法律声明

- 本课件包括：演示文稿，示例，代码，题库，视频和声音等，小象学院拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。

关注 小象学院

第3讲　动态规划方法

强化学习

主讲人：叶梓

上海交通大学博士

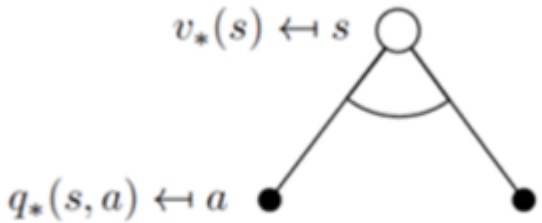
主要研究方向：机器学习、深度学习、人工智能

本章内容

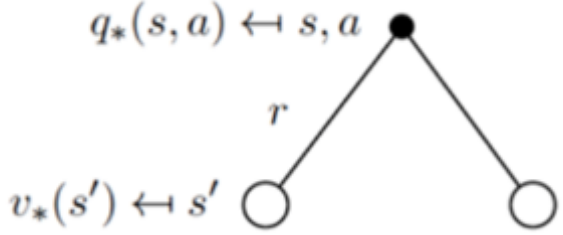
- 价值函数的计算困难
- 求解最优策略的基本思路
- 动态规划方法的原理
- 主要研究问题
- 开源库介绍
- 环境搭建

最优值函数的递归定义

- 最优状态价值函数：即考虑到这个状态下，可能发生的所有后续动作，并且都挑最好的动作来执行的情况下，这个状态的价值。
- 最优状态动作值函数：即在这个状态下执行了一个特定的动作，然后考虑到执行这个动作后所有可能处于的后续状态并且在这些状态下总是选取最好的动作来执行所得到的长期价值。



$$v^*(s) = \max_a q^*(s,a)$$



$$q^*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v^*(s')$$

价值函数的计算困难

- Bellman方程：

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \right)$$

- 忽略策略的随机性，Bellman方程可以写作

$$v_\pi = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_\pi$$

- 直接求解，则有：

$$v_\pi = (I - \gamma \mathcal{P}^\pi)^{-1} \mathcal{R}^\pi$$

- Bellman方程的复杂度为$O(n^3)$。
- 真实场景：P和R的规模都太大，很难直接求解出来。

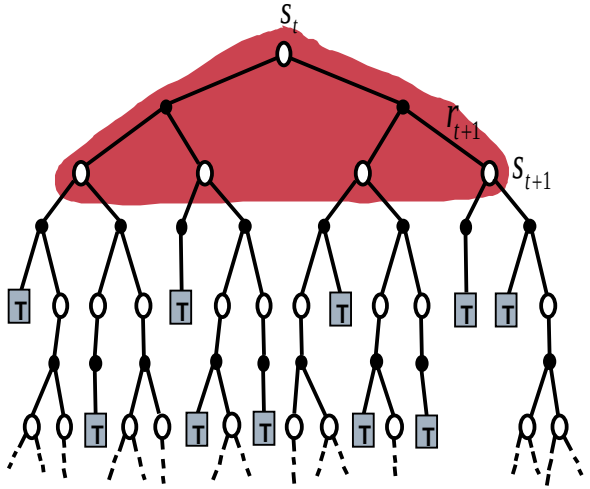
迭代计算价值函数

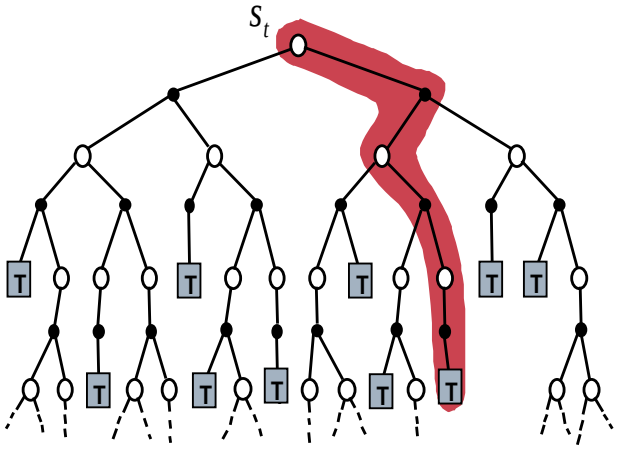
- 常用的价值函数计算方法都是迭代进行的。
- 动态规划法：已知环境模型P和R，每步进行迭代。
  - 在实际应用中，很少使用动态规划来解决大规模强化学习问题。
- Monte Carlo法：没有环境模型，根据经验学习。只考虑有终任务，任务结束后对所有的回报进行平均。
- 时序差分法：没有环境模型，根据经验学习。每步进行迭代，不需要等任务完成。

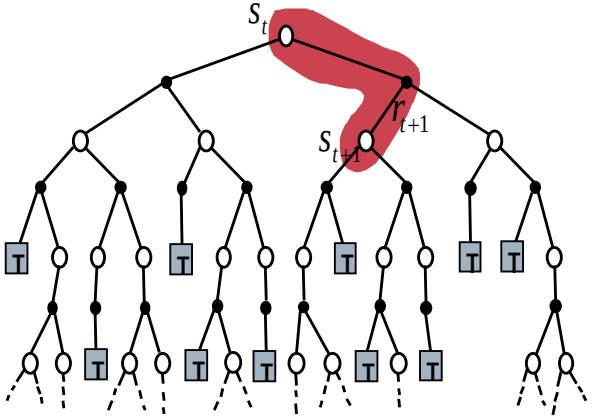
求解最优策略

- 最优策略基本的解法有三种：动态规划法、蒙特卡罗方法、时序差分法。

动态规划	简单的蒙特卡罗	简单的TD方法







动态规划方法示例

- 设有一座高度是10级台阶的楼梯，从下往上走，每一步只能向上跨1级或者跨2级台阶。
- 要求用算法来求出一共有多少种走法？

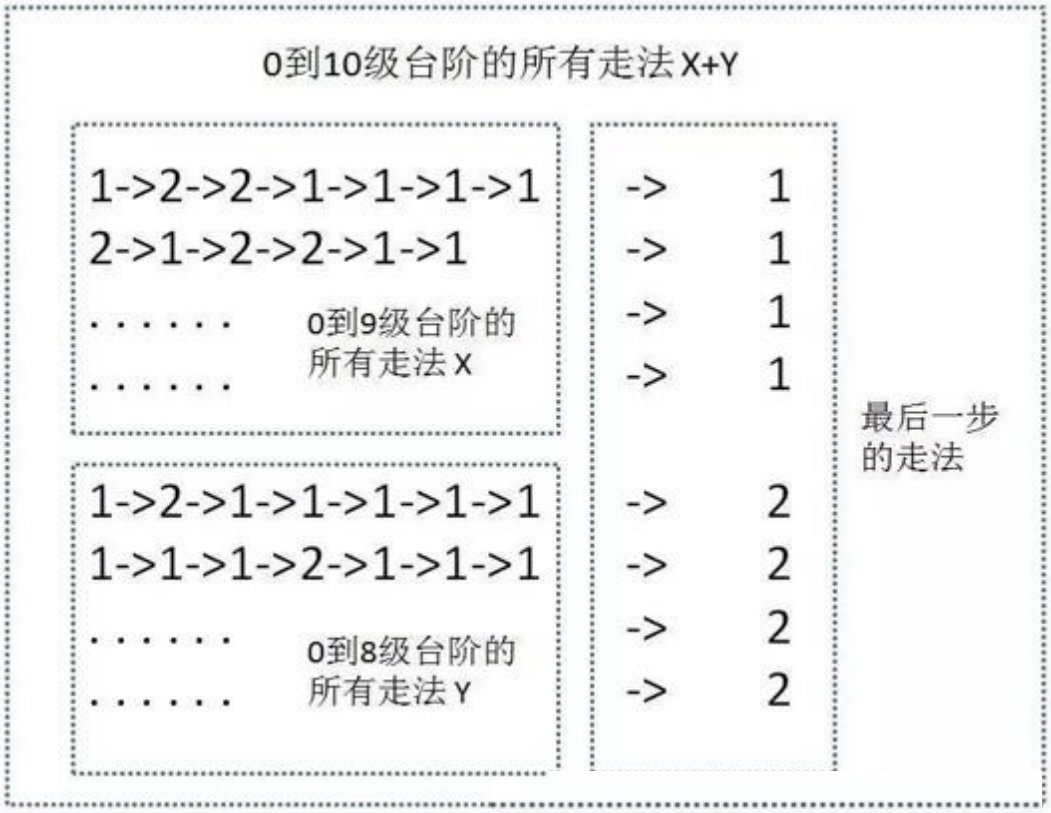
1. 每次走1级台阶，一共走10步，简写成：1,1,1,1,1,1,1,1,1,1

2. 每次走2级台阶，一共走5步，简写成 2,2,2,2,2。

N. 除此之外，还有很多很多种走法.....

动态规划方法示例

- Dynamic Programming 是一种分阶段求解决策问题的数学思想。



基于模型的动态规划方法

- DP的思想：把复杂的问题分阶段进行简化，逐步简化成简单的问题

- **边界**
- F(1)、F(2)直接有结果，无需继续简化

- **最优子结构**
- F(9)、F(8) 是 F(10)的最优子结构

- **状态转移方程**
- 核心，决定了问题一个阶段和下一阶段的关系

当只有1级台阶和两级台阶的时候，分别是1种、2种走法，我们可以归纳如下公式：

F(1) = 1 ;
F(2) = 2 ;
……
……
F(8) = F(7) + F(6) ;
F(9) = F(8) + F(7) ;
F(10) = F(9) + F(8) ;
F(n) = F(n-1) + F(n-2)

动态规划问题求解

- 迭代进行，每一次迭代过程中，只要保留之前的两个状态，就可以推导出新的状态。
- 不需要像备忘录那样保留全部的子状态。
- 实现了时间和空间上的最优化。

台阶数	走法数
1	1
2	2
3	F(1)+F(2)
4	F(2)+F(3)
5	F(3)+F(4)
6	F(4)+F(5)
7	F(5)+F(6)
8	F(6)+F(7)
9	F(7)+F(8)
10	F(8)+F(9)

- 什么是动态规划（DP）？
- “动态”指的是该问题的时间序列部分，“规划”指的是去优化一个计划，即，优化一个策略。

动态规划的条件与步骤

- 并不是所有问题都能用动态规划求解，动态规划方法需要待解决的问题包含两个性质：
  - a）最优子结构：保证问题能够使用最优性准则，从而问题的最优解可以分解为子问题最优解；
  - b）重叠子问题：子问题重复出现多次，因而我们可以缓存并重用子问题的解。
- 动态规划通常分为三步：
  - a）将问题分解为子问题；
  - b）求解子问题；
  - c）合并子问题的解。

最优子结构

- 最优子结构是依赖特定问题和子问题的分割方式而成立的条件。各子问题具有最优解，就能求出整个问题的最优解，此时条件成立。
- 反之，如果不能利用子问题的最优解获得整个问题的最优解，那么这种问题就不具有最优子结构。
- 动态规划问题都是递归问题，假设当前决策结果是f[n]，则最优子结构就是要让f[n-k]最优，最优子结构性质就是能让转移到n的状态是最优的。

MDP的求解

- MDP正好满足动态规划问题的两个性质：
  - 贝尔曼方程给出了问题的迭代分解，
  - 值函数保存和重用问题的解。
- 可以利用动态规划方法来求解MDP规划问题，
- 注意：此时假定MDP的模型是已知的。
- 动态规划方法既可用于预测问题，也可用于控制问题。

MDP的求解

- 强化学习目标是找到最优策略，即MDP的最优解。
- 求解过程一般分为两步：
  - **预测**：给定策略，评估相应的状态价值函数和状态-动作价值函数。
  - **控制**：状态值函数和状态-动作值函数求出后，就可以得到当前状态对应的最优动作。

.	状态价值	动作价值
预测	V_π	q_π
控制	V_*	q_*

评估给定的策略π

- **解决方案：迭代应用Bellman期望方程**
- 即在每次迭代过程中，对于第k+1次迭代，所有的状态s的价值都用$v_k(s')$计算，并更新该状态第k+1次迭代中所使用的价值$v_k(s)$，其中s'是s的后继状态。
- 此种方法通过反复迭代最终将收敛至V_π。

$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s'))$$

本轮

上一轮

示例——格子世界

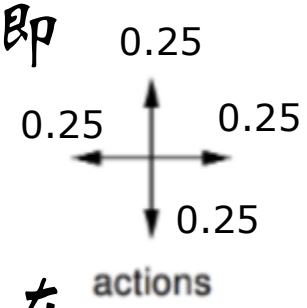
- **状态空间S**：S1 - S14非终止状态，ST终止状态，即灰色方格所示两个位置；
- **行为空间A**：{n, e, s, w} 对于任何非终止状态可以有东南西北移动四个行为；
- **转移概率P**：任何试图离开方格世界的动作其位置将不会发生改变，其余条件下将100%地转移到动作指向的状态；

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

$P^{e}_{1,2}=1$； $P^{s}_{1,5}=1$；
$P^{e}_{2,3}=1$； $P^{w}_{2,1}=1$；
$P^{w}_{3,2}=1$； $P^{s}_{3,7}=1$；
$P^{n}_{4,0}=1$； $P^{n}_{5,1}=1$；
……
$P^{n}_{1,1}=1$； $P^{w}_{4,4}=1$；
$P^{s}_{12,12}=1$； $P^{e}_{7,7}=1$；
……

示例——格子世界

- 即时奖励R：任何在非终止状态间的转移得到的即时奖励均为-1，进入终止状态即时奖励为0；
- 衰减系数γ：1；
- 当前策略π：Agent采用随机行动策略，在任何一个非终止状态下有均等的几率采取任一移动方向这个行为，
  - 即：$\pi(n|\bullet) = \pi(e|\bullet) = \pi(s|\bullet) = \pi(w|\bullet) = 1/4$。
- 问题：求该方格世界在给定策略下的（状态）价值函数，也就是求解在给定策略下，该方格世界里每一个状态的价值。



示例——格子世界

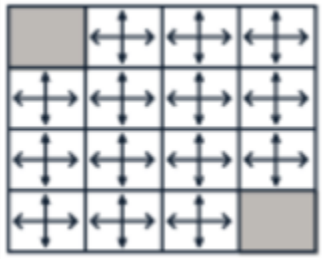
$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \right)$$

开始迭代

$k=0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

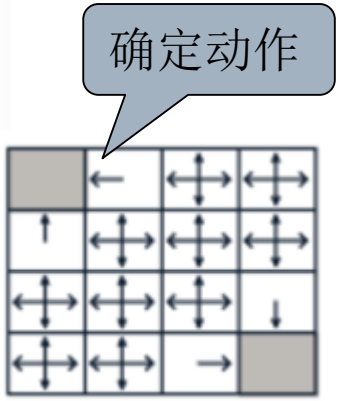
依据当前的状态价值，无法得出比随机策略更好的策略。



$k=1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0

$$-1.0 = 0.25 * (-1 + 1.0 * 0) + 0.25 * (-1 + 1 * 0) + 0.25 * (-1 + 1 * 0) + 0.25 * (-1 + 1 * 0)$$



示例——格子世界

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \right)$$

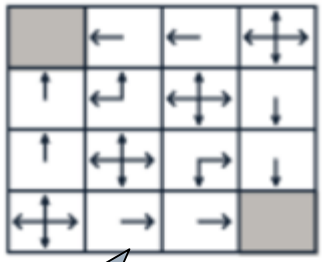
k=2

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0

按当前策略

-1.7 = 0.25 *(-1 + 1.0 * 0) + 0.25 *(-1 + 1*-1) + 0.25 *(-1 + 1*-1) + 0.25 *(-1 + 1*-1)

目前还没有得到最优策略



确定动作

k=3

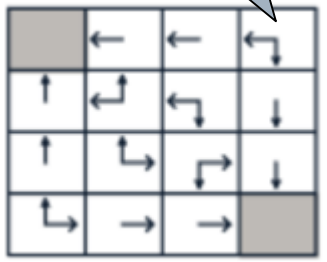
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0

按当前策略

-2.4 = 0.25 *(-1 + 1.0 * 0) + 0.25 *(-1 + 1*-2.0) + 0.25 *(-1 + 1*-2.0) + 0.25 *(-1 + 1*-1.7)

依据该状态价值函数已经可以得到最优策略

调整概率

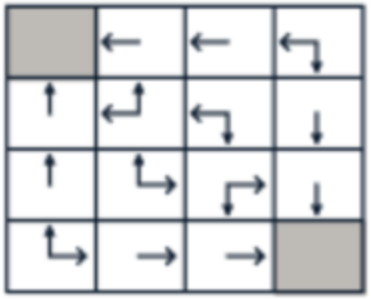


示例——格子世界

- 可以一直迭代下去，直到收敛。
- 但对于发现最优策略而言，其实是没必要的。

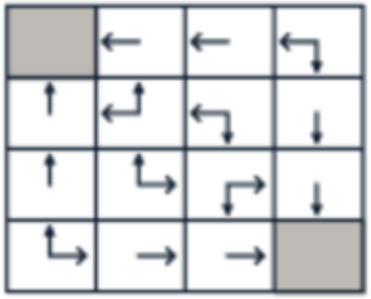
$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0



$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0



最优解的计算方法

- 策略迭代（Policy iteration）算法通过构建值函数（而不是最优值函数）来评估策略，然后利用这些值函数，寻找新的、改进的策略。
- 值迭代（Value iteration）算法寻找最优值函数，最优值函数包括每个状态或者每个状态动作对的最大回报。最优值函数用来计算最优策略。

策略迭代

- 通过方格世界的例子，我们得到了一个优化策略的办法，分为两步：
- 首先我们在一个给定的策略下迭代更新价值函数：

$$v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + ... | S_t = s\right]$$

- 随后，在当前策略基础上，贪婪地选取行为，使得后继状态价值增加最多：

$$\pi' = \text{greedy}(v_\pi)$$

策略迭代-I

- 使用策略迭代算法，每次迭代分两步走：
- 第1步：先任意假设一个策略π_k，使用这个策略迭代价值函数直到收敛，

$$V_{i+1}^{\pi_k}(s) \leftarrow \sum_{s'} P(s, \pi_k(s), s') \left[R(s, \pi_k(s), s') + \gamma V_i^{\pi_k}(s') \right]$$

- 最后得到的V(s)就是我们用策略π_k，能够取得的最好价值函数V(s)了。
- (其实是策略的一种评估)

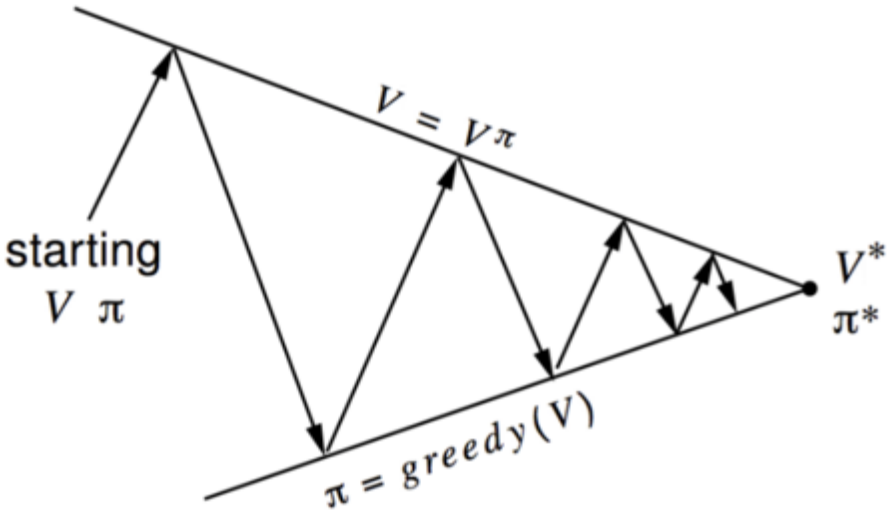
策略迭代-II

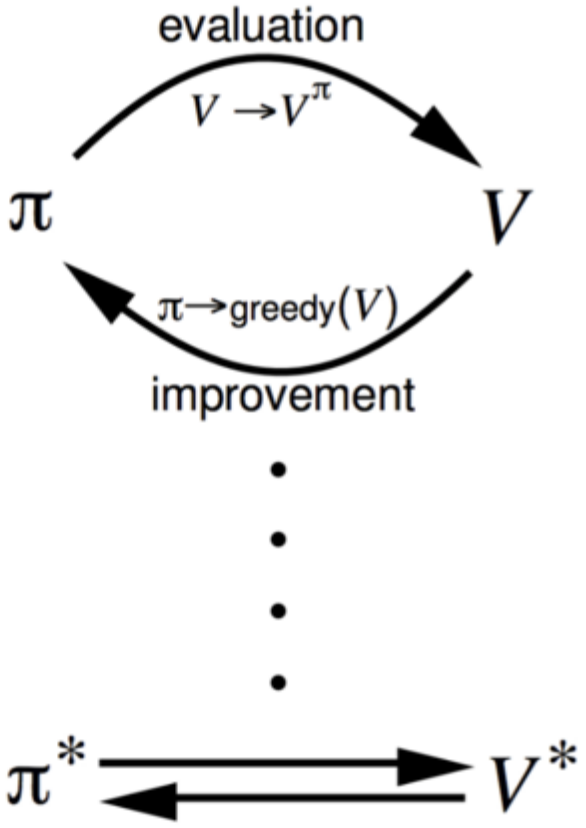
- 第2步：重新审视每个状态所有可能的行动Action，优化策略π_k，看看有没有更好的Action可以替代老的Action：

$$\pi_{k+1}(s) = \arg\max_a \sum_{s'} P(s,a,s') \left[R(s,a,s') + \gamma V^{\pi_k}(s') \right]$$

- 策略迭代最后得到了每个状态应有的最佳策略。

策略迭代





Policy evaluation Estimate v_π

Any policy evaluation algorithm

Policy improvement Generate $\pi' \geq \pi$

Any policy improvement algorithm

改进的策略迭代

- 有时候不需要持续迭代至最有价值函数，可以设置一些条件提前终止迭代，比如：
  - 设定一个ε，比较两次迭代的价值函数平方差；
  - 直接设置迭代次数；
  - 每迭代一次，就更新一次策略等。

策略迭代的伪码

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\left[r + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $a \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
   If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return V and π; else go to 2

策略的随机性，和动作后状态转移的随机性，都有体现。

基于某种策略

直到不再变化

策略迭代伪码的说明

- 1.initialization 初始化所有状态的v(s)以及π(s)
- 2.poicy evaluation 用当前的v(s)对当前策略进行评估，计算出每一个状态的v(s)，直到v(s)收敛，才算计算好了这个状态价值函数V(s)
- 3.policy improvement 既然上一步已经得到了当前策略的评估函数V(s),那么就可以利用这个评估函数进行策略改进。
- 在每个状态s时，对每个可能的动作a，都计算一下采取这个动作后到达的下一个状态的期望价值。看看哪个动作可以到达的状态的期望价值函数最大，就选取这个动作。以此更新了π(s)
- 然后再次循环上述2、3步骤，直到V(s)与π(s)都收敛。

最优价值函数

- **最优状态值函数**

  最优值函数$v_*(s)$是在从所有策略产生的状态价值函数中，选取使状态s价值最大的函数：

$$v_*(s) = \max_{\pi} v_\pi(s)$$

- **最优状态行为值函数**

  最优状态行为值函数$q_*(s,a)$是从所有策略产生的行为价值函数中，选取是状态行为对<s,a>价值最大的函数:

$$q_*(s,a) = \max_{\pi} q_\pi(s,a)$$

Bellman最优公式

最优状态值函数

$$V^*(s) = \max_{a \in A(s)} E\left\{ r_{t+1} + \gamma V^*(s_{t+1}) \middle| s_t = s, a_t = a \right\}$$

$$= \max_{a \in A(s)} \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V^*(s') \right]$$

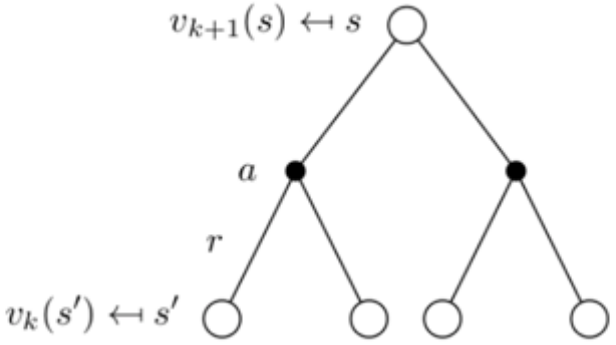
最优状态行为值函数

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

$$= R(s,a) + \gamma E_{P(s'|s,a)}[\max_{a'} Q^*(s',a')]$$

评估给定的策略π

- 通过前面的分析，我们通过迭代使用贝尔曼期望方程backup（反演/推演）来进行“预测问题”的求解。
- 同步backup：



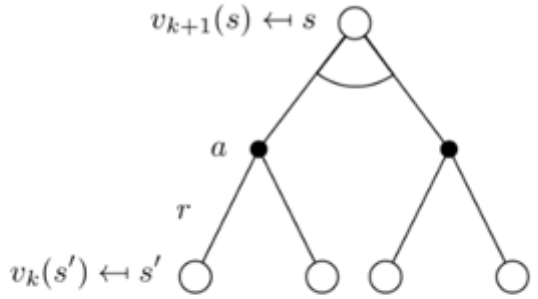
- Using *synchronous* backups,
  - At each iteration $k+1$
  - For all states $s \in \mathcal{S}$
  - Update $v_{k+1}(s)$ from $v_k(s')$
  - where s' is a successor state of s

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

$$\mathbf{v}^{k+1} = \mathcal{R}^{\pi} + \gamma \mathcal{P}^{\pi} \mathbf{v}^k$$

价值迭代

- 先更新每个状态s的长期价值V(s)，这个价值是立即回报r(s,a)与下一个状态s+1的长期价值的综合，从而获得更新。
- 每次迭代，对于每个状态s，都要更新价值函数V(s)；
- 对于每个状态s的价值更新，需要考虑所有行动Action的可能性。



$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

$$\mathbf{v}_{k+1} = \max_{a \in \mathcal{A}} \mathcal{R}^a + \gamma \mathcal{P}^a \mathbf{v}_k$$

价值迭代的伪码

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat

$\Delta \leftarrow 0$

For each $s \in \mathcal{S}$:

$v \leftarrow V(s)$

$V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that

$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

仅体现的动作后状态转移的随机性，每一种可能的a都要考虑，选哪个动作是由max()决定的，是确定的。

同上。

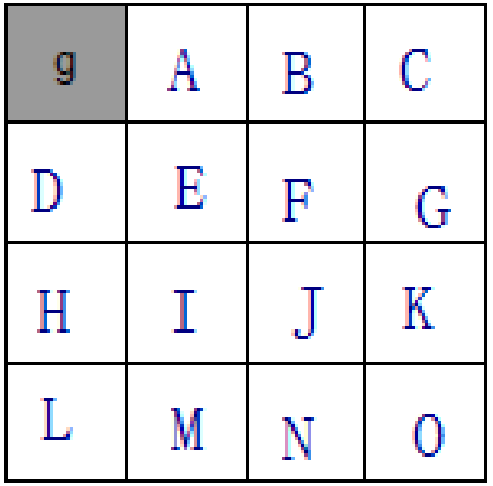
值迭代的说明

- **Initialization**：初始化所有状态的v(s)
- **finding optimal value function**：（找到最优的价值函数）注意伪代码里的max：
  - 对每一个当前状态s,对每个可能的动作a,都计算一下采取这个动作后到达的下一个状态的期望价值。
  - 看哪个动作可以到达的状态的期望价值函数最大，就将这个最大的期望价值函数作为当前状态的价值函数v(s)
  - 循环执行这个步骤，直到价值函数收敛,就可以得到最优optimal的价值函数了
- **policy extraction**：利用上面步骤得到的optimal价值函数和状态转移概率，就计算出每个状态应该采取的optimal动作，这个是deterministic。

值迭代示例

- 又是一个格子世界：除了终点（写着g的灰色格）的reward=0，其他的状态reward=-1。

- 在迭代的过程中，距离g最近的那两个格子（即A和D）的V(s)最先可以算出最大值。



Problem

值迭代示例

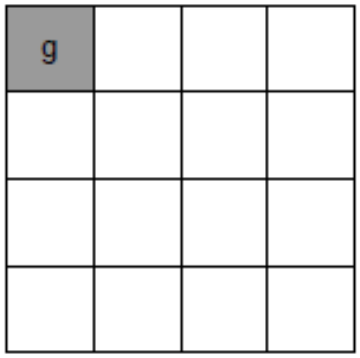
$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

- 对于A来说，只要选择向左，就可以达到终点获得reward。而向右则没有奖赏。
- 所以根据公式中的max，A就会选择左走，它的V值最大，所以第一次迭代它就已经可以获得最大值了。
- 然后接下来就是对于B来说，第一次迭代向左和向右都一样，但是在第二次迭代中就会发现两次向左就可以达到最大值。

g	A	B	C
D	E	F	G
H	I	J	K
L	M	N	O

Problem

值迭代示例



Problem

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

V_1

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

V_2

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

V_3

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

V_4

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

V_5

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

V_6

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

策略迭代与值迭代的异同

1. Initialization
   $v(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $temp \leftarrow v(s)$
         $v(s) \leftarrow \sum_{s'} p(s'|s, \pi(s)) \left[r(s, \pi(s), s') + \gamma v(s') \right]$
         $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
   until $\Delta < \theta$ (a small positive number)

对当前策略进行评估

3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
      $temp \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s'} p(s'|s, a) \left[r(s, a, s') + \gamma v(s') \right]$
      If $temp \neq \pi(s)$, then $policy\text{-}stable \leftarrow false$
   If $policy\text{-}stable$, then stop and return v and π; else go to 2

finding optimal value function

Initialize array v arbitrarily (e.g., $v(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
      $temp \leftarrow v(s)$
      $v(s) \leftarrow \max_a \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma v(s')]$
      $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
until $\Delta < \theta$ (a small positive number)

策略无关，找最优

Output a deterministic policy, π, such that
$\pi(s) = \arg\max_a \sum_{s'} p(s'|s, a) \left[r(s, a, s') + \gamma v(s') \right]$

找到能达到最优的下一状态的动作。

两者的区别

- policy iteration：使用bellman方程来更新value，最后收敛的value即v^{π}是当前policy下的value值（所以叫做对policy进行评估），目的是为了后面的policy improvement得到新的policy。

- value iteration：使用bellman 最优方程来更新value，最后收敛得到的value即v*就是当前state状态下的最优的value值。因此，只要最后收敛，那么最优的policy也就得到的。

区别及联系

- 策略迭代的第二步policy evaluation与值迭代的第二步finding optimal value function十分相似，除了后者用了max操作，前者没有max。
  - 因此后者可以得出optimal value function，而前者不能得到optimal function。
- 策略迭代的收敛速度更快一些。
  - 在状态空间较小时，最好选用策略迭代方法；当状态空间较大时，值迭代的计算量更小一些。
  - 本质上依赖于模型，而且理想条件下需要遍历所有的状态，这在稍微复杂一点的问题上就基本不可能了。

异步动态规划的常见形式

- in-place 动态规划： 不对上一周期的value进行备份，直接使用这一周期的value。
- 其实，本周期的value本来就是上一周期优化的结果，只是少了备份这一步，节省了一些内存……

for all s in $\mathcal{S}$

$$v_{new}(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_{old}(s') \right)$$

$$v_{old} \leftarrow v_{new}$$

for all s in $\mathcal{S}$

$$v(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right)$$

异步动态规划的常见形式

- Prioritised Sweeping：计算优化目标值和现实值之差（Bellman error），对多个S计算后排序，差值大的在前，依次优化对应的s的value。

$$\left| \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right) - v(s) \right|$$

- real-time 动态规划：应该可以翻译为实时动态规划，但是real-time 实际上指的是 real time step，意思是真正经历的时间步，也就是真正经历的S，只更新经历过的S。

$$v(S_t) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_{S_t}^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{S_t s'}^a v(s') \right)$$

动态规划的不足

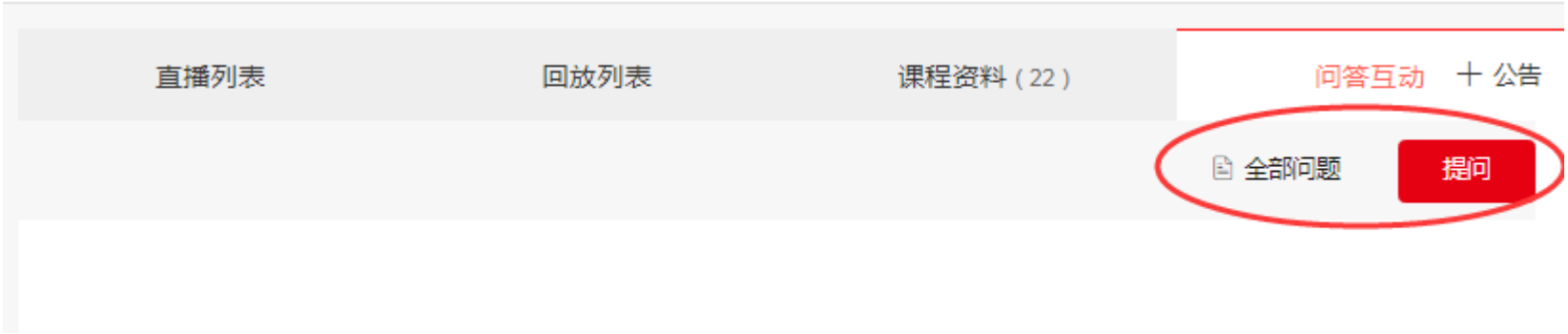
- DP方法的缺点显而易见：必须知道状态转移概率才能进行最优策略的计算。这在我们真实的使用场景中几乎不可能实现。
- 对于DP而言，它的推演是整个树状散开的，这种方法被称为Full-Width Backup方法。
- 在这种方法中，对于每一次backup来说，所有的后继状态和动作都要被考虑在内，并且需要已知MDP的转移矩阵与奖励函数，因此DP将面临维数灾难问题。
- 所以就有了Sample Backup……

参考资料

- https://blog.csdn.net/u012151283/article/details/54926888
- https://blog.csdn.net/songrotek/article/details/51378582
- https://blog.csdn.net/panglinzhuo/article/details/77752574
- https://blog.csdn.net/liweibin1994/article/details/79093453
- https://blog.csdn.net/u013745804/article/details/78218140

问答互动

在所报课的课程页面，
1、点击“全部问题”显示本课程所有学员提问的问题。
2、点击“提问”即可向该课程的老师和助教提问问题。



联系我们

小象学院：互联网新技术在线教育领航者

– 微信公众号：**小象学院**