



# Reinforcement learning (RL) 学习

## 强化学习学习笔记

作者：WANG Maonan

组织：BUPT & UiA

时间：October 26, 2020

版本：2.1



文艺数学君

换一种姿势学数学

模板来自，*ElegantBook*, 感谢模版的制作。

*Victory won't come to us unless we go to it. — M. Moore*

*The only stupid question is the one you were afraid to ask but never did. — Rich Sutton*

# 目 录

---

<b>1 马尔科夫决策过程 (Markov Decision Processes, MDP)</b>	<b>1</b>	2.2 Policy Evaluation . . . . .	<b>21</b>
1.1 马氏过程 (Markov Processes) . . . . .	1	2.2.1 关于收敛的说明	22
1.2 Markov Reward Processes	2	2.2.2 Policy Evaluation 的例子 . . . . .	23
1.2.1 Bellman Equation for MRP (贝尔曼等式) . . . . .	4	2.3 Policy Iteration . . . . .	26
1.2.2 Bellman Equation for MRP 举例 . . . . .	5	2.3.1 Policy Iteration 的例子 . . . . .	27
1.3 Markov Decision Processes	8	2.4 Value Iteration . . . . .	29
1.3.1 Policy function of MDP . . . . .	9	2.4.1 Value Iteration 的例子 . . . . .	30
1.3.2 Two Value Functions of MDP . . . . .	9	<b>3 Model-Free Prediction</b>	<b>32</b>
1.3.3 Bellman Equation of MDP . . . . .	10	3.1 Monte-Carlo Learning . . . . .	32
1.3.4 Optimal Value Function . . . . .	13	3.1.1 MC 方法总体介绍	32
<b>2 Planning by Dynamic Programming</b>	<b>16</b>	3.1.2 MC 方法详细步骤	33
2.1 动态规划介绍 . . . . .	16	3.1.3 Incremental Mean	33
2.1.1 递归的思想 (recursion)--预备知识 . . . . .	16	3.2 Temporal-Difference Learning . . . . .	35
2.1.2 动态规划的思想--预备知识 . . . . .	18	3.2.1 TD 方法思想 . . . . .	35
2.1.3 动态规划解 MDP 问题 . . . . .	20	3.2.2 TD 与 DP 和 MC 的比较 . . . . .	36
		3.3 TD( $\lambda$ ) . . . . .	37
		3.3.1 n-Step Prediction	37
		3.3.2 $\lambda$ -return . . . . .	38
<b>4 Model-Free Control</b>	<b>43</b>		
		4.1 On and Off-Policy Learning	43
		4.2 On-Policy Monte-Carlo Control . . . . .	43
		4.2.1 Monte-Carlo Control 的基本思想 . . . . .	43

4.2.2	$\epsilon$ -Greedy Exploration . . . . .	44	5.1.2	Linear Value Function Approximation . . .	65
4.2.3	Annealing $\epsilon$ -greedy mechanism	45	5.1.3	Table Lookup Features . . . . .	65
4.2.4	Greedy in the Limit with Infinite Exploration (GLIE) . . . . .	46	5.1.4	Incremental Prediction Algorithms	66
4.2.5	BlackJack 与 GLIE	46	5.1.5	Control with Value Function Approximation . . . . .	67
4.2.6	Windy Grid-world Play-ground 与 GLIE	47	5.1.6	The example of Mountain Car . . . . .	67
4.3	On-Policy Temporal-Difference Control . . . . .	49	5.2	Batch Reinforcement Learning . . . . . . .	68
4.3.1	Sarsa . . . . .	50	5.2.1	Deep Q-Networks (DQN)	69
4.3.2	n-Step Sarsa . . . . .	51	5.2.2	Deep Q-Networks (DQN) 的例子 . . . . .	71
4.3.3	Eligibility Traces	52			
4.3.4	Sarsa 与 Sarsa( $\lambda$ ) 的比较 . . . . .	53	<b>6 Policy Gradient</b>		73
4.4	Off-Policy Learning . . . . .	54	6.1	Advantages of Policy-Based RL . . . . .	73
4.4.1	Importance Sampling . . . . .	55	6.1.1	Rock-Paper-Scissors . . . . .	73
4.4.2	Importance Sampling for Off-Policy Monte-Carlo	55	6.1.2	Aliased Gridworld	74
4.4.3	Importance Sampling for Off-Policy TD . . . . .	56	6.2	Policy Objective Function and Optimisation . . . . .	75
4.4.4	Q-Learning . . . . .	57	6.2.1	Policy Objective Function . . . . .	75
4.4.5	N-step Q-Learning	58	6.2.2	Policy Optimisation . . . . .	76
<b>5</b>	<b>Value Function Approximation</b>	<b>64</b>	6.3	Monte-Carlo Policy Gradient . . . . .	77
5.1	Incremental Methods . . . . .	64	6.3.1	Score Function . . . . .	77
5.1.1	Feature Vectors . . . . .	64	6.3.2	One-Step MDPs	77

---

6.3.3	Policy Gradient Theorem . . . . .	78	6.4.3	Compatible Function Ap- proximation . . . . .	80
6.3.4	Monte-Carlo Policy Gradient (REINFORCE) . . . . .	78	6.4.4	Reducing Vari- ance Using Base- line . . . . .	81
6.4	Actor-Critic Policy Gra- dient . . . . .	79	6.4.5	Estimating the Advantage Func- tion . . . . .	82
6.4.1	Reduce Variance Using a Critic . . . . .	79	6.5	Actor-Critic Baseline (A2C) 实验 . . . . .	83
6.4.2	Estimating the Action-Value Function . . . . .	79			

# 第1章 马尔科夫决策过程 (Markov Decision Processes, MDP)

## 内容提要

- Markov Processes
- Markov Decision Processes
- Markov Reward Processes

## 1.1 马氏过程 (Markov Processes)

首先从最基础的概念介绍起. 首先介绍, 马尔科夫性质 (**Markov Property**).

### 定义 1.1. 马尔科夫性质 (Markov Property)

马尔科夫性质的数学表示方式如下:

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, \dots, S_t] \quad (1.1)$$

也就是, 未来状态的条件概率只依赖于当前的状态, 与过去状态是条件独立的.

于是从状态  $s$  转移到下一个状态  $s'$  的概率, 我们可以写成下面的表达式.

$$P_{ss'} = P[S_{t+1} = s' | S_t = s] \quad (1.2)$$

于是, 在马氏过程中, 我们可以将从一个 state 达到另一个 state 的概率表示为一个状态转移矩阵 (state transition matrix).

$$P = \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nn} \end{bmatrix} \quad (1.3)$$

于是, 对于整个 Markov Processes, 我们可以有如下的定义:

### 定义 1.2. Markov Processes

一个马尔科夫过程是无记忆的随机过程 (a Markov process is a memoryless random process).

马尔科夫过程是一个二元组  $\langle S, P \rangle$ , 其中:

- $S$  表示有限状态的集合.
- $P$  表示状态转移矩阵, 其中每一个元素为,  $P_{ss'} = P[S_{t+1} = s' | S_t = s]$ .

下面我们来看一个 Markov process 的例子. 这一部分例子和例子图片来自于UCL Course on RL.

### 例 1.1

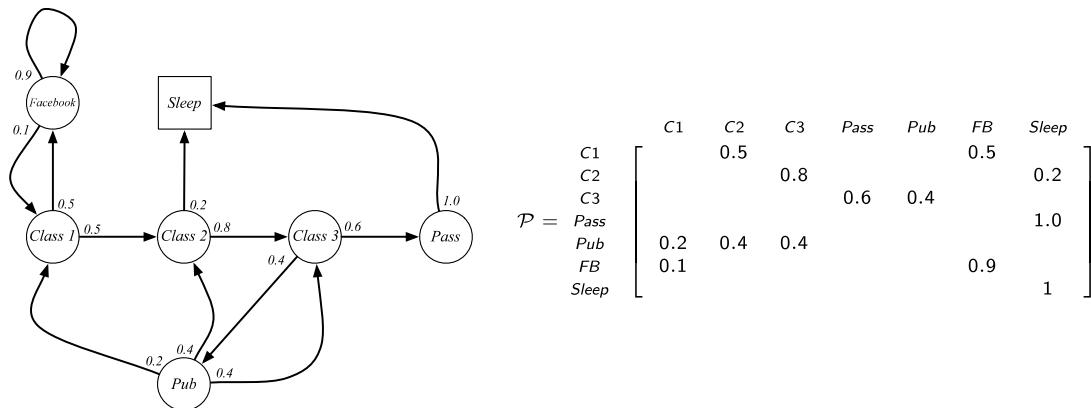


图 1.1: 马尔科夫过程例子.

在上面的图中, 左侧表示所有的 state 的连接关系, 右侧是他们的状态转移矩阵. 注意最后 sleep 的状态, 这是一个自吸收的状态, 自己转移到自己, 概率是 1.

## 1.2 Markov Reward Processes

Markov Reward Processes 是在 Markov Processes 的基础上加上了 **values**. 这些 value 会告诉我们这个 state 有多好. 我们在之前的 Markov Processes 中增加两样东西, 分别是  $R$  (reward function) 和  $\gamma$  (折损因子). 我们来看他完整的定义.

### 定义 1.3. Markov Reward Processes

Markov Reward Processes 是一个元祖  $\langle S, P, R, \gamma \rangle$ . 其中:

- $S$  表示有限状态的集合.
- $P$  表示状态转移矩阵, 其中每一个元素为,  $P_{ss'} = P[S_{t+1} = s' | S_t = s]$ .

- $R$  是 reward function,  $R_s = E[R_{t+1}|S_t = s]$ , 表示能从状态  $s$  获得多少 reward, 这是 immediate reward (当前时刻获得的 reward).
- $\gamma$ , 在计算累计收益的时候, 对未来的收益乘上的一个因子. 可以避免在解 bellman equation 时出现矩阵不可逆, 后面会有讲到.

1.4 所示.

$$U_t = R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot R_{t+2} + \cdots + \gamma^{T-t} \cdot R_T. \quad (1.4)$$

因为环境是随机的, 所以我们可能每一次求  $U_t$  结果都是不同的. 于是我们希望求  $U_t$  的期望. 于是我们定义了 value function  $v(s)$ .

#### 定义 1.4. State Value Function $v(s)$

state value function  $v(s)$  是从状态  $s$  开始的期望回报:

$$v(s) = E[U_t|S_t = s]. \quad (1.5)$$

那么我们如何求  $v(s)$  呢, 一个最简单的想法就是可以进行多次随机试验. 从我们要计算的 state 开始, 多次模拟, 计算累计收益之后求平均. 下面来看一个简单的例子.

**例 1.2** 对于 Markov process 的例子 (图1.1), 我们在这里加上 reward. 加上 reward 的例子如图1.2所示.

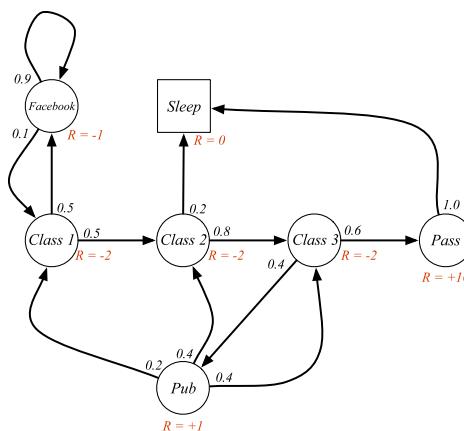


图 1.2: Markov Reward Processes 例子.

我们假设我们要计算  $v(s_1 = c_1)$ , 设此时  $\gamma = \frac{1}{2}$ .

那么我们进行多次模拟, 可能得到如下的序列:

- $\{C1, C2, C3, Pass, Sleep\}$ :

- $v(s_1 = c_1) = -2 + (-2) \cdot \frac{1}{2} + (-2) \cdot (\frac{1}{2})^2 + 10 \cdot (\frac{1}{2})^3 = -2.25$
- $\{C1, FB, FB, C1, C2, Sleep\}$ :
  - $v(s_1 = c_1) = -2 + (-1) \cdot \frac{1}{2} + (-1) \cdot (\frac{1}{2})^2 + (-2) \cdot (\frac{1}{2})^3 + (-2) \cdot (\frac{1}{2})^4 = -3.125$
  - ...

### 1.2.1 Bellman Equation for MRP (贝尔曼等式)

上面介绍的 value function  $v(s) = E[U_t | S_t = s]$  可以分成两个部分:

- 当前立即的回报 (immediate reward),  $R_{t+1}$ ;
- 下一个状态的 discounted value,  $\gamma \cdot v(S_{t+1})$

这是因为, 我们可以按照下面的方式进行分解:

$$\begin{aligned}
 v(s) &= E[U_t | S_t = s] \\
 &= E[R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot R_{t+2} + \gamma^3 \cdot R_{t+3} + \dots | S_t = s] \\
 &= E[R_t + \gamma \cdot (R_{t+1} + \gamma^1 \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \dots) | S_t = s] \\
 &= E[R_t + \gamma \cdot U_{t+1} | S_t = s] \\
 &= E[R_t + \gamma \cdot v(S_{t+1}) | S_t = s]
 \end{aligned} \tag{1.6}$$

于是我们就有了下面的式子,

$$v(s) = E[R_t + \gamma \cdot v(S_{t+1}) | S_t = s] \tag{1.7}$$

他表示当前状态是  $s$  的情况下的 value function 的值. 里面出现的  $v(S_{t+1})$  中的  $S_{t+1}$  是一个随机变量, 这里  $S_{t+1} \in State\_Set$ .

我们给一个更加详细的解释, 如下图 1.3 所示. 该图表示的是从当前状态  $s$  到下一步的状态  $s'$ . 且我们知道从状态  $s$  到下一步的状态  $s'$  的概率, 也就是  $P_{ss'}$ .

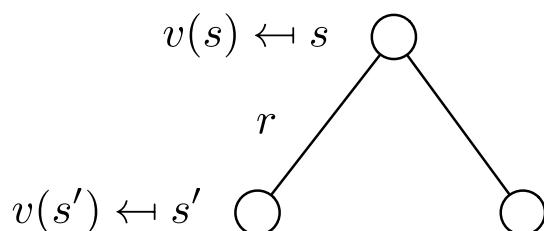


图 1.3: Bellman Equation 的解释.

于是, 我们可以对上面的 Bellman Equation 做进一步的化简.

$$\begin{aligned}
 v(s) &= E[R_t + \gamma \cdot v(S_{t+1}) | S_t = s] \\
 &= R_t + \gamma \cdot \sum_{s' \in S} P_{ss'} \cdot v(s')
 \end{aligned} \tag{1.8}$$

我们可以将上面的表达式使用矩阵来进行表示, 也就是下面的是式子1.9:

$$\begin{bmatrix} v(s_1) \\ \vdots \\ v(s_n) \end{bmatrix} = \begin{bmatrix} R_1 \\ \vdots \\ R_n \end{bmatrix} + \gamma \cdot \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nn} \end{bmatrix} \begin{bmatrix} v(s_1) \\ \vdots \\ v(s_n) \end{bmatrix} \tag{1.9}$$

因为 Bellman Equation 是一个 linear equation, 如果我们知道转移矩阵  $P$  和每一个 state 的 reward  $R$ , 我们就将  $\vec{v}$  计算出来.

$$\begin{aligned}
 \vec{v} &= \vec{R} + \gamma P \vec{v} \\
 (I - \gamma \cdot P) \vec{v} &= \vec{R} \\
 \vec{v} &= (I - \gamma \cdot P)^{-1} \vec{R}
 \end{aligned} \tag{1.10}$$

上面  $I$  表示对角矩阵.

## 1.2.2 Bellman Equation for MRP 举例

下面我们举一个具体的例子, 来看一下如何求解 state value  $v$ . 在这里我们对环境是已知的, 也就是知道状态转移矩阵  $P$  的.

**例 1.3** 我们还是使用上面的 Markov process 的例子. 我们假设此时  $\gamma = 1$ . 图1.4是详细的状态转移图.

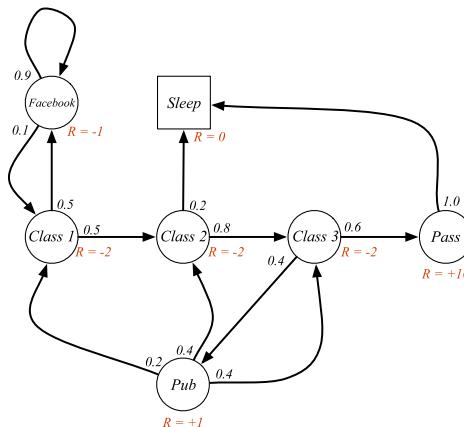


图 1.4: Markov Reward Processes 例子.

此时, 状态转移矩阵为:

$$\begin{matrix}
 & c1 & c2 & c3 & pass & pub & FB & sleep \\
 c1 & 0 & 0.5 & 0 & 0 & 0 & 0.5 & 0 \\
 c2 & 0 & 0 & 0.8 & 0 & 0 & 0 & 0.2 \\
 p = & c3 & 0 & 0 & 0 & 0.6 & 0.4 & 0 & 0 \\
 & pass & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
 & pub & 0.2 & 0.4 & 0.4 & 0 & 0 & 0 & 0 \\
 & FB & 0.1 & 0 & 0 & 0 & 0 & 0.9 & 0 \\
 & sleep & 0 & 0 & 0 & 0 & 0 & 0 & 1
 \end{matrix} \tag{1.11}$$

此时的 reward 为:

$$\begin{bmatrix} r(c1) \\ r(c2) \\ r(c3) \\ r(pass) \\ r(pub) \\ r(fb) \\ r(sleep) \end{bmatrix} = \begin{bmatrix} -2 \\ -2 \\ -2 \\ +10 \\ +1 \\ -1 \\ 0 \end{bmatrix} \tag{1.12}$$

于是, 根据上面的 Bellman Equation, 我们可以求解得到  $\vec{v}$ . 其实下面在具体求得时候会出现问题, 也就是当  $\gamma = 1$  的时候, 矩阵  $(I - \gamma \cdot P)^{-1}$  是不可逆的, 存在全 0 行. 我在实际求得时候, 取了  $\gamma = 0.99999$ .

$$\begin{bmatrix} v(c1) \\ v(c2) \\ v(c3) \\ v(pass) \\ v(pub) \\ v(fb) \\ v(sleep) \end{bmatrix} = \left( \begin{matrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.8 & 0 & 0 & 0 & 0.2 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.6 & 0.4 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0.2 & 0.4 & 0.4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0.1 & 0 & 0 & 0 & 0 & 0.9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{matrix} \right)^{-1} \begin{bmatrix} -2 \\ -2 \\ -2 \\ +10 \\ +1 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} -13 \\ -1.4 \\ 4.3 \\ +10 \\ +0.8 \\ -22 \\ 0 \end{bmatrix} \tag{1.13}$$

关于上面式子的求解, 可以使用 numpy 进行完成.

```

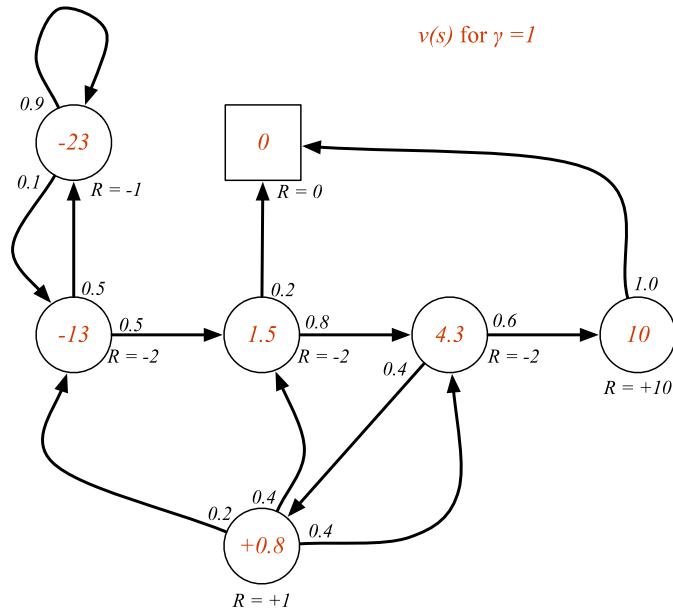
1 p = np.array([[0, 0.5, 0, 0, 0.5, 0], [0, 0, 0.8, 0, 0, 0.2],
2 [0, 0.0, 0.6, 0.4, 0, 0], [0, 0, 0, 0, 0, 1],
3 [0.2, 0.4, 0.4, 0, 0, 0], [0.1, 0, 0, 0, 0.9, 0], [0, 0, 0, 0, 0, 1]])
4
5 r = np.array([-2,-2,-2,10,1,-1,0])
6 I = np.eye(7)
7
8 np.dot(np.linalg.inv(I-0.99999*p), r.reshape(-1,1))

```

array([-12.54073351, 1.45690179, 4.32117045, 10., 0.80308417, -22.53857963, 0.])

图 1.5: 求解 Bellman Equation.

最终, 每个 state 的 state value 如图1.6所示:

图 1.6:  $\gamma = 1$  时的 state value.

我们验证其中的一个 state, 查看是否符合  $v(s) = R_t + \gamma \cdot \sum_{s' \in S} P_{ss'} \cdot v(s')$ . 假设现在在 class3 这个状态, 也就是图1.7中红色的部分.

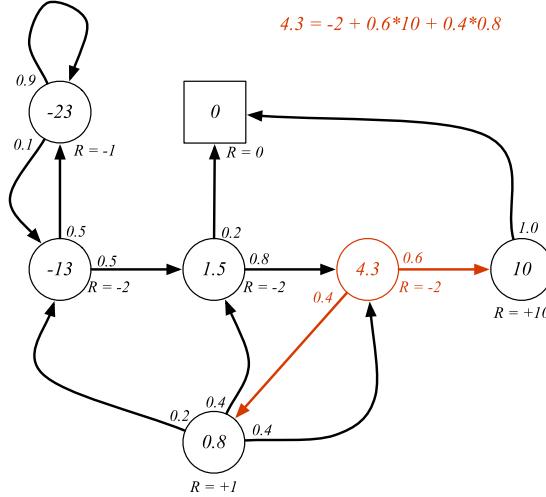


图 1.7:  $\gamma = 1$  时的 state value 的结果验证.

- 在 class3 这个状态可以获得的 reward 是 -2.
  - 从 class3 有 0.6 的概率进入 pass, 这个 state 的 v 值是 10.
  - 从 class3 有 0.4 的概率进入 pub, 这个 state 的 v 值是 0.8.
- 于是, 我们有:

$$v(class3) = -2 + 0.6 \cdot 10 + 0.4 \cdot 0.8 = 4.32 \quad (1.14)$$

这个结果和上面我们算出来是一样的, 也就是上面的结果是满足 bellman equation 的.

但是对  $n \cdot n$  矩阵求逆的时间复杂度为  $O(n^3)$ , 所以上面直接求解一般用在规模较小的问题上面. 对于一些较大规模的问题, 可以使用下面的方式求解 (后面都会涉及):

- Dynamic programming
- Monte-Carlo evaluation
- Temporal-Difference learning

## 1.3 Markov Decision Processes

接下来我们来将 Markov Decision Processes (MDP) 的相关内容. Markov Decision Processes 是在 Markov Reward Processes 的基础上加上了 action. 之前我们是根据状态转移矩阵  $P$  来计算从状态  $s$  到状态  $s'$  的概率. 而我们现在可以根据 action 进行转移. 也就是说状态转移矩阵要写成  $P_{ss'}^a$ .

首先我们给出 Markov Decision Processes 的定义.

### 定义 1.5. Markov Decision Processes

Markov Decision Processes 是一个元祖  $\langle S, A, P, R, \gamma \rangle$ . 其中:

- $S$  表示有限状态的集合.
- $A$  表示有限的动作集合.
- $P$  表示状态转移矩阵, 其中每一个元素为,  $P_{ss'}^a = P[S_{t+1} = s' | S_t = s, A_t = a]$ . 也就是对于每一个 action, 都有一个对应的状态转移矩阵
- $R$  是 reward function,  $R_s^a = E[R_{t+1} | S_t = s, A_t = a]$ , 表示能在状态  $s$  做动作  $a$  获得多少 reward, 这是 immediate reward(当前时刻获得的 reward).
- $\gamma$ , 在计算累计收益的时候, 对未来的收益乘上的一个因子. (可以避免在解 bellman equation 时出现矩阵不可逆, 见例 1.3)



### 1.3.1 Policy function of MDP

在有了 action 之后, 我们想的就是可以有一套策略, 告诉我们在什么时候该做什么事情, 使得我们可以获得最大的累计 reward. 为了形式化的表示, 我们首先定义一个 policy 的函数.

### 定义 1.6. Policy 的定义

Policy  $\pi$  就是在给定 state 情况下, 采取不同 action 的概率.

$$\pi(a|s) = P[A_t = a | S_t = s] \quad (1.15)$$

我们需要注意的是, MDP Policies 只依赖于当前的 state. 且目前我们考虑静态的 policy, 也就是无论什么时候在这个 state,  $\pi(a|s)$  都是一样的.



policy 不要与状态转移矩阵弄混淆. 这里 policy 只是返回在状态为  $s$  的情况下做出各种 action 的概率 (也就是 policy 函数返回的是 action 的概率). 然后环境才是根据当前的状态和做出的动作返回新的环境, 也就是  $P_{ss'}^a$ . Policy 不是返回新环境  $s'$  的概率.

### 1.3.2 Two Value Functions of MDP

因为我们要找出最优的策略, 需要计算累计 reward. 于是从 **action** 角度出发和从 **state** 角度出发会有两种不同的 value function (在 MRP 的时候因为没有 action, 所以只有 state value function).

接下来是 state value function of MDP 的定义. 在 MDP 中, 使用不同的 action 的组合, 也就是使用不同的 policy, 会导致不同的累计 reward.

所以我们的 state value function 可以按下面方式进行定义.



**定义 1.7. State Value Function  $v^\pi(s)$  of MDP**

State value function of MDP  $v_\pi(s)$  是策略  $\pi$  从状态  $s$  开始的期望回报:

$$v_\pi(s) = E_\pi[U_t | S_t = s]. \quad (1.16)$$

这样我们就有了 state value function  $v_\pi(s)$ , 他告诉我们这个 state 会有多好.

同样, 我们还可以定义 action value function, 他告诉我们在特定的 state 下, 采取某个 action 的好坏.

**定义 1.8. Action Value Function  $q_\pi(s, a)$  of MDP**

Action value function of MDP  $q_\pi(s, a)$  是使用策略  $\pi$ , 从状态  $s$  开始, 执行动作  $a$  的期望累计回报:

$$q_\pi(s, a) = E_\pi[U_t | S_t = s, A_t = a]. \quad (1.17)$$

### 1.3.3 Bellman Equation of MDP

于是, 同样的想法, 我们可以将上面的  $v_\pi(s)$  和  $q_\pi(s, a)$  进行分解. **我们通过分解可以找出一个递推关系式.** 分解的结果为两部分:

- Immediate reward
- Discounted value of successor state

首先是对 state value function  $v_\pi(s)$  进行分解:

$$\begin{aligned} v_\pi(s) &= E_\pi[U_t | S_t = s] \\ &= E_\pi[R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot R_{t+2} + \gamma^3 \cdot R_{t+3} + \dots | S_t = s] \\ &= E_\pi[R_t + \gamma \cdot (R_{t+1} + \gamma^1 \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \dots) | S_t = s] \quad (1.18) \\ &= E_\pi[R_t + \gamma \cdot U_{t+1} | S_t = s] \\ &= E_\pi[R_t + \gamma \cdot v_\pi(S_{t+1}) | S_t = s] \end{aligned}$$

接着对 action value function  $q_\pi(s, a)$  进行分解.

$$\begin{aligned} q_\pi(s, a) &= E_\pi[U_t | S_t = s, A_t = a] \\ &= E_\pi[R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot R_{t+2} + \gamma^3 \cdot R_{t+3} + \dots | S_t = s, A_t = a] \\ &= E_\pi[R_t + \gamma \cdot (R_{t+1} + \gamma^1 \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \dots) | S_t = s, A_t = a] \quad (1.19) \\ &= E_\pi[R_t + \gamma \cdot U_{t+1} | S_t = s, A_t = a] \\ &= E_\pi[R_t + \gamma \cdot q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \end{aligned}$$

我们对  $v_\pi(s)$  可以进行进一步的化简. 图1.8表示, 在状态为  $s$  下, 下一步会采取的动作.

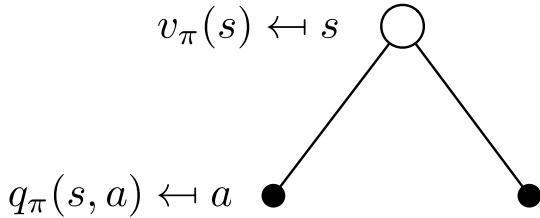


图 1.8: State Value function 的 Bellman Equation 在 MDP 的解释.

如图1.8中所示:

- 空心圆表示 state.
- 从当前 state 到每一个 action 的概率为  $\pi(a|s)$ .
- 实心黑色圆表示 action, 且在  $state = s, action = a$  能获得的累计收益是  $q_\pi(s, a)$ .

于是, 我们可以进一步化简  $v_\pi(s)$ .

$$\begin{aligned} v_\pi(s) &= E_\pi[R_t + \gamma \cdot v_\pi(S_{t+1}) | S_t = s] \\ &= \sum_{a \in A} \pi(a|s) \cdot q_\pi(s, a) \end{aligned} \quad (1.20)$$

我们只需要按不同 action 出现概率进行加权平均即可. 因为只是从  $s \rightarrow a$ , 所以这里还没有 immediate reward, 这一部分包含在  $q_\pi(s, a)$ .

接下来, 我们用同样的思路对  $q_\pi(s, a)$  进行分解. 我们使用图1.9来进行解释.

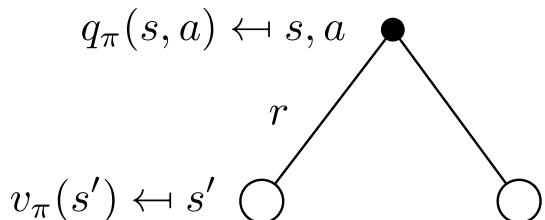


图 1.9: Action Value function 的 Bellman Equation 在 MDP 的解释.

如图1.9中所示:

- 实心黑色圆表示 action, 且在  $state = s, action = a$  能获得的累计收益是  $q_\pi(s, a)$ .

- 环境会根据状态  $s$  和采取的动作  $a$ , 返回一个 reward,  $R_t^a$
  - 同时, 环境会根据状态  $s$  和采取的动作  $a$ , 产生新的 state  $s'$ , 这个概率是  $P_{ss'}^a$ .
  - 这个新的 state  $s'$  的期望累计收益会是  $v_\pi(s')$
- 于是, 我们可以进一步化简  $q_\pi(s, a)$ .

$$\begin{aligned} q_\pi(s, a) &= E_\pi[R_t + \gamma \cdot q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \\ &= R_t^a + \gamma \cdot \sum_{s' \in S} P_{ss'}^a \cdot v_\pi(s') \end{aligned} \quad (1.21)$$

我们将上面两个部分合起来一起考虑, 也是为了消除式子1.20中的  $q_\pi(s, a)$ , 产生递推式子.

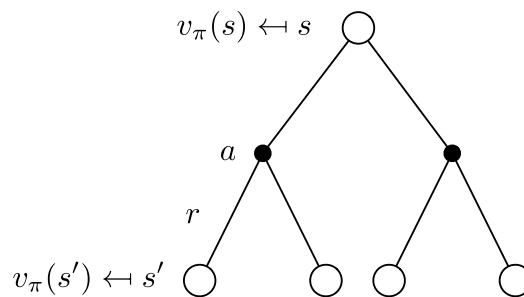


图 1.10: State Value function 的 Bellman Equation 在 MDP 的解释.

如图1.10中所示:

- 我们从某个 state 出发, 以  $\pi(a|s)$  的概率采取不同的 action.
- 黑色实心点表示采取的不同的 action.
- 环境会根据状态  $s$  和采取的动作  $a$ , 返回一个 reward,  $R_t^a$
- 环境会根据当前的 state 和采取的 action, 生成新的 state  $s'$ , 这个概率是  $P_{ss'}^a$ .
- 从这个新的 state  $s'$  开始的期望累计收益为  $v^\pi(s')$ .

我们把上面描述的用数学表达式写出来.

$$\begin{aligned} v_\pi(s) &= E_\pi[R_t + \gamma \cdot v_\pi(S_{t+1}) | S_t = s] \\ &= \sum_{a \in A} \pi(a|s) \cdot q_\pi(s, a) \\ &= \sum_{a \in A} \pi(a|s) \cdot (R_s^a + \gamma \cdot \sum_{s' \in S} P_{ss'}^a \cdot v_\pi(s')) \end{aligned} \quad (1.22)$$

对于  $q_\pi(s, a)$ , 我们使用同样的方式进行分解. 这里可以消除式子1.21中的  $v_\pi(s')$ .

如图1.11中所示:

- 我们从某个 action 出发, 以  $P_{ss'}^a$  的达到新的 state  $s'$ .
- 环境会根据状态  $s$  和采取的动作  $a$ , 返回一个 reward,  $R_t^a$

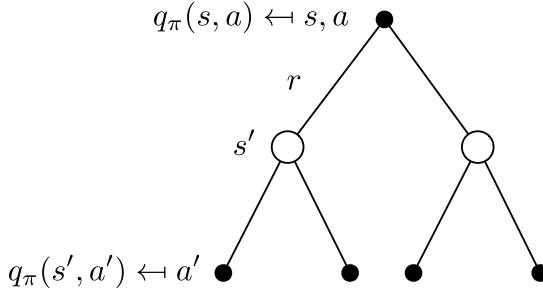


图 1.11: Action Value function 的 Bellman Equation 在 MDP 的解释.

- 白色空心点表示可能会达到的不同的新的 state  $s'$ .
- 接着, 在新的 state, 会有  $\pi(a|s')$  的概率采取动作.
- 在新的动作  $a'$  时, 此时的收益为  $q_\pi(s', a')$ .

我们把上面描述的用数学表达式写出来.

$$\begin{aligned}
 q_\pi(s, a) &= E_\pi[R_t + \gamma \cdot q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \\
 &= R_s^a + \gamma \cdot \sum_{s' \in S} P_{ss'}^a \cdot v^\pi(s') \\
 &= R_s^a + \gamma \cdot \sum_{s' \in S} P_{ss'}^a \cdot \sum_{a' \in A} \pi(a'|s') \cdot q_\pi(s', a')
 \end{aligned} \tag{1.23}$$

### 1.3.4 Optimal Value Function

接下来, 我们讲一下我们最关心的问题, 如何找到在 MDP 中最好的一个策略. 那么什么是最好的策略, 最好的策略就是可以让我们的累计的 reward 获得最大的策略. 我们可以按照下面方式来定义 policy 的好坏.

#### 定义 1.9. Ordering over Policies

对于两个 policy 的好坏:

$$\pi > \pi', \text{ if } \forall s, v_\pi(s) \geq v_{\pi'}(s) \tag{1.24}$$

接下来, 我们定义两个 optimal value function.

#### 定义 1.10. Optimal State Value Function

Optimal state value function  $v_*(s)$ , 是在从状态  $s$  开始, 不同的 policy 中可以获得最大 reward 的:

$$v_*(s) = \max_\pi v_\pi(s). \tag{1.25}$$

### 定义 1.11. Optimal Action Value Function

Optimal action value function  $q_*(s, a)$  是在状态  $s$  开始, 进行一步动作  $a$ , 可以获得的最大的累计奖励:

$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a). \quad (1.26)$$

如果我们知道了  $q_*(s, a)$ , 我们就可以知道每一步的 action. 这是因为, 如果我们知道了  $q_*(s, a)$ , 那么  $a = \arg \max_a q_*(s, a)$  就是我们在状态  $s$  要进行的 action. 我们将其使用数学式子表达出来.

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg \max_a q_*(s, a) \\ 0 & \text{otherwise} \end{cases} \quad (1.27)$$

接下来, 我们还是试着将  $v_*(s)$  和  $q_*(s, a)$  写成递推关系是, 也就是写成上面 Bellman Equation 的样子, 这里叫做 Bellman Optimality Equation.

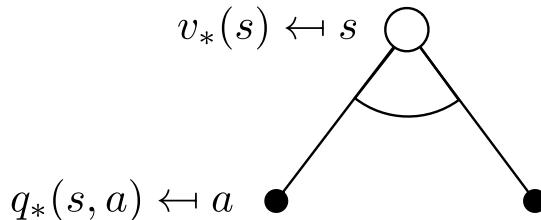


图 1.12: Optimal State Value function 的 Bellman Equation 在 MDP 的解释.

从当前的 state 出发, 会执行不同的 action, 但是  $v_*(s)$  就是最大的  $q_*(s, a)$ , 也就是:

$$v_*(s) = \max_a q_*(s, a). \quad (1.28)$$

对于  $q_*(s, a)$ , 我们同样进行分解.

在这里, 我们从一个 action 出发, 看会有何种 state 发生. 但是在这种新的 state 是不受我们控制的, 所以我们就按照出现的概率将其进行累加 (上面的 action 我们是可以进行选择的, 所以取最大值, 这里是受到环境影响, 我们取平均).

$$q_*(s, a) = R_s^a + \gamma \cdot \sum_{s' \in S} P_{ss'}^a v_*(s'). \quad (1.29)$$

我们还是将上面两部分放在一起, 这个时候我们就可以获得  $v_*(s)$  和  $q_*(s, a)$

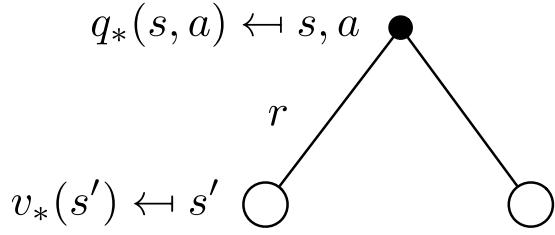


图 1.13: Optimal Action Value function 的 Bellman Equation 在 MDP 的解释.

的递推关系.

$$\begin{aligned} v_*(s) &= \max_a q_*(s, a). \\ &= \max_a R_s^a + \gamma \cdot \sum_{s' \in S} P_{ss'}^a v_*(s') \end{aligned} \quad (1.30)$$

上面的式子说明, 要想求得  $v_*(s)$ , 从状态  $s$  开始的累计最优奖励, 我们只需要使得其下一状态  $s'$  是最优的. 也就是我们是可以将这个问题拆分成小问题进行解决的.

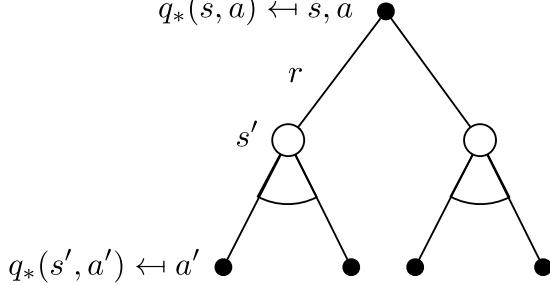


图 1.14: Optimal Action Value function 的 Bellman Equation 在 MDP 的解释.

$$q_*(s, a) = R_s^a + \gamma \cdot \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s', a'). \quad (1.31)$$

对于求解 Bellman Optimality Equation, 我们无法直接进行求解 (没有一个显示的公式), 因为他是非线性问题. 但是有下面的一些迭代求解的方式, 之后介绍的方法就是迭代求解的:

- Value Iteration
- Policy Iteration
- Q-learning
- Sarsa

# 第 2 章 Planning by Dynamic Programming

## 内容提要

Policy Iteration

Value Iteration

这一部分，我们会介绍当我们对环境的信息全部已知的时候，如何使用动态规划的思想解决 Bellman Optimality Equation. 在具体进行数码之前，我们会从递归开始，对动态规划做一个简单的介绍。

## 2.1 动态规划介绍

### 2.1.1 递归的思想 (recursion)--预备知识

递归就是函数自己调用自己。用一个比较经典的电影 < 盗梦空间 > 来说明就是：

- 我们可以从一个梦境进入下一个梦境，直到进入最后一个 (递归的结束条件)；
- 接着我们从最后一个梦境向前面的梦境返回，返回的时候通常会对从上一个梦境带来的内容进行操作 (递归的时候的数据处理，所以数据处理的步骤要写在函数调用自己的后面)。
- 最后就是走出梦境 (结束递归)

总的过程如下图所示，先下到最后的梦境，再逐层返回，返回的过程中对数据处理。

下面我们来看一个具体的递归的例子。

**例 2.1** 例如我们要计算一个数的 N 次方，也就是计算  $x^n$ 。如果我们直接乘 N 次，那么时间复杂度是  $O(n)$ 。

但是，我们这里使用递归的思想，也就是将  $x^n = x^{\frac{n}{2}} \cdot x^{\frac{n}{2}}$ 。实际需要考虑 n 的奇偶，完整的递推关系如式子 2.1 所示：

$$pow(x, n) = \begin{cases} pow(x, n//2)^2 & \text{if } n \% 2 == 0 \\ pow(x, n//2)^2 \cdot x & \text{if } n \% 2 == 1 \end{cases} \quad (2.1)$$

例如  $2^7$  就是  $2^7 = (2^3) * (2^3) * 2$ 。也就是对应上面  $n \% 2 == 1$  (n 为基数) 的情况。这个时候时间复杂度为  $O(\log_2(n))$ 。

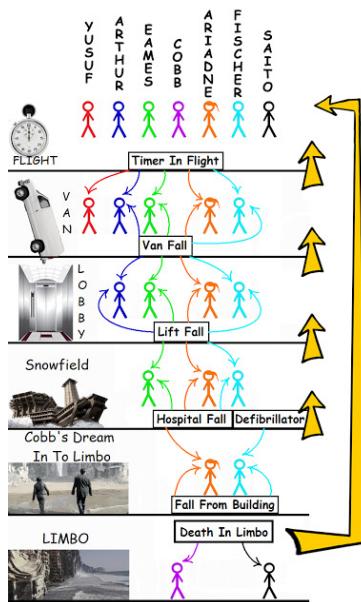


图 2.1: 电影 &lt; 盗梦空间 &gt;.

下面是具体解决的时候的代码.

```

class Solution(object):
    def showTrace(self, level, n, a):
        """打印递归过程函数
        """
        print('='*level, end=' ')
        print(">level:{} , 2^{}, Result:{}".format(level, n, a))

    def myPow(self, x: float, n: int, level=0) -> float:
        # 终止条件
        if n == 0:
            self.showTrace(level, n, a=1)
            return 1
        elif n == -1:
            self.showTrace(level, n, a=1/x)
            return 1/x

        # 调用递归
        a = self.myPow(x=x, n=n//2, level=level+1)

        # 进行处理
        if n % 2 == 0:

```

```

        a = a * a
        self.showTrace(level, n, a=a)
    elif n % 2 == 1:
        a = a * a * x
        self.showTrace(level, n, a=a)
    return a

if __name__ == "__main__":
    sol = Solution()
    sol.myPow(x=2, n=17)

```

最终的显示结果如图所示:

```

=====> level:5, 2^0, Result:1
=====> level:4, 2^1, Result:2
====> level:3, 2^2, Result:4
==> level:2, 2^4, Result:16
=> level:1, 2^8, Result:256
> level:0, 2^17, Result:131072

```

图 2.2: 递归流程显示.

可以看到, 为了计算  $2^{17}$ :

- 首先拆分为  $(2^8) * (2^8) * 2$ .
- 接着计算  $2^8$ , 拆分为  $(2^4) * (2^4)$
- 接着计算  $2^4$ , 拆分为  $(2^2) * (2^2)$
- 接着计算  $2^2$ , 拆分为  $(2^1) * (2^1)$
- 接着计算  $2^1$ , 拆分为  $(2^0) * (2^0) * 2$ .
- 有了  $2^1$ , 就可以逐层向上倒推了.

关于更多关于递归和分治的内容, 可以查看文章, 数据结构与算法基础-递归与分治<sup>1</sup>.

## 2.1.2 动态规划的思想--预备知识

动态规划能解决的问题, 需要拥有**最优子结构**. 也就是, 可以将一个大的问题, 分解成两个或是多个小问题. 得到这些小问题的最优解, 也就解决了最后大问题的最优解.

但是有的时候, 动态规划的 dp 方程不是很好想, 我自己的做法是先从递推开始想起, 加上存储, 然后将递归顺序倒过来 (这个是我看 leetcode 上算法题的一些经验). 总结一下也就是按照下面的顺序进行思考动态规划的问题:

<sup>1</sup><https://mathpretty.com/12162.html>

- 使用递推的方式进行推导.
- 递推 + 存储中间过程.
- 将递推的顺序倒过来, 从初始状态开始思考.

我们看下面的一个例子, 如图2.3所示: 要求就是, 我们要计算从起点 (**start**) 开始, 到终点 (**end**), 一共有多少种不同的走法. 其中有红色涂色的地方表示是不可以通行的. 且我们只能向右或是向下行走

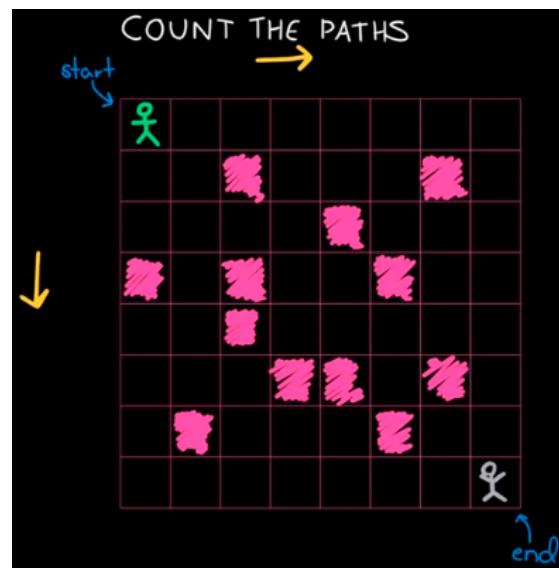


图 2.3: 动态规划例子.

我们首先用递推的想法去考虑这个问题. 要计算从起点到终点的路径的总个数, 可以将问题转换为计算从 **A** 到终点的路径总和加上从 **B** 到终点的路径总和. 也就是如图2.4所示:

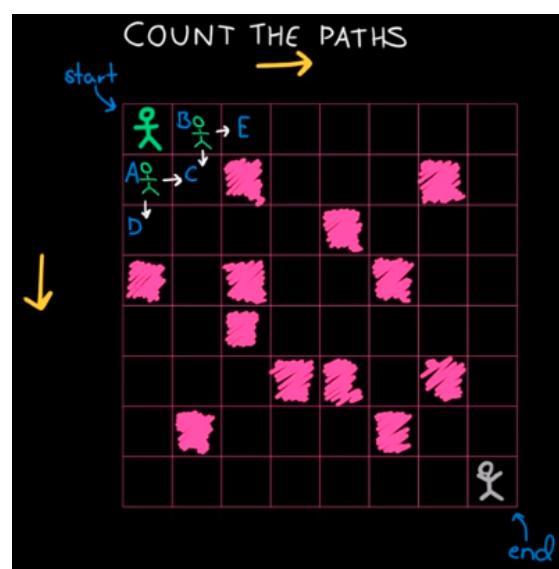


图 2.4: 动态规划例子-递推想法.

也就是  $\text{paths}(\text{start}, \text{end}) = \text{paths}(A, \text{end}) + \text{paths}(B, \text{end})$ . 我们用同样的想法

计算  $\text{paths}(A, \text{end})$  和  $\text{paths}(B, \text{end})$ . 也就是可以用下图2.5进行表示.

$$\begin{aligned} \text{paths}(\text{start}, \text{end}) &= \\ \text{paths}(A, \text{end}) + \text{paths}(B, \text{end}) & \\ \underbrace{\text{paths}(D, \text{end}) + \text{paths}(C, \text{end})}_{\Downarrow} & \quad \underbrace{\text{paths}(C, \text{end}) + \text{paths}(E, \text{end})}_{\Downarrow} \end{aligned}$$

图 2.5: 动态规划例子-递推公式.

到这里就可以用递推的方法解决上面的问题. 但是我们看上面的递推公式, 发现  $\text{paths}(c, \text{end})$  被重复计算了, 于是我们考虑在计算的过程中加上存储, 将这些值保存下来.

最后, 我们把上面的问题从终点开始思考, 结合上面的递推公式. 设整个格子用变量  $\text{checkerboard}$  表示, 为一个  $8 \times 8$  的向量.

- 初始状态

- 初始状态为  $\text{checkerboard}[7, 7] = 1, \text{checkerboard}[7, 6] = 1, \text{checkerboard}[6, 7] = 1.$

- dp 方程

- $\text{checkerboard}[i, j] = \text{checkerboard}[i + 1, j] + \text{checkerboard}[i, j + 1].$
- $\text{checkerboard}[i, j] = 0$ , 如果  $(i, j)$  是不能走的地方.
- $\text{checkerboard}[i, j] = 0$ , 如果  $i > 7$  或是  $j > 7$ (超出棋盘).

上面只是大概讲了一下思想, 关于更多动态规划的内容(详细的代码), 可以查看下面两个链接:

- 数据结构与算法基础-动态规划<sup>2</sup>.
- 数据结构与算法基础-动态规划习题<sup>3</sup>.

### 2.1.3 动态规划解 MDP 问题

使用 dynamic programming 来解决 MDP 问题, 我们假设我们完全了解 MDP 的结构. 也就是, 我们了解 MDP 的内部奖励和状态转移, 即我们了解这个五元组的所有信息  $\langle S, A, P, R, \gamma \rangle$ . 但是在 reinforcement learning 中我们可能不能获得这些所有的信息.

在这里, 我们会分为两个问题来进行介绍, 分别是 prediction 和 control.

- 对于 prediction 的问题, 我们是已知  $\langle S, A, P, R, \gamma \rangle$  和策略  $\pi$ , 需要求解 value function  $v_\pi$ . 也就是想要知道这个  $\pi$  怎么样, 能获得多少的奖励.

<sup>2</sup><https://mathpretty.com/12166.html>

<sup>3</sup><https://mathpretty.com/12174.html>

- 对于 control 问题, 我们已知  $\langle S, A, P, R, \gamma \rangle$ , 但我们不知道 policy  $\pi$ , 现在想知道  $v_*$ , 也就是找到最好的一个策略  $\pi$ .

## 2.2 Policy Evaluation

这部分要讨论的就是, 当我们知道了 MDP, 知道了 policy, 我们要衡量这个 policy 的好坏.

我们要衡量一个 policy 的好坏, 也就是要计算得到  $v_\pi(s)$ . 我们可以使用上一章推导出来的式1.22, 也就是下面这个式子进行迭代.

这里有一个问题, 式子2.2是课件中的式子, 式子2.3是我认为正确的式子. 我认为 reward 不仅和采取的 action 有关, 还与下一个 state 有关, 也就是图2.6中  $r$  所在的位置.

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \cdot (R_s^a + \gamma \cdot \sum_{s' \in S} P_{ss'}^a \cdot v_\pi(s')) \quad (2.2)$$

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P_{ss'}^a \cdot (R_{ss'}^a + \gamma \cdot v_\pi^0(s)) \quad (2.3)$$

也就是, 我们初始化一个  $v_1$ , 可能就是一个全 0 的向量. 我们通过上面式子2.2进行迭代 (因为对 MDP 是全部已知的, 所以知道  $P_{ss'}^a$ ), 生成  $v_1 \rightarrow v_2 \rightarrow v_3 \dots \rightarrow v_\pi$  (最终可以收敛).

在每一轮, 我们对于所有的状态  $s \in S$ , 都进行更新  $v_{k+1}(s) = v_k(s')$ , 其中  $s'$  是  $s$  的下一步状态. 我们结合下图2.6进行说明.

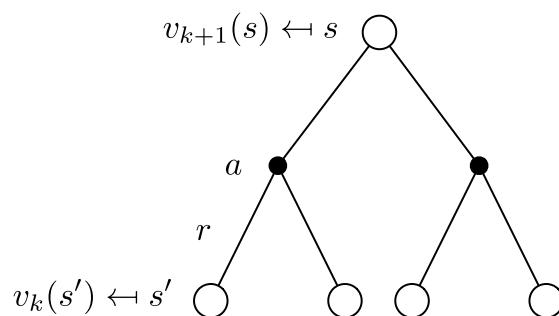


图 2.6: 对于 Policy Evaluation 的解释.

如图2.6所示:

- 目前是第  $k$  轮迭代,  $s$  表示目前要更新的状态, 更新之前的 value 是  $v_k(s)$ .
- 因为要更新  $s$ , 我们需要用到  $s$  采取 action 之后可能达到的新的状态  $s'$ .
- $v_k(s')$  表示我们使用状态  $s'$  的上一个 value 在这里进行计算, 生成状态  $s$  的新的 value,  $v_{k+1}(s)$ .

下面说一下这样进行迭代为什么能最后收敛, 也就是通过上面式子2.2迭代之后,  $v_i$  可以最终收敛到  $v_\pi$ .

### 2.2.1 关于收敛的说明

要证明收敛性, 需要用到 Contraction Mapping Theorem(收缩映射定理), 这里好像只说明了为什么收敛, 没说一定收敛到  $v_\pi(s)$ .

#### 定理 2.1. Contraction Mapping Theorem

设空间  $V$  是完备度量空间. 设  $T : V \rightarrow V$  是一个压缩映射, 也就是存在一个非负实数,  $\gamma < 1$ , 使得  $distance(T(x), T(y)) \leq \gamma \cdot distance(x, y)$ .

那么映射  $T$  在空间  $V$  中只有一个不动点,  $x^*$ , 也就是说  $T(x^*) = x^*$ . 更进一步, 这个不动点可以用以下的方法来求出:

- 从  $V$  内的任意一个元素  $x_0$  开始
- 定义一个迭代序列  $x_n = T(x_{n-1})$ , 其中  $n = 1, 2, 3, \dots$
- 那么, 这个序列收敛, 极限为  $x^*$ .



我们将上面的式2.2写成矩阵形式, 如式2.7所示:

$$v_\pi = R^\pi + \gamma P^\pi v_\pi \quad (2.4)$$

于是, 我们可以定义一个映射,  $T^\pi$ , 有式2.5:

$$T^\pi(v) = R^\pi + \gamma P^\pi v \quad (2.5)$$

下面只需要证明映射  $T^\pi$  在空间  $v$ (value function) 中是一个压缩映射即可. 于是我们任取两个向量  $u$  和  $v$ :

$$\begin{aligned} \|T^\pi(v) - T^\pi(u)\|_\infty &= \|(R^\pi + \gamma P^\pi v) - (R^\pi + \gamma P^\pi u)\|_\infty \\ &= \|\gamma P^\pi(v - u)\|_\infty \\ &\leq \|\gamma P^\pi\| \|v - u\|_\infty \\ &\leq \gamma \|v - u\|_\infty \end{aligned} \quad (2.6)$$

其中,  $\|v - u\|_\infty = \max_{s \in S} |v(s) - u(s)|$ .



## 2.2.2 Policy Evaluation 的例子

我们在这里做一个简单的例子, 来实现对一个策略  $\pi$  的评价. 实验的代码可以查看链接, 强化学习 (Reinforcement Learning) 代码笔记<sup>4</sup>.

我们在这里使用 Gym 提供的环境, FrozenLake-v0. 环境如下所示:

SFFF	(S: starting point, safe)
FHFH	(F: frozen surface, safe)
FFFF	(H: hole, fall to your doom)
HFFG	(G: goal, where the frisbee is located)

对上面环境的一个简单的介绍:

- 我们的任务就是从 **start point** 开始到 **goal**. 也就是上面从 **S** 到 **G**.
- F 表示可以行走; H 表示不能行走的, 走入 H 游戏结束.
- 因为 F 是结冰的冰面, 所以我们走的方向与实际的方向可能会不一样, 系统有  $\frac{1}{3}$  的概率往别的方向走 (环境有随机性).
- 为了使得游戏尽快结束, 每走一步的  $reward = -1$ , 到达终点则  $reward = 10$ .

我们对上面的环境进行一个抽象的表示, 如图2.7所示, 其中

- 每一个数字表示格子的标号.
- 红色的数字表示此处为 hole.
- 我们要从数字 0 到数字 15.

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

图 2.7: FrozenLake-v0 示意图.

这里我们想要对一个随机 **policy** 进行评价 ( $\pi$  如式子2.7所示), 也就是向每一个方向走的概率都是一样的.

$$\pi(up|\cdot) = \pi(down|\cdot) = \pi(left|\cdot) = \pi(right|\cdot) = 0.25 \quad (2.7)$$

于是就可以将上面的 policy 用表格来进行表示, 如图2.8所示. 每一行表示一个 state, 一行的四个值表示在这个 state 往四个方向走的概率.

<sup>4</sup>[https://github.com/wmn7/ML\\_Practice/tree/master/2020\\_04\\_13](https://github.com/wmn7/ML_Practice/tree/master/2020_04_13)

	P(L)	P(D)	P(R)	P(U)
State 0	0.25	0.25	0.25	0.25
State 1	0.25	0.25	0.25	0.25
⋮	⋮	⋮	⋮	⋮
State 15	0.25	0.25	0.25	0.25

图 2.8: 每一个 state 执行 action 的概率表格.

我们使用 numpy 生成来生成数组来存储 policy.

```
import gym
import numpy as np

environment = gym.make('FrozenLake-v0')
policy = np.ones((environment.nS, environment.nA)) / environment.nA
```

最终的 policy 会如图2.9所示:

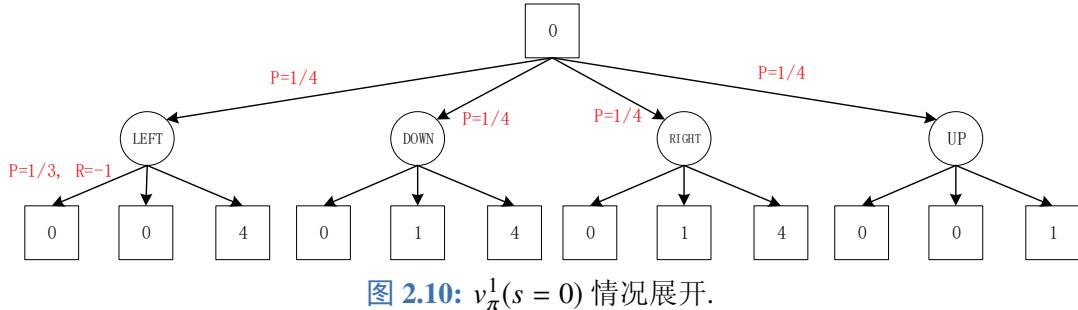
```
array([[0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25],
       [0.25, 0.25, 0.25, 0.25]])
```

图 2.9: 每一个 state 执行 action 的概率表格-Array.

接下来就是按照上面式2.2进行迭代. 我们来看一个具体计算的例子, 我们初始化的 value 为全 0 矩阵, 且  $\gamma = 0.9$  (不过因为初始为 0, 所以不影响).

我们来看一下第一轮的情况下,  $v_\pi^1(s = 0)$  的值是如何计算的. 在  $s = 0$  的情况下, 可以执行 4 种不同的动作, 每一个动作可以到不同的新的状态, 整体状态如图2.10所示:

于是,  $v_\pi^1(s = 0)$  的计算如式子2.8所示.



$$\begin{aligned}
 v_\pi^1(s = 0) &= \sum_{a \in A} \pi(a|s = 0) \sum_{s' \in S} P_{ss'}^a \cdot (R_{ss'}^a + \gamma \cdot v_\pi^0(s)) \\
 &= \frac{1}{4} \cdot \frac{1}{3} \cdot [(-1 + 0.9 * 0) + (-1 + 0.9 * 0) + (-1 + 0.9 * 0)] \\
 &\quad + \frac{1}{4} \cdot \frac{1}{3} \cdot [(-1 + 0.9 * 0) + (-1 + 0.9 * 0) + (-1 + 0.9 * 0)] \\
 &\quad + \frac{1}{4} \cdot \frac{1}{3} \cdot [(-1 + 0.9 * 0) + (-1 + 0.9 * 0) + (-1 + 0.9 * 0)] \\
 &\quad + \frac{1}{4} \cdot \frac{1}{3} \cdot [(-1 + 0.9 * 0) + (-1 + 0.9 * 0) + (-1 + 0.9 * 0)] \\
 &= -1
 \end{aligned} \tag{2.8}$$

下面, 我们分别画出迭代 10 次, 20 次, 199 次后的每一个 state 对应的 value, 如图2.11所示.

```

Evaluation iterations:10
[[-6.4507704 -6.49730403 -6.56394722 -6.51457622]
 [-6.43929532 -6.12579511 -6.30160344 -6.12579511]
 [-6.36727564 -6.09779259 -5.54196666 -6.12579511]
 [-6.12579511 -5.34285003 -2.45118631 -6.12579511]]

Evaluation iterations:20
[[-8.74838238 -8.75837677 -8.73701108 -8.76067403]
 [-8.7174594 -8.64914828 -8.5234858 -8.64914828]
 [-8.60194468 -8.22714016 -7.6795972 -8.64914828]
 [-8.64914828 -7.44191556 -4.56304673 -8.64914828]]

Policy evaluated in 199 iterations
[[-9.95075012 -9.95355297 -9.88926567 -9.95469959]
 [-9.92605845 -9.99999999 -9.7103292 -9.99999999]
 [-9.79456232 -9.3663229 -8.82330857 -9.99999999]
 [-9.99999999 -8.56578645 -5.69360823 -9.99999999]]

```

图 2.11: 迭代不同次数之后的对  $\pi$  的 value.

我们对图2.11进行简单的解释 (我们就看 199 次迭代之后的每个 state 的 value 值).

- 比如终点左侧的 state 的 value 是 -5.7, 这个表示从这个 state 出发到终点的 reward 是 -5.7.

- 这个意思是可能还需要走 15 步骤才可以走到 (因为到终点会获得 10 的 reward)
- 这是因为我们现在使用的 policy 就是一个完全随机的 policy. 所以走的步数会偏多.

我们画出每次迭代之后, 前后 value 的变化, 如图2.12所示. 我在这里计算的是一轮迭代之后, 当下这个 value 的表格的平均值, 前后的变化.

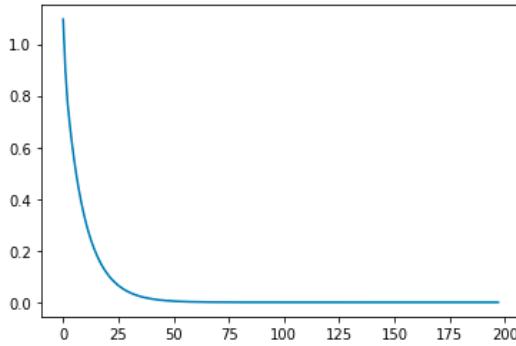


图 2.12: Policy Evaluation 迭代收敛速度.

可以看到, 在迭代 30 轮之后, 基本就对这个 policy 评价完成了, value 的变化后面就不大了.

## 2.3 Policy Iteration

在上面, 我们介绍了如何评价一个 policy 的好坏. 也就是给定任何一个策略  $\pi$ , 我们可以计算得到该策略  $\pi$  在每一个 state  $s$  下的累计期望获利  $v_\pi(s)$ .

下面, 我们来看一下如何对一个策略  $\pi$  进行改进, 使  $\pi$  可以变好 (这里变好指的是对于新的  $\pi$ ,  $v_\pi(s)$  变大). 于是, 我们可以使用下面的方式对  $\pi$  进行优化:

- 给定一个策略  $\pi$ ;
- 衡量一个策略  $\pi$  的好坏, 下面是在  $t$  时刻在状态  $s$  下的累计收益的计算:
  - $v_\pi(s) = E[R_{t+1} + \gamma R_{t+2} + \dots | S_t = s]$ ;
- 改善一个策略, 使用贪婪的方式进行改善,  $\pi' = greedy(v_\pi)$ , 下面具体说明迭代的方式:
  - 新的策略的生成方式:  $\pi'(s) = \arg \max_{a \in A} q_\pi(s, a)$
  - 上面式子的解释: 也就是计算从当前状态  $s$ , 采取下一步动作  $a$  之后会获得的累计收益, 从中选择一个最大的动作  $a$  进行执行;
  - $q_\pi(s, a)$  的计算, 参考图2.13:  $q_\pi(s, a) = R_t^a + \gamma \cdot \sum_{s' \in S} P_{ss'}^a \cdot v_\pi(s')$ , 这里有困惑, 感觉应该写成下面这个样子,  $q_\pi(s, a) = \sum_{s' \in S} P_{ss'}^a \cdot (R_{ss'}^a + \gamma \cdot v_\pi(s'))$ ;
  - 在我们当前的条件下,  $P_{ss'}^a$  是已知的;  $v_\pi(s')$  在我们进行策略估计的时候计算出来了;

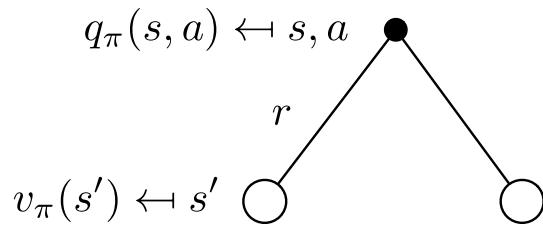


图 2.13: Action Value function 的化简.

### 2.3.1 Policy Iteration 的例子

下面我们来看一个例子, 上面的步骤是如何进行迭代的.

**例 2.2** 在这里我们还是使用 FrozenLake-v0 的例子, 即图2.14的  $4 * 4$  的格子. 其中, 每一个数字表示格子的标号, 红色的数字表示此处为 hole, 我们要从数字 0 到数字 15.

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

图 2.14: FrozenLake-v0 示意图.

我们从随机策略开始, 也就是此时的策略如2.9所示:

$$\pi(up|\cdot) = \pi(down|\cdot) = \pi(left|\cdot) = \pi(right|\cdot) = 0.25 \quad (2.9)$$

我们使用上面介绍的 Policy Evaluation, 计算出此策略的  $v_\pi$ , 如图2.15所示, 左侧为每一个格子的编号, 右边为在那个格子时的  $v_\pi(s)$ , 最后一个格子的  $value = -9.99$ , 出现这个情况是因为最后是吸收状态, 一直自己到自己:

				$V$
0	1	2	3	-9. 95
4	5	6	7	-9. 93
8	9	10	11	-9. 79
12	13	14	15	-9. 99
				-8. 57
				-5. 69
				-9. 99

图 2.15: FrozenLake-v0 中策略的  $v_\pi$ .

现在假设我们在格子 14, 也就是终点左侧的一个格子, 我们看一下如何使用上面的贪婪算法更新策略, 即得到  $\pi(s = 14)$ .

在格子 14, 我们有 4 种不同的 action, 每一种 action 会出现不同的新的 state, 我们可以使用下面的命令查看每一种环境出现的概率. 下面表示从 state=14, 向左走, 分别都会走到格子 10, 13, 14.

```
# LEFT = 0, DOWN = 1, RIGHT = 2, UP = 3
environment.P[14][0]
>>
"""
[(0.3333333333333333, 10, 0.0, False),
 (0.3333333333333333, 13, 0.0, False),
 (0.3333333333333333, 14, 0.0, False)]
"""

```

于是, 在原有的 state, 执行不同的动作, 会到达不同的新的 state, 总的状态如下图2.16所示:

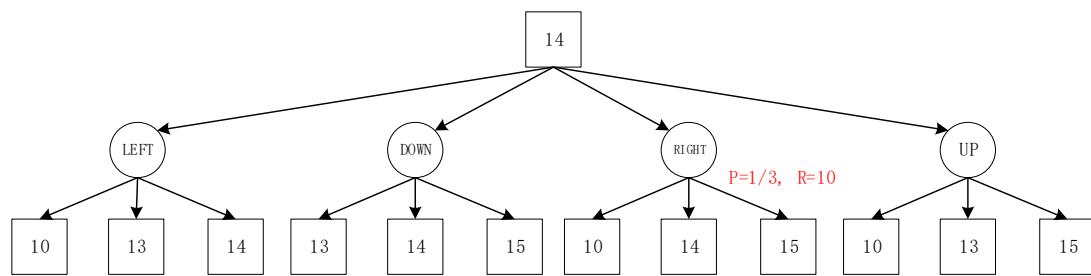


图 2.16: FrozenLake-v0 中 state 14 的所有下一步的可能性.

在上一图中, 执行一个 action 到下一个 state 的概率都是  $\frac{1}{3}$ . 此时, 我们执行贪

心算法, 计算  $q_\pi(s, a)$ , 如式子2.10所示:

$$\begin{cases} q_\pi(s = 14, a = 'LEFT') = \frac{1}{3} \cdot (-1 + 1 * -8.82) + \frac{1}{3} \cdot (-1 + 1 * -8.57) + \frac{1}{3} \cdot (-1 + 1 * -5.69) \\ q_\pi(s = 14, a = 'DOWN') = \frac{1}{3} \cdot (-1 + 1 * -8.57) + \frac{1}{3} \cdot (-1 + 1 * -5.69) + \frac{1}{3} \cdot (10 + 1 * -9.99) \\ q_\pi(s = 14, a = 'RIGHT') = \frac{1}{3} \cdot (-1 + 1 * -8.82) + \frac{1}{3} \cdot (-1 + 1 * -5.69) + \frac{1}{3} \cdot (10 + 1 * -9.99) \\ q_\pi(s = 14, a = 'UP') = \frac{1}{3} \cdot (-1 + 1 * -8.82) + \frac{1}{3} \cdot (-1 + 1 * -8.57) + \frac{1}{3} \cdot (10 + 1 * -9.99) \end{cases} \quad (2.10)$$

最终计算的结果如式2.11所示:

$$\begin{cases} q_\pi(s = 14, a = 'LEFT') = -8.69 \\ q_\pi(s = 14, a = 'DOWN') = -5.42 \\ q_\pi(s = 14, a = 'RIGHT') = -5.5 \\ q_\pi(s = 14, a = 'UP') = -6.46 \end{cases} \quad (2.11)$$

按照  $\pi'(s) = \arg \max_{a \in A} q_\pi(s, a)$ , 我们应该选择上面最大的, 即向'DOWN'走. 在这里, 我们一旦使用了 **Greedy Exploration**, 之后的动作就会固定, 所以之后我们会介绍  $\epsilon$ -**Greedy Exploration**, 他可以保证其他的 action 也有出现的机会.

这个和我们的直觉是不一样的, 我们直觉可能认为向左走是最快的, 但是因为环境存在随机性, 向下走才是最稳的.

## 2.4 Value Iteration

在 value iteration 的部分, 我们使用式子2.12来进行迭代:

$$v_*(s) = \max_{a \in A} R_s^a + \gamma \cdot \sum_{s' \in S} P_{ss'}^a v_*(s') \quad (2.12)$$

下面我们来看一下 value iteration 的具体的步骤, 我们的目标仍然是找到一个最优的 policy (find optimal policy  $\pi$ ):

- 我们从一个任意的  $v_1(s)$  开始.
- 使用式子  $v_*(s) = \max_{a \in A} R_s^a + \gamma \cdot \sum_{s' \in S} P_{ss'}^a v_*(s')$  进行迭代, 这个过程如图2.17所示.
- 最终,  $v_1(s), \dots, v_n(s)$  会收敛.

下面我们来看一下 **value iteration** 与 **policy iteration** 的区别:

- 在 **Policy Iteration** 的时候, 我们每一步都有一个 policy, 通过 policy  $\pi$  来计算  $v_\pi(s)$ ;
- 在 **Value Iteration** 的时候, 我们直接对  $v_n(s)$  进行迭代, 这个  $v_n(s)$  可能没有一个 policy  $\pi$  与之对应; 但是在迭代的最后, 我们获得了最优的 policy 的

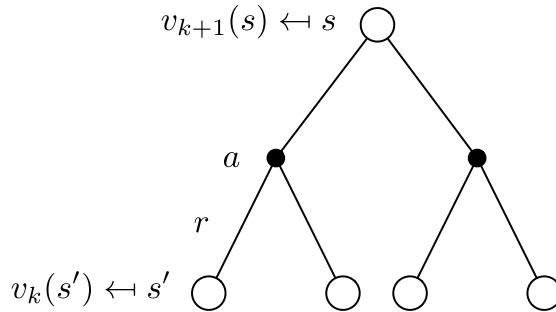


图 2.17: 对于 Value Iteration 的解释.

 $v(s)$ ;

### 2.4.1 Value Iteration 的例子

接下来我们看一个 value iteration 的例子. 在这里, 会给出一个具体的对 value 进行更新的例子. 同时有, 若我们知道了最优的 value, 如何求出最优的 policy, 即指导我们行动.

#### 例 2.3

我们还是使用 FrozenLake-v0 的例子, 图2.18是迭代第 6 次和第 7 次的时候的 value 的值. 我们看一下  $s = 0$  处 (红字加粗的), 是如何从  $-4.11$  变为  $-4.70$  的.

Step:6				Step:7							
0	1	2	3	<b>-4.11</b>	-4.13	-4.07	-4.10	<b>-4.70</b>	-4.69	-4.60	-4.69
4	<b>5</b>	6	<b>7</b>	-4.12	-4.10	-3.84	-4.10	-4.65	-4.69	-4.32	-4.69
8	9	10	<b>11</b>	-3.94	-3.22	-2.36	-4.10	-4.38	-3.56	-2.77	-4.69
<b>12</b>	13	14	15	-4.10	-1.78	1.74	-4.10	-4.69	-2.08	1.34	-4.69

图 2.18: Value Iteration 迭代第 6 和第 7 次的结果.

因为  $v_n(s) = \max_{a \in A} \sum_{s' \in S} P_{ss'}^a (R_{ss'}^a + \gamma \cdot v_{n-1}(s))$ , 即  $v_n(s) = \max_{a \in A} q_n(s, a)$ . 对于  $s = 0$  来说, 执行不同的动作会有下图2.19的可能性:

于是,  $q_\pi(s, a)$  的计算如式子2.13所示:

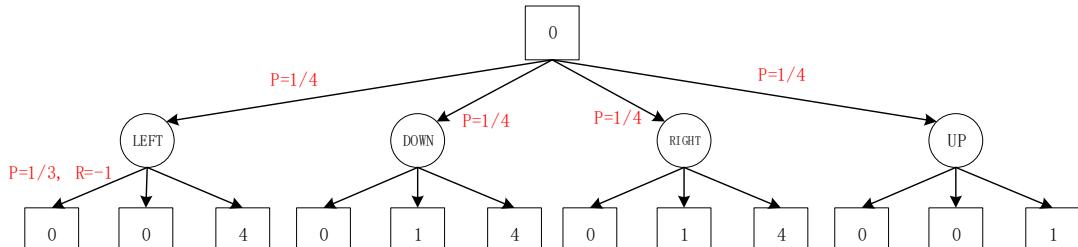


图 2.19: FrozenLake-v0 中 state 0 的所有下一步的可能性.

$$\begin{cases} q_\pi(s = 0, a = 'LEFT') = \frac{1}{3} \cdot (-1 + 0.9 * -4.11) + \frac{1}{3} \cdot (-1 + 0.9 * -4.11) + \frac{1}{3} \cdot (-1 + 0.9 * -4.12) \\ q_\pi(s = 0, a = 'DOWN') = \frac{1}{3} \cdot (-1 + 0.9 * -4.11) + \frac{1}{3} \cdot (-1 + 0.9 * -4.13) + \frac{1}{3} \cdot (-1 + 0.9 * -4.12) \\ q_\pi(s = 0, a = 'RIGHT') = \frac{1}{3} \cdot (-1 + 0.9 * -4.11) + \frac{1}{3} \cdot (-1 + 0.9 * -4.13) + \frac{1}{3} \cdot (-1 + 0.9 * -4.12) \\ q_\pi(s = 0, a = 'UP') = \frac{1}{3} \cdot (-1 + 0.9 * -4.11) + \frac{1}{3} \cdot (-1 + 0.9 * -4.11) + \frac{1}{3} \cdot (-1 + 0.9 * -4.13) \end{cases} \quad (2.13)$$

式子2.13的计算结果如式子2.14所示:

$$\begin{cases} q_\pi(s = 0, a = 'LEFT') = -4.702 \\ q_\pi(s = 0, a = 'DOWN') = -4.708 \\ q_\pi(s = 0, a = 'RIGHT') = -4.708 \\ q_\pi(s = 0, a = 'UP') = -4.705 \end{cases} \quad (2.14)$$

我们从上面式子2.14中选取最大值更新  $V_7(s = 0) = -4.702$ , 这个就是从  $-4.11$  变为  $-4.70$  的计算步骤.

接着, 我们来看一下如何根据计算得到的  $v(s)$  得到最后的 policy. 我们根据式子2.15来得到最终的策略.

$$\pi(s) = \arg \max_{a \in A} P_{ss'}^a (R_{ss'}^a + \gamma V(s')) \quad (2.15)$$

我们会依次扫描每一个 state, 查看采取不同的动作, 到达不同的 next state  $s'$ . 关于具体的计算的例子, 可以查看 github 仓库中的代码<sup>5</sup>.

<sup>5</sup>[https://github.com/wmn7/ML\\_Practice/blob/master/2020\\_04\\_13/03\\_Example\\_of\\_Value\\_Iteration.ipynb](https://github.com/wmn7/ML_Practice/blob/master/2020_04_13/03_Example_of_Value_Iteration.ipynb)

# 第 3 章 Model-Free Prediction

## 内容提要

- ❑ Monte-Carlo Learning
- ❑ TD( $\lambda$ )
- ❑ Temporal-Difference Learning

在上一章的时候, 我们的假设是我们完全了解 MDP 的结构, 了解内部的状态转移, 来求解 value function  $v_\pi$ . 在这一章的时候, 我们假设我们对 MDP 的内部是未知的, 需要来估计给定一个策略  $\pi$ , 来估计他的 value function  $v_\pi$ . 会分为三个方法进行介绍:

- Monte-Carlo Learning
- Temporal-Difference Learning
- TD( $\lambda$ )

## 3.1 Monte-Carlo Learning

### 3.1.1 MC 方法总体介绍

蒙特卡洛学习的思想很简单, 现在我们要对给定一个策略  $\pi$ , 估计他的  $v_\pi$ . 那么我们只需要使用该策略  $\pi$  进行多次完整的实验, 这时候对于每一个 state 我们都有此时对应获得的 reward, 就可以根据这个计算出  $v_\pi$ . 下面是 MC 方法的一些特点:

- MC 方法是 model-free 的, 不需要了解 MDP 中的转移和 reward.
- MC 方法需要完成完整的一次游戏 (complete episodes).
- MC 方法最终的 idea 就是  $value = mean\ return$  (进行多次实验, 求平均值).  
接着我们对 MC 方法进行一个大体的说明.
  - 明确目标, learn  $v_\pi$  from episodes of experience under policy  $\pi$ .
  - 对于状态  $s$ , 我们要计算的  $v_\pi(s) = E_\pi[G_t | S_t = s]$ 
    - 其中  $G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T$
    - $R_{t+n}$  我们都是可以在模拟的时候得到的.
  - 于是我们使用策略  $\pi$  进行一次完整的实验, 可以得到  $s_1, a_1, r_2, \dots, s_k \sim \pi$ .
  - 进行多次实验后, 我们计算得到多个  $G_t$ , 我们使用 empirical mean return 来代替 expected return.

### 3.1.2 MC 方法详细步骤

上面介绍了 MC 方法的一个大致的思路, 这里我们详细说明一下如何估计 state  $s$ .

- 进行第  $i$  轮 episode 时, 从第一次访问  $s$  到结束, 计算得到  $G_i(s)$ .
- 增加计数器,  $N(s) \leftarrow N(s) + 1$ .
- 增加 total return,  $S(s) \leftarrow S(s) + G_i(s)$ .
- 计算  $V_\pi(s)$ , 求平均,  $V_\pi(s) = S(s)/N(s)$ .
- 重复上面的步骤, 当  $N(s) \rightarrow \infty$ ,  $V_\pi(s)$  可以逼近于  $\pi$  的 value function.

我们使用 **BlackJack** 的例子来进行说明, 在这个环境中, observation 是一个三元组:

- the players current sum (目前手上牌的总和)
- the dealer's one showing card (1-10 where 1 is ace) (对手牌面的牌)
- whether or not the player holds a usable ace (0 or 1) (自己手上是否有 A, 这个牌可以当做 1 或是 11)

我们使用的策略就是当前自己牌的和小于 20, 就一直要牌. 我们要对这个策略进行评估.

例如现在我们进行了一次模拟(第 10 轮), 得到观测值为  $(15, 8, False), (17, 8, False), (20, 8, False)$ , 分别对应的 reward 是 0, 0, 1. 假设我们要计算状态  $s = (17, 8, False)$ .

此时已有

- $N(s = (17, 8, False))_9 = 4$ , 目前进行了 9 次模拟, 出现了 4 次.
- $S(s = (17, 8, False))_9 = -6$ , 目前的总的累计的奖励.
- $\gamma = 0.9$ , 折扣因子.

那么(这一部分的代码可以查看 github 仓库中的 **04\_MonteCarlo\_normalMean.ipynb** 文件), 这时候迭代各个变量应该进行如下变化:

- $N(s = (17, 8, False))_{10} = 4 + 1$
- $G = 0 + 0.9 \cdot 1 = 0.9$ , 注意这里的计算.
- $S(s = (17, 8, False))_{10} = -6 + 0.9 = -5.1$
- $V(s = (17, 8, False)) = S(s = (17, 8, False))_{10}/N(s = (17, 8, False))_{10} = -1.02$

### 3.1.3 Incremental Mean

我们还可以对上面求平均的方式进行进一步的优化, 这个称之为 incremental mean.

$$\begin{aligned}
\mu_k &= \frac{1}{k} \sum_{j=1}^k x_j \\
&= \frac{1}{k} (x_k + \sum_{j=1}^{k-1} x_j) \\
&= \frac{1}{k} (x_k + (k-1) \mu_{k-1}) \\
&= \mu_{k-1} + \frac{1}{k} (x_k - \mu_{k-1})
\end{aligned} \tag{3.1}$$

也就是在更新的时候, 使用下面的方式进行更新  $v$ . 这样做的好处是不需要保存历史的  $S$  值, 每次在现有的平均值的基础上进行更新. 也就是每次在修正  $V(S_t)$  的值.

- 增加计数器,  $N(s) \leftarrow N(s) + 1$ .
- $V(S_t) \leftarrow V(S_t) + \frac{1}{N} (G_t - V(S_t))$ .

最终的 value function 如图3.1所示:

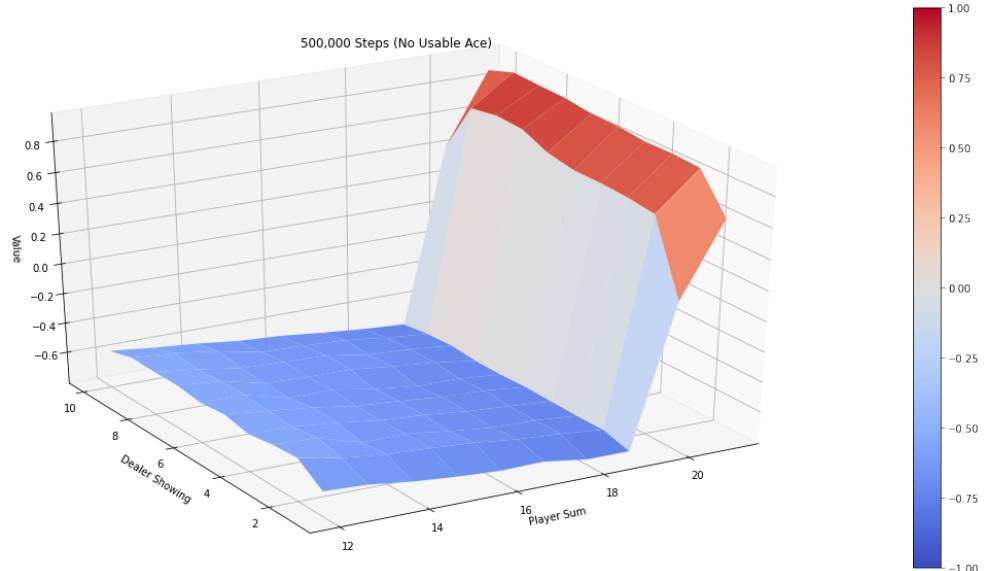


图 3.1: BlackJack 指定策略  $\pi$  的 Value Function 示意图.

对于式子  $V(S_t) \leftarrow V(S_t) + \frac{1}{N} (G_t - V(S_t))$  来说, 我们可以使用  $\alpha$  来代替  $\frac{1}{N(S_t)}$ . 也就是最终的式子为3.2. 这样可以迭代次数也不需要进行记忆. 这个式子会在讲 TD 的时候再次用到, 看 TD 与 MC 的联系:

$$V(S_t) \leftarrow V(S_t) + \alpha (G_t - V(S_t)) \tag{3.2}$$

但是 MC 方法还是需要跑完完整的一次实验, 才可以获得  $G_t$ , 不能走一步更新一次.

## 3.2 Temporal-Difference Learning

### 3.2.1 TD 方法思想

在上面介绍的 MC 方法中, 有下面两个缺点:

- 我们需要等整个过程结束之后才可以进行更新.
- 我们需要知道最终的结果 (final outcome) 才可以对 value function 进行更新.

但是这样效率就不会很高, 因为有的时候, 完成一个完整的回合是需要花费很多时间的. 这时, 就要用到 **Temporal-Difference(TD)** 的方法. 我们用一个简单的开车的例子来说明 TD 方法的思想.

我们现在驾车要从地点 A 到地点 C, 途中会经过地点 B. 我们希望可以知道从地点 A 到地点 C 需要的时间, 以及地点 B 到地点 C 需要的时间. 于是我们驾车出发, 得到下图3.2的信息:

	Place A	Place B	Place C
预计时间	30 min	20 min	0 min
花费时间	0 min	15 min	10 min
累计花费时间	0 min	15 min	25 min

图 3.2: TD 算法思想解释, Driving Home 的例子.

我们详细解释一下图3.2的内容:

- 当我们刚出发的时候, 我们预计从 A 到 C 需要 30min, 此时还没开车, 花费时间为 0;
- 当我们达到地点 B 的时候, 我们花费了 15min. 我们估计从 B 到 C 需要 20min, **TD 算法在这个时候就会更新 A 到 C 的预测时间, 改为  $20+15=35$ min.** 这里 15 相当于就是获得的 reward. 20 就是对下一阶段的预测, 也就是  $v_\pi(s)$ ;
- 当到达 C 的时候, 我们从 B 到 C 花费了 10min. **TD 算法在这个时候就会更新 B 到 C 的预测时间, 改为  $0+10=10$ min;**
- 使用 **MC** 方法的时候, 我们需要等整个过程结束之后才进行更新, A 到 C 的花费是 25min;

我们可以将上面从地点 A 到地点 C 需要的时间看成地点 A 到地点 C 的累计奖励, 每次花费的时间为即时奖励. 于是我们可以得到递推式子3.3, 是 TD 算法的一个更新的式子(在上面的例子中, 由于  $\alpha = 1$ , 所以  $V(S_t)$  被消掉了):

$$V(S_t) \leftarrow V(S_t) + \alpha (R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \quad (3.3)$$

其中:

- $R_{t+1}$  表示离开这个状态获得的即时奖励;
- $V(S_{t+1})$  表示下一状态的累计奖励.
- $R_{t+1} + \gamma V(S_{t+1})$  称作是 TD target;
- $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$  称作是 TD error;

我们可以对比式子3.2, 我们将其中  $G_t$ (实际奖励) 换成了  $R_{t+1} + \gamma V(S_{t+1})$ (当前奖励 + 下一状态预估累计奖励).

### 3.2.2 TD 与 DP 和 MC 的比较

首先我们将 **TD** 与 **DP(dynamic program)** 进行比较. DP 我们是应用了贝尔曼方程进行求解, 具体的式子为1.18, 最后可以写为式子3.4:

$$v_\pi(s) = E_\pi[R_t + \gamma \cdot v_\pi(S_{t+1}) | S_t = s] \quad (3.4)$$

我们比较式子3.3和式子3.4, 可以发现当3.3中  $\alpha = 1$  的时候, 3.3就是式子3.4期望里面的值. 我们还是通过多次实验, 使用 empirical mean return 来代替 expected return. 我们可以这么理解 TD 的式子, 每一次我们获得了一些信息(新的 reward), 就对之前的预测进行更正.

接着是关于 **TD** 与 **MC** 进行比较. 因为 MC 的计算涉及  $G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$ , 其中  $R_t$  每次实验结果会不同, 会有较大的方差, 所以总体方差较大. 但是 TD 方法只会用到当前一步的 reward( $R_{t+1}$ ), 所以方差较小.

关于 TD 与 MC 的不同, 我们再通过下面一个例子进行说明.

**例 3.1** 假设现在只有两个状态, A 和 B.  $\gamma = 1$ , 我们进行了 8 次实验, 结果如表3.1所示, 其中二元组 (A,0) 表示 A 为状态, 0 为获得的 reward:

下面分别使用 MC 方法和 TD 方法计算 A 和 B 的价值. 使用 **MC 方法**:

- $V(s = A) = 0 + 0 = 0$
- $V(s = b) = \frac{6}{8}$

使用 **TD 方法**:

- $V(s = B) = \frac{6}{8}$
- $V(s = A) = R + V(s = B) = 0 + \frac{6}{8} = \frac{6}{8}$



序号	状态转移
1	(A, 0), (B, 0)
2	(B, 1)
3	(B, 1)
4	(B, 1)
5	(B, 1)
6	(B, 1)
7	(B, 1)
8	(B, 0)

表 3.1: MC 和 TD 差异的例子.

## 3.3 TD( $\lambda$ )

### 3.3.1 n-Step Prediction

n-步预测指从状态序列的当前状态  $S_t$  开始往序列终止状态方向观察至状态  $S_{t+n-1}$ , 使用这 n 个状态产生的即时奖励  $R_{t+1}, R_{t+2}, \dots, R_{t+n}$  以及状态  $S_{t+n}$  的预估价值来计算当前状态  $S_t$  的价值. 简单的来讲, 就是走 n 步, 用这 n 步的结果来预测后面的累计价值. 我们用公式表将 n-step prediction 表示出来就是如式3.5所示:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) \quad (3.5)$$

有了式3.5之后, 我们就可以使用  $G_t^{(n)}$  来更新价值函数, 如式3.6所示:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^{(n)} - V(S_t)) \quad (3.6)$$

图3.3比较取不同的 n 的情况:

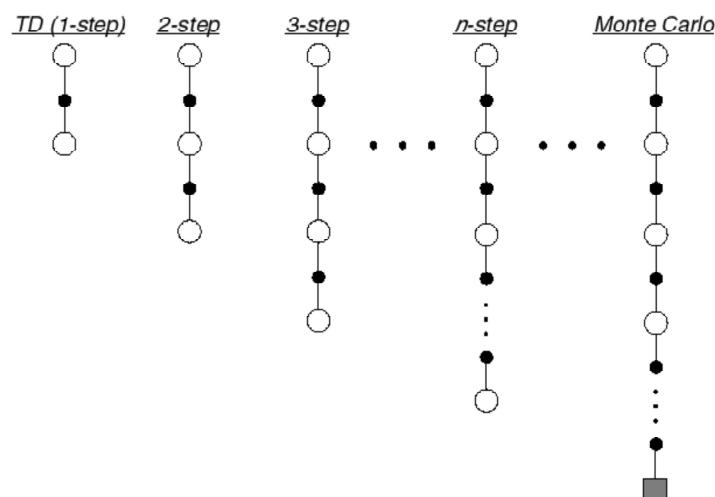


图 3.3: 取不同 n 的情况.

可以看到:

- $n = 1$  的时候, 就是 TD 方法, 更新式子为,  $V(S_t) \leftarrow V(S_t) + \alpha (R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$ .
- $n = \infty$  的时候, 就是 MC 方法, 更新式子为,  $V(S_t) \leftarrow V(S_t) + \alpha (G_t - V(S_t))$ , 这里因为做了完整的实验, 所以  $G_t$  是可以得到的.

### 3.3.2 $\lambda$ -return

上面我们介绍了不同的 n-step prediction, 那么取何种  $n$  的时候, 是最优的呢. 实际上, 我们可以不指定一个特定的  $n$ , 而是将其平均起来. 比如说, 现在我们有 2-step return  $G^{(2)}$  和 4-step return  $G^{(4)}$ , 那么平均就是  $\frac{1}{2} \cdot (G^{(2)} + G^{(4)})$ . 那么, 是否有一种有效的方式, 可以让我们将所有的 time-steps 全部合并起来.

我们使用  $\lambda$ -return,  $G_t^\lambda$  可以将 n-step returns  $G_t^{(n)}$  都合起来. 其中  $G_t^\lambda$  的计算方式如式3.7所示:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \quad (3.7)$$

之后的更新的公式如式3.8所示:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^\lambda - V(S_t)) \quad (3.8)$$

注意, 这里涉及一个无穷级数求和, 如式3.9所示:

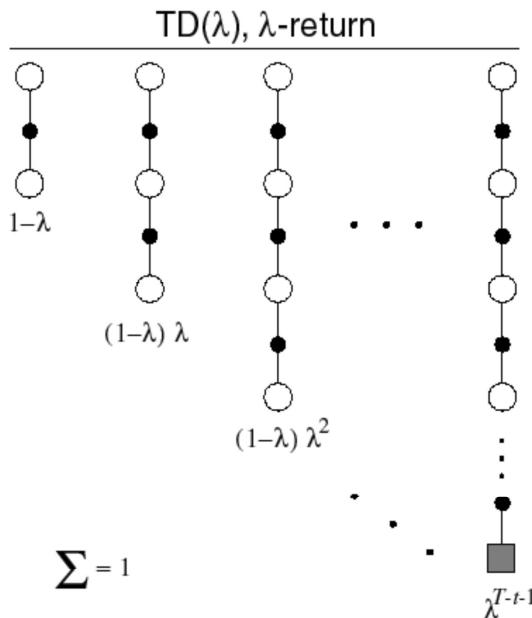
$$\sum_{k=0}^n \lambda^k = \frac{1 - \lambda^{n+1}}{1 - \lambda} \quad (3.9)$$

于是,  $\lambda$ -return 的整体计算步骤如图3.4所示, 表示从  $t$  时刻状态  $s$  第一次被访问到, 从这个时间开始的累计奖励.

因为是有限步, 这里是有  $T - t - 2$  步 (其中  $T$  是整个流程完成需要的 steps), 所以最后的系数是  $\lambda^{T-t-1}$ , 此时的系数和为 1. 系数和的计算步骤如下所式3.10所示:

$$\begin{aligned} & (1 - \lambda) \sum_{k=0}^{T-t-2} \lambda^k + \lambda^{T-t-1} \\ & = (1 - \lambda) \frac{1 - \lambda^{T-t-1}}{1 - \lambda} + \lambda^{T-t-1} \\ & = 1 \end{aligned} \quad (3.10)$$

也就是前面按照几何加权, 最后一个系数为一个固定的系数, 保证最后的系数和为 1. 注意, 这时候

图 3.4:  $G_t^\lambda$  的计算方式.

- $\lambda = 0$  就是 TD;
- $\lambda = 1$  就是 MC;

在引入  $\lambda$  之后, 我们发现要更新一个状态的价值函数  $V(S_t)$ , 就必须走完一整个 episode, 获得每一个状态的即时奖励. 这个与 MC 算法是一样的.

### 3.3.2.1 Eligibility Traces

下面我们换一个角度, 来看上面的  $\lambda$ -return, 我们着重分析一下式子 3.8 中的 error,  $G_t^\lambda - V(S_t)$ . 我们将其进行展开 (拆开), 进行化简. 首先我们按照各个的定义进行展开:

$$\begin{aligned}
 & G_t^\lambda - V(S_t) \\
 &= -V(S_t) \\
 &\quad + (1 - \lambda)\lambda^0(R_{t+1} + \gamma V(S_{t+1})) \\
 &\quad + (1 - \lambda)\lambda^1(R_{t+1} + \gamma R_{t+1} + \gamma^2 V(S_{t+2})) \\
 &\quad + (1 - \lambda)\lambda^2(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 V(S_{t+1})) \\
 &\quad + \dots
 \end{aligned} \tag{3.11}$$

其中(对上面式子接着进行化简):

$$\begin{aligned}
 & (1-\lambda)\lambda^0(R_{t+1} + \gamma V(S_{t+1})) \\
 = & \lambda^0(R_{t+1} + \gamma V(S_{t+1}) - \lambda R_{t+1} - \lambda \gamma V(S_{s+1})) \\
 = & (\gamma \lambda)^0[R_{t+1} + \gamma V(S_{s+1}) - \lambda \gamma V(S_{t+1})] \\
 & - \lambda R_{t+1}
 \end{aligned} \tag{3.12}$$

$$\begin{aligned}
 & (1-\lambda)\lambda^1(R_{t+1} + \gamma R_{t+1} + \gamma^2 V(S_{t+2})) \\
 = & \lambda^1(R_{t+1} + \gamma R_{t+1} + \gamma^2 V(S_{t+2}) - \lambda R_{t+1} - \lambda \gamma R_{t+1} - \lambda \gamma^2 V(S_{s+1})) \\
 = & (\gamma \lambda)^1[R_{t+2} + \gamma V(S_{s+2}) - \lambda \gamma V(S_{t+2})] \\
 & + \lambda R_{t+1} \\
 & - \lambda^2(R_{t+1} + \gamma R_{t+2})
 \end{aligned} \tag{3.13}$$

同理,我们可以对使用相同的方式进行如下的化简.

$$\begin{aligned}
 & (1-\lambda)\lambda^2(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 V(S_{t+1})) \\
 = & (\gamma \lambda)^2[R_{t+3} + \gamma V(S_{s+3}) - \lambda \gamma V(S_{t+3})] \\
 & + \lambda^2(R_{t+1} + \gamma R_{t+2}) \\
 & - \lambda^3(R_{t+1} + \gamma R_{t+2} + \lambda \gamma^2 R_{t+3})
 \end{aligned} \tag{3.14}$$

所以,  $G_t^\lambda - V(S_t)$  可以接着进行化简.



$$\begin{aligned}
& G_t^\lambda - V(S_t) \\
&= -V(S_t) \\
&\quad + (1-\lambda)\lambda^0(R_{t+1} + \gamma V(S_{t+1})) \\
&\quad + (1-\lambda)\lambda^1(R_{t+1} + \gamma R_{t+1} + \gamma^2 V(S_{t+2})) \\
&\quad + (1-\lambda)\lambda^2(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 V(S_{t+1})) \\
&\quad + \dots \\
&= -V(S_t) \\
&\quad + (\gamma\lambda)^0[R_{t+1} + \gamma V(S_{s+1}) - \lambda\gamma V(S_{t+1})] \\
&\quad + (\gamma\lambda)^1[R_{t+2} + \gamma V(S_{s+2}) - \lambda\gamma V(S_{t+2})] \\
&\quad + (\gamma\lambda)^2[R_{t+3} + \gamma V(S_{s+3}) - \lambda\gamma V(S_{t+3})] \\
&\quad + \dots \\
&= (\gamma\lambda)^0[R_{t+1} + \gamma V(S_{s+1}) - V(S_t)] \\
&\quad + (\gamma\lambda)^1[R_{t+2} + \gamma V(S_{s+2}) - V(S_{t+1})] \\
&\quad + (\gamma\lambda)^2[R_{t+3} + \gamma V(S_{s+3}) - V(S_{t+2})] \\
&\quad + \dots
\end{aligned} \tag{3.15}$$

从上面可以看到, 每次连加的时候, 都只有下一个状态的 reward, 这个时候就不需要对所有的 reward 进行存储了. 上面式子 3.15 里面就是 1-step prediction, 外面是系数, 之前外面只有  $\gamma$ , 现在有了  $\lambda$ .

我们定义 Eligibility traces 为 (每一个 state 都有一个对应的值), 来表示外面的系数 (这个是每一轮都需要重置, 就是一把游戏结束就初始化为 0):

$$E_t(s) = \begin{cases} 0 & t = 0 \\ \gamma\lambda E_{t-1}(s) & S_t \neq s \\ \gamma\lambda E_{t-1}(s) + 1 & S_t = s \end{cases} \tag{3.16}$$

$V(s)$  可以使用如下方式进行更新.

$$V(S_t) \leftarrow V(S_t) + \alpha E_t(s)(R_{t+1} + \gamma V(S_{s+1}) - V(S_t)) \tag{3.17}$$

我们假设  $k$  时刻第一次达到状态  $s$ (这里只考虑一次访问状态  $s$ , 多次访问就是在后面进行累加即可), 于是  $E_t(s)$  可以化简为:

$$E_t(s) = \begin{cases} 0 & t < k \\ (\gamma\lambda)^{t-k} & t \geq k \end{cases} \tag{3.18}$$

因为式子3.17是每一轮进行迭代, 也就是总的修改量如式子3.19所示, 迭代修改与一起修改值是一样的:

$$\begin{aligned}
 & \sum_{t=0}^T \alpha E_t(s)(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \\
 &= \alpha(\gamma\lambda)^0[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \\
 &\quad + \alpha(\gamma\lambda)^1[R_{t+2} + \gamma V(S_{t+2}) - V(S_{t+1})] \\
 &\quad + \alpha(\gamma\lambda)^2[R_{t+3} + \gamma V(S_{t+3}) - V(S_{t+2})] \\
 &\quad + \dots \\
 &= \alpha(G_t^\lambda - V(S_t))
 \end{aligned} \tag{3.19}$$

所以使用 Eligibility traces 和式子3.8是等价的, 也就是  $V(S_t) \leftarrow V(S_t) + \alpha(G_t^\lambda - V(S_t))$ . 一次更新  $G_t^\lambda - V(S_t)$  和迭代更新  $E_t(s)(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$  是一样的.

我们对上面的 Eligibility traces 进行解释, 他其实考虑了两个因素, 分别是出现的次数与出现的位置.

例如, 现在有一个情况为" 响铃-> 响铃-> 响铃-> 灯亮-> 拿到食物", 那么问" 响铃" 和" 灯亮" 两个动作, 哪个导致了最后拿到食物. Eligibility traces 就是同时考虑了两种情况, 当出现同一个情况的时候, state 会直接加 1, 同时, 随着时间的推移, 又会乘上一个衰减系数. 由上面式子3.15的证明, 可以看到  $G_t^\lambda$  与这里的 Eligibility traces 是等价的. 在使用 Eligibility traces 的时候, 我们可以进行一步, 更新一次  $V(s)$ , 不需要等整个 episodes 结束之后, 才能更新.

# 第4章 Model-Free Control

## 内容提要

- |  |  |
|--|--|
| <input type="checkbox"/> On-Policy Monte-Carlo Control | Learning                                     |
| <input type="checkbox"/> On-Policy Temporal-Difference | <input type="checkbox"/> Off-Policy Learning |

在上一章的时候, 我们是在不了解 MDP 的结构的情况下, 对于一个给定的策略  $\pi$ , 来求解 value function  $v_\pi$ . 在这一章的时候, 我们会更进一步, 如何对策略进行改进, 使得新的策略  $\pi'$ , 有  $v_{\pi'}(s) > v_\pi(s)$ . 会分为三个方法进行介绍:

- On-Policy Monte-Carlo Control
- On-Policy Temporal-Difference Learning
- Off-Policy Learning

## 4.1 On and Off-Policy Learning

因为之后的策略分为两种, 分别是 **On Policy** 和 **Off Policy**, 我们在这里首先介绍一下这两种的区别.

- On Policy Learning
  - Learn on the job;
  - Learn about policy  $\pi$  from experience sampled from  $\pi$ ;
  - 也就是我们用策略  $\pi$  做模拟的时候, 我们同时在对  $\pi$  进行优化;
- Off Policy Learning
  - Look over someone's shoulder;
  - Learn about policy  $\pi$  from experience sampled from  $\mu$ ;
  - 我们可能从一个更好的策略  $\mu$  进行抽样, 来优化现有的策略  $\pi$ ;

## 4.2 On-Policy Monte-Carlo Control

### 4.2.1 Monte-Carlo Control 的基本思想

之前我们在" 动态规划" 那一部分介绍过, 策略的更新主要分为两个步骤:

- **Policy Evaluation**, Estimate  $v_\pi$  (首先对当前策略进行评估, 得到  $v_\pi$ );
- **Policy Improvement**, Generate  $\pi' > \pi$  (使用 greedy policy improvement 的思想, 来优化策略  $\pi$ );

但是在“动态规划”那一章中，我们使用式子4.1来生成新的策略。在那一章，我们对  $P_{ss'}^a$  是已知的。但是因为现在是 model-free 的情况下，所以我们不知道这些值。也就是我们无法从  $v(s)$  求出对应的策略  $\pi$ 。

$$\pi(s) = \arg \max_{a \in A} P_{ss'}^a(R_{ss'}^a + \gamma V(s')) \quad (4.1)$$

如图4.2所示，我们知道  $q(s, a) = P_{ss'}^a(R_{ss'}^a + \gamma V(s'))$ 。也就是对上面的式子4.1进行一些替换。

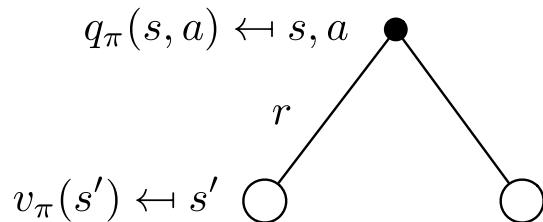


图 4.1: Action Value function 的解释。

替换完毕之后如式4.2所示。所以，我们实际上可以通过式子4.2来进行新的策略的生成。我们也是可以通过模拟的方法来求出  $q(s, a)$  的值。

$$\pi(s) = \arg \max_{a \in A} q(s, a) \quad (4.2)$$

## 4.2.2 $\epsilon$ -Greedy Exploration

现在又产生了一个新的问题，就是如果采用 greedy policy 的话，我们很可能陷入循环，不会探索新的路径。我们看下面的例子。

### 例 4.1

现在有两扇门，打开左侧和打开右侧的门会获得不同的奖励。设当前所在的状态是  $s$ 。

- 我们打开左边的门（然后整个过程结束），获得 reward=0，得到  $q(s, 'left')=0$ ；
- 我们打开右边的门，获得 reward=1，得到  $q(s, 'right')=1$ ；
- 这个时候，因为打开右侧的门获得的收益比打开左侧的门大，之后 policy 就会打开右侧的门，因为  $right = \arg \max_{a \in A} q(s, a)$ ；
- 也就是下一轮，打开右侧的门获得 reward=3， $q(s, 'right') = \frac{3+1}{2} = 2$ ；
- 下一轮还是会打开右侧的门，此时 reward=2，则  $q(s, 'right') = \frac{1+3+2}{3} = 2$ ；

但是我们不能确定打开右侧的门就是最好的，我们只尝试开过一次左侧的门。

为了解决上面的问题, 我们会使用  $\epsilon$ -Greedy Exploration 的方法. 这个方法的思想很简单, 我们设定一个  $\epsilon$  值, 来控制是 Explore(探索) 还是 Exploit(利用).

- Explore(探索) 就是不选择收益最大的那个 action.
- Exploit(利用) 就是按照  $a^* = \arg \max_{a \in A} q(s, a)$  进行选择动作.

比如  $\epsilon = 0.1$ , 那么有 10% 的可能性用在探索上 (Explore), 90% 用在利用上 (Exploit). 具体实施的时候, 我们按照式子 4.3 给每一个动作一个概率.

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon & \text{if } a^* = \arg \max_{a \in A} (q(s, a)) \\ \frac{\epsilon}{m} & \text{otherwise} \end{cases} \quad (4.3)$$

上式中,  $m$  表示所有可能的动作. 也就是对于使得  $q(s, a)$  最大的动作, 我们以  $\frac{\epsilon}{m} + 1 - \epsilon$  的概率执行. 其他的每一个动作都以  $\frac{\epsilon}{m}$  的概率执行.

这里概率和是 1, 如式子 4.4 所示.

$$(m - 1) * \frac{\epsilon}{m} + \frac{\epsilon}{m} + 1 - \epsilon = 1 \quad (4.4)$$

这里当  $\epsilon = 1$  的时候, 是随机的策略, 也就是每个 action 的概率都是一样的.

那么使用  $\epsilon$ -Greedy Exploration, 得到的新的策略  $\pi'$  一定会比之前的好吗. 这个是正确的, 有一个定理可以证明和保证 (暂时先不放证明, 详细的课件上有).

### 4.2.3 Annealing $\epsilon$ -greedy mechanism

之前我们有说到过  $\epsilon$ -greedy 策略, 但是如何选取合适的  $\epsilon$  呢. 一个比较流行的方法就是使用 annealing  $\epsilon$ -greedy mechanism, 这时  $\epsilon$  会随着游戏轮数 (episode i) 的进行逐渐下降.

$\epsilon$  的计算方法如式子 4.5:

$$\epsilon = \max\left(\frac{\epsilon_m - 1}{M_\epsilon} * i + 1, \epsilon_m\right) \quad (4.5)$$

其中  $\epsilon_m$  表示在  $M_\epsilon$  之后希望得到的  $\epsilon$  的值. 比如希望在 50 轮之后 ( $M_\epsilon = 50$ ), 使得  $\epsilon = 0.01$  (也就是  $\epsilon_m = 0.01$ ).

我们简单看一下  $i$  取值不同时的情况:

- 当  $i = 0$  的时候, 此时  $\epsilon = 1$ , 此时每一个 action 执行的概率是相等的.
- 当  $i = 1$  的时候, 如果  $M_\epsilon = 50$ ,  $\epsilon_m = 0.01$ , 那么此时  $\epsilon = 0.98$ .
- 当  $i = M_\epsilon$  时,  $\epsilon = \epsilon_m$ .
- 当  $i > M_\epsilon$  时,  $\epsilon = \epsilon_m$ .

所以, 当  $i$  从 0 到  $\infty$  的过程时,  $\epsilon$  就是从 1 开始逐渐递减, 直到  $\epsilon = \epsilon_m$  结束.

#### 4.2.4 Greedy in the Limit with Infinite Exploration (GLIE)

我们把上面的思路结合起来, 在每一个 **episode** 的时候 (我们不需要把  $q$  估计的很准确, 也就是  $q$  的更新可能只要更新一次即可):

- **Policy Evaluation**, 我们估计  $q$ ;
- **Policy Improvement**, 我们使用  $\epsilon$ -Greedy Policy Improvement;

我们称上面的方法为 GLIE Monte-Carlo Control. 下面我们再来看一下详细的步骤.

- 在第  $k$  轮的迭代中 ( $k$ th episode), 我们使用策略  $\pi$  得到了  $\{S_1, A_1, R_2, \dots, S_T\}$ ;
- 对于每一个状态  $S$ , 和动作  $A$  二元组,  $\{S_t, A_t\}$ , 我们进行如下操作 (**更新 Q**):
  - $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$ , 出现次数 +1;
  - $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t))$ , 使用 incremental mean;
- 更新策略,  $\pi \leftarrow \epsilon - greedy(Q)$

在使用 GLIE 的时候, 需要保证下面的两个条件:

- 所有的 (state, action) 对必须要被访问到无数次, 也就是  $\lim_{k \rightarrow \infty} N_k(s, a) = \infty$ ;
  - 最终的 policy 必须要收敛为 greedy, 也就是  $\lim_{k \rightarrow \infty} \pi_k(a|s) = 1$ , 比如  $\epsilon = \frac{1}{k}$ ;
- 在实际使用的时候, 不满足这两个条件, 也是可以进行使用的.

#### 4.2.5 BlackJack 与 GLIE

我们还是使用 **BlackJack** 的例子来进行说明. 因为 **BlackJack** 一把很短, 所以很适合使用 **MC** 的方法, 因为很容易就完成一局.

我们按照上面的 GLIE 的方法进行迭代, 每次更新  $Q$ , 从而更新策略  $\pi$ . 我们会将  $Q$  按照下面的方式进行存储:

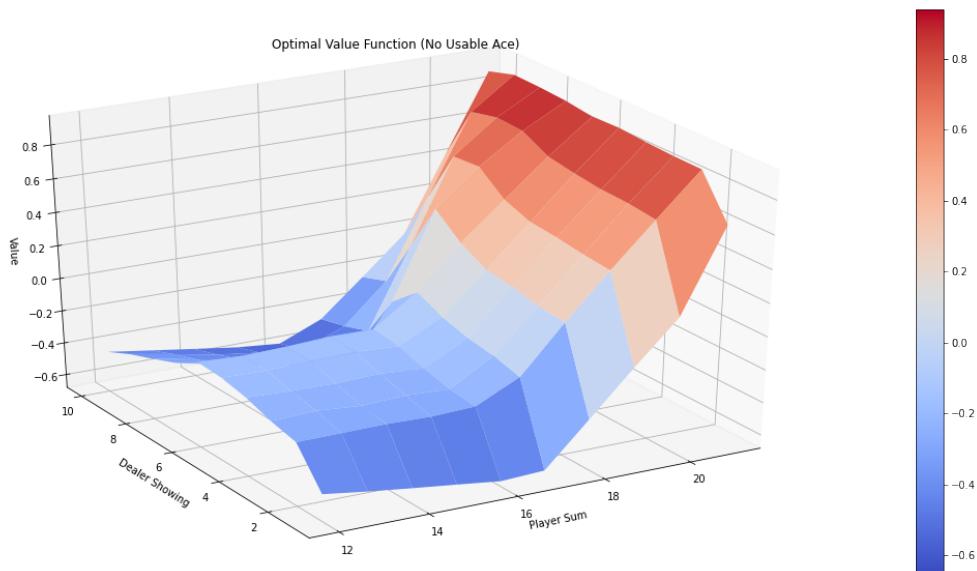
- 存储为字典类型;
- key 为 state;
- value 为两个动作对应的  $q$  值;

```
{(12, 1, False): array([-0.77219167, -0.51320247]),
 (12, 1, True): array([-0.69767442, -0.2746601 ]),
 (12, 2, False): array([-0.29160021, -0.30458926])}
```

最后得到最终的策略, 并绘制出此时的  $V$  的图像, 如图4.2所示. 可以看到与图3.1比起来, 图像的 value 会比之前的要高.

关于上面  $V$  的计算, 我们根据  $V(s) = \max_{a \in A} Q(s)$  来计算. 最后的策略不需要探索, 也就是不需要  $\epsilon$ , 故  $V(s)$  可以按上式计算.

最终的 policy 如图4.3所示, 其中 hit 表示要牌, stick 表示不要牌. 其中可以看到, 当没有 Ace 的时候 (图4.3(b)), 即使当我们的牌比较小, 当对手牌面也比较小,

图 4.2: BlackJack 最优策略  $\pi$  的 Value Function 示意图。

我们选择不要牌 (stick).

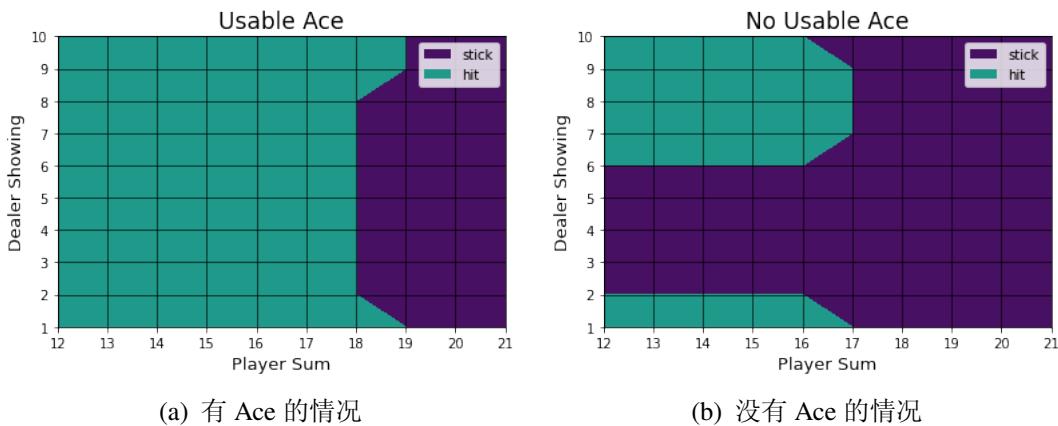


图 4.3: BlackJack Policy 示意图

我实际用这个策略在在线游戏上测试了一下<sup>1</sup>, 不能说是一直在赢, 但是从初始 1500 到最后 4750. 我是每次都压 500. 最终的结果如图4.4所示.

## 4.2.6 Windy Gridworld Playground 与 GLIE

在这里会介绍一个新的环境 Windy Gridworld Playground, 以及使用 GLIE 来寻找最优策略. 上面的 BlackJack 因为一般一整把游戏就 1-3 回合就结束了, 不是很适合来作为 TD 的例子. 同时, 因为我们也不知道 BlackJack 最优的策略是长什么样子的, 我们不知道最后结果的好坏, 所以我们使用一个新的环境来举例子.

首先我们来简单介绍一下 Windy Gridworld Playground 环境. 环境总体如图4.5所示:

<sup>1</sup><https://www.arkadium.com/games/blackjack/>

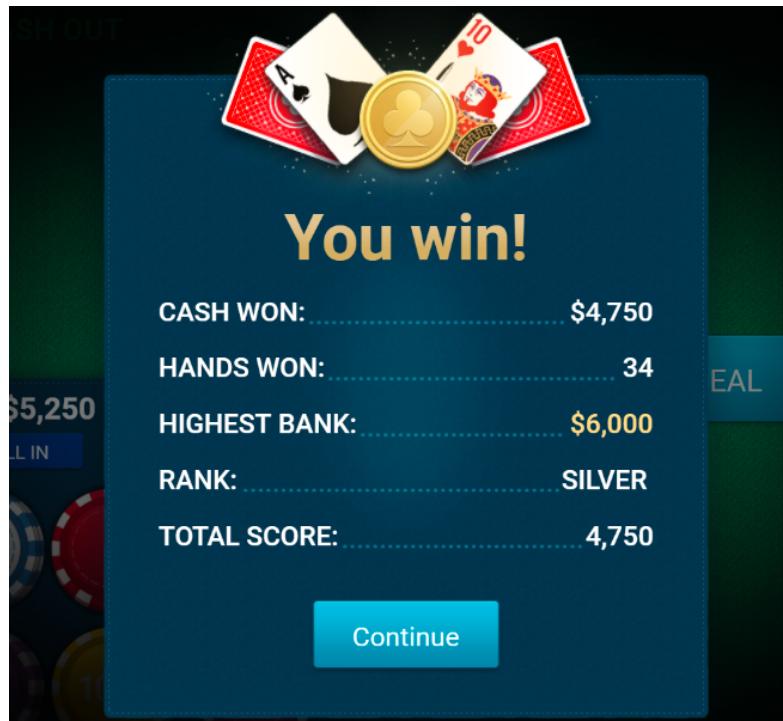


图 4.4: 使用上述策略实际测试.

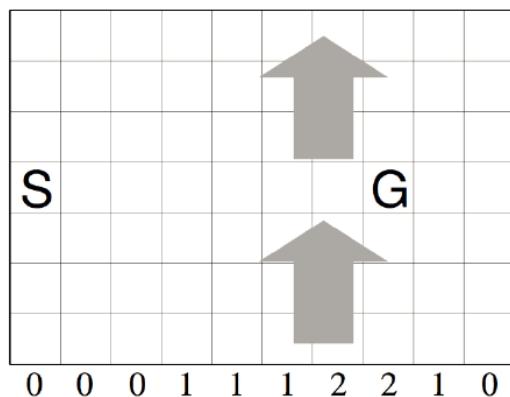


图 4.5: Windy Gridworld Playground 环境介绍.

- **目标:** 我们要从 **S** 出发, 到达 **G**.
  - **环境介绍:** 每一列下面有数字 0,1,2, 这个表示风力, 走在这些列上的时候会受到相应向上的风力的影响;
  - **action:** 我们有上下左右四个方向可以走, 对应关系分别如下;
    - 0, UP
    - 1, RIGHT
    - 2, DOWN
    - 3, LEFT
  - **observation:** 每次返回的观察为格子所在的编号, 起始的编号是 30;
  - **reward:** 每走一步 reward=-1, reward 越大也就是走的步数越少;
- 我们使用 GLIE 来寻找一下最佳路径. 总的思路还是和上面 BlackJack 的例子是一样的. 最终的结果如图4.6所示:
- 图4.6(a)表示随着 episode 的回合次数变多, reward 变大 (说明走的步数变少);
  - 图4.6(b)表示随着 episode 的回合次数变多, episode length 变少 (说明走的步数变少);

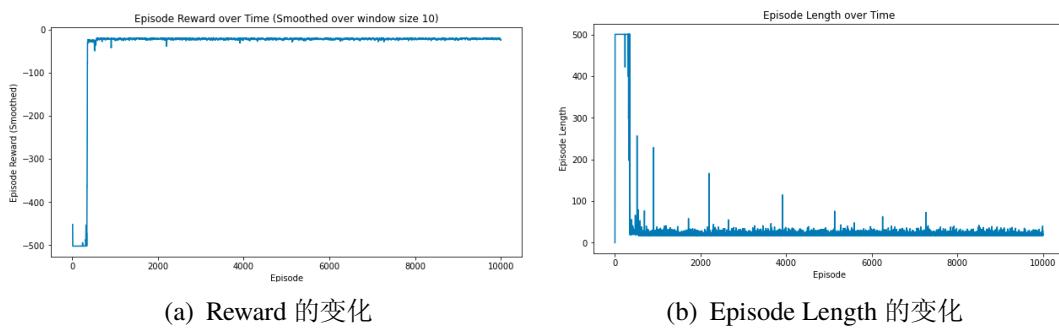


图 4.6: Windy Gridworld Playground 使用 GLIE 效果图

这里最后的结果是大约走 17 步. 这个不是最优的解, 最优解是 14 步.

### 4.3 On-Policy Temporal-Difference Control

我们在上一章讲 prediction 的时候说到, TD Learning 比起 MC 会有一些优点:

- Low variance (低的方差);
- Online (可以实时进行学习, 不需要等整个 episode 完成);
- Incomplete sequences (不完整的 episode 也是可以学习的);

于是, 我们很自然的想法就是使用 TD 来替代上面的 MC 方法. 于是整体的流程如下所示, 每一个 time-step 进行下面的更新:

- 使用 TD 算法来估计  $Q(S, A)$  的值.
- 使用  $\epsilon$ -greedy policy 来更新策略.

### 4.3.1 Sarsa

上面说的这个想法就是 **Sarsa** 方法, 接下来我们就详细介绍如何使用 **Sarsa** 方法, 更新 action-value function, 也就是更新  $Q(s, a)$ . 如图4.7所示:

- 当前 agent 处于状态  $S$  中, 首先基于  $\epsilon$ -greedy 选择一个行为  $A$ ;
- agent 执行这个动作, 获得奖励,  $R$ ;
- 环境到达一个新的状态  $S'$ ;
- 再次基于  $\epsilon$ -greedy 选择一个行为  $A'$ , 此时不执行动作, 只是为了获得  $Q(S', A')$  来更新  $Q(S, A)$  (查看式子4.6);

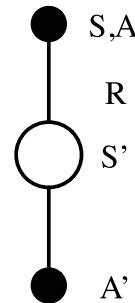


图 4.7: Sarsa 示意图.

这也是为什么叫 Sarsa 这个名字, 也就是上面的五个状态依次出现. 于是, 式子4.6表示  $Q(S, A)$  的更新, 这个和式子3.3是一样的:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)] \quad (4.6)$$

在使用 Sarsa 的时候, 需要保证下面的两个条件. 前面的两个条件是和 GLIE 中的条件是一样的.

- 所有的 (state, action) 对必须要被访问到无数次, 也就是  $\lim_{k \rightarrow \infty} N_k(s, a) = \infty$ ;
- 最终的 policy 必须要收敛为 greedy, 也就是  $\lim_{k \rightarrow \infty} \pi_k(a|s) = 1$ , 比如  $\epsilon = \frac{1}{k}$ ;
- 系数  $\alpha$  需要满足,  $\sum_{t=1}^{\infty} \alpha_t = \infty$ , 且  $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$ ;

下面把 Sarsa 详细的算法说明一下, 其中:

- $\alpha$  表示步长, 注意步长一般随着迭代逐渐减少, 这样才可以保证算法收敛;
- $\gamma$  表示衰减因子, 也就是折扣因子;
- $\epsilon$  表示探索率, 这个探索率也是需要逐渐减少, 最后整个 policy 变为 greedy;

Sarsa 算法可以写成下面的样子:

---

**Algorithm 1:** Sarsa 算法简介
 

---

```

Initialize  $Q(s, a)$ ,  $\forall s \in S, a \in A(s)$ ,  $Q(s, a) = 0$ ,  $N = 0$  ;
while  $N < Max\ Episode$  do
    Initialize  $S$  ;
    Choose  $A$  from  $S$  using policy derived  $Q$  (e.g., using  $\epsilon$ -greedy) ;
    while  $S$  is not terminal do
        Take action  $A$ , observe  $R, S'$  ;
        Choose  $A'$  from  $S'$  using policy derived  $Q$  (e.g., using  $\epsilon$ -greedy) ;
        更新  $Q(S, A)$ ,  $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$  ;
         $S \leftarrow S'$ ,  $A \leftarrow A'$  ;
    end
end
  
```

---

关于 Sarsa 的收敛性, 最终可以得到  $Q(s, a) \leftarrow q_*(s, a)$ . 详细的在课件中, 之后补充收敛性的证明.

这里有一个问题, 这里使用系数  $\alpha$  会比使用  $\frac{1}{N}$  收敛更快一些. 但是实际上, 根据 incremental mean, 也就是式子3.1, 使用  $\frac{1}{N}$  才更好. 这个之后可以再做一下实验.

下面我们来看一个 Sarsa 的例子, 我们还是使用 Windy Gridworld Playground 的例子. 最终的结果如图4.8所示:

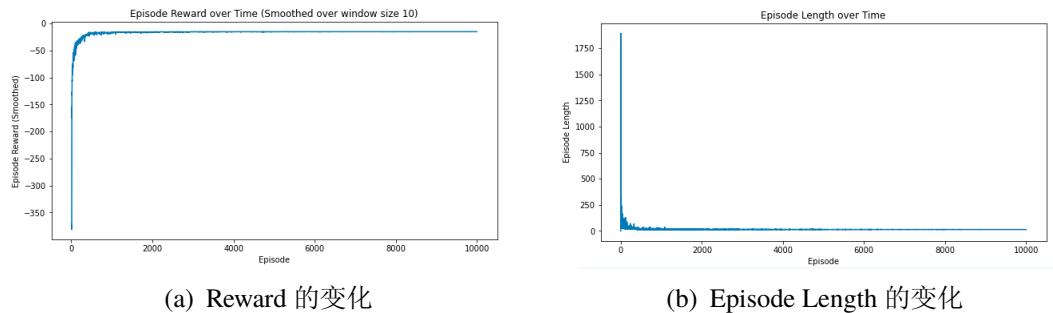


图 4.8: Windy Gridworld Playground 使用 Sarsa 效果图

可以看到, 相比图4.6 (使用 GLIE), 收敛更快一些.

### 4.3.2 n-Step Sarsa

我们还是和之前的想法一样, 是否可以走  $n$  步, 然后给出预测呢. 于是我们使用式子4.7来表示  $n$ -step Q-return.

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}) \quad (4.7)$$

最终的更新式子如4.8所示:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[q_t^{(n)} - Q(S_t, A_t)] \quad (4.8)$$

### 4.3.3 Eligibility Traces

同样, 我们会考虑  $n$  取多少的时候, 结果是最优的呢. 实际上, 我们可以不指定一个特定的  $n$ , 而是将其平均起来. 那么, 是否有一种有效的方式, 可以让我们将所有的 time-steps 全部合并起来.

我们使用  $\lambda$ -return,  $q^\lambda$  可以将  $n$ -step returns  $q_t^{(n)}$  都合起来. 其中  $q^\lambda$  的计算方式如式4.9所示:

$$q_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)} \quad (4.9)$$

之后, 我们就可以使用  $q_t^\lambda$  来更新  $Q(S_t, A_t)$  了, 更新的式子如4.10所示:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[q_t^\lambda - Q(S_t, A_t)] \quad (4.10)$$

当然, 使用上面这种方式进行更新存在一个问题, 也就是需要完成整个 episode 才可以进行更新. 为了可以每一步进行更新, 我们使用 eligibility traces 的方法. 关于 eligibility traces 与  $q_t^\lambda$  等价的证明, 可以参考上一章的证明.

之前我们是对每一个 state 需要保存一个参数, 这里我们是要对每一个  $(S, A)$  保存一个参数. 其中参数的计算如式4.11所示:

$$E_t(s, a) = \begin{cases} 0 & t = 0 \\ \gamma \lambda E_{t-1}(s, a) & S_t \neq s \text{ or } A_t \neq a \\ \gamma \lambda E_{t-1}(s, a) + 1 & S_t = s \text{ and } A_t = a \end{cases} \quad (4.11)$$

之后  $Q(s, a)$  的更新我们就按照式4.12进行更新:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha E_t(s, a)[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (4.12)$$

详细的  $Sarsa(\lambda)$  的算法如下所示, 我们有两点是需要注意的:

- 在每一个新的 episode, 都要进行  $E_t(s, a) = 0, \forall s \in S, a \in A(s)$  (仔细看代码中的 **eligibility traces** 初始化的位置);
- 每一次更新 Q, 不仅要更新这一次的二元组  $(S, A)$ , 其他的也是要进行更新的 (看代码中的第二重循环).

下面是 Sarsa( $\lambda$ ) 的算法流程:

---

**Algorithm 2: Sarsa( $\lambda$ ) 算法简介**


---

```

Initialize  $Q(s, a)$ ,  $\forall s \in S, a \in A(s)$ ,  $Q(s, a) = 0$ ,  $N = 0$  ;
while  $N < Max\ Episode$  do
    初始化  $E_t(s, a) = 0$ ,  $\forall s \in S, a \in A(s)$  ;
    Initialize  $S$  ;
    Choose  $A$  from  $S$  using policy derived  $Q$  (e.g., using  $\epsilon$ -greedy) ;
    while  $S$  is not terminal do
        Take action  $A$ , observe  $R, S'$  ;
        Choose  $A'$  from  $S'$  using policy derived  $Q$  (e.g., using  $\epsilon$ -greedy) ;
         $E_t(s, a) = E_{t-1}(s, a) + 1$  ;
         $\delta \leftarrow R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$  ;
        for  $\forall s \in S, a \in A(s)$  (注意这里是要更新所有的) do
            更新  $Q(S, A)$ ,  $Q(S, A) \leftarrow Q(S, A) + \alpha \delta E(s, a)$  ;
            更新  $E(s, a)$ ,  $E(s, a) \leftarrow \gamma \lambda E(s, a)$ 
        end
         $S \leftarrow S'$ ,  $A \leftarrow A'$ ;
    end
end

```

---

我们还是使用 Windy Gridworld Playground 来作为示例, 最终的结果如图4.9所示.

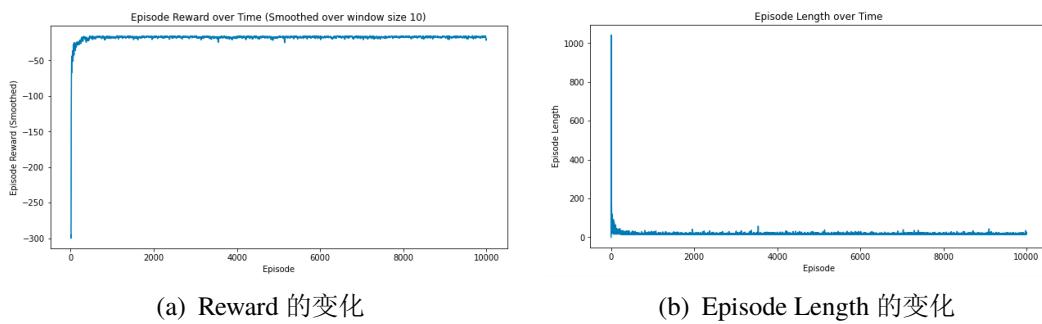


图 4.9: Windy Gridworld Playground 使用 Sarsa( $\lambda$ ) 效果图

#### 4.3.4 Sarsa 与 Sarsa( $\lambda$ ) 的比较

这里我们再用一个简单的例子, 来对比一下 Sarsa 与 Sarsa( $\lambda$ ). 其中我们要注意 Sarsa( $\lambda$ ) 更新速度比 Sarsa 更快.

下面是对环境的简单介绍:

- 有三个状态 A, B, C. 就是从 A->B->C;

- A->B 的 reward=0, B->C 的 reward=1;

- 初始化:

- $V(A) = V(B) = V(C) = 0;$
- $\alpha = 1, \gamma = 1;$
- $E(A) = E(B) = E(C) = 0$ , 这是 eligibility traces, 用在 Sarsa( $\lambda$ ) 的时候.
- $\lambda = 0.5$ , 用在 Sarsa( $\lambda$ ) 的时候, 更新 eligibility traces 的时候用到.

使用 Sarsa 的时候:

- 从 A->B, reward=0, 于是  $V(A) = V(A) + \alpha[0 + \gamma V(B) - V(A)] = 0;$

- 从 B->C, reward=1, 于是  $V(B) = V(B) + \alpha[1 + \gamma V(C) - V(B)] = 1;$

可以看到, 每一轮只更新了一个 state 的值, 也就是这里只会更新  $V(B)$  的值.

$V(A)$  需要下一轮的时候才会有更新.

使用 Sarsa( $\lambda$ ) 的时候:

- 从 A->B

- $E(A) = E(A) + 1 = 1;$
- $\delta = 0 + \gamma V(B) - V(A) = 0;$
- 更新 Q 值 (对于访问过的所有 state)
  - $V(A) = V(A) + \alpha E(A)\delta = 0;$

- 从 B->C

- 更新 eligibility traces
  - $E(A) = \lambda\gamma E(A) = 0.5;$
  - $E(B) = E(B) + 1 = 1;$
- $\delta = 1 + \gamma V(C) - V(B) = 1;$
- 更新 Q 值 (对于访问过的所有 state, 这里有 A 和 B)
  - $V(A) = V(A) + \alpha E(A)\delta = 0.5;$
  - $V(B) = V(B) + \alpha E(B)\delta = 1;$

可以看到, 使用 Sarsa( $\lambda$ ) 的时候, 在第一轮的时候, 由于 eligibility traces 的关系, 所有所有访问过的状态都会进行更新. 也就是我们在最后获得了 reward=1 的反馈, 但是我们 state=A 也就是可以得到更新.

## 4.4 Off-Policy Learning

Off-Policy 的想法就是我们使用策略  $\mu$  来进行模拟, 并使用  $\mu$  得到的观测值来更新策略  $\pi$ .

例如我们现在按照策略  $\mu(a|s)$  来进行游戏, 得到一系列观测值,  $\{S_1, A_1, R_2, \dots, S_T\} \sim \mu$ . 接着我们使用这一系列的观测值来更新策略  $\pi$ .

为什么 Off-Policy 的方法是重要的:

- 策略  $\mu$  可以是一些成熟的行为, 例如是人类的行为, 于是可以让 agent 通过观察人类的行为或者其他 agent 的行为进行学习;
- 可以重复利用 old policies( $\pi_1, \pi_2, \dots, \pi_{t-1}$ ) 产生的数据 (这个在 DQN 的时候会用到);
- 可以用一个探索性的策略进行探索, 但是最后学习一个确定性策略 (q-Learning 的主要思想);
- 可以使用一个策略进行采样, 同时学习多个策略;

#### 4.4.1 Importance Sampling

上面我们说到我们可以通过在策略  $\mu$  中进行抽样, 来更新策略  $\pi$ . 但是, 因为从不同的策略进行抽样, 所以更新的时候需要一个权重. 这里就需要使用 **importance sampling**.

我们现在有从  $Q$  中进行抽样得到的数据  $X$ , 并且用这个  $X$  可以得到  $f(X)$ . 我们要求  $f(X)$  在  $P$  中的期望, 即  $E_{X \sim P}[f(X)]$ . 这个时候因为  $X$  是从  $Q$  中进行抽样的, 所以我们不能直接将  $f(X)$  进行求平均. 解决的方法是在  $f(x)$  前面乘一个系数 (权重)  $\frac{P(X)}{Q(X)}$ . 也就是 ( $P$  中出现的概率)/( $Q$  中出现的概率).

我们将上面的  $Q$  与  $P$  与行为策略  $\mu$  和最终策略  $\pi$  结合起来.

- $Q$  对应着我们行为策略  $\mu$ , 我们根据策略  $\mu$  得到样本  $R$ , 可以计算  $E(R_\mu | S = s, A = a)$ . 这对应这在  $Q$  中抽样得到  $X$ .
- 我们希望求  $E(R_\pi | S = s, A = a)$ , 这个时候需要乘权重.
- $E(R_\pi | S = s, A = a) = E\left(\frac{\pi(a|S=s)}{\mu(a|S=s)}R_\mu | S = s, A = a\right)$ .

关于 **importance sampling** 的证明如式子4.13所示:

$$\begin{aligned}
 & E_{X \sim P}[f(X)] \\
 &= \sum P(X)f(X) \\
 &= \sum Q(X)\frac{P(X)}{Q(X)}f(X) \\
 &= E_{X \sim Q}\left[\frac{P(X)}{Q(X)}f(X)\right]
 \end{aligned} \tag{4.13}$$

#### 4.4.2 Importance Sampling for Off-Policy Monte-Carlo

有了上面的 **importance sampling** 之后, 我们就可以实现 off-policy 了.

- 我们使用策略  $\mu$  进行模拟, 得到一系列观测值,  $\{S_1, A_1, R_2, \dots, S_T\} \sim \mu$ ;
- 使用上面的观测值计算  $G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n}$  (这个是策略  $\mu$  的, 不能直接用来更新策略  $\pi$ );
- 为了可以使得  $G_t$  可以用来更新  $\pi$ , 我们将他们乘上系数  $G_t^\pi$ ;

- $G_t^\pi = \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \dots \frac{\pi(A_T|S_T)}{\mu(A_T|S_T)} G_t$ ;
- 之后可以使用  $G_t^{\pi_i}$  进行更新  $V(s)$  或是  $Q(s, a)$ , 这个就是对策略  $\pi$  的评价;
- 更新的式子为,  $V(S_t) \leftarrow V(S_t) + \alpha(G_t^{\pi_i} - V(S_t))$ ;

但是, 因为 MC 方法的方差本来就比较大, 现在给了一个系数之后 (这些概率也是估计), 方差就更大了.

关于 MC 的 off-policy 的实验, 我们这里设置执行的策略  $\mu$  是  $\epsilon$ -greedy 策略,  $\pi$  是 greedy 策略. 这个时候,  $\pi(a|s)$  只有两个值, 分别是 0 或是 1. 也就是一旦策略  $\pi$  和策略  $\mu$  选择的 action 不同, 那么  $\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} = 0$ .

我简单说明一下整体的思路, 用下面的例子做一下说明.

- 使用策略  $\mu$ , 得到如下观察值,  $S_1, A_1, R_2, S_2, A_2, R_3, S_3$ .
- 更新  $Q(S_2, A_2) = Q(S_2, A_2) + \alpha[R_3 - Q(S_2, A_2)]$ , 这里  $G = R_3$ .
- 更新  $Q(S_1, A_1) = Q(S_1, A_1) + \alpha[\frac{\pi(A_2|S_2)}{\mu(A_2|S_2)}(R_2 + \lambda * R_3) - Q(S_1, A_1)]$ , 这里  $G = R_2 + \lambda * R_3$ .

在更新  $Q(S_2, A_2)$  的时候, 是没有 importance sampling 的系数的, 这是因为这里的  $R_3$  可以看成就是在  $(S_2, A_2)$  获得的 reward. 但是在更新  $Q(S_1, A_1)$  的时候, 需要乘上系数, 这是因为此时有两个 reward, 分别是  $R_2$  和  $R_3$ . 其中  $R_3$  是与  $A_2$  有关的, 这个  $A_2$  又是从策略  $\mu$  中选的, 所以要乘系数. 但是, 这里系数的位置有两种, 一种对应课件, 一种对应书.

- 课件:  $Q(S_1, A_1) = Q(S_1, A_1) + \alpha[\frac{\pi(A_2|S_2)}{\mu(A_2|S_2)}(R_2 + \lambda * R_3) - Q(S_1, A_1)]$
- 书本:  $Q(S_1, A_1) = Q(S_1, A_1) + \alpha \frac{\pi(A_2|S_2)}{\mu(A_2|S_2)} [R_2 + \lambda * R_3 - Q(S_1, A_1)]$

但是不管是上面哪一种, 在 10000 轮之后, 结果都没有收敛 (暂时不确定是这个方法本来就是不会收敛, 还是我的代码存在问题). 下面是代码和关于书本上更新公式的来源.

- 为什么 Q-learning 不用重要性采样
- 实验代码, 05\_Importance\_Sampling\_Random\_MC\_Windy\_Gridworld.ipynb

#### 4.4.3 Importance Sampling for Off-Policy TD

上面我们说了 MC 方法的方差比较大, 可能不适用于 **importance sampling**. 于是我们就想到使用 TD 方法, 他的方差会比 MC 的要小.

- 我们使用策略  $\mu$  进行模拟, 在状态  $S_t$  执行  $A_t$  得到,  $R_{t+1}$ , 并进入下一个状态  $S_{t+1}$ ;
- 使用上面的观测值计算  $R_{t+1} + \gamma V(S_{t+1})$  (这个是策略  $\mu$  的, 不能直接用来更新策略  $\pi$ );
- 乘上系数  $\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}$ ;
- 更新的式子为,  $V(S_t) \leftarrow V(S_t) + \alpha[\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}[R_{t+1} + \gamma V(S_{t+1})] - V(S_t)]$ ;



我觉得上面式子有问题, 这里是 1-step, 所以不需要乘权重, 类似于 Q-Learning 不要乘权重

上面的方法, 与 MC 的比起来, 方差会小很多 (因为只需要预测一步, 计算一步). 同样, 在使用的时候, 因为我们不知道转移概率, 所以我们会预估  $Q(s, a)$  值来转换为策略.

我们使用行为策略  $\mu$  为随机策略, 优化策略  $\pi$  是 greedy 策略来进行实验. 此时  $Q(s, a)$  的更新如式子4.14所示 (这里的更新的式子已经和下面的 Q Learning 的更新式子很相似了):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t)] \quad (4.14)$$

其中:

- $A'$  是从策略  $\pi$  中选的, 也就是在看到状态  $S_{t+1}$  的时候,  $\pi$  所做的决定.
- 但是, 下一步的动作, 我们还是通过策略  $\mu$ (随机策略) 来选择, 也就是  $A_{t+1} \sim \mu(\cdot | S_{t+1})$ .

实验步骤与结果在链接, Importance Sampling TD. 最终策略  $\pi$  是可以优化到较好的值的.

#### 4.4.4 Q-Learning

在 Off-Policy 中, 会有两个策略, 我们根据一个策略 ( $\mu$ ) 进行行动, 来指导另外一个策略 ( $\pi$ ) 的的更新. 我们在这里将执行的策略  $\mu$  设置为  $\epsilon$ -greedy 策略,  $\pi$  设置为 greedy 策略. 这个时候, 该算法中的行为策略  $\mu$  与目标策略  $\pi$  是从同一个值  $\mathbf{Q}$  中学习得到, 这样两个策略能同时得到改善.

于是,  $Q(S_t, A_t)$  的更新过程如式子4.15所示 (与式子4.14是一样的):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t)] \quad (4.15)$$

其中:

- $A'$  是从策略  $\pi$  中选的, 也就是在看到状态  $S_{t+1}$  的时候,  $\pi$  所做的决定.
- 但是, 下一步的动作, 我们还是通过策略  $\mu$  来选择, 也就是  $A_{t+1} \sim \mu(\cdot | S_{t+1})$ .

注意, 这里是不需要乘权重系数的, 因为这里是 1-step 的情况, 我们会在后面详细说明, 并给出 2-step 和 3-step 的情况.

我们进一步化简上面的式子4.15. 我们知道策略  $\pi$  是 greedy 策略, 也就是  $\pi(S_{t+1}) = \arg \max_{a'} Q(S_{t+1}, a')$ . 于是我们将  $A' = \arg \max_{a'} Q(S_{t+1}, a')$  代入式子4.15进行化简, 如式子4.16所示:

$$\begin{aligned}
 & R_{t+1} + \gamma Q(S_{t+1}, A') \\
 & = R_{t+1} + \gamma Q(S_{t+1}, \arg \max_{a'} Q(S_{t+1}, a')) \\
 & = \gamma \max_{a'} Q(S_{t+1}, a')
 \end{aligned} \tag{4.16}$$

于是,  $Q(S_t, A_t)$  的更新式子可以化简为4.17:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)] \tag{4.17}$$

于是, Q-Learning 的整体的算法如下所示.

---

**Algorithm 3: Q Learning**


---

Initialize  $Q(s, a)$  arbitrarily;

**while**  $S$  is not terminal **do**

    Initialize  $s$ ;

**while**  $s$  is not terminal **do**

        Choose  $A$  from  $S$  using policy driven from  $Q$  (e.g.,  $\epsilon$ -greedy 策略);

        Take action  $A$ , observe  $R(reward)$ ,  $S'$ ;

$Q(S, A) \leftarrow Q(S, A) + \alpha \cdot (R + \gamma \cdot \max_a Q(S', a) - Q(S, A))$ ;

$S \leftarrow S'$ ;

**end**

**end**

---

我们同样使用 Q-Learning 在 Windy Gridworld 上面进行实验. 实验代码链接为, [05\\_Q-Learning\\_Windy\\_Gridworld.ipynb](#). 最终的结果如图4.10所示.

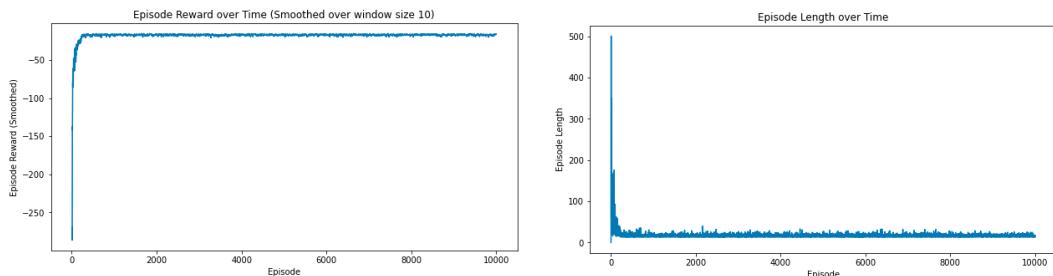


图 4.10: Windy Gridworld Playground 使用 Q-Learning 效果图

#### 4.4.5 N-step Q-Learning

我们在这里首先说明一下, 为什么 Q-Learning 是 Off-policy 的方法, 但是却没有 importance sampling.

- 在 Q-Learning 的时候, 我们现在更新  $Q(S_t, A_t)$ , 虽然这里的  $A_t$  是根据策略  $\mu$  得到的, 但是我们可以将其看成是一组数据, 即在状态  $S_t$  执行  $A_t$  会获得奖励  $R_t$ . 接着在  $S_{t+1}$  我们使用的是根据策略  $\pi$  获得的动作  $A'$ , 所以整体就是在更新策略  $\pi$  下的  $Q$ , 所以不需要进行 importance sampling.
- 同时, 我们也可以认为此时动作  $A$  已经确定了, 是没有随机性在里面的.
- 现在如果是进行 2 步, 这个时候第二个动作选择就有随机性了, 就需要进行 importance sampling.

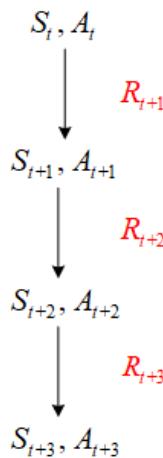


图 4.11: N-Step 的解释.

上面我们说了, 因为 Q-Learning 是 1-step 的情况, 下面我们来看一下 2-step 和 3-step 的情况. 这个时候就需要进行 importance sampling 了. 我们使用图 4.11 来说明 1-step, 2-step 和 3-step.

- 1-step:
  - 在状态  $S_t$  的情况下, 根据策略  $\mu$  选择了动作  $A_t$ , 来到下一个状态  $S_{t+1}$ ;
  - 在状态  $S_{t+1}$  的情况下, 根据策略  $\pi$  选择动作  $A'$  进行更新 (这个动作只用于更新, 不会被真正执行).
- 2-step:
  - 在状态  $S_t$  的情况下, 根据策略  $\mu$  选择了动作  $A_t$ , 来到下一个状态  $S_{t+1}$ ;
  - 在状态  $S_{t+1}$  的情况下, 根据策略  $\mu$  选择了动作  $A_{t+1}$ , 来到下一个状态  $S_{t+2}$ , 此时, 这个动作  $A_{t+1}$  的选择就与策略  $\mu$  有关, 因为在状态  $S_{t+1}$  的情况下, 有不同的动作的选择;
  - 在状态  $S_{t+2}$  的情况下, 根据策略  $\pi$  选择动作  $A'$  进行更新 (这个动作只用于更新, 不会被真正执行).
- 3-step:
  - 在状态  $S_t$  的情况下, 根据策略  $\mu$  选择了动作  $A_t$ , 来到下一个状态  $S_{t+1}$ ;
  - 在状态  $S_{t+1}$  的情况下, 根据策略  $\mu$  选择了动作  $A_{t+1}$ , 来到下一个状态  $S_{t+2}$ , 此时, 这个动作  $A_{t+1}$  的选择就与策略  $\mu$  有关, 因为在状态  $S_{t+1}$  的

情况下,有不同的动作的选择;

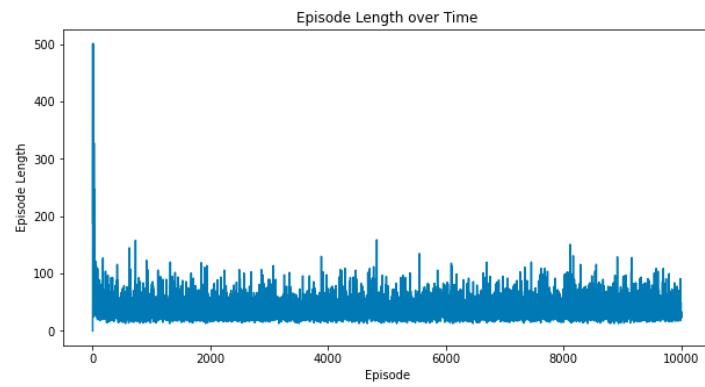
- 在状态  $S_{t+2}$  的情况下, 根据策略  $\mu$  选择了动作  $A_{t+2}$ , 来到下一个状态  $S_{t+3}$ , 此时, 这个动作  $A_{t+2}$  的选择就与策略  $\mu$  有关, 因为在状态  $S_{t+2}$  的情况下,有不同的动作的选择;
- 在状态  $S_{t+3}$  的情况下, 根据策略  $\pi$  选择动作  $A'$  进行更新 (这个动作只用于更新, 不会被真正执行).

上面大致描述了三种的步骤, 以及什么时候需要使用 importance sampling 的内容. 下面我们对上面三个内容进行实验. 注意, 下面所有的实验, 行为策略  $\mu$  设置为  $\epsilon$ -greedy,  $\pi$  设置为 greedy 策略.

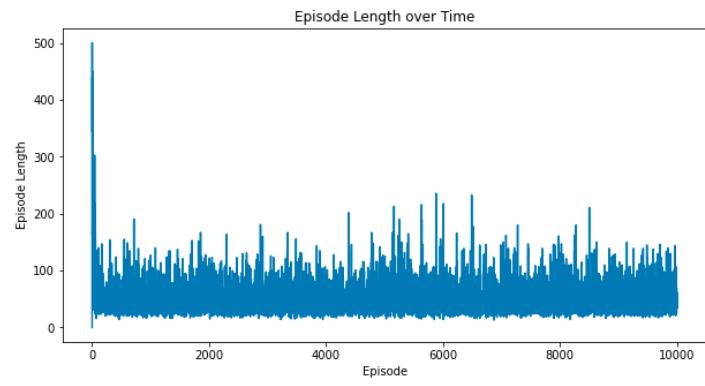
实际上, 最终的结果会与策略  $\mu$  的选择有关. 如果  $\mu$  选择随机策略, 在这里也就是每一个 action 的概率是 0.25, 那么因为权重  $\frac{\pi(\cdot|s)}{\mu(\cdot|s)}$  的取值只有 0 和 4. 那么在 n-step 的时候, 例如 3-step 的时候, 很可能前面的权重系数会变得很大, 最后 Q 和可能会不收敛, 趋于无穷.

关于 2-step Q-Learning 的实验, 这里  $Q(S_t, A_t)$  的更新我尝试了下面的四种方式, 这一部分的实验代码链接为, 2-Step TD:

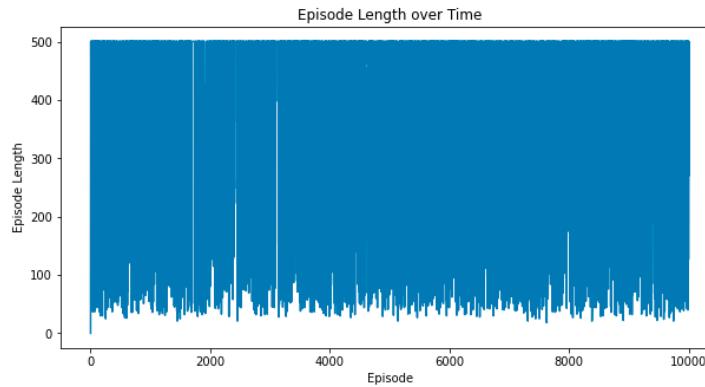
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} [R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}, A')] - Q(S_t, A_t)$ , 如图4.12(a), 收敛;
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \gamma R_{t+2} + \gamma^2 Q(S_{t+2}, A')] - Q(S_t, A_t)$ , 如图4.12(b), 收敛;
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [\frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} (R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}, A')) - Q(S_t, A_t)]$ , 如图4.12(c), 不收敛;
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}, A')] - Q(S_t, A_t)$ , 如图4.12(d), 收敛;



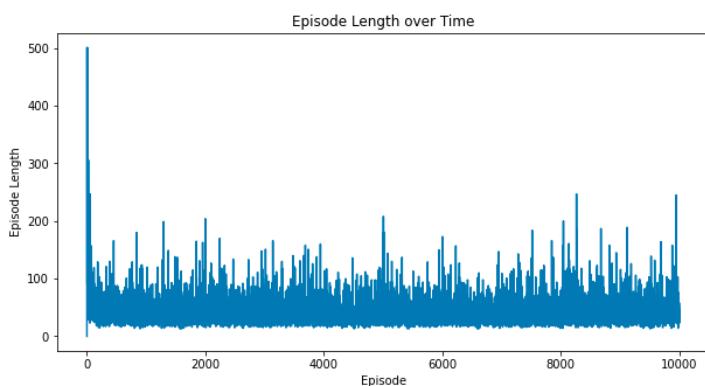
(a) Episode Length 的变化



(b) Episode Length 的变化



(c) Episode Length 的变化

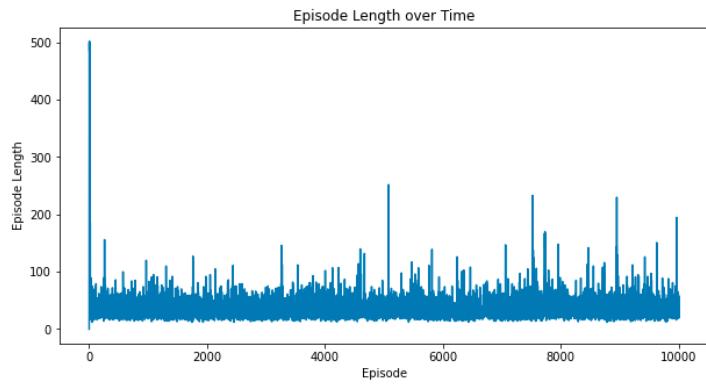


(d) Episode Length 的变化

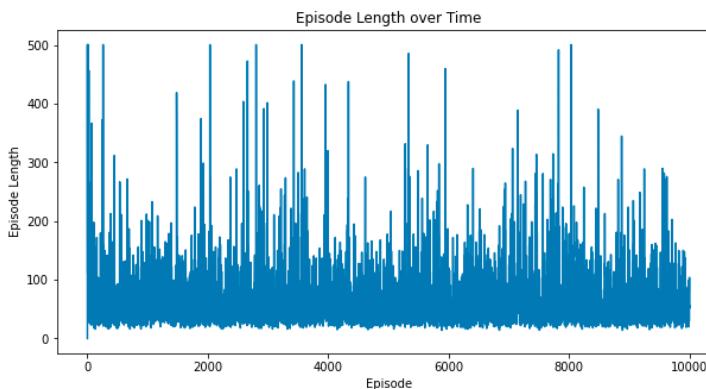
图 4.12: 2-step Q-Learning 实验

关于 **3-step Q-Learning** 的实验, 我们同样尝试下面四种方式: 这一部分的实验代码链接为, **3-Step TD**:

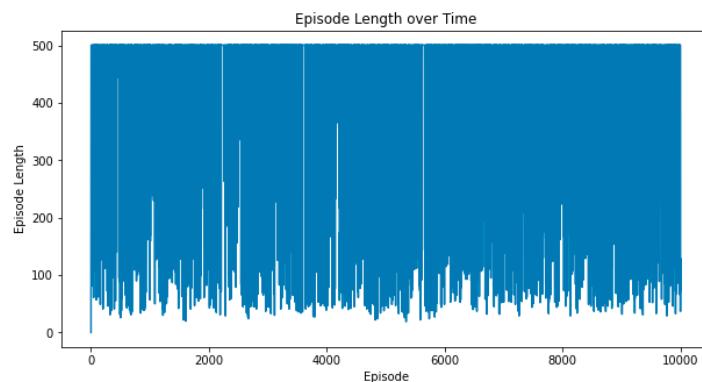
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \frac{\pi(A_{t+2}|S_{t+2})}{\mu(A_{t+2}|S_{t+2})} [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 Q(S_{t+3}, A') - Q(S_t, A_t)]$ , 如图**4.13(a)**, 收敛;
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \gamma R_{t+2} + \frac{\pi(A_{t+2}|S_{t+2})}{\mu(A_{t+2}|S_{t+2})} \gamma^2 R_{t+3} + \gamma^3 Q(S_{t+3}, A') - Q(S_t, A_t)]$ , 如图**4.13(b)**, 不收敛;
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [\frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \frac{\pi(A_{t+2}|S_{t+2})}{\mu(A_{t+2}|S_{t+2})} (R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 Q(S_{t+3}, A')) - Q(S_t, A_t)]$ , 如图**4.13(c)**, 特别不收敛;
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 Q(S_{t+3}, A') - Q(S_t, A_t)]$ , 如图**4.13(d)**, 收敛;



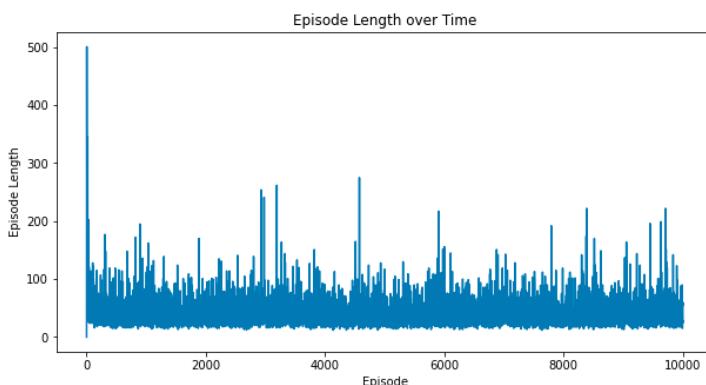
(a) Episode Length 的变化



(b) Episode Length 的变化



(c) Episode Length 的变化



(d) Episode Length 的变化

图 4.13: 3-step Q-Learning 实验

# 第 5 章 Value Function Approximation

## 内容提要

- ❑ Incremental Methods
- ❑ Deep Q-Network
- ❑ Batch Reinforcement Learning

在上一章的时候, 我们在最后讲了 Q Learning, 这已经可以帮助我们解决一些优化问题. 在 Q Learning 中, 我们通过更新  $Q(s, a)$ , 来获得最终的策略. 在之前的实验中, 所有的  $Q(s, a)$  是通过表格的形式进行存储的. 寻找某一个状态下, 使用某个动作的价值需要进行查表操作, 但在许多实际问题中, 拥有大量状态和行为空间. 此时查表得方法变得不是很有效.

因此在这一章中, 我们主要来解决价值函数得近似表示的问题. 对于状态和行为空间都比较大的情况下, 精确获得各种  $V(s)$  和  $Q(s, a)$  几乎是不可能的. 这时候需要找到近似的函数, 具体可以使用线性组合, 神经网络以及其他方法来近似价值函数:

例如, 式子5.1就是  $Q(s, a)$  的近似表达:

$$\hat{Q}(s, a, \mathbf{w}) \approx Q_\pi(s, a) \quad (5.1)$$

其中  $w$  是近似函数的参数. 这里使用近似函数  $\hat{Q}(s, a, \mathbf{w})$  的好处在于, 对于未知的状态  $s$  也可以给出结果. 传统的查表对于未知情况是不能给出结果的.

## 5.1 Incremental Methods

下面举例子我们都使用近似  $V(s)$  作为例子. 近似  $Q(s, a)$  也是类似.

### 5.1.1 Feature Vectors

首先, 对于每一个状态  $S$  来说, 我们都可以使用一个向量  $x(S)$  来进行表示. 例如式子5.2所示:

$$x(S) = \begin{pmatrix} x_1(S) \\ \vdots \\ x_n(S) \end{pmatrix} \quad (5.2)$$

其中  $x_i(s)$  可以表示:

- 距离环境的各种距离;
  - agent 此时的自身状态, 速度, 方向等.
- 但是这些  $x_i(s)$  需要人为进行定义.

### 5.1.2 Linear Value Function Approximation

在有了上面的  $x(S)$  之后, 我们可以通过一个线性式子来对 value function 进行估计. 式子如5.3所示:

$$\hat{v}(s, \mathbf{w}) = x(S)^\top \mathbf{w} = \sum_{i=1}^n x_i(S) w_i \quad (5.3)$$

我们现在需要求出参数  $w$ . 同时, 我们的目标是希望  $\hat{v}(s, \mathbf{w}) \approx v_\pi(S)$ . 于是我们有了 object function, 如式子5.4所示:

$$J(w) = E_\pi[(v_\pi(S) - x(S)^\top w)^2] \quad (5.4)$$

于是, 我们就可以通过梯度下降法来更新参数  $w$ .

$$\begin{aligned} \Delta w &= -\frac{1}{2}\alpha \nabla_w J(w) \\ &= \alpha E_\pi[(v_\pi(S) - x(S)^\top w)] \nabla_w (x(S)^\top w) \\ &= \alpha(v_\pi(S) - x(S)^\top w)x(S) \end{aligned} \quad (5.5)$$

### 5.1.3 Table Lookup Features

现在我们在来回看一下查表的方法, 会发现这个是上面 linear value function approximation 的一种特殊情况. 比如有 4 个 state, 那么  $x(S)$  就会写成如式子5.6所示:

$$x^{table}(S) = \begin{pmatrix} 1(S = s_1) \\ 1(S = s_2) \\ 1(S = s_3) \\ 1(S = s_4) \end{pmatrix} \quad (5.6)$$

比如说, 若当前在状态  $s = 1$  的情况下, 则就是式子5.7:

$$x^{table}(S = 1) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (5.7)$$

参数  $w_1$  此时就是对这个状态  $s_1$  价值的估计:

$$\hat{v}(S = 1, w) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}^\top \cdot \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix} \quad (5.8)$$

于是, 对于一般情况下, 查表法可以写成下面的线性近似的方法:

$$\hat{v}(S, w) = \begin{pmatrix} 1(S = s_1) \\ \vdots \\ 1(S = s_n) \end{pmatrix}^\top \cdot \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} \quad (5.9)$$

### 5.1.4 Incremental Prediction Algorithms

对于上面的式子5.5, 要想更新参数  $w$ , 还存在一个问题, 就是我们不知道  $v_\pi(s)$ . 于是我们想到使用每次获得的 reward 来代替  $v_\pi(s)$ .

对于 MC 方法来说, 我们使用  $\textcolor{red}{G_t}$  来代替  $v_\pi(s)$ . 于是更新的式子如5.10所示:

$$\Delta w = \alpha(\textcolor{red}{G_t} - \hat{v}(S_t, w))x(S) \quad (5.10)$$

对于 TD(0) 来说, 我们使用  $\textcolor{red}{R_{t+1}} + \gamma \hat{v}(S_{t+1}, w)$  来代替  $v_\pi(s)$ . 于是更新的式子如5.11所示:

$$\Delta w = \alpha(\textcolor{red}{R_{t+1}} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w))x(S) \quad (5.11)$$

#### 5.1.4.1 MC Learning with Value Function Approximation

我们详细说明一下上面使用 MC 方法来更新参数  $w$ . 使用 MC 方法获得的  $G_t$  是无偏的 (unbiased), 但是会有较大的方差.

在实际使用中, 我们完成了一轮游戏, 会获得如下的值:

$$(S_1, G_1), (S_2, G_2), \dots, (S_T, G_T) \quad (5.12)$$

有了这些值之后, 我们可以把问题看成一个监督学习的问题. 上面的就是数据集, 我们通过式子5.10来更新参数  $w$ , 从而获得对  $v_\pi(s)$  的估计.

### 5.1.4.2 TD Learning with Value Function Approximation

使用 TD 算法来更新参数  $w$ . 此时每一次获得的  $R_{t+1} + \gamma \hat{v}(S_{t+1}, w)$  对于真实的  $v_\pi(s)$  来说是有偏估计, 但是方差较小.

与上面 MC 方法不同的是, 在 TD 的时候, 每执行一步, 得到一个  $(S_t, R_{t+1} + \gamma \hat{v}(S_{t+1}, w))$ , 就按照式子5.11更新一次参数  $w$ .

### 5.1.5 Control with Value Function Approximation

有了上面对于状态函数的估计, 下面就可以用来对策略  $\pi$  进行更新. 更新得过程如图5.1所示:

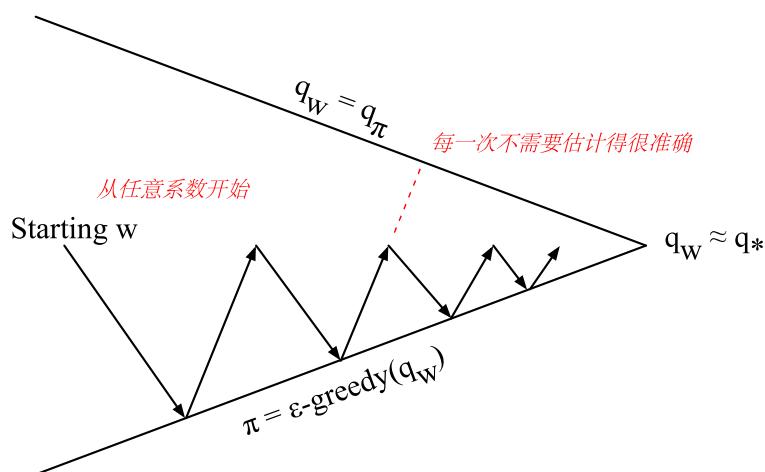


图 5.1: 更新策略步骤.

对上面的过程进行简单的描述:

- 我们从从任意一个系数  $w$  开始, 估计当前策略  $\pi$  的 value function, 这里是估计  $Q_\pi(s, a)$ .
- 我们不需要估计的太准确, 有了估计的  $\hat{Q}(s, a)$  之后, 使用  $\epsilon$ -greedy 策略更新策略.
- 有了新的策略之后重新估计  $Q_\pi(s, a)$ , 依次继续下去.

### 5.1.6 The example of Mountain Car

下面按照上面的方法, 来进行一个实验. 这里使用的是 Mountain Car 来进行测试. Mountain Car 的整个场景如图5.2所示:

其中:



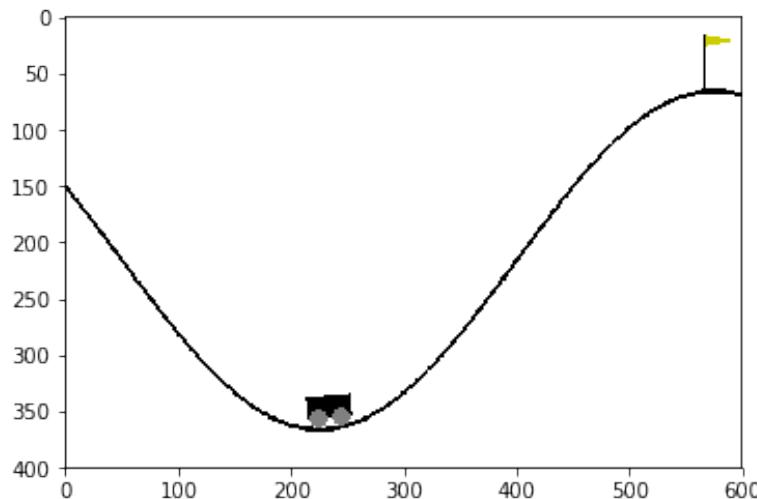


图 5.2: Mountain Car 简单介绍.

- **目标:** 到达右侧插旗的位置.
- **动作:** 正向的最大油门和反向的最大油门(向左走和向右走).
- 在这里, 单靠小车自身的动力是不足以在谷底由静止一次性冲上右侧目标位置的. 也就是不能通过一直向一个方向前进到达终点.
- 能完成任务的策略应该是, 当小车加速上升到一定位置时, 让小车回落, 同时反向加速, 使其加速冲向谷底, 到达终点.

我们使用 TD 的方法来进行估计, 最终的结果如下所示, 可以看到随着迭代的进行, 每一轮所走的步数是逐渐减少的:

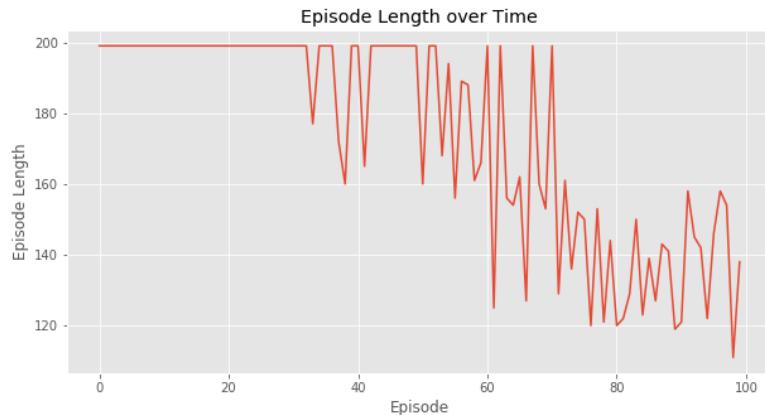


图 5.3: Mountain Car 结果介绍.

## 5.2 Batch Reinforcement Learning

在上面介绍的算法中, 存在一个问题, 就是数据利用不充分. 在更新过一次  $w$  之后, 我们所使用的数据之后就不会再用到了. 于是 batch method 的想法就是把之前的数据也收集起来. 在这类方法中, 很重要的一种方法就是 Deep Q-Networks.

下面是一些参考资料:

- Paper, Playing Atari with Deep Reinforcement Learning
- A pair of interrelated neural networks in Deep Q-Network
- REINFORCEMENT LEARNING (DQN) TUTORIAL

### 5.2.1 Deep Q-Networks (DQN)

DQN 会使用到两个技术, **experience replay** 和 **fixed Q-targets**. 在讲 DQN 之前, 首先我们回顾一下 Q Learning 的更新公式:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)] \quad (5.13)$$

在式子 5.13 中,  $Q(S_t, A_t)$  是使用表格进行表示的. 于是很自然我们想到可以把 Q-Learning 中的 Q Tabel 转换为网络表示,  $\hat{Q}(S, A, W)$ . 但是如果此时网络的输入是  $a$  和  $s$ , 我们很难计算  $\max_{a'} Q(S, a')$ . 于是在 DQN 中, 网络结构如图 5.4 所示, 目前我们探讨 **action** 是有限个数的情况.

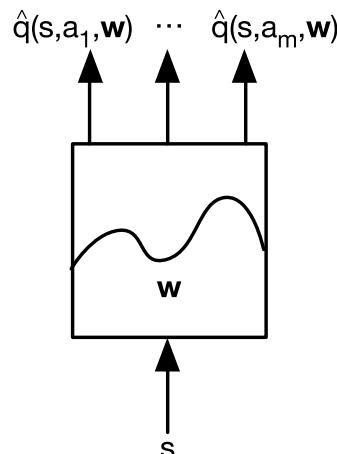


图 5.4: DQN 输入和输出.

其中:

- 输入是状态  $s$ , 如果是游戏, 可以输入游戏的画面.
- 如果有  $n$  个 action, 输出就是  $n$  个值, 分别对应  $\hat{q}(s, a_i, w)$ .

这个时候我们更新  $\hat{Q}(S, A, \theta)$  的时候就不是按照式子 5.13 的方式进行迭代了. 我们是使用梯度下降的方式进行更新.

$$\begin{aligned} L(\theta) &= E_\pi[(y - \hat{Q}(S, A, \theta))^2] \\ &= E_\pi[(r + \gamma \max_a \hat{Q}(S, a, \theta) - \hat{Q}(S, A, \theta))^2] \end{aligned} \quad (5.14)$$

### 5.2.1.1 Pair of Q-Networks: local and target

但是直接按照上面这么做网络的训练结果是不稳定的。这是因为式子5.14中的  $\hat{Q}(S, A, \theta)$  和 target value ( $r + \gamma \max_a \hat{Q}(S, a, \theta)$ ) 之间有关联。我的理解是，如果一个网络的 predict ( $\hat{Q}(S, A, \theta)$ ) 和 target ( $r + \gamma \max_a \hat{Q}(S, a, \theta)$ ) 都是由这个网络产生的，训练就会不稳定。

于是解决的方法就是同时训练两个 network，分别叫做 **Policy Network** 和 **Target Network**。其中：

- **Policy Network** 和 **Target Network** 结构完全一模一样。
- **Policy Network**
  - **Policy Network** 会产生 action.
  - **Policy Network** 会每一轮都进行更新.
  - $\hat{Q}(S, A, \theta)$  中的  $Q$  为 **Policy Network**.
- **Target Network**
  - 每 N 轮，**Target Network** 会直接拷贝 **Policy Network** 的参数.
  - $\max_a \hat{Q}(S, a, \theta)$  为 **Target Network**.

于是，上面式子5.14可以改为式子5.15。

$$\begin{aligned} L(\theta) &= E_{\pi}[(y - \hat{Q}(S, A, \theta))^2] \\ &= E_{\pi}[(r + \gamma \max_a \hat{Q}_{target}(S, a, \theta_{target}) - \hat{Q}_{policy}(S, A, \theta_{policy}))^2] \end{aligned} \quad (5.15)$$

### 5.2.1.2 Experience replay

为了减少数据之间的相关性 (reduce correlations)，在 DQN 时会用到一种叫 experience replay 的机制。该方法思想很简单：

- 在每走一步之后，我们会得到  $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ ，我们将  $e_t$  存储到一个数据集  $D$  中去。
- 每次更新 Q 的时候，我们会从  $D$  中进行 sample，然后使用 minibatch updates.
- 这个 memory 大小是固定的，也就是说新的记忆的加入会消除老的记忆。

### 5.2.1.3 DQN 算法整体流程

根据上面讲的，图5.5是整个 Deep Q-Networks 的流程。

下面是 DQN 的整个流程：

- 用 Policy Network 生成每一个 action 的概率，接着根据  $\epsilon$ -greedy 策略选取一个 action  $a_t$ ；
- 接着将得到的值  $(s_t, a_t, r_{t+1}, s_{t+1})$  存储再 memory  $D$  中；



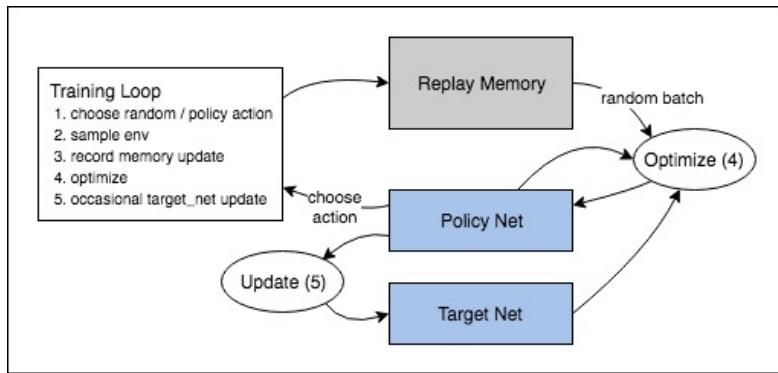


图 5.5: Deep Q-Networks 整体流程图.

- 从  $D$  中选取 mini-batch 的  $(s, a, r', s')$ ;
- 计算得到  $\hat{Q}_{policy}(s, a, \theta_{policy})$  和  $r' + \max_a \hat{Q}_{target}(s', a, \theta_{target})$ .
- 得到 loss function, 式子5.16.
- 使用梯度下降法更新参数  $\theta_{policy}$ .

$$L(\theta_{policy}) = E_{s,a,r^s,s^s \sim D} [(r' + \max_a \hat{Q}_{target}(s', a, \theta_{target}) - \hat{Q}_{policy}(s, a, \theta_{policy}))^2] \quad (5.16)$$

## 5.2.2 Deep Q-Networks (DQN) 的例子

因为在线上环境 Google Colab 里面无法渲染图像, 所以我选择使用 Cliff Walking PlayGround 来进行 Deep Q-Network 的实验.

### 5.2.2.1 Cliff Walking 环境介绍

Cliff Walking 总的环境如图5.6所示, 是一个 4\*12 大小的棋盘:

- 我们从 S 出发, 要达到 G;
- 每一个网格有四个 action, 分别是上下左右;
- 每走一步的 reward 是-1, 如果走到 cliff 那么 reward 是-100, 走到 G 没有特别的 reward;
- 同时若走到 The Cliff, 会回到 S 重新开始;

### 5.2.2.2 Pytorch 实现 Deep Q-Network

Cliff Walking 一共有 48 个状态, 每次环境返回的是 state 的编号, 也就是现在在哪个格点上面. 为了更好的模拟 Deep Q-Network, 我们不直接使用环境返回的 state. 我们将 state 转换为 one-hot 编码. 共 48 维. 来作为网络的输入. 所以, 我们这里的网络结构是 **48->24-12->4**.

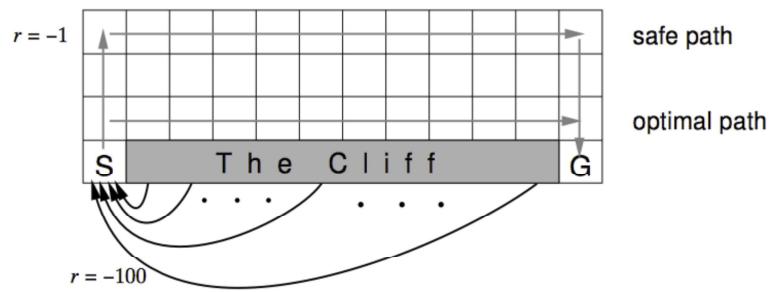


图 5.6: Cliff Walking Playground 介绍.

关于详细的训练过程及代码可以查看链接, [Pytorch 实现 Deep Q-Learning\(Cliff Walking PlayGround\)](#). 最终的实验结果如下所示, 图5.7是 Episode Length 的变化, 可以看到一开始没有收敛, 最后可以在 20 步以内完成一个 epoch.

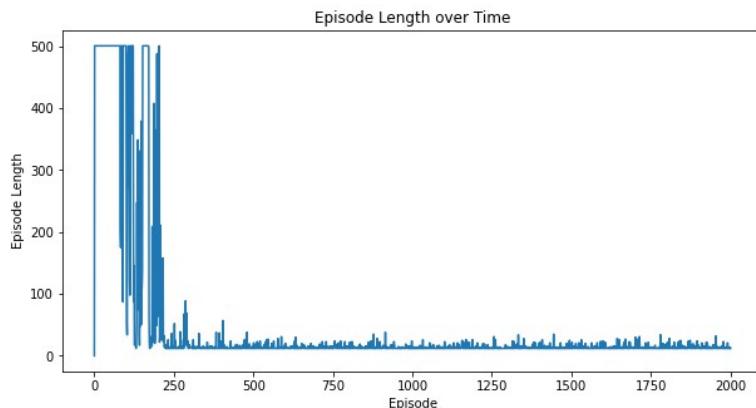


图 5.7: Pytorch 实现 Deep Q-Network 的结果.

# 第 6 章 Policy Gradient

## 内容提要

- Score Function
- Actor-Critic Policy Gradient
- REINFORCE 算法

在上一章的时候, 我们介绍了如何近似 value function ( $v_\pi$ ) 和 action-value function ( $Q_\pi$ ). 在有了 value function 之后, 我们使用  $\epsilon$ -greedy 来生成最终的策略  $\pi$ . 最终, 我们生成的策略是固定的. 因为使用了 **greedy** 的方式选择 **action**, 所以给定一个 **state**, 做出某个 **action** 的概率是 **100%**.

但是在这一部分, 我们直接近似策略, 而不是近似价值函数 (value function). 如式子6.1, 我们参数化策略函数  $\pi_\theta$ . 策略函数确定了在给定的状态和一定的参数设置下, 采取任何可能行为的概率. 因此事实上它是一个概率密度函数. 在实际应用策略产生行为时, 是按照这个概率分布进行行为采样的, 从而获得实际的行为. 这样, 每一个 action 执行的概率就不是 100% 了.

$$\pi_\theta(s, a) = P[a|s, \theta] \quad (6.1)$$

我们的最终要做的: 利用参数化的策略函数,  $\pi_\theta(s, a)$ , 通过调整参数  $\theta$  来得到一个较优策略. 较优策略的原则是, 遵循这个策略产生的行为可以得到较多的奖励. 具体的机制是设计一个目标函数, 对其使用梯度上升 (Gradient Ascent) 算法优化参数以最大化奖励.

## 6.1 Advantages of Policy-Based RL

这里介绍一个使用 Policy-Based 的方法的好处, 就是可以学到随机策略 (**stochastic policies**). 在基于 value function 的算法中, 一旦价值函数 (或者 q-table) 确定, 那么最终的策略也会是固定的. 有的时候随机策略是最优策略. 下面看两个例子.

### 6.1.1 Rock-Paper-Scissors

在石头剪刀布的游戏里, 一旦有一方是确定性的策略, 那么对手就可以针对这个策略. 所以, 这个时候最好的策略就是随机策略.

## 6.1.2 Aliased Gridworld

Aliased Gridworld 的游戏如下图6.1所示:

- 个体需要尽快找到钱袋, 而避免骷髅.
- 如果我们选择描述 state 的特征不当, 可能会出现两个灰色格子无法区分. 这就是状态重名 (Aliased).
- 在这里, 我们定义描述 state 的函数为,  $\phi(s, a)$ .

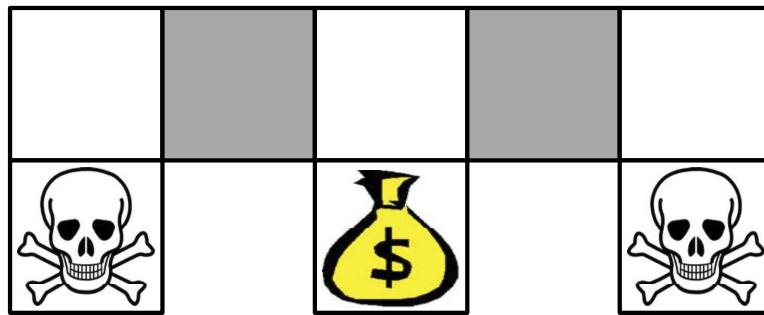


图 6.1: Aliased Gridworld 游戏介绍.

在上面这种情况下, 我们分别比较 **value-based RL** 和 **policy-based RL** 的不同. 在 **value-based RL** 的情况下, value function 的近似如式子6.2所示:

$$Q_\theta(s, a) = f(\phi(s, a), \theta). \quad (6.2)$$

在使用 **value-based RL** 的情况下, 最终会得到一个确定性的策略 (**deterministic policy**). 同时在状态重名 (Aliased) 的情况下, 会导致两个灰色格子 (grey state) 的 action 是一样的. 最终的确定性策略可能如图6.2所示:

这也就会导致 agent 陷入循环 (get stuck), 从而一直无法获得 money. 同时, 由于在训练的时候使用  $\epsilon$ -greedy 或是 greedy 的方法, 有可能 agent 长时间在左侧两个格子反复, 无法到达钱袋, 获取 reward.

在 **policy-based RL** 的情况下, 策略的近似 (parametrised policy) 如式子6.3所示, 最终得到的是每一个 action 的概率:

$$\pi_\theta(s, a) = g(\phi(s, a), \theta). \quad (6.3)$$

在使用 **policy-based RL** 的情况下, 最终可以得到一个随机性的策略 (**stochastic policy**). 这样, 虽然在状态重名 (Aliased) 的情况下, 但是在灰色格子中向两个

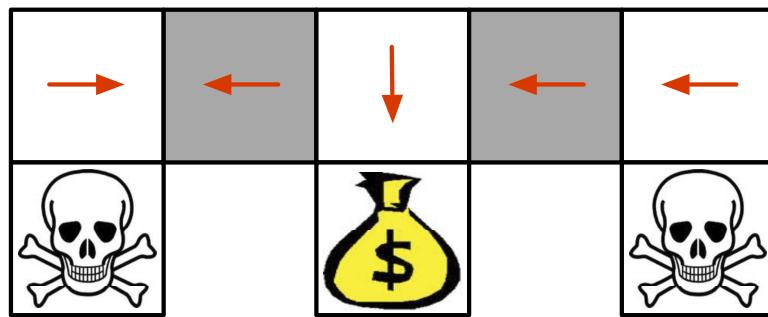


图 6.2: Aliased Gridworld 在 value-based 情况下可能的策略.

方向走的可能性都是存在的. 最终的策略可能如图6.3所示, 灰色格子上向左和向右走的概率都是 0.5:

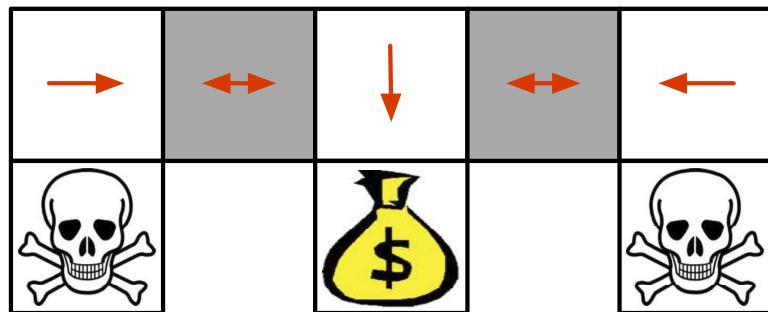


图 6.3: Aliased Gridworld 在 policy-based 情况下可能的策略.

## 6.2 Policy Objective Function and Optimisation

### 6.2.1 Policy Objective Function

我们的目标是给定一个带有参数  $\theta$  的策略函数,  $\pi_\theta(s, a)$ , 我们要找出最优的参数  $\theta$ . 那么如何衡量策略函数  $\pi_\theta$  的好坏呢, 我们希望使用这个策略函数  $\pi_\theta$  可以获得尽可能多的累计 reward. 于是针对不同的情况, 我们会设计不同的策略目标函数 (policy objective function),  $J_\theta$ . 下面三个式子的目标都是同一个目标, 都是试图描述个体在某一时刻的价值.

### 6.2.1.1 Using Start Value

在能够产生完整的 Episode 的环境下 (也就是个体可以达到最终状态的时候), 我们可以用这样一个值来衡量整个策略的优劣: 从某状态  $s_1$  算起, 直到终止状态个体获得的累计奖励. 这个值称为 **start value**.

**Start value** 的意思是说, 如果个体从某个状态  $s_1$  开始, 那么从该状态开始到 Episode 结束个体将会得到怎样的最终奖励. 这个时候算法真正关心的是, 找到一个策略, 当把这个个体放在这个状态  $s_1$ , 让它执行当前的策略, 能够获得 start value 的奖励. 这样我们的目标就变成最大化这个 start value:

这个时候的  $J_\theta$  的定义如6.4所示.  $V^{\pi_\theta}(s_1)$  就是使用策略  $\theta$  在状态  $s_1$  下的累计 reward. 我们希望这个累计的 reward 的期望最大.

$$J_1(\theta) = V^{\pi_\theta}(s_1) = E_{\pi_\theta}[v_1]. \quad (6.4)$$

### 6.2.1.2 Using Average Value

在连续的环境中, 我们使用 average value. 假设我们知道在使用当前策略  $\theta$  下, 每个状态  $s$  出现的概率为  $d^{\pi_\theta}(s)$ , 那么  $J_\theta$  的定义如6.5所示.

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s). \quad (6.5)$$

### 6.2.1.3 Using Average reward per time-step

又或者, 在连续的环境下, 我们选取一个时间段 (一个时间步长), 在这个时间内个体可以到达的所有状态, 以及采取行动可以获得的及时的 reward. 于是, 此时的  $J_\theta$  的定义如6.6所示. 注意, 这里是及时奖励  $R$ , 而不是累计奖励  $V$ .

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) R_s^a. \quad (6.6)$$

## 6.2.2 Policy Optimisation

在找到了目标函数  $J_\theta$  之后, 下面就是优化问题. 我们希望找到一个  $\theta$ , 使得上面的三个目标函数  $J_\theta$  最大.

在这一章的部分, 我们集中在使用梯度的方法来优化目标函数  $J_\theta$ . 同时探讨基于序列结构片段 (sequential structure) 进行优化. 这里的序列结构片段 (sequential structure) 就是指不用完整的一个 Episode, 类似于 TD 的想法.

## 6.3 Mente-Carlo Policy Gradient

### 6.3.1 Score Function

下面我们会详细说明如何进行梯度下降. 假设策略函数  $\pi_\theta$  是处处可微的, 且我们知道梯度  $\nabla_\theta \pi_\theta(s, a)$ . 在这里我们用一个小变换, 如式子6.7所示:

$$\begin{aligned}\nabla_\theta \pi_\theta(s, a) &= \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} \\ &= \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a)\end{aligned}\tag{6.7}$$

我们称  $\nabla_\theta \log \pi_\theta(s, a)$  是 **score function**. Score function 有一个较好的性质, 他的期望是 0, 具体的推导如式子6.8所示.

$$\begin{aligned}E_{\pi_\theta(s,a)}[\nabla_\theta \log \pi_\theta(s, a)] &= \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) \\ &= \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} \\ &= \sum_s d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s, a) \\ &= \nabla_\theta \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) \\ &= \nabla_\theta 1 \\ &= 0\end{aligned}\tag{6.8}$$

期望是 0 是一个非常好的性质, 他可以使得式子6.9成立.

$$E_{\pi_\theta(s,a)}[\nabla_\theta \log \pi_\theta(s, a) - b] = E_{\pi_\theta(s,a)}[\nabla_\theta \log \pi_\theta(s, a)]\tag{6.9}$$

在式子6.9中,  $b$  被称作 **baseline**, 他可以降低估计的方差. 我们会在后面的推导中用到这个性质. 关于更多 score function 的内容, 可以参考链接, **Score Function Trick** 及其在机器学习中的应用.

### 6.3.2 One-Step MDPs

在这里, 我们初始状态是从  $s \sim d(s)$  中获得. 且执行一步 action 之后, 会获得 reward 是  $r = R_s^a$ , 且停止.

由于是单步的过程, 因此上面介绍的三种目标函数此时是一样的. 此时目标函数如式子6.10所示:

$$\begin{aligned} J(\theta) &= E_{\pi_\theta}[r] \\ &= \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(s, a) R_s^a. \end{aligned} \quad (6.10)$$

若直接对式子6.10进行求梯度是困难的,于是我们进行进一步化简. 此时  $J(\theta)$  相应的梯度为6.11,这里利用了式子6.7进行化简:

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta E[r] \\ &= \nabla_\theta \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(s, a) R_s^a \\ &= \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} \nabla_\theta \pi_\theta(s, a) R_s^a \\ &= \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) R_s^a \\ &= E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot R_s^a] \end{aligned} \quad (6.11)$$

可以看到在单步MDPs中,目标函数的梯度等于策略函数对数梯度与即时奖励两部分乘积的期望.这两部分都是容易获得的,于是  $J(\theta)$  相应的梯度也是容易计算的. 这也就是为什么要使用 score function 进行计算,可以将原本的对期望求导,将求导转到 score function,这样是容易计算的. 现在我们将单步MDP推广到多步MDP.

### 6.3.3 Policy Gradient Theorem

对于多步MDPs来说,我们只需要将式子6.11中的及时reward  $r$  替换成  $Q$  值即可.这对于下面三种目标函数都是通用的.于是有如下的定理:

#### 定理 6.1. Policy Gradient Theorem

对于任何的可微的策略  $\pi_\theta(s, a)$ ,对于任何目标函数(policy objective function),

$J = J_1, J = J_{avR}$ , 或是  $J = \frac{1}{1-\gamma} J_{avV}$ , 策略梯度都是如式6.12所示:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot Q^{\pi_\theta}(s, a)] \quad (6.12)$$

### 6.3.4 Monte-Carlo Policy Gradient (REINFORCE)

在有了上面的定理之后,我们结合MC的想法,有了REINFORCE算法.该算法有以下的特点:

- 使用 return  $v_t$  作为  $Q^{\pi_\theta}(s, a)$  的无偏估计.
- 通过随机梯度上升对参数  $\theta$  进行更新.于是,  $\theta$  的梯度如式6.13所示.

- 注意在式子6.13中, 我们使用  $v_t$  作为  $Q^{\pi_\theta}(s, a)$  的无偏估计

$$\Delta\theta = \alpha \cdot \nabla_\theta \log \pi_\theta(s, a) \cdot v_t. \quad (6.13)$$

下面是 REINFORCE 的算法描述:

---

#### Algorithm 4: REINFORCE 算法简介

---

```

Initialize  $\theta$  ;
for each episode  $\{s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$  do
    for  $t=1$  to  $T-1$  do
         $\theta = \theta + \alpha \cdot \nabla_\theta \log \pi_\theta(s, a) \cdot v_t.$ 
    end
end
return  $\theta$ 

```

---

## 6.4 Actor-Critic Policy Gradient

### 6.4.1 Reduce Variance Using a Critic

上面我们使用 MC 方法中的  $v_t$  作为  $Q^{\pi_\theta}(s, a)$  的无偏估计. 虽然是无偏的, 但是有较大的方差. 我们希望可以相对准确的估计  $Q^{\pi_\theta}(s, a)$ , 用他来指导策略的更新. 这就是 **Actor-Critic** 的主要思想.

我们使用 **Critic** 来估计 action-value function, 也就是估计  $Q^{\pi_\theta}(s, a)$ . **Critic** 也是一个近似函数, 有参数  $w$  来控制, 如式子6.14所示:

$$Q_w(s, a) \approx Q^{\pi_\theta}(s, a). \quad (6.14)$$

在 **Actor-Critic** 算法中, 主要包含两个参数:

- **Critic**: 近似 action-value function, 包含系数  $w$ ;
- **Actor**: 近似 policy function, 包含系数  $\theta$ ;

于是, 在 **Actor-Critic** 算法中,  $\theta$  的梯度如6.15所示:

$$\Delta\theta = \alpha \cdot \nabla_\theta \log \pi_\theta(s, a) \cdot Q_w(s, a). \quad (6.15)$$

### 6.4.2 Estimating the Action-Value Function

对于 **Critic** 的任务, 是来估计 action-value function. 他是用来估计当给定策略  $\pi_\theta$  的时候, 来评价这个策略的好坏. 关于对给定策略的评价, 我们之前有讲过, 可

以使用 MC 或是 TD 的方法进行评价. 若使用 TD 的方式来估计  $Q_w(s, a)$ , 可以按照式子6.16进行迭代:

$$Q_w(s, a) = Q_w(s, a) + \beta[r + \gamma Q_w(s', a') - Q_w(s, a)]. \quad (6.16)$$

需要与 value-based 的方法做一下区分, 在 policy-based 的方法中, 我们不需要使用  $\epsilon$ -greedy 或是 greedy 的方法来更新策略. 策略是根据参数  $\theta$ , 根据  $\pi_\theta$  直接得到的.

### 6.4.3 Compatible Function Approximation

上面我们通过  $Q_w(s, a)$  来近似  $Q^{\pi_\theta}(s, a)$ . 但是这样也会引入一个偏差 (bias), 这会导致找不到最优解. 但是, 我们可以通过选择  $Q_w(s, a)$  来避免引入任何偏差 (avoid introducing any bias).

在这里, 没有偏差就是指使用  $Q_w(s, a)$  估计得到的  $\nabla_\theta J(\theta)$ , 与使用  $Q^{\pi_\theta}(s, a)$  得到的是相同的. 也就是如式子6.17:

$$E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot Q^{\pi_\theta}(s, a)] = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot Q_w(s, a)]. \quad (6.17)$$

那么什么样子的  $Q_w(s, a)$  才是满足条件的呢, 满足下面两个条件的  $Q_w(s, a)$  可以使得估计是无偏的. 如下面的定理说明:

#### 定理 6.2. Compatible Function Approximation Theorem

如果下面的两个条件满足:

- 近似价值函数的梯度完全等同于策略函数对数的梯度, 如式6.18所示:

$$\nabla_w Q_w(s, a) = \nabla_\theta \log \pi_\theta(s, a). \quad (6.18)$$

- 价值函数参数  $w$  使得均方差最小, 如式6.19所示:

$$\epsilon = E_{\pi_\theta}[(Q_w(s, a) - Q^{\pi_\theta}(s, a))^2]. \quad (6.19)$$

如果满足上面两个条件, 那么式子6.20就是没有偏差的.

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot Q_w(s, a)]. \quad (6.20)$$

下面简单说明一下, 为什么满足上面定理的两个条件的  $Q_w(s, a)$ , 是可以满足式子6.17的. 也就是估计是无偏的.

首先因为参数  $w$  使得均方差最小, 也就是  $\epsilon$  关于  $w$  的梯度为 0. 否则, 我们可以通过梯度下降, 进一步减少 loss. 也就是, 此时满足式子6.21:

$$\nabla_w \epsilon = 0. \quad (6.21)$$

我们对式子6.21进一步化简, 可以得到式子6.22:

$$E_{\pi_\theta}[(Q_w(s, a) - Q^{\pi_\theta}(s, a)) \cdot \nabla_w Q_w(s, a)] = 0. \quad (6.22)$$

有因为式子6.18, 有  $\nabla_w Q_w(s, a) = \nabla_\theta \log \pi_\theta(s, a)$ , 我们将其代入式子6.22进一步化简, 得到式子6.23:

$$\begin{aligned} E_{\pi_\theta}[(Q_w(s, a) - Q^{\pi_\theta}(s, a)) \cdot \nabla_\theta \log \pi_\theta(s, a)] &= 0 \\ E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot Q^{\pi_\theta}(s, a)] &= E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot Q_w(s, a)] \end{aligned} \quad (6.23)$$

可以看到, 式子6.23可以化简到式子6.17, 证毕.

#### 6.4.4 Reducing Variance Using Baseline

上面我们给出了什么样子的  $Q_w(s, a)$  可以使得是无偏估计. 这里我们讲一个小的技巧, 可以减少方差.

我们从 policy gradient 中抽取一个叫 baseline function  $B(s)$  的函数. 该  $B(s)$  只与状态  $s$  有关, 但是与行为  $a$  是无关的. 通过  $B(s)$ , 我们希望不改变  $E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot Q^{\pi_\theta}(s, a)]$ , 但是减少他的方差. 即满足式子6.24, 这个其实我们在上面的 score function 的时候有进行简单的说明, 在这里我们再次进行简单的证明:

$$E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot Q^{\pi_\theta}(s, a)] = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot (Q^{\pi_\theta}(s, a) - B(s))]. \quad (6.24)$$

下面我们只需要证明  $E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot B(s)] = 0$  即可. 证明过程如式子6.25所示:

$$\begin{aligned} E_{\pi_\theta(s, a)}[\nabla_\theta \log \pi_\theta(s, a) \cdot B(s)] &= \sum_{s \in S} d_{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} B(s) \\ &= \sum_{s \in S} d_{\pi_\theta}(s) \sum_{a \in A} \nabla_\theta \pi_\theta(s, a) B(s) \\ &= \sum_{s \in S} d_{\pi_\theta}(s) \cdot B(s) \cdot \nabla_\theta \sum_{a \in A} \pi_\theta(s, a) \\ &= \sum_{s \in S} d_{\pi_\theta}(s) \cdot B(s) \cdot \nabla_\theta 1 \\ &= \sum_{s \in S} d_{\pi_\theta}(s) \cdot B(s) \cdot 0 \\ &= 0 \end{aligned} \quad (6.25)$$

这里有一个问题, 及减去一个数, 方差不一定会变小的,  $D(x - y) = D(x) + D(y) - Cov(x, y)$ , 所以为什么这里减去 baseline 可以使得方差变小.

$B(s)$  的最佳取值就是  $V^{\pi_\theta}(s)$ , 也就是  $B(s) = V^{\pi_\theta}(s)$ . 于是我们定一个 advantage function  $A^{\pi_\theta}(s, a)$ . 他的含义是, 当个体采取行为  $a$  离开  $s$  状态时, 比该状态  $s$  平均价值要好多少.  $A^{\pi_\theta}(s, a)$  的表达式如6.26所示:

$$\begin{aligned} A^{\pi_\theta}(s, a) &= Q^{\pi_\theta}(s, a) - B(s) \\ &= Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \end{aligned} \quad (6.26)$$

于是, 最终的目标函数的梯度为式6.27:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot \textcolor{red}{A^{\pi_\theta}(s, a)}] \quad (6.27)$$

#### 6.4.5 Estimating the Advantage Function

上面我们介绍了 Advantage Function  $A^{\pi_\theta}(s, a)$ , 他可以减少方差. 所以, critic 实际近似的是  $A^{\pi_\theta}(s, a)$ . 这就需要我们同时近似  $V^{\pi_\theta}(s)$  和  $Q^{\pi_\theta}(s, a)$ .

但是实际情况下, 我们不需要近似两个函数, 这是因为  $V^{\pi_\theta}(s)$  的 TD 误差  $\delta^{\pi_\theta}$  是  $A^{\pi_\theta}(s, a)$  的无偏估计.  $\delta^{\pi_\theta}$  的计算如式子6.28所示, 当前状态为  $s$ , 当前动作为  $a$ :

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s) \quad (6.28)$$

我们对  $\delta^{\pi_\theta}$  求期望, 如式6.29所示:

$$\begin{aligned} E_{\pi_\theta}[\delta^{\pi_\theta}|s, a] &= E_{\pi_\theta}[r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)|s, a] \\ &= E_{\pi_\theta}[r + \gamma V^{\pi_\theta}(s')|s, a] - E_{\pi_\theta}[V^{\pi_\theta}(s)|s, a] \\ &= Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \\ &= A^{\pi_\theta}(s, a) \end{aligned} \quad (6.29)$$

所以最终的梯度公式, 可以化简为6.30:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) \cdot \textcolor{red}{\delta^{\pi_\theta}}] \quad (6.30)$$

在实际运算中, 我们会使用近似函数  $V_v(s)$  来近似  $V^{\pi_\theta}(s)$ . 我们只需要更新一组参数  $v$ , 来近似  $V_v(s)$ , 最后根据式子6.31来得到近似的  $\delta^{\pi_\theta}$ .

$$\delta^{\pi_\theta} \approx \delta_v = r + \gamma V_v(s') - V_v(s). \quad (6.31)$$

下面是 One-Step Actor-Critic Baseline 的算法流程, 也就是 TD 的, MC 的需要

稍作修改.

---

**Algorithm 5:** Actor-Critic Baseline (A2C) 算法简介
 

---

```

Initialize actor function  $\pi_\theta(a|s)$  ;
Initialize critic function  $V_v(s)$  ;
Initialize parameter  $\theta \in \mathbb{R}$ , and  $v \in \mathbb{R}'$  ;
for each episode do
  while Until Done do
     $A \sim \pi_\theta(\cdot|s)$  ;
    Take action A, and observe S' and R ;
     $\delta \leftarrow R + \gamma V_v(S') - V_v(S)$  ;
    Update parameter  $v$ ,  $v \leftarrow v + lr \cdot \delta \cdot \nabla_v V_v(S)$  ;
    Update parameter  $\theta$ ,  $\theta \leftarrow \theta + lr \cdot \delta \cdot \nabla_\theta \log \pi_\theta(s, a)$  ;
     $S \leftarrow S'$  ;
  end
end

```

---

我们详细解释上面的算法, 要计算式子6.32, 因为  $\delta \leftarrow R + \gamma V_v(S') - V_v(S)$ , 所以我们将 loss 设置为  $\delta^2$  即可. 对  $\delta^2$  求导即可得到式子6.32. 也就是 critic function 的 loss function 就是  $\delta^2$ .

$$v \leftarrow v + lr \cdot \delta \cdot \nabla_v V_v(S). \quad (6.32)$$

接着是对 actor function 的参数  $\theta$  进行更新. 更新是式子如6.33所示. 这里我们把  $\delta$  看成一个常数, 不需要进行求导. 此时的 loss function 就是  $\delta \cdot \log \pi_\theta(s, a)$ .

$$\theta \leftarrow \theta + lr \cdot \delta \cdot \nabla_\theta \log \pi_\theta(s, a). \quad (6.33)$$

## 6.5 Actor-Critic Baseline (A2C) 实验

关于 Actor-Critic Baseline, 我们同样使用 Pytorch 在 Cliff Walking 的环境中进行实现. 关于环境的介绍, 可以参考5.2.2.1部分. 这里我们重点说一下 Actor-Critic Baseline (A2C) 的实验部分, 详细的 notebook 链接还是会放在后面.

首先, 具体实施中, 因为 actor 和 critic 两个网络的输入部分是一样的, 都是输入 state, 也就是他们可以共用特征处理模块. 这样, 我们的网络可以定义为如图6.4所示的样子.

在我们这次的实验中, 我们使用的是 MC 的方式, 会完整进行完一整个 episode, 接着有了每一个 state 的  $G_t$  之后, 进行更新. 最终的结果如图6.5所示:



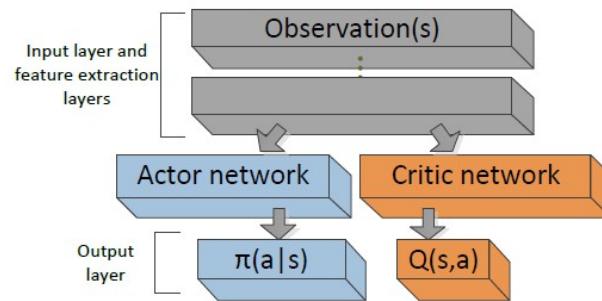


图 6.4: Actor-Critic Network 结构介绍.

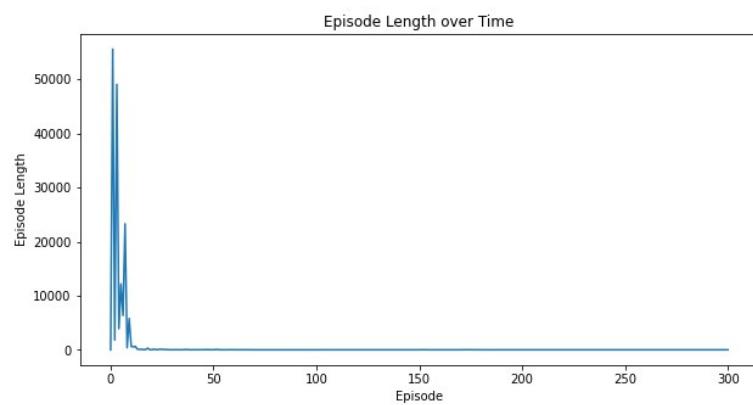


图 6.5: Actor-Critic Baseline 实验结果.

最后, jupyter notebook 的链接为 [Actor\\_Critic\\_Baseline\\_\(A2C\)\\_Pytorch.ipynb](#)