第二章- MDP、K臂老虎机及贝尔曼方程

Markov Decision Process

Markov Property

State s_t is Markovian if and only if:

$$p\left(s_{t+1} \mid s_t\right)=p\left(s_{t+1} \mid h_t\right)$$
$$p\left(s_{t+1} \mid s_t, a_t\right)=p\left(s_{t+1} \mid h_t, a_t\right)$$

The future is independent of the past given the present. 未来的情况只和当前状态有关，和再之前的状态无关。

Markov Reward Process

Markov Reward Process is a Markov Chain + reward

Definition of Markov Reward Process (MRP)

1. S is a (finite) set of states ($s \in S$)
2. P is dynamics/transition model that specifies $P(S_{t+1}=s' | s_t=s)$
3. R is a reward function $R(s_t=s)=\mathbb{E}[r_t | s_t=s]$
4. Discount factor $\gamma \in [0,1]$

If finite number of states, R can be a vector

Return and Value function

- Horizon的定义：一个回合内的最大时间步
- Return的定义：$G_t=R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+\gamma^3 R_{t+4}+\ldots+\gamma^{T-t-1} R_T$
- Value function的定义（表示未来奖励的值）：

$$\begin{aligned} V_t(s) &= \mathbb{E}\left[G_t \mid s_t=s\right] \\ &= \mathbb{E}\left[R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+\ldots+\gamma^{T-t-1} R_T \mid s_t=s\right] \end{aligned}$$

为什么要有γ：

- 避免在循环的马尔可夫过程中有无限的return
- 关于未来的不确定性不能完全地展示，那这个γ其实就表征了一定的不确定性
- 比起未来的奖励人类更偏爱即时奖励
- 如果$\gamma=0$,表示我们只关心立即奖励而不关心未来的奖励；如果$\gamma=1$,表示我们觉得未来的奖励和现在的即时奖励一样重要，所以就可以把γ作为超参数来得到不同行为的agent

Markov Descion Process

Markov Decision Process is Markov Reward Process with decisions.

Definition of MDP

1. S is a finite set of states
2. A is a finite set of actions
3. P^a is dynamics/transition model for each action $P(s_{t+1} = s' | s_t = s, a_t = a)$
4. R is a reward function $R(s_t = s, a_t = a) = \mathbb{E}[r_t | s_t = s, a_t = a]$
5. Discount factor $\gamma \in [0, 1]$

MDP is a tuple: (S, A, P, R, γ)

MDP中的重要定义

- 在有限MDP中, 状态, 动作和奖励 $(\mathcal{S}, \mathcal{A}$ 和 $\mathcal{R})$ 的集合都具有有限数量的元素。 在这种情况 下, 随机变量 R_t 和 S_t 具有明确定义的离散概率分布, 仅取决于先前的状态和动作。 也就是说对于这些随机变量的特定值, $s' \in \mathcal{S}$ 和 $r \in \mathcal{R}$, 在给定前一状态和动作的特定值的情况下，存在这些值在时间t发生的概率:

$$p(s', r \mid s, a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \tag{1}$$

- $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ 是四个参数的普通确定性函数。 但这里只是提醒我们 p 指定 s 和 a 的每个选择的概率分布, 即对所有 $s \in \mathcal{S}, a \in \mathcal{A}(s)$ $\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1$。
- 从四参数动力学函数p中，可以计算出人们可能想知道的关于环境的任何其他信息，例如状态转移概率 (我们将其略微滥用符号表示为三参数函数 $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$),

$$p(s' \mid s, a) \doteq \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \tag{2}$$

- 我们还可以将状态 - 动作对的预期奖励计算为双参数函数 $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$

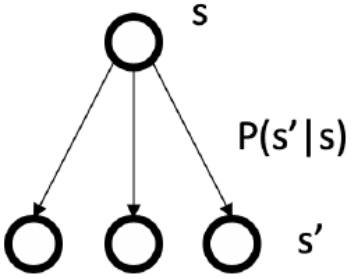
$$r(s, a) \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a) \tag{3}$$

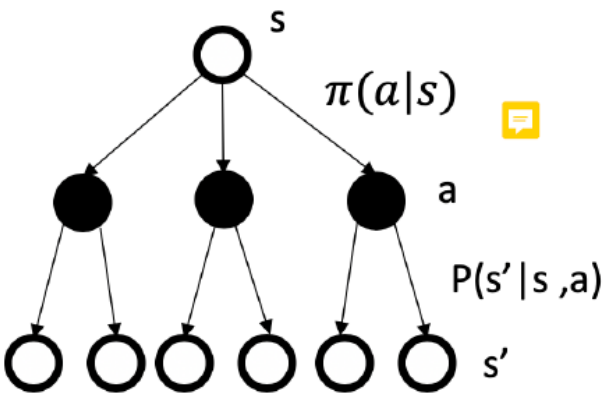
- 以及状态 - 行动 - 下一状态三元组的预期奖励作为三个参数函数 $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$

$$r(s, a, s') \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r \mid s, a)}{p(s' \mid s, a)} \tag{4}$$

MRP和MDP的对比

MDP相当于是在MRP的中间加了一层决策层，相当于MRP是一个随波逐流的过程，但是MDP是一个有agent在掌舵的过程。



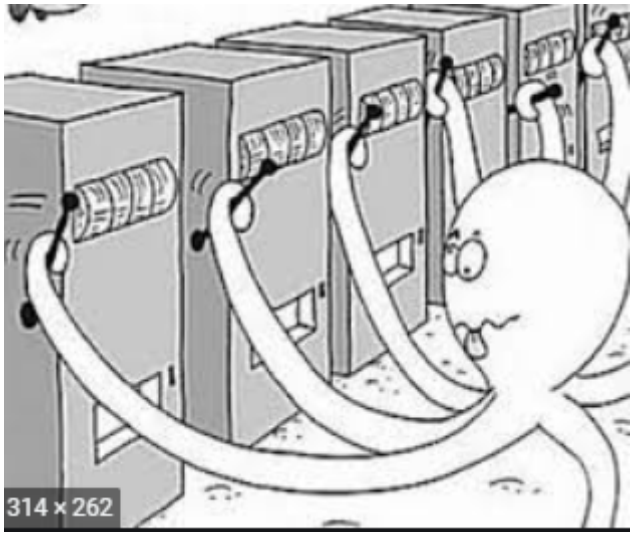


K臂老虎机介绍及其Python实现

如果大家想对K臂老虎机做一个比较深入的了解的话，建议大家去阅读这篇博客，作者写的挺清楚的，而且还推荐了很多的其他材料，我这里主要是对K臂老虎机做一个简要的介绍。

定义

K臂老虎机（Multi-armed bandit，简称MAB）最早的场景是在赌场里面。赌场里面有K台老虎机，每次去摇老虎机都需要一个代币，且老虎机都会以一定概率吐出钱，你手上如果有T个代币，也就是你一共可以摇T次，你怎么才能使你的期望回报最大？当然我们要先假设每个老虎机吐钱的概率是不一样的，不然你怎么摇就都是一样的了。



我们一般也将所有的下面这种形式的问题成为K臂老虎机问题：你可以反复面对 k 种不同的选择或行动。在每次选择之后，你会收到一个数值奖励，该奖励取决于你选择的行动的固定概率分布。 你的目标是在一段时间内最大化预期的总奖励。

如果我们是贝叶斯人，我们在实际对老虎机进行操作之前其实对老虎机吐钱的概率就已经有了一个先验的分布，然后我们不断地进行试验，根据试验的结果取调整我们前面的分布；而如果我们是频率学家，那我们一开始对这些机器吐钱的概率其实是没有先验的，我们会通过实验去预测出每台机器吐钱的概率，然后根据这个概率去不断优化我们的决策。

但不管从哪种角度出发，**K臂老虎机的问题其实就是一个探索与利用的问题**，就比如说我们先进行来m次实验（m<T），发现了第一个臂吐钱的频率更高，那接下来我们是一直去摇第一个臂（利用：exploitation）还是说我们还去试着摇一下其他的臂（探索：exploration），从短期来看利用是好的，但是从长期来看探索是好的。

基本概念

在我们的K臂老虎机中，只要选择了该动作, k 个动作的每一个都有预期的或平均的奖励, 让我们 称之为该行动的 价值。我们将在时间步 t 选择的动作表示为 A_t, 并将相应的奖励表示为 R_t。 然后, 对于任意动作 a 的价值, 定义 $q_*(a)$ 是给定 a 选择的预期奖励：

$$q_*(a) \doteq \mathbb{E}\left[R_t \mid A_t = a\right] \tag{5}$$

如果我们知道每个动作的价值, 那么解决 K臂老虎机将是轻而易举的：你总是选择具有最高价值的动作。但是**我们不知道实际动作价值, 尽管你可能有估计值。** 我们将在时间步骤t 的动作 a 的估计值表示为 $Q_t(a)$ 。 我们希望 $Q_t(a)$ 接近 $q_*(a)$ 。

K臂老虎机的变种

我们在上面定义中介绍的K臂老虎机其实是最简单的一种场景，K臂老虎机还有很多其他的变形：

- 如果那些臂的吐钱的概率分布在一开始就设定好了，而且之后不再改变，则称为**oblivious adversary setting**。
- 如果那些臂吐钱的概率设定好了之后还会发生变化，那么称为**adaptive adversary setting**。
- 如果把待推荐的商品作为MAB问题的arm，那么我们在推荐系统中我们就还需要考虑用户作为一个活生生的个体本身的兴趣点、偏好、购买力等因素都是不同的，也就是我们需要考虑同一臂在不同上下文中是不同的, 这种也被称为**contextual bandits**。
- 如果每台老虎机每天摇的次数有上限，那我们就得到了一个**Bandit with Knapsack**问题。

greedy和$\epsilon-greedy$

greedy（贪婪）的算法也就是选择具有最高估计值的动作之一：$A_t = \underset{a}{\operatorname{argmax}}\, Q_t(a)$，也就是相当于我们只做exploitation ; 而$\epsilon-greedy$以较小的概率 ϵ地从具有相同概率的所有动作中随机选择, 相当于我们在做exploitation的同时也做一定程度的exploration。greedy的算法很容易陷入执行次优动作的怪圈，当reward的方差更大时，我们为了做更多的探索应该选择探索度更大的$\epsilon-greedy$，但是当reward的方差很小时，我们可以选择更greedy的方法，在实际当中我们很多时候都会让ϵ 从一个较大的值降低到一个较小的值，比如说从1降低到0.1，相当于我们在前期基本上只做探索，后期只做利用。

- Trade-off between exploration and exploitation
- ϵ -Greedy Exploration: Ensuring continual exploration
    - Q All actions are tried with non-zero probability
    - With probability $1-\epsilon$ choose the greedy action
    - B With probability ϵ choose an action at random

$$\pi(a \mid s)=\begin{cases}\epsilon/|\mathcal{A}|+1-\epsilon & \text{if } a^*=\arg\max_{a\in\mathcal{A}} Q(s,a)\\ \epsilon/|\mathcal{A}| & \text{otherwise}\end{cases} \tag{6}$$

Policy improvement theorem: For any ϵ -greedy policy π, the ϵ -greedy policy π' with respect to q_π is an improvement, $v_{\pi'}(s) \geq v_\pi(s)$

$$\begin{aligned}
q_\pi\left(s, \pi'(s)\right) &= \sum_{a\in\mathcal{A}} \pi'(a \mid s) q_\pi(s,a) \\
&= \frac{\epsilon}{|\mathcal{A}|}\sum_{a\in\mathcal{A}} q_\pi(s,a) + (1-\epsilon)\max_a q_\pi(s,a) \\
&\geq \frac{\epsilon}{|\mathcal{A}|}\sum_{a\in\mathcal{A}} q_\pi(s,a) + (1-\epsilon)\sum_{a\in\mathcal{A}} \frac{\pi(a \mid s)-\frac{\epsilon}{|\mathcal{A}|}}{1-\epsilon} q_\pi(s,a) \\
&= \sum_{a\in\mathcal{A}} \pi(a \mid s) q_\pi(s,a) = v_\pi(s)
\end{aligned}$$

Therefore, $v_{\pi'}(s) \geq v_\pi(s)$ from the policy improvement theorem

MC with $\epsilon-$ Greedy Exploration

Algorithm 1

Initialize $Q(S,A)=0, N(S,A)=0, \epsilon=1, k=1$
$\pi_k = \epsilon$-greedy(Q)
loop
  Sample k-th episode $(S_1, A_1, R_2, ..., S_T) \sim \pi_k$
  for each state S_t and action A_t in the episode **do**
    $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t))$
  end for
  $k \leftarrow k+1, \epsilon \leftarrow 1/k$
  $\pi_k = \epsilon$-greedy(Q)
end loop

softmax 方法

softamx是另一种兼顾探索与利用的方法，它既不像greedy算法那样贪婪，也没有像$\epsilon-$ greedy那样在探索阶段做随机动作而是使用 softmax函数计算每一个arm被选中的概率，以更高的概率去摇下平均收益高的臂，以更地的概率去摇下平均收益低的臂。 arm_i 表示第i 个手柄， U_i 表示手柄的平均收 益, k 是手柄总数。

$$p\left(arm_i\right)=\frac{e^{u_i}}{\sum_j^k e^{u_i}} \tag{7}$$

当然这里有一个问题是为什么要用softmax，我们直接用某一个臂得到的平均收益除以总的平均收益不行吗？**我理解上感觉softmax方法是在agrmax方法和直接除这种方法之间的方法**，因为softmax加上e之后其实会让平均收益低的臂和平均收益高的臂走向极端，**也就是让策略越来越激进，甚至到最终收敛成argmax**？而且我感觉图像分类里面经常用softmax一方面是因为求梯度比较好计算，另一方面是因为有时候softmax之前得到的分数可能有负数，那我们这里的好处好可以加上就是刚开始某一个臂的平均收益是0的时候我们依旧会有一定概率选它而不会像下面公式里面的这种一样不选它。

$$p\left(arm_i\right)=\frac{u_i}{\sum_j^k u_i} \tag{8}$$

所以总的来说softmax有三个好处：

- 便于求梯度
- 在刚开始某一个臂收益为0的时候这个臂依旧有被选上的可能
- softma**x算法让平均收益低的臂和平均收益高的臂走向极端，也就是让策略越来越激进，甚至到最终收敛成argmax**，就有点像$\epsilon-greedy$中ϵ不断下降一样。

一个简单的赌博机算法

简单的赌博机算法

初始化，a 从 1 到 k:

$$Q(a) \leftarrow 0$$
$$N(a) \leftarrow 0$$

循环:

$$A \leftarrow \begin{cases} argmax_a Q(a) & 以1-\varepsilon概率（随意打破关系） \\ 随机动作 & 以\varepsilon概率 \end{cases}$$
$$R \leftarrow bandit(a)$$
$$N(A) \leftarrow N(A)+1$$
$$Q(A) \leftarrow Q(A)+\frac{1}{N(A)}[R-Q(A)]$$

循环的最后一步其实用到了

$$\begin{aligned} Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\ &= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\ &= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\ &= \frac{1}{n}(R_n + (n-1)Q_n) \\ &= \frac{1}{n}(R_n + nQ_n - Q_n) \\ &= Q_n + \frac{1}{n}(R_n - Q_n) \end{aligned}$$

也就是：新估计←旧估计+步长[目标−旧估计]

Python 代码实现

在代码里面实现了$\epsilon-greedy$、softmax，以及直接根据当前各个臂的平均收益去决策三种方法，完整的代码放在github上了，写的比较匆忙，后面会再更新一下放到github的仓库之中

```python
# 作者：曾云辉
# 创建时间：2021/1/13 23:54
# IDE：PyCharm
# encoding: utf-8

import random
import math
import numpy as np
import matplotlib.pyplot as plt

ARM_NUM = 5
E_GREEDY_FACTOR = 0.9
SEED = random.randint(1, 10000)
TEST_STEPS = 1000


```

```python
class MAB:
    def __init__(self, arm_num: int) -> None:
        """
        :param arm_num:  the number of arms  臂的数量
        """
        self.arm_num = arm_num    # 设置臂的数量
        self.probability = dict({})  # 设置每个臂能摇出一块钱的概率
        self.try_time = dict({})  # 每个臂已经摇过的次数
        self.reward = dict({})  # 每个臂已经获得的钱
        self.reward_all = 0  # 所有臂获得的收益之和
        self.try_time_all = 0  # 总的尝试的次数

    def reset(self, seed: int) -> None:
        """
        Each arm is initialized, and each arm is set the same when passing in the same random seed
        对每一个臂进行初始化，当传入的随机种子一样时，每个臂的设置相同
        :param seed: random seed  传入的随机种子
        """
        print("We have %d arms" % self.arm_num)
        for num in range(self.arm_num):
            random.seed(num+seed)
            self.probability[str(num + 1)] = random.random()
            self.try_time[str(num + 1)] = 0
            self.reward[str(num + 1)] = 0

    def step(self, arm_id: str):
        """
        Change the arm according to the arm_id
        当传入每次要摇下的臂的编号后，老虎机的状态发生变化
        :param arm_id: the id of the arm in this step 这一步控制摇下杆的id
        """
        self.try_time[arm_id] += 1
        self.try_time_all += 1
        if random.random() < self.probability[arm_id]:
            self.reward[arm_id] += 1
            self.reward_all += 1

    def render(self):
        """
           draw the multi-armed bandit,including tried times and reward
           for each arm, and total tried times and rewards.
        """
        if self.arm_num <= 10:
            print('*' * 8 * (self.arm_num + 1) + '**')
            title = str(self.arm_num) + " arm bandit"
            title_format = '{:^' + str(8 * (self.arm_num + 1)) + 's}'
            print('*' + title_format.format(title) + '*')

            print('*' + ' ' * 8 * (self.arm_num + 1) + '*')

            print('*{:^8s}'.format('arm'), end='')
            for arm in range(self.arm_num):
                print('{:^8d}'.format(arm + 1), end='')
            print('*\n')

            print('*{:^8s}'.format('tried'), end='')
            for arm in range(self.arm_num):
```

```
                print('{:^8d}'.format(self.try_time[str(arm + 1)]), end='')
            print('*\n')

            print('*{:^8s}'.format('reward'), end='')
            for arm in range(self.arm_num):
                print('{:^8d}'.format(self.reward[str(arm + 1)]), end='')
            print('*\n')

            print('*' + title_format.format("total tried:" + str(self.try_time_all)) + '*')
            print('*' + title_format.format("total rewards:" + str(self.reward_all)) + '*')
            print('*' + ' ' * 8 * (self.arm_num + 1) + '*')
            print('*' * 8 * (self.arm_num + 1) + '**')


def e_greedy_method(mab):
    """
       e greedy method: define a e_greedy_factor and create a random number,
       when the random number is less then e_greedy_factor, then pick a arm
       randomly, else pick the arm with argmax q_table.
       :param mab: the class MBA
       :return: selected arm_id
    """
    q_table = []
    for arm_num in range(mab.arm_num):
        if mab.try_time[str(arm_num+1)] != 0:
 q_table.append(mab.reward[str(arm_num+1)]/mab.try_time[str(arm_num+1)])
        else:
            q_table.append(0)
    if random.random() < E_GREEDY_FACTOR:
        arm_id = random.randint(1, mab.arm_num)
    else:
        arm_id = np.argmax(q_table) + 1
    return arm_id


def sofmax_method(mab):
    """
    softmax method: calculate the softmax value of each arm's avarage reward,
    and pick the arm with greatest softmax value.
    :param mab: the class MBA
    :return: selected arm_id
    """
    exp_sum = 0
    softmax_list = []

    for arm_num in range(mab.arm_num):
        if mab.try_time[str(arm_num+1)] > 0:
            exp_sum += math.exp(mab.reward[str(arm_num+1)] / mab.try_time[str(arm_num+1)])
        else:
            exp_sum += math.exp(0)
```

```
    assert exp_sum > 0
    for arm_num in range(mab.arm_num):
        if mab.try_time[str(arm_num+1)] == 0:
            avg_reward_temp = 0
        else:
            avg_reward_temp = mab.reward[str(arm_num+1)] /
mab.try_time[str(arm_num+1)]
        softmax_list.append(math.exp(avg_reward_temp) / exp_sum)
    arm_id = np.random.choice(mab.arm_num, 1, p=softmax_list)[0]
    print("The softmax list is", softmax_list)
    print("The id of returned arm is ", arm_id+1)
    return arm_id + 1


def average_method(mab):
    """
    decide the arm_id according to the average return of each arm but
don't do the math.exp() operation like softmax
    :param mab: the class MBA
    :return: selected arm_id
    """
    sum_average = 0
    softmax_list = []

    for arm_num in range(mab.arm_num):
        if mab.try_time[str(arm_num + 1)] > 0:
            sum_average += (mab.reward[str(arm_num + 1)] /
mab.try_time[str(arm_num + 1)])
        else:
            sum_average += 0
    if sum_average == 0:
        arm_id = np.random.choice(mab.arm_num) + 1
    else:
        for arm_num in range(mab.arm_num):
            if mab.try_time[str(arm_num + 1)] == 0:
                avg_reward_temp = 0
            else:
                avg_reward_temp = mab.reward[str(arm_num + 1)] /
mab.try_time[str(arm_num + 1)]
            softmax_list.append(avg_reward_temp / sum_average)
        arm_id = np.random.choice(mab.arm_num, 1, p=softmax_list)[0]
    print("The softmax list is", softmax_list)
    print("The id of returned arm is ", arm_id + 1)
    return arm_id + 1


if __name__ == '__main__':
    reward_list = []
    mab_test = MAB(ARM_NUM)
    print("****Multi-armed Bandit***")
    mab_test.reset(SEED)
    mab_test.render()
    for i in range(TEST_STEPS):
        mab_test.step(str(average_method(mab_test)))
        reward_list.append(mab_test.reward_all/mab_test.try_time_all)
        if (i+1) % 20 == 0:
            print("We have test for %i times" % (i+1))
            mab_test.render()
```

```
    plt.plot(reward_list)
    plt.show()

```

Bellman Equation

贝尔曼方程定义了状态之间的迭代关系，是强化学习里面它特别重要的一个知识点。

V(s)

价值函数其实从一个更长远的角度定义的一个状态的好坏，价值函数不仅仅考虑了短期的即时奖励，更重要的是价值函数考虑了到达这个状态会带来的长期的效益，因此相比于reward是一个更长远地衡量状态的方式。

$V(s)=R(s)+\gamma \sum_{s' \in S} P\left(s' \mid s\right) V\left(s'\right)$ （第一部分可以看作立即奖励，第二部分可以看作未来的奖励）

$$\begin{bmatrix} V\left(s_1\right) \\ V\left(s_2\right) \\ \vdots \\ V\left(s_N\right) \end{bmatrix} = \begin{bmatrix} R\left(s_1\right) \\ R\left(s_2\right) \\ \vdots \\ R\left(s_N\right) \end{bmatrix} + \gamma \begin{bmatrix} P\left(s_1 \mid s_1\right) & P\left(s_2 \mid s_1\right) & \ldots & P\left(s_N \mid s_1\right) \\ P\left(s_1 \mid s_2\right) & P\left(s_2 \mid s_2\right) & \ldots & P\left(s_N \mid s_2\right) \\ \vdots & \vdots & \ddots & \vdots \\ P\left(s_1 \mid s_N\right) & P\left(s_2 \mid s_N\right) & \ldots & P\left(s_N \mid s_N\right) \end{bmatrix} \begin{bmatrix} V\left(s_1\right) \\ V\left(s_2\right) \\ \vdots \\ V\left(s_N\right) \end{bmatrix}$$

$$V = R + \gamma PV$$

$$V = (I - \gamma P)^{-1} R$$

在求解的过程中有两个比较重要的问题：

- 怎么保证矩阵是可逆的?
- 矩阵求逆的复杂度是$O\left(N^3\right)$，所以这种求解方式只对小的MRP的问题有效

Q（s,a)

动作价值函数，我们一般也会直接称q函数定义了状态-动作对的好坏，其实我们只要知道了V函数，Q函数包括我们后面讲的A（优势）函数都可以求出来，而如果知道这三个函数中的任何一个，整个MDP其实就是可解的。但是我们没办法一开始就知道，所以我们一般都会通过一个初始化的价值函数或者直接初始化一个策略去采样去然后去估算价值函数，然后再根据新的价值函数再去采样，这样理想情况下我们对价值函数的估计越来越准，决策也越来越好。

The action-value function $q^\pi(s,a)$ is the expected return starting from state s, taking action a, and then following policy π

$$q^\pi(s,a) = \mathbb{E}_\pi \left[G_t \mid s_t = s, A_t = a \right]$$

$$v^\pi(s) = \sum_{a \in A} \pi(a \mid s) q^\pi(s,a)$$

Bellman Expection Eqution

$v^\pi(s) = E_\pi \left[R_{t+1} + \gamma v^\pi \left(s_{t+1} \right) \mid s_t = s \right]$

$q^\pi(s,a) = E_\pi \left[R_{t+1} + \gamma q^\pi \left(s_{t+1}, A_{t+1} \right) \mid s_t = s, A_t = a \right]$

$$v^\pi(s) = \sum_{a \in A} \pi(a \mid s) q^\pi(s,a)$$

$$q^\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P\left(s' \mid s, a\right) v^\pi\left(s'\right)$$

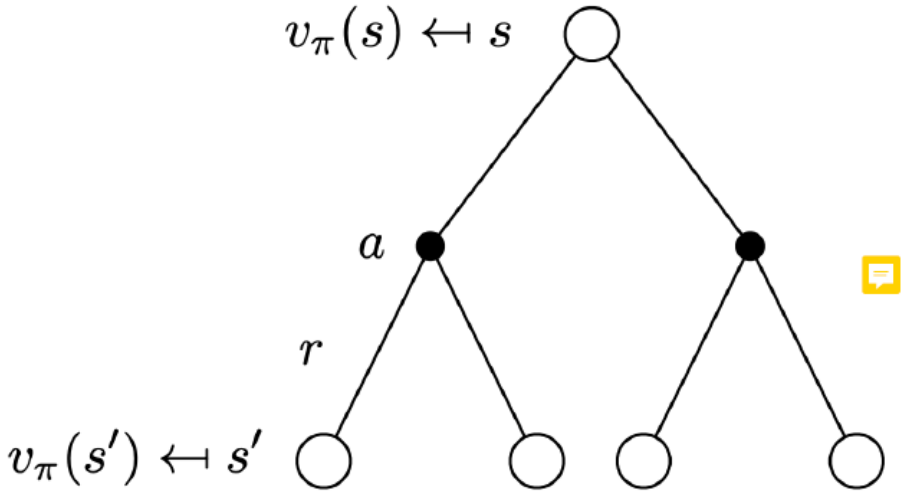
Thus，将两个方程互相带入彼此

$$v^{\pi}(s)=\sum_{a\in A}\pi(a\mid s)\left(R(s,a)+\gamma\sum_{s'\in S}P\left(s'\mid s,a\right)v^{\pi}\left(s'\right)\right) \tag{9}$$

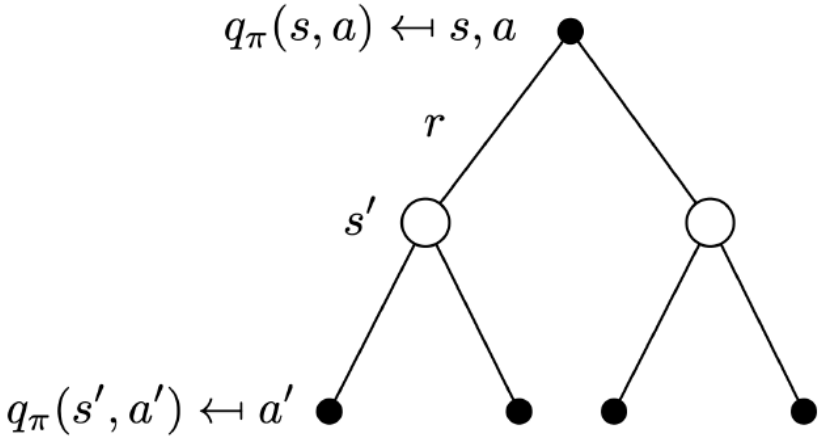
$$q^{\pi}(s,a)=R(s,a)+\gamma\sum_{s'\in S}P\left(s'\mid s,a\right)\sum_{a'\in A}\pi\left(a'\mid s'\right)q^{\pi}\left(s',a'\right) \tag{10}$$

以上等式表征了当前的Q函数和V函数与将来的Q函数和V函数之间的关系

其实就是下面的两个图：



$$v^{\pi}(s)=\sum_{a\in A}\pi(a|s)(R(s,a)+\gamma\sum_{s'\in S}P(s'|s,a)v^{\pi}(s'))$$



$$q^{\pi}(s,a)=R(s,a)+\gamma\sum_{s'\in S}P(s'|s,a)\sum_{a'\in A}\pi(a'|s')q^{\pi}(s',a')$$

Bellman Optimality Equation

$$v^*(s) = \max_a q^*(s,a)$$
$$q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P\left(s' \mid s,a\right) v^*\left(s'\right)$$

thus

$$v^*(s) = \max_a R(s,a) + \gamma \sum_{s' \in S} P\left(s' \mid s,a\right) v^*\left(s'\right)$$
$$q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P\left(s' \mid s,a\right) \max_{a'} q^*\left(s',a'\right)$$

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}\left[G_t \mid S_t = s, A_t = a\right] \\ &= \max_a \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a\right] \\ &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*\left(S_{t+1}\right) \mid S_t = s, A_t = a\right] \\ &= \max_{a \in \mathcal{A}(s)} \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_*\left(s'\right)\right] \end{aligned}$$

最后两个方程是 v_* 的贝尔曼最优方程的两种形式, q_* 的贝尔曼最优方程为

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \sum_{a'} q_*\left(S_{t+1,a'}\right) \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma \sum_{a'} q_*\left(s', a'\right)\right] \end{aligned}$$

贝尔曼最优方程其实阐述了一个事实：**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。**

对于有限的MDP， v_* 的贝尔曼最优方程具有唯一解。贝尔曼最优方程实际上是一个方程组，每个状态一个方程，所以如果有 n 个状态, 则有 n 个未知数的 n 个方程。 如果环境的动态 p 是已知，则原则上可以使用解决非线性方程组的各种方法中的任何一种来求解该 v_* 的方程组。 同样, 可以求解 q_* 的一组相关方程。

一旦有 v_*, 确定最优策略就比较容易了。对于每个状态, 将在贝尔曼最优方程中获得最大价值的一 个或多个动作。 任何仅为这些操作分配非零概率的策略都是最优策略。你可以将其视为一步步的搜索。 **如果具有最优价值函数 v_*, 则在一步搜索之后出现的动作将是最优动作。 另一种说法的方法是任何对最优评估函数 v_* 贪婪的策略是最优策略。**

计算机科学中使用术语贪楚来描述任何基于本地或直接考虑来选择替代搜索或决策程序的方法，而不考虑这种选择可能阻止未来获得更好的替代方法的可能性。因此，它描述了根据其短期结果选择行动的策略。

v_* 的美妙之处在于，如果用它来评估行动的短期结果，具体来说是一步到位的结果， 那么贪婪的策略在我们感兴趣的长期意义上 实际上是最优的，因为 v_* 已经考虑到所有可能的未来动作的奖励结果。**定义 v_* 的意义就在于，我们可以将最优的长期（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。**因此, 一步一步的搜索产生长期的最佳动作。

有 q_* 使选择最优动作更容易。 给定q_*, agent基至不需要进行单步搜索的过程， 对于任何状态 s 它可以简单地发现任何使 $q_*(s,a)$ 最大化的动作。　　动作价值函数有效地缓存了所有单步搜索的结果。 它将最优的长期回报的期望作为本地并立即可用于每个状态一动作对的值。因此, 代表状 态-动作对的功能而不仅仅是状态的代价， 最优动作-价值函数允许选择最优动作而不必知道关于可能的后继状态及其值的任何信息，即不需要知道任何环境的动态变化特性了。

总结也就是说有了v_*之后，还需要做单步搜索，但是有了q_*后，就不需要了。

显示地求解贝尔曼最优方程提供了求解最优策略的一条途径，从而为解决强化学习问题提供了依据。 但是, 这个解决方案很少直接有用。它类似于穷举搜索，展望所有可能性,计算每种可能性出现的概率及其期望收益。这个解决方案依赖于至少三个假设，在实践中很少是这样的:

- **我们准确地知道环境的动态特性**;
- **我们有足够的计算资源来完成解决方案的计算;**
- **马尔可夫性。**

对于我们感兴趣的任务，通常不能完全满足这三个条件。 例如，虽然第一和第三个假设对于西洋双陆棋游戏没有任何问题, 但第二个是主要的障碍。 因为游戏有 10^{20} 个状态, 所以今天最快的电脑需要数千年的时间才能计算出 v_* 的贝尔曼方程式, 而找到 q_* 也是如此。在强化学习中，我们通常只能用近似解法来解决这类问题。