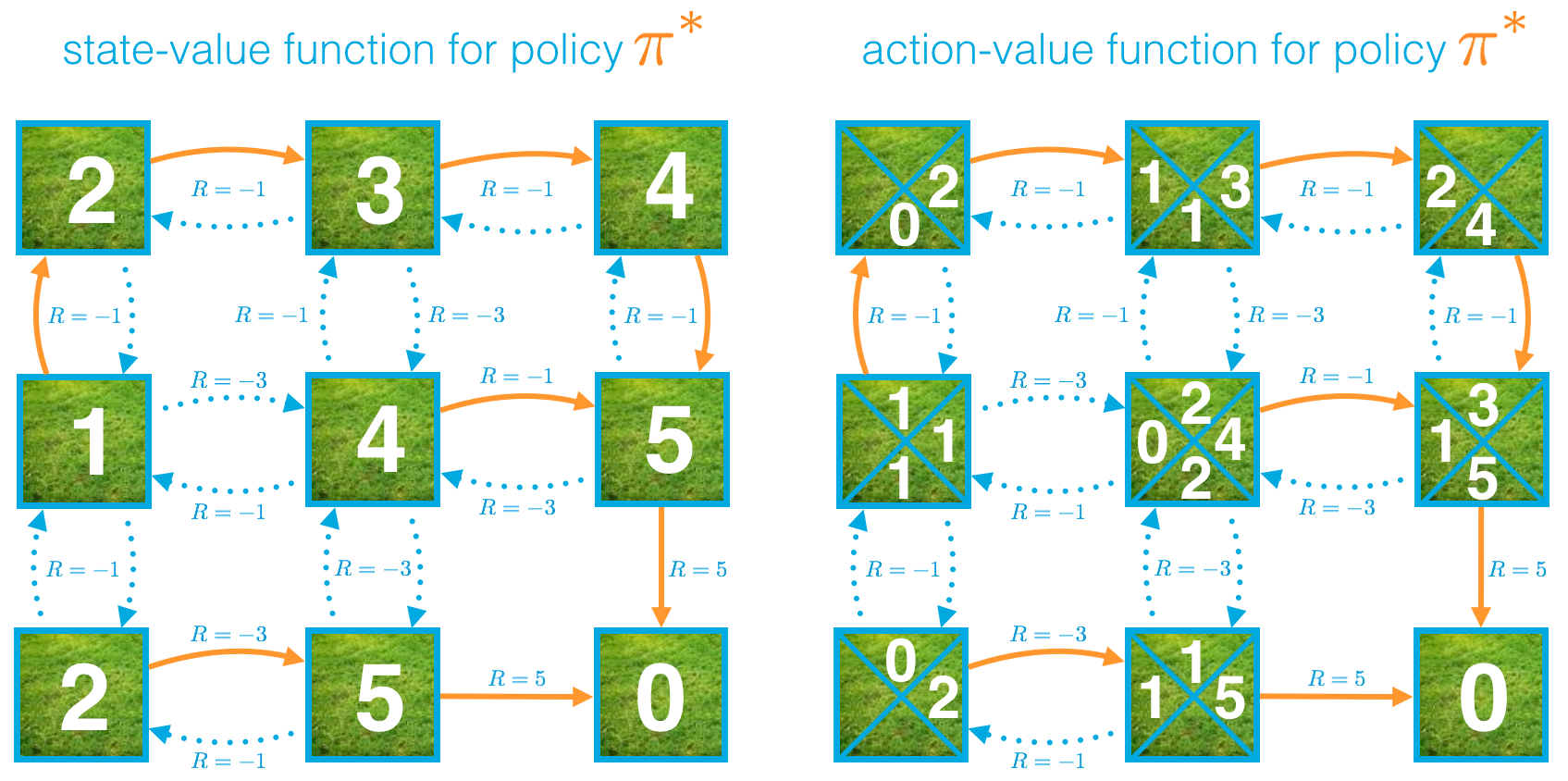
动作价值函数的贝尔曼方程：



最优动作简直函数记为。



udacity动作价值的示例计算：



标黄的箭头表示最佳策略，最后一个草地为目的地，其它草地分四格，格子中的数字代表动作价值函数值。

例1，从第8个草地，如果向右直接到达目的地，则获得奖励5，所以其动作价值函数值为5.

例2，从第4个草地，按照最佳策略，向上开始，动作价值函数值为：

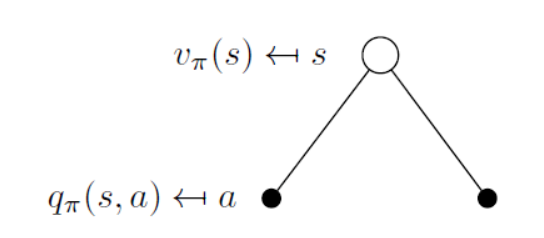
(-1)+(-1)+(-1)+(-1)+5=1，第1个草地中对应的动作向下格子为0，是因为其要先向下到第4个草地，然后按照最佳策略，计算出来的为0.

* 1. **策略、价值函数和动作价值函数**

首先要了解，在MDP中，有两个概率。其中一个是状态转移概率，表示在状态s下执行动作a，下一个状态为的概率；另一个是策略，表示在状态s下选择动作a的概率（如果为确定策略，）。

在MDP中，一般使用期望值，原因是奖励和下一个状态是根据MDP的一步动态特征选择的，奖励和下一个状态是从概率分布抽取的。

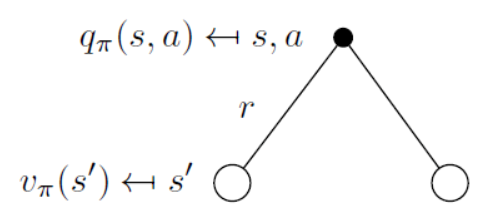
和之间的关系：



上图中，白色点表示状态，黑色点表示动作，在状态s下，Agent可能执行左边的动作a，也可能执行右边的动作，所以公式(1)：



和之间的关系：



上图中，状态s下执行动作a，获得奖励r，状态可能转变为左边的状态，也可能转移为右边的状态，所以公式(2)：



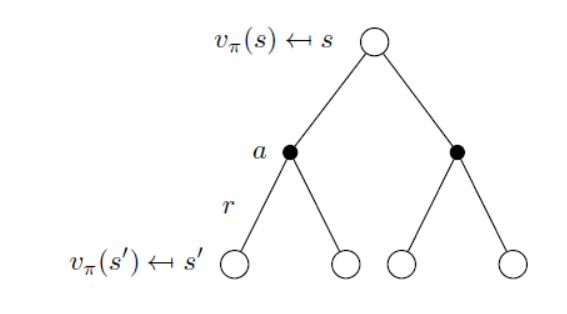
对于确定策略，价值函数为：



其中表示剔除终止状态的状态集。

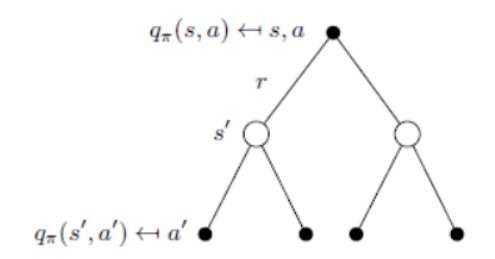
对于随机策略，价值函数为：通过将公式(2)代入公式(1)





同理，对于动作价值函数：动作价值函数，动作a已确定，所以没有。将公式(1)代入公式(2)





上述的方程表明了任何状态动作对（根据任意策略）相对于后序状态（根据同一策略）的预期值。

和的关系：根据公式(1)，并根据：



得到公式(3)



和的关系：根据公式(1)，得到公式(4)：



的贝尔曼最优方程：将公式(4)代入公式(3)



上述的方程表示任何状态根据最优策略相对于后续状态（根据最优策略）的值。

的贝尔曼最优方程：将公式(3)代入公式(4)



上述方程表示任何状态动作对根据最优策略相对于后续状态动作对（根据最优策略）的值。

1. **动态规划**

假设Agent已经了解了环境的所有信息，知道环境如何决定下个状态，知道环境如何奖励。

* 1. **策略评估**

价值函数的贝尔曼方程为：



如果策略确定了，求解每个状态的价值时，可以通过解方程组的方式，但一般通过迭代方式来解决。

伪代码如下：

输入：MDP(包括状态集，动作集，奖励集)，策略，一个极小的正数。

输出：价值函数

初始化：，先猜测每个状态的价值都为0

repeat until :



for :

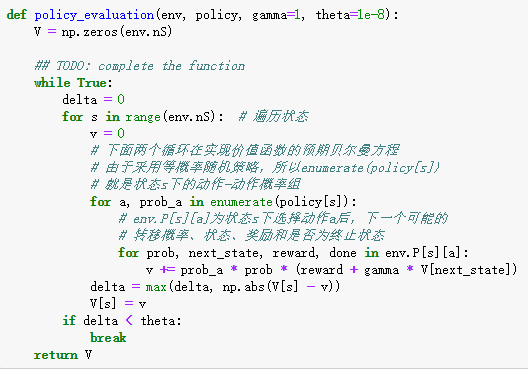






return 

真实代码：



* 1. **策略改进**

寻找更好的策略。

伪代码如下：

输入：MDP，价值函数。

输出：更好的策略

for :

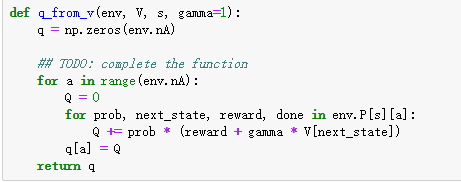
for :

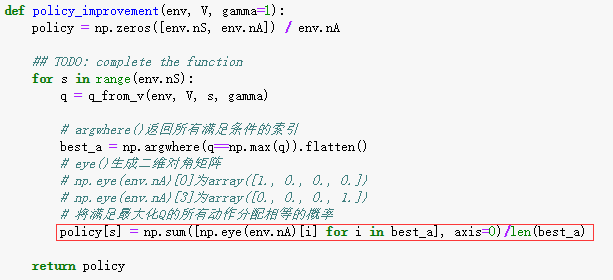




return 

在上述最大化动作价值函数时，如果有多个动作都满足，则可以考虑每个动作都分配相等的概率。例：





* 1. **策略迭代**

寻找最优策略。

伪代码如下：

输入：MDP，一个很小的正数

输出：策略。

初始化：任意随机的策略，一般可以取等概率策略，，for  and 

repeat:

策略评估：

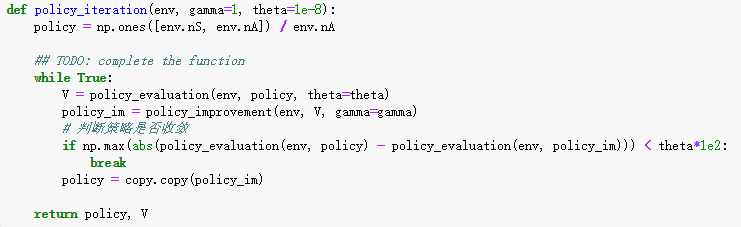
策略改进：

if  then break



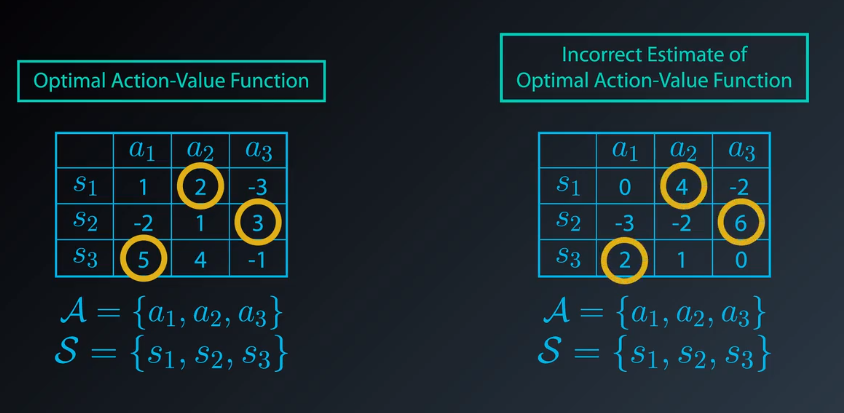
return 

真实代码：



* 1. **截断策略评估**

如果我们希望获取最佳策略，需要进行策略评估，但策略评估有可能会耗时较长，例如设置的很小。但实际上，策略评估得到的各个状态的价值不一定要十分精确，只需要相对准确即可，例如：



上图中，左边为最优价值函数，右边为不太准确的价值函数，根据它们得到的最佳策略是一样的。

截断策略评估：

输入：MDP(包括状态集，动作集，奖励集)，策略，最大迭代次数max\_iterations。

输出：价值函数

初始化：，先猜测每个状态的价值都为0



while:

for :

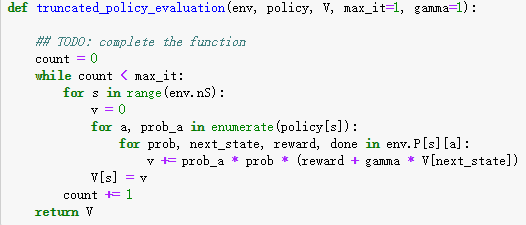




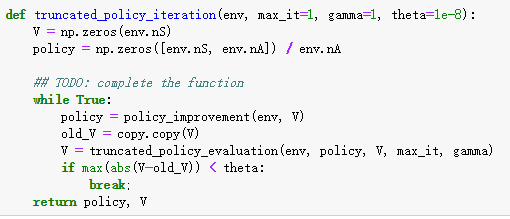


return 

真实代码：



截断策略迭代：



* 1. **值迭代**

值迭代同样是寻找最佳策略。

在策略迭代中，先假设一个策略，然后根据策略评估求值价值函数，然后策略改进，得到更好的策略，循环执行，直到策略和价值都收敛。

在值迭代中，直接求每个动作最大的价值，即最优价值函数，然后使用策略改进，得到最佳策略。

伪代码如下：

输入：MDP，一个很小的正数。

输出：最佳策略，

初始化：，先猜测每个状态的价值都为0

repeat until :



for :



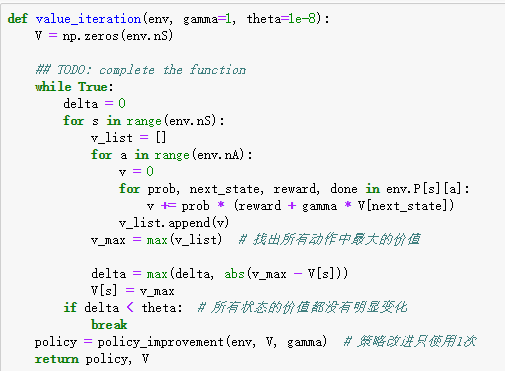




策略改进：

return 

实际代码：



1. **蒙特卡洛方法**

在动态规划中，Agent完全了解环境及其各种特性（状态集，动作集，转移概率等），但如果Agent不完全了解环境，就需要通过互动来学习环境状态。

蒙特卡洛方法一般用在episode task，即阶段性任务，在有限的时间内一定会到达终止状态。

蒙特卡洛方法不用知道环境的状态转移概率，所以它需要与环境进行多个episode，然后根据得到一系列：



根据这些经验，来进行估计。

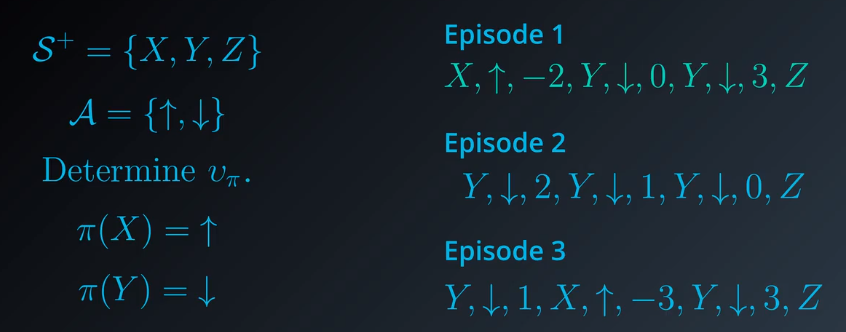
* 1. **蒙特卡洛策略估计**

对于多个episode得到的状态-动作-奖励序列，估计。蒙特卡洛根据每个episode中状态开始计算其折扣回报：



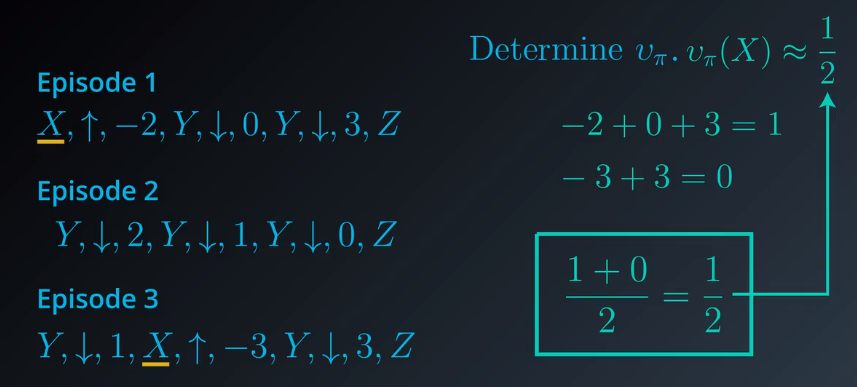
然后求所有回报的平均值来估计。

Udacity例子：



其中Z为终止状态。

估计状态X的价值：



折扣率为1

Episode1中，G(X)=-2+0+3=1

Episode2中，没有X，不考虑

Episode3中，G(X)=-3+3=0

估计状态Y的价值：

从上面的序列图中可以看出，在episode中，Y的状态出现次数不止一次，这里有两种方式来解决问题。

1. first-visit MC，根据第1个出现的Y来计算回报

Episode1中，G(Y)=0+3=3

Episode2中，G(Y)=2+1+0=3

Episode3中，G(Y)=1+(-3)+3=1

V(Y)=7/3

1. every-visit MC，

Episode1中，G(Y1)=0+3=3, G(Y2)=3

Episode2中，G(Y1)=2+1+0=3, G(Y2)=1+0=1, G(Y3)=0

Episode3中，G(Y)=1+(-3)+3=1, G(Y2)=3

V(Y)=(3+3+3+1+0+1+3) / 7=2

蒙特卡洛策略估计伪代码：

输入：策略，正整数num\_episodes。

输出：价值函数

初始化：状态s在所有episodes出现的次数，

状态s在所有episodes的回报和returns\_sum(s)=0，

for  to num\_episodes

使用策略生成一个episode 

for  to T-1

if  is a first visit(with return ) then



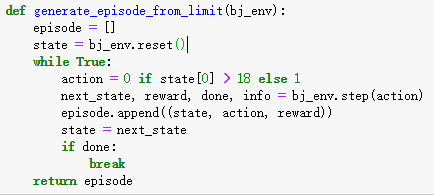
returns\_sum()  returns\_sum()



return V

真实代码：

生成episode



蒙特卡洛策略估计：

