第5章 时序差分

5.1 时序差分简介

前面几章分别介绍了蒙特卡罗强化学习和动态规划强化学习。蒙特卡罗强化学习需要学习完整的采样轨迹，才能去更新值函数和改进策略，学习效率很低。而动态规划强化学习需要采用自举(bootstapping)的方法，用后继状态的值函数估计当前值函数，可以在每执行一步策略之后就进行值函数的更新，相比较而言，效率较高。本章介绍的时序差分方法充分结合了动态规划的自举和蒙特卡罗的采样，通过学习后继状态的值函数来逼近当前状态值函数，实现对不完整轨迹进行学习，可以高效地解决免模型强化学习问题。

时序差分学习最早由 A. Sammuel 在他著名的跳棋算法中提出，这个程序具有自学习能力，可通过分析大量棋局来逐渐辨识出当前局面的好棋和坏棋，不断提高棋艺水平。1988年，Sutton 首次证明了时序差分方法(TD(0))在最小均方误差(MSE)上的收敛性。之后，时序差分法被广泛应用在无法产生完整轨迹的无模型强化学习问题上。

第 4 章已经介绍过，蒙特卡罗使用实际的累积回报平均值 G_t 作为值函数的估计来更新值函数：

$$V(s_t) \leftarrow V(s_t) + \alpha(G_t - V(s_t))$$

而时序差分方法的应用场景是不完整轨迹，无法获得累积回报。它在估计状态 S_t 的值函数时，用的是离开该状态的立即回报 R_{t+1} 与下一状态 S_{t+1} 的预估折扣值函数 $\gamma V(S_{t+1})$ 之和：

$$V_\pi(S_t) = E_\pi[G_t \mid S_t = s] = E_\pi[R_{t+1} + \gamma V(S_{t+1}) \mid S_t = s]$$

上式符合 Bellman 方程的描述。用 $R_{t+1} + \gamma V(S_{t+1})$ 代替 G_t，就有了时序差分方法(TD)的值函数更新公式：

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

其中：$R_{t+1} + \gamma V(S_{t+1})$ 称为 TD 目标值；$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(s_t)$ 称为 TD 误差。

5.2 三种方法的性质对比

接下来从值函数估计方式、偏差与方差、马尔可夫属性等方面对时序差分、动态规划和蒙特卡罗三种方法进行对比。

1. 值函数估计

三种方法最大的不同体现在值函数的更新公式上。蒙特卡罗(MC)方法使用的是值函数最原始的定义,该方法依靠采样,学习完整的轨迹,利用实际累积回报的平均值 G_t 估计值函数,如图 5-1 所示。

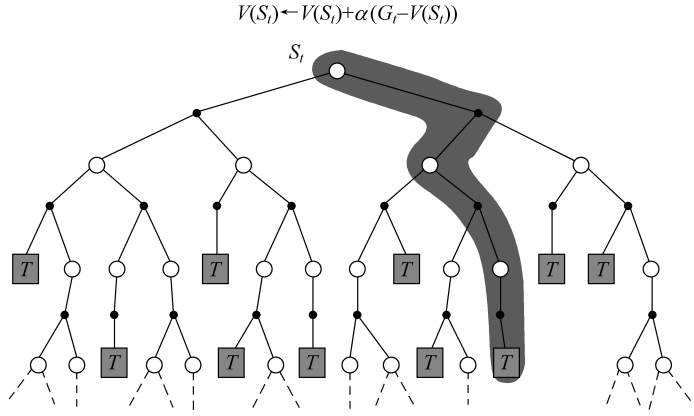


图 5-1 MC 方法(见彩插)

时序差分(TD)和动态规划(DP)则利用一步预测方法计算当前状态值函数,其共同点是利用了自举,使用后继值函数逼近当前值函数。不同的是,动态规划方法无须采样,直接根据完整模型,通过当前状态 S 所有可能的转移状态 S'、转移概率、立即回报来计算当前状态 S 的值函数,如图 5-2 所示。而时序差分(TD)方法是无模型方法,无法获得当前状态的所有后继状态及回报等,仅能通过采样学习轨迹片段,用下一状态的预估状态价值更新当前状态预估价值,如图 5-3 所示。

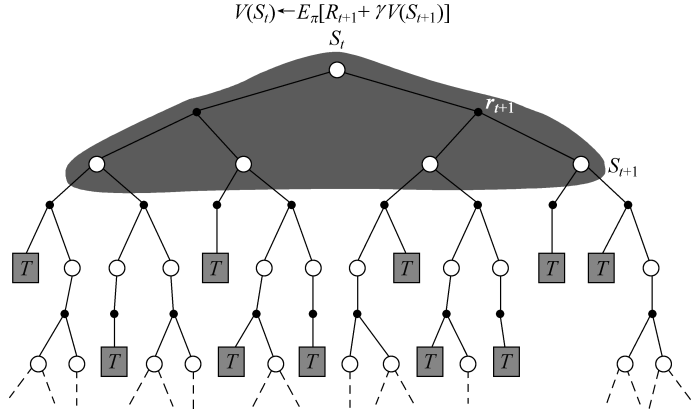


图 5-2 DP 方法(见彩插)

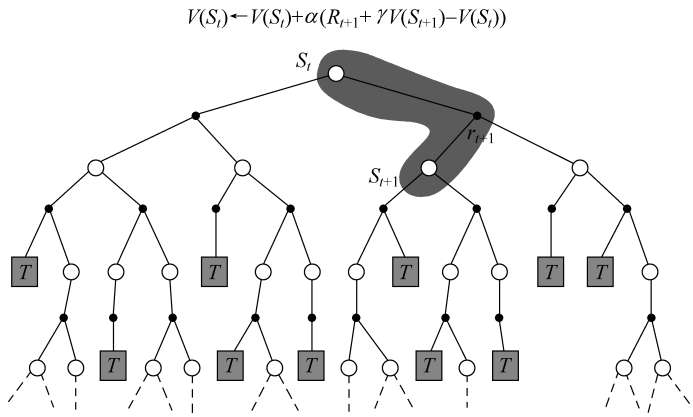


图 5-3　TD 方法(见彩插)

三种方法的值函数估计公式及对比见表 5-1。

表 5-1　时序差分、动态规划和蒙特卡罗三种方法

方　法	值函数估计	是否自举	是否采样
DP	$V_\pi(S_t)=E_\pi[R_{t+1}+\gamma V(S_{t+1})\mid S_t=s]$	自举	无须采样
MC	$V_\pi(S_t)\approx G_t\mid S_t=s$	不自举	采样,完整轨迹
TD	$V_\pi(S_t)\approx R_{t+1}+\gamma V(S_{t+1})\mid S_t=s$	自举	采样,不完整轨迹

2. 偏差/方差

蒙特卡罗(MC)和时序差分(TD)均是利用样本去估计值函数,可以从统计学的角度来对比两种方法的期望和方差两个指标。

蒙特卡罗在估计值函数时,使用的是累积回报 G_t 的平均值。G_t 期望便是值函数的定义,因此蒙特卡罗方法是无偏估计。

$$G_t=r_{t+1}+\gamma r_{t+2}+\cdots+\gamma^{T-1}r_T=\sum_{k=0}^{T}\gamma^k r_{t+k+1}$$

蒙特卡罗在计算 G_t 值时,需要计算从当前状态到最终状态之间所有的回报,在这个过程中要经历很多随机的状态和动作,因此每次得到的随机性很大。所以尽管期望等于真值,但方差无穷大。

时序差分方法使用 $R_{t+1}+\gamma V(S_{t+1})$(也叫 TD 目标)估计值函数,若 TD 目标采用真实值,是基于下一状态的实际价值对当前状态实际价值进行估计,则 TD 估计也是无偏估计。然而在实际中 TD 目标用的是估计值,即基于下一状态预估值函数计算当前预估值函数,因此时序差分估计属于有偏估计。跟蒙特卡罗相比,时序差分只用到了一步随机状态和动作,因此 TD 目标的随机性比蒙特卡罗方法中的 G_t 要小,其方差也比蒙特卡罗方法的方差小。

动态规划方法利用模型计算所有后继状态,借助贝尔曼方程,利用后继状态得到当前状

态的真实值函数，不存在偏差和方差，三种方法的偏差与方差对比见表 5-2。

表 5-2 三种方法的偏差与方差对比

方 法	偏 差	方 差
DP	无偏差	无方差
MC	无偏差	高方差
TD	无偏(真实 TD 目标) 有偏(预估 TD 目标)	低方差

3. 马尔可夫性

动态规划方法是基于模型的方法，基于现有的一个马尔可夫决策模型 MDP 的状态转移概率和回报，求解当前状态的值函数，因此该方法具有马尔可夫性。

蒙特卡罗和时序差分方法都是无模型方法，都需要通过学习采样轨迹估计当前状态值函数。所不同的是，应用时序差分(TD)算法时，时序差分算法试图利用现有的轨迹构建一个最大可能性的马尔可夫决策模型，即首先根据已有经验估计状态间的转移概率：

$$\hat{P}^{a}_{s,s'}=\frac{1}{N(s,a)}\sum_{k=1}^{K}\sum_{t=1}^{T_k}I(s_t^k,a_t^k,s_{t+1}^k=s,a,s')$$

同时估计某一个状态的立即回报：

$$\hat{R}^{a}_{s}=\frac{1}{N(s,a)}\sum_{k=1}^{K}\sum_{t=1}^{T_k}I(s_t^k,a_t^k=s,a)r_t^k$$

最后计算该马尔可夫决策模型的状态值函数。

而蒙特卡罗算法并不试图构建马尔可夫决策模型，该算法试图最小化状态值函数与累积回报的均方误差：

$$\sum_{k=1}^{K}\sum_{t=1}^{T_k}(G_t^k-V(s_t^k))^2$$

通过比较可以看出，时序差分和动态规划均使用了马尔可夫决策模型问题的马尔可夫属性，在马尔可夫环境下更有效；但是蒙特卡罗方法并不利用马尔可夫属性，通常在非马尔可夫环境下更有效，见表 5-3。

表 5-3 三种方法马尔可夫性对比

方 法	是否使用马尔可夫属性
DP	是
MC	否
TD	是

5.3 Sarsa：在线策略 TD

与蒙特卡罗、动态规划一致，时序差分方法也遵循了广义策略迭代框架，由策略评估和策略改进两个步骤交替进行，直至获取最优解。因为是无模型方法，所以策略评估是针对采样数据进行的。同样地，根据产生采样数据的策略和评估改进的策略是否为同一个策略，时序差分方法也可以分为在线策略法(on-policy)和离线策略法(off-policy)。

先来介绍在线策略时序差分方法，也就是下面要介绍的 Sarsa 方法，此方法由 Rummmy 和 Niranjan 于 1994 年提出。

Sarsa 的名称来源如图 5-4 所示，序列描述：基于状态 S，遵循当前策略，选择一个行为 A，形成第一个状态行为对(S,A)。与环境交互，得到回报 R，进入下一个状态 S'，再次遵循当前策略，产生一个行为 A'，产生第二个状态行为对(S',A')。利用后一个状态行为对(S',A')的行为值函数 $Q(S',A')$值更新前一个状态行为对(S,A)的 $Q(S,A)$值。

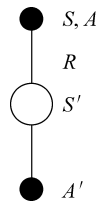


图 5-4 Sarsa 的名称来源

对应的行为值函数更新公式如下：

$$Q(S,A)\leftarrow Q(S,A)+\alpha(R+\gamma Q(S',A')-Q(S,A))$$

可见在具体执行时，单个轨迹内，每进行一个时间步，都会基于这个时间步的数据对行为值函数进行更新，产生采样的策略和评估改进的策略都是 ε-贪心策略。算法流程如下。

算法：Sarsa 算法

输入：环境 E，状态空间 S，动作空间 A，折扣回报 γ，初始化行为值函数 $Q(s,a)=0$，$\pi(a|s)=\dfrac{1}{|A|}$

For $k=0,1,\cdots,m$ do(针对每一条轨迹)
　初始化状态 s
　在 E 中通过 π 的 ε-贪心策略采取行为 a，得到第一个状态行为对(s,a)
　For $t=0,1,2,3\cdots$ do(针对轨迹中的每一步)
　　$r,s'=$在 E 中执行动作 a 产生的回报和转移的状态；
　　基于 s'，通过 π 的 ε-贪心策略采取行为 a'，得到第二个状态行为对(s',a')
　　更新(s,a)的 Q 值
　　$Q(s,a)\leftarrow Q(s,a)+\alpha(r+\gamma Q(s',a')-Q(s,a))$；
　　$s\leftarrow s',a\leftarrow a'$
　end for　s 是一个终止状态
　$\forall s_t\in S'$：

$$\pi(s_t)=\underset{a_t\in A}{\operatorname{argmax}}Q(s_t,a_t)$$

　end for

输出：最优策略 π

5.4 Q-learning：离线策略 TD 方法

离线策略时序差分(TD)学习的任务是借助策略 $\mu(a|s)$ 的采样数据来评估和改进另一个策略 $\pi(a|s)$。

离线策略 TD 也使用了重要性采样的方法。假设在状态 s_t 下遵循两个不同的策略产生了同样的行为 a_t，则两种情形下产生行为 a_t 的概率大小不一样。

首先考虑使用原始策略 π 来评估策略 π。情形如下：基于状态 s_t，遵循策略 π，产生行为 a_t，得到回报 R_{t+1}，进入新的状态 s_{t+1}，再次遵循策略 π，产生行为 a_{t+1}。评估策略 π 时对应的 TD 目标为

$$R_{t+1}+\gamma Q(s_{t+1},a_{t+1})$$

若是改用行为策略 μ 来评估策略 π，则需要给 $R_{t+1}+\gamma Q(s_{t+1},a_{t+1})$ 乘以一个重要性采样比率，对应的 TD 目标变为：

$$\frac{\pi(a_t\mid s_t)}{\mu(a_t\mid s_t)}(R_{t+1}+\gamma Q(s_{t+1},a_{t+1}))$$

离线策略 TD 方法策略评估对应的具体数学表示为：

$$Q(s_t,a_t)\leftarrow Q(s_t,a_t)+\alpha\left(\frac{\pi(a_t\mid s_t)}{\mu(a_t\mid s_t)}(R_{t+1}+\gamma Q(s_{t+1},a_{t+1}))-Q(s_t,a_t)\right)$$

这个公式可以这样解释：在状态 s_t 时，分别比较依据策略 $\pi(a_t|s_t)$ 和当前策略 $\mu(a_t|s_t)$ 产生行为 a_t 的概率大小，比值作为 TD 目标的权重，依此调整原来状态 s_t 的价值 $Q(s_t,a_t)$。

应用这种思想表现最好的方法是 Q-学习(Q-learning)方法。Q-learning 方法由 Watkins 和 Dayan 于 1992 年提出。它的要点在于，更新一个状态行为对的 Q 值时，采用的不是当前遵循策略(行为策略 μ)的下一个状态行为对的 Q 值，而是待评估策略(目标策略 π)产生的下一个状态行为对的 Q 值。

更新公式如下：

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

式中，TD 目标 $R_{t+1}+\gamma Q(s_{t+1},A')$ 是基于目标策略 π 产生的行为 A' 得到的 Q 值和一个立即回报的和。在 Q-learning 方法中，实际与环境交互时遵循的策略 μ 是一个基于原始策略的 ε-贪心策略，它能保证经历足够丰富的新状态。而目标策略 π 是单纯的贪心策略，保证策略最终收敛到最佳策略。

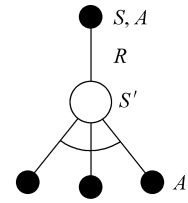
接下来，对 Q-learning 方法的更新公式进行变换。因为 A' 是基于目标策略 π 产生的行为，目标策略 π 是基于行为值函数的贪心策略，所以 A' 可表示为：

$$\pi(S_{t+1})=\operatorname{argmax}_{a'}Q(S_{t+1},a')$$

则 Q-learning 的 TD 目标为：

$$R_{t+1}+\gamma Q(S_{t+1},A')=R_{t+1}+\gamma Q(S_{t+1},\underset{a'}{\operatorname{argmax}}Q(S_{t+1},a'))=R_{t+1}+\underset{a'}{\max}\gamma Q(S_{t+1},a')$$

图 5-5 所示是 Q-learning 具体的更新公式和图解。



$$Q(S,A)\leftarrow Q(S,A)+\alpha(R+\gamma\underset{a'}{\max}Q(S',a')-Q(S,A))$$

图 5-5 Q-learning 公式及图解

可见，在状态 s_t 遵循 ε-贪心策略得到的 Q 值将朝着最大价值的方向更新。

同 Sarsa 方法一样，Q-learning 在具体执行时，单个轨迹内，每进行一个时间步，也会基于这个时间步的数据对行为值函数进行更新。其中产生采样的策略是 ε-贪心策略，而评估改进的策略是贪心策略。算法流程如下。

算法：Q-learning 算法

输入：环境 E，状态空间 S，动作空间 A，折扣回报 γ，初始化行为值函数 $Q(s,a)=0$，$\pi(a|s)=\dfrac{1}{|A|}$

For $k=0,1,\cdots,m$ do(针对每一条轨迹)

初始化状态 s

For $t=0,1,2,3\cdots$ do(针对轨迹中的每一步)

在 E 中通过 π 的 ε-贪心策略采取行为 a

$r,s'=$在 E 中执行动作 a 产生的回报和转移的状态；

$Q(s,a)\leftarrow Q(s,a)+\alpha(r+\gamma\underset{a'}{\max}Q(s',a')-Q(s,a))$；

$s\leftarrow s'$，

end for s 为终止状态

end for

$$\pi^*(s)=\underset{a\in A}{\operatorname{argmax}}Q(s,a)$$

输出：最优策略 π^*

5.5 实例讲解

本节以带陷阱的网格世界寻宝为例，分别使用 Sarsa 方法和 Q-learning 方法寻找最优策略并比较了两种算法在处理上的异同，同时给出核心代码。

5.5.1 迷宫寻宝

1. 环境描述

迷宫是一个 5×5 的网格世界，对应的马尔可夫决策模型一共有 24 个状态，如图 5-6 所示。

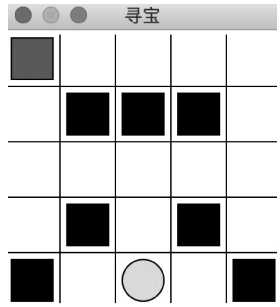


图 5-6 迷宫环境(见彩插)

网格世界每个格子的边长是 40 像素，空心方块表示智能体，边长为 30 像素。状态用空心智能体移动至当前格子时，空心方块与网格格子中心重叠后，空心方块左上和右下角的坐标表示。例如，网格世界第一行第一列的格子所代表的状态可表示为(5,5,35,35)，以此类推，可得到迷宫游戏的全部状态空间。其中有七个陷阱(图 5-6 实心方块所在位置)和一个宝藏区(实心圆所在位置)。

空心方块表示智能体，可执行的行为分别为朝上、下、左、右移动一步，则动作空间标记为 $A=\{0,1,2,3\}$，0、1、2、3 分别对应上、下、左、右。

在这个迷宫游戏中，智能体一旦进入陷阱位置，获得负 1 回报，游戏终止。智能体一旦进入宝藏区，获得正 1 回报，游戏终止。除此之外，智能体的任何移动，回报为 0。并且当智能体位于网格世界边缘格子时，任何使得智能体试图离开格子世界的行为都会使得智能体停留在移动前的位置。

对于智能体来说，它不清楚整个格子世界的构造。它不知道格子是长方形还是正方形，不知道格子世界的边界在哪里，也不清楚陷阱和宝藏的具体位置。智能体能做的就是不断进行上下左右移动，与环境进行交互，通过环境反馈的回报不断调整自己的行为。

假设在此网格世界游戏中，智能体状态转移概率 $p_{ss'}^{a}=1$，折扣因子 $\gamma=1$。求解此网格世界寻找宝藏的最优策略。

2. 环境代码

接下来根据上述描述构建网格寻宝环境，环境代码主要由一个 Maze 类构成，包含如下方法。

- def _build_maze(self)：构建迷宫的方法，该方法给出了陷阱位置、宝藏位置及智能体的初始位置。并且定义了动作空间，给出了状态转换过程以及行为回报。
- def step(self,action)：根据当前行为，返回下一步的位置、立即回报，以及判断游戏是否终止。
- def reset(self)：根据当前状态，重置画布。
- def render_by_policy(self,policy,result_list)：根据传入策略，进行界面渲染。

环境代码如下。

```
import numpy as np
import time
import sys
if sys.version_info.major == 2:
    import Tkinter as tk
else:
    import Tkinter as tk
UNIT = 40                    # 每个格子的大小
MAZE_H = 5                   # 行数
MAZE_W = 5                   # 列数
class Maze(tk.Tk, object):
    def __init__(self):
        super(Maze, self).__init__()
        self.action_space = ['u', 'd', 'l', 'r']
        self.n_actions = len(self.action_space)
        self.title('寻宝')
        self.geometry('{0}x{1}'.format(MAZE_H * UNIT, MAZE_H * UNIT))
        self._build_maze()
    def _build_maze(self):
        # 创建一个画布
        self.canvas = tk.Canvas(self, bg = 'white',
                                height = MAZE_H * UNIT,
                                width = MAZE_W * UNIT)
        # 在画布上画出列
        for c in range(0, MAZE_W * UNIT, UNIT):
            x0, y0, x1, y1 = c, 0, c, MAZE_H * UNIT
            self.canvas.create_line(x0, y0, x1, y1)
        # 在画布上画出行
        for r in range(0, MAZE_H * UNIT, UNIT):
            x0, y0, x1, y1 = 0, r, MAZE_H * UNIT, r
            self.canvas.create_line(x0, y0, x1, y1)
        # 创建探险者起始位置(默认为左上角)
        origin = np.array([20, 20])
        # 陷阱 1
        hell1_center = origin + np.array([UNIT, UNIT])
        self.hell1 = self.canvas.create_rectangle(
            hell1_center[0] - 15, hell1_center[1] - 15,
            hell1_center[0] + 15, hell1_center[1] + 15,
            fill = 'black')
        # 陷阱 2
        hell2_center = origin + np.array([UNIT * 2, UNIT])
        self.hell2 = self.canvas.create_rectangle(
            hell2_center[0] - 15, hell2_center[1] - 15,
            hell2_center[0] + 15, hell2_center[1] + 15,
```

```
            fill = 'black')
        # 陷阱 3
        hell3_center = origin + np.array([UNIT * 3, UNIT])
        self.hell3 = self.canvas.create_rectangle(
            hell3_center[0] - 15, hell3_center[1] - 15,
            hell3_center[0] + 15, hell3_center[1] + 15,
            fill = 'black')
        # 陷阱 4
        hell4_center = origin + np.array([UNIT, UNIT * 3])
        self.hell4 = self.canvas.create_rectangle(
            hell4_center[0] - 15, hell4_center[1] - 15,
            hell4_center[0] + 15, hell4_center[1] + 15,
            fill = 'black')
        # 陷阱 5
        hell5_center = origin + np.array([UNIT * 3, UNIT * 3])
        self.hell5 = self.canvas.create_rectangle(
            hell5_center[0] - 15, hell5_center[1] - 15,
            hell5_center[0] + 15, hell5_center[1] + 15,
            fill = 'black')
        # 陷阱 6
        hell6_center = origin + np.array([0, UNIT * 4])
        self.hell6 = self.canvas.create_rectangle(
            hell6_center[0] - 15, hell6_center[1] - 15,
            hell6_center[0] + 15, hell6_center[1] + 15,
            fill = 'black')
        # 陷阱 7
        hell7_center = origin + np.array([UNIT * 4, UNIT * 4])
        self.hell7 = self.canvas.create_rectangle(
            hell7_center[0] - 15, hell7_center[1] - 15,
            hell7_center[0] + 15, hell7_center[1] + 15,
            fill = 'black')
        # 宝藏位置
        oval_center = origin + np.array([UNIT * 2,UNIT * 4])
        self.oval = self.canvas.create_oval(
            oval_center[0] - 15, oval_center[1] - 15,
            oval_center[0] + 15, oval_center[1] + 15,
            fill = 'yellow')
        # 将探险者用矩形表示
        self.rect = self.canvas.create_rectangle(
            origin[0] - 15, origin[1] - 15,
            origin[0] + 15, origin[1] + 15,
            fill = 'red')
        # 画布展示
        self.canvas.pack()
    # 根据当前的状态重置画布(为了展示动态效果)
    def reset(self):
```

```
            self.update()
            time.sleep(0.5)
            self.canvas.delete(self.rect)
            origin = np.array([20, 20])
            self.rect = self.canvas.create_rectangle(
                origin[0] - 15, origin[1] - 15,
                origin[0] + 15, origin[1] + 15,
                fill = 'red')
            return self.canvas.coords(self.rect)
        # 根据当前行为,确定下一步的位置
        def step(self, action):
            s = self.canvas.coords(self.rect)
            base_action = np.array([0, 0])
            if action == 0:                          # 上
                if s[1] > UNIT:
                    base_action[1] -= UNIT
            elif action == 1:                        # 下
                if s[1] < (MAZE_H - 1) * UNIT:
                    base_action[1] += UNIT
            elif action == 2:                        # 左
                if s[0] > UNIT:
                    base_action[0] -= UNIT
            elif action == 3:                        # 右
                if s[0] < (MAZE_W - 1) * UNIT:
                    base_action[0] += UNIT
            # 在画布上将探险者移动到下一位置
            self.canvas.move(self.rect, base_action[0], base_action[1])
            # 重新渲染整个界面
            s_ = self.canvas.coords(self.rect)     # next state
            # 根据当前位置来获得回报值及是否终止
            if s_ == self.canvas.coords(self.oval):
                reward = 1
                done = True
                s_ = 'terminal'
            elif s_ in [self.canvas.coords(self.hell1),self.canvas.coords(self.hell2), self.
    canvas.coords(self.hell3),self.canvas.coords(self.hell4), self.canvas.coords(self.hell5),
    self.canvas.coords(self.hell6), self.canvas.coords(self.hell7)]:
                reward = -1
                done = True
                s_ = 'terminal'
            else:
                reward = 0
                done = False
            return s_, reward, done
        def render(self):
            time.sleep(0.1)
            self.update()
```

5.5.2 Sarsa 方法

1. 算法详情

本节使用 Sarsa 方法对带陷阱的网格世界马尔可夫决策问题进行求解。总体思路是以 ε-贪心策略采样数据，生成轨迹。针对每一条轨迹的每个时间步，进行一次策略评估，根据下式更新状态行为对的行为值函数：

$$Q(s_1,a_1) \leftarrow Q(s_1,a_1)+\alpha(r+\gamma Q(s_2,a_2)-Q(s_1,a_1))$$

每条轨迹结束，根据更新的值函数，对策略进行改进。规定轨迹总数目为 100 条。超出轨迹数目之后，输出最优策略。具体操作过程如下。

(1) 初始化全部行为值函数 $Q(s,a)=0$。当前的 q 值以 q 表形式存储，创建 q 表。

```
self.q_table = pd.DataFrame(columns=self.actions, dtype=np.float64)
```

初始化值函数。

```
self.q_table = self.q_table.append(
    pd.Series(
        [0]*len(self.actions),
        index=self.q_table.columns,
        name=state,
    )
)
```

(2) 初始化环境，得到初始状态 s_1。这里指的是智能体初始位置，$s_1=(5,5,35,35)$。

```
observation = env.reset()
```

(3) 基于状态 s_1，遵循 ε-贪心策略选择行为 a_1。例如，得到动作 $a_1=2$(表示向右移动一格)，得到第一个状态行为对(s_1,a_1)。

```
# 基于当前状态选择行为
action = RL.choose_action(str(observation))

def choose_action(self, observation):
    self.check_state_exist(observation)
    # 从均匀分布的[0,1)中随机采样，当小于阈值时采用选择最优行为的方式，当大于阈值时采用
    # 选择随机行为的方式，这样人为增加随机性是为了解决陷入局部最优
    if np.random.rand() < self.epsilon:
        # 选择最优行为
        state_action = self.q_table.ix[observation, :]
```

```
            # 因为一个状态下最优行为可能会有多个,所以在碰到这种情况时,需要随机选择一个行
            # 为进行
            state_action = state_action.reindex(np.random.permutation(state_action.index))
            action = state_action.idxmax()
        else:
            # 选择随机行为
            action = np.random.choice(self.actions)
        return action
```

(4) 动作 a_1 作用于环境,获得立即回报 R_1 和下一个状态 s_2,同时得到了轨迹是否终止的标识。这里:$s_2=(45,5,75,35)$,$R_1=0$,done=false(表示轨迹未终止)。

```
observation_, reward, done, oval_flag = env.step(action)
```

(5) 基于状态 s_2,继续遵循 ε-贪心策略,得到行为 a_2。这里动作 $a_2=0$(表示向右移动一格),得到第二个状态行为对(s_2,a_2)。

```
action_ = RL.choose_action(str(observation_))
```

(6) 通过第二个状态行为对(s_2,a_2)的行为值函数 $Q(s_2,a_2)$更新第一个状态行为对(s_1,a_1)的行为值函数 $Q(s_1,a_1)$。根据公式 $Q(s_1,a_1)\leftarrow Q(s_1,a_1)+\alpha(r+\gamma Q(s_2,a_2)-Q(s_1,a_1))$,计算得到:$Q(s_1,a_1)=0$,紧接着,令 $s_2=s_1$。

```
RL.learn(str(observation), action, reward, str(observation_), action_)

def learn(self, s, a, r, s_, a_):
    self.check_state_exist(s_)
    q_predict = self.q_table.ix[s, a]
    if s_ != 'terminal':
        # 使用公式: Q_target = r + γQ(s',a')
        q_target = r + self.gamma * self.q_table.ix[s_, a_]
    else:
        q_target = r
    # 更新公式: Q(s,a)←Q(s,a) + α(r + γQ(s',a') - Q(s,a))
    self.q_table.ix[s, a] += self.lr * (q_target - q_predict)

observation = observation_
```

(7) 重复步骤(3)~(6),直至轨迹结束。

(8) 结合更新后的行为值函数,采用 ε-贪心法对原始策略进行更新。

```
# 开始输出最终的 Q 表
q_table_result = RL.q_table
# 使用 Q 表输出各状态的最优策略
policy = get_policy(q_table_result)

def get_policy(q_table,rows = 5,cols = 5,pixels = 40,origin = 20):
    policy = []

    for i in range(rows):
        for j in range(cols):
            # 求出每个格子的状态
            item_center_x, item_center_y = (j * pixels + origin), (i * pixels + origin)
            item_state = [item_center_x - 15.0, item_center_y - 15.0, item_center_x +
15.0, item_center_y + 15.0]

            # 如果当前状态为各终止状态,则值为 - 1
            if item_state in [env.canvas.coords(env.hell1), env.canvas.coords(env.hell2),
env.canvas.coords(env.hell3), env.canvas.coords(env.hell4), env.canvas.coords(env.hell5),
env.canvas.coords(env.hell6), env.canvas.coords(env.hell7), env.canvas.coords(env.oval)]:
                policy.append( - 1)
                continue

            if str(item_state) not in q_table.index:
                policy.append((0, 1, 2, 3))
                continue
            # 选择最优行为
            item_action_max = get_action(q_table,str(item_state))
            policy.append(item_action_max)

    return policy
```

(9) 重复步骤(2)～(8),直至轨迹数＝100。最终得到的最优策略如图 5-7 所示。

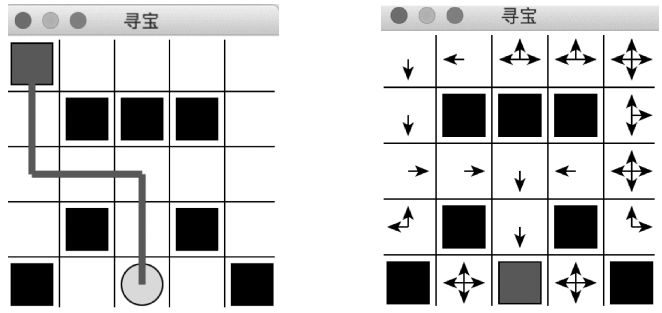


图 5-7 Sarsa 方法得到的最优策略(见彩插)

图 5-7 为智能体从起点出发找到宝藏的最优路径。最优路径所在状态经历了多次探索，可以得到比较准确的最优行为。而其他状态经历次数很少，给出的最优行为不精确。例如，宝藏左右两侧的网格位置，因为智能体从没有经历过，因此其四个方向的行为值函数均为 0，对应的最优行为为四个方向中的任意一个。为简要地说明问题，仅列出最优路径经历的状态及采取的最优行为，见表 5-4。其中最优行为是带有 ε 随机性的随机行为，以 ε 的概率选择当前最优动作，以 1－ε 的概率随机选择一个行为。

表 5-4　最优路径经历的状态及采取的最优行为

状　　态	行　　为
(5.0,5.0,35.0,35.0)	1
(5.0,45.0,35.0,75.0)	1
(5.0,85.0,35.0,115.0)	3
(45.0,85.0,75.0,115.0)	3
(85.0,85.0,115.0,115.0)	1
(85.0,125.0,115.0,155.0)	1

2. 核心代码

Sarsa 最核心的方法是 update()方法，循环遍历 100 条轨迹中的每一个时间步；进行行为的选择和行为值函数的更新，并基于行为值函数进行策略改进。

update()方法调用的其他基础方法均写在 RL 类中，例如：

def choose_action(str(observation))：基于输入状态，根据 ε-贪心策略选择行为。

def learn(str(observation),action,reward,str(observation_),action_)：Sarsa 的值函数更新方法。由代码可见，Sarsa 方法自始至终都在维护一个 Q 表(q_table，行为值函数表)，此表记录了智能体所经历过的状态行为对的行为值函数。

def get_policy(…)：基于当前 Q 表，绘制最优策略图。

具体代码如下。

```
def update():
    for episode in range(100):
        # 初始化状态
        observation = env.reset()
        c = 0
        tmp_policy = {}
        while True:
            # 渲染当前环境
            env.render()
            # 基于当前状态选择行为
            action = RL.choose_action(str(observation))
            state_item = tuple(observation)
```

```
                tmp_policy[state_item] = action
                # 采取行为获得下一个状态和回报及是否终止
                observation_, reward, done, oval_flag = env.step(action)
                # 基于下一个状态选择行为
                action_ = RL.choose_action(str(observation_))
                # 基于变化 (s, a, r, s, a)使用 Sarsa 进行 Q 的更新
                RL.learn(str(observation), action, reward, str(observation_), action_)
                # 改变状态和行为
                observation = observation_
                c += 1
                # 如果为终止状态,结束当前的局数
                if done:
                    break
        print('游戏结束')
        # 开始输出最终的 Q 表
        q_table_result = RL.q_table
        print(q_table_result)
        # 使用 Q 表输出各状态的最优策略
        policy = get_policy(q_table_result)
        policy_result = np.array(policy).reshape(5,5)
        env.render_by_policy_new(policy_result)
        # env.destroy()

    if __name__ == "__main__":
        env = Maze()
        RL = SarsaTable(actions = list(range(env.n_actions)))
        env.after(100, update)
        env.mainloop()

    class RL(object):
        def __init__(self, action_space, learning_rate = 0.01, reward_decay = 0.9, e_greedy = 0.9):
            self.actions = action_space
            self.lr = learning_rate
            self.gamma = reward_decay
            self.epsilon = e_greedy
            self.q_table = pd.DataFrame(columns = self.actions, dtype = np.float64)
        def check_state_exist(self, state):
            if state not in self.q_table.index:
                # 如果状态在当前的 Q 表中不存在,将当前状态加入 Q 表中
                self.q_table = self.q_table.append(
                    pd.Series(
                        [0] * len(self.actions),
                        index = self.q_table.columns,
                        name = state,
                    )
                )
```

```
    def choose_action(self, observation):
        self.check_state_exist(observation)
        # 从均匀分布的[0,1]中随机采样,当小于阈值时采用选择最优行为的方式,当大于阈值时采用
        # 选择随机行为的方式,这样人为增加随机性是为了解决陷入局部最优
        if np.random.rand() < self.epsilon:
            # 选择最优行为
            state_action = self.q_table.ix[observation, :]
            # 因为一个状态下最优行为可能会有多个,所以在碰到这种情况时,需要随机选择一
            # 个行为进行
             state_action = state_action.reindex(np.random.permutation(state_action.
index))
            action = state_action.idxmax()
        else:
            ## 选择随机行为
            action = np.random.choice(self.actions)
        return action
    def learn(self, *args):
        pass


# 在线策略 Sarsa
class SarsaTable(RL):
    def __init__(self, actions, learning_rate=0.01, reward_decay=0.9, e_greedy=0.9):
        super(SarsaTable, self).__init__(actions, learning_rate, reward_decay, e_greedy)
    def learn(self, s, a, r, s_, a_):
        self.check_state_exist(s_)
        q_predict = self.q_table.ix[s, a]
        if s_ != 'terminal':
            # 使用公式: Q_target = r + γQ(s',a')
            q_target = r + self.gamma * self.q_table.ix[s_, a_]
        else:
            q_target = r
        # 更新公式: Q(s,a)←Q(s,a) + α(r + γQ(s',a') - Q(s,a))
        self.q_table.ix[s, a] += self.lr * (q_target - q_predict)
   def get_action(q_table,state):
    # 选择最优行为
    state_action = q_table.ix[state, :]
    # 因为一个状态下最优行为可能会有多个,所以在碰到这种情况时,需要随机选择一个
    state_action_max = state_action.max()

    idxs = []
    for max_item in range(len(state_action)):
        if state_action[max_item] == state_action_max:
            idxs.append(max_item)
    sorted(idxs)
```

```
        return tuple(idxs)


    def get_policy(q_table,rows = 5,cols = 5,pixels = 40,origin = 20):
        policy = []
        for i in range(rows):
            for j in range(cols):
                # 求出每个格子的状态
                item_center_x, item_center_y = (j * pixels + origin), (i * pixels + origin)
                item_state = [item_center_x - 15.0, item_center_y - 15.0, item_center_x +
    15.0, item_center_y + 15.0]
                if item_state in [env.canvas.coords(env.hell1), env.canvas.coords(env.hell2),
                 env.canvas.coords(env.hell3), env.canvas.coords(env.hell4),
                 env.canvas.coords(env.hell5), env.canvas.coords(env.hell6),
                 env.canvas.coords(env.hell7), env.canvas.coords(env.oval)]:
                 policy.append(-1)
                    continue
                if str(item_state) not in q_table.index:
                    policy.append((0, 1, 2, 3))
                    continue
                # 选择最优行为
                item_action_max = get_action(q_table,str(item_state))
                policy.append(item_action_max)
            return policy
```

5.5.3 Q-learning 方法

1. 算法详情

使用 Q-learning 方法对带陷阱的网格世界马尔可夫问题进行求解，总体思路是以 ε-贪心策略采样数据，生成轨迹。针对每一条轨迹的每个时间步，进行一次策略评估，更新状态行为对的行为值函数。Q-learning 在对策略进行评估的同时，通过 max 操作实现贪心算法，对策略进行改进。

$$Q(s_1,a_1) \leftarrow Q(s_1,a_1) + \alpha(r + \gamma \max_{a'} Q(s_2,a') - Q(s_1,a_1))$$

规定轨迹总数目为 100 条。超出轨迹数目之后，得到最终的行为值函数。通过对行为值函数进行贪婪操作，得到最优策略。

(1) 初始化行为值函数 $Q(s,a)=0$。当前的 q 值以 q 表形式存储，创建 q 表。

```
self.q_table = pd.DataFrame(columns = self.actions, dtype = np.float64)
```

初始化值函数。

```
self.q_table = self.q_table.append(
    pd.Series(
        [0] * len(self.actions),
        index = self.q_table.columns,
        name = state,
    )
)
```

(2) 初始化环境,得到初始状态 s_1,也即智能体初始位置,s_1=(5,5,35,35)。

```
observation = env.reset()
```

(3) 基于状态 s_1,遵循 ε-贪心策略选择行为 a_1。选择动作 a_1=0,表示向上移动一格;得到第一个状态行为对(s_1,a_1)。

```
action = RL.choose_action(str(observation))
def choose_action(self, observation):
    self.check_state_exist(observation)
    # 从均匀分布的[0,1)中随机采样,当小于阈值时采用选择最优行为的方式,
    # 当大于阈值时采用选择随机行为的方式,这样人为增加随机性是为了解决陷入局部最优
    if np.random.rand() < self.epsilon:
        # 选择最优行为
        state_action = self.q_table.ix[observation, :]
        # 因为一个状态下最优行为可能有多个,所以碰到这种情况时,需要随机选择一个行为
        # 进行
        state_action = state_action.reindex(np.random.permutation(state_action.index))
        action = state_action.idxmax()
    else:
        # # 选择随机行为
        action = np.random.choice(self.actions)
    return action
```

(4) 获得立即回报 R_1 和下一个状态 s_2 及是否终止的标识。s_2=(5,5,35,35),R_1=0,done=false 表示轨迹未终止。

```
observation_, reward, done, oval_flag = env.step(action)
```

(5) 获得 s_2 对应的所有行为值函数(共 4 个),并找出其中最大的值,得到 $\max_{a'} Q(s_2,a')$。

```
RL.learn(str(observation), action, reward, str(observation_))
def learn(self, s, a, r, s_):
    self.check_state_exist(s_)
    q_predict = self.q_table.ix[s, a]
    if s_ != 'terminal':
```

```
            # 如果下一个状态不是终止状态,使用公式:Q_target = r + γ maxQ(s',a')计算
            q_target = r + self.gamma * self.q_table.ix[s_, :].max()
        else:
            q_target = r
```

(6) 通过值函数 $\max_{a'} Q(s_2,a')$更新值函数 $Q(s_1,a_1)$,得到: $Q(s_2,a_2)=0$,接着令 $s_2=s_1$。

```
# 更新公式: Q(s,a)←Q(s,a) + α(r + γ maxQ(s',a') - Q(s,a))
self.q_table.ix[s, a] += self.lr * (q_target - q_predict)

observation = observation_
```

(7) 重复步骤(3)～(6),直至轨迹结束。

(8) 结合更新后的行为值函数,采用贪心法对原始策略进行更新。

```
# 开始输出最终的 Q 表
q_table_result = RL.q_table
# 使用 Q 表输出各状态的最优策略
policy = get_policy(q_table_result)
```

(9) 重复步骤(2)～(7),直至轨迹数=100,得到最优行为值函数和最优策略。

(10) 同 Sarsa 方法一样,Q-learning 方法也可以找到智能体从起点出发找到宝藏的最优路径,如图 5-7 所示。

表 5-5 仅列出最优路径经历的状态及采取的最优行为。

表 5-5 最优路径经历的状态及采取的最优行为

状 态	行 为
(5.0,5.0,35.0,35.0)	1
(5.0,45.0,35.0,75.0)	1
(5.0,85.0,35.0,115.0)	3
(45.0,85.0,75.0,115.0)	3
(85.0,85.0,115.0,115.0)	1
(85.0,125.0,115.0,155.0)	1

2. 核心代码

Q-learning 最核心的方法是 update()方法,循环遍历 100 条轨迹中的每一个时间步来更新行为值函数,并基于行为值函数进行策略改进。update()方法调用的其他基础方法均写在 RL 类中。例如:

- def choose_action(str(observation)): 基于当前状态,使用 ε-贪心策略产生行为。
- def learn(str(observation),action,reward,str(observation_)): Q-learning 的值函

数更新方法。同 Sarsa 方法一样，Q-learning 也在维护一个 Q 表(q_table)，此表记录了智能体所经历过的状态行为对的行为值函数。

- get_policy(…)：基于当前 Q 表，绘制最优策略图。

具体代码如下：

```
    def update():
     for episode in range(100):
         # 初始化状态
         observation = env.reset()
         c = 0
         tmp_policy = {}
         while True:
             # 渲染当前环境
             env.render()
             # 基于当前状态选择行为
             action = RL.choose_action(str(observation))
             state_item = tuple(observation)
             tmp_policy[state_item] = action
             # 采取行为获得下一个状态和回报及是否终止
             observation_, reward, done, oval_flag = env.step(action)
               # 根据当前的变化开始更新 Q
             RL.learn(str(observation), action, reward, str(observation_))
             # 改变状态和行为
             observation = observation_
             c += 1
             # 如果为终止状态,则结束当前的局数
             if done:
                 break
     print('游戏结束')
     # 开始输出最终的 Q 表
     q_table_result = RL.q_table
     print(q_table_result)
     # 使用 Q 表输出各状态的最优策略
     policy = get_policy(q_table_result)
     policy_result = np.array(policy).reshape(5,5)
     env.render_by_policy_new(policy_result)
     # env.destroy()

if __name__ == "__main__":
     env = Maze()
     RL = QLearningTable(actions = list(range(env.n_actions)))
     env.after(100, update)
     env.mainloop()
     class RL(object):
     def __init__(self, action_space, learning_rate = 0.01, reward_decay = 0.9, e_greedy = 0.9):
```

```
        self.actions = action_space
        self.lr = learning_rate
        self.gamma = reward_decay
        self.epsilon = e_greedy
        self.q_table = pd.DataFrame(columns = self.actions, dtype = np.float64)
    def check_state_exist(self, state):
        if state not in self.q_table.index:
            # 如果状态在当前的Q表中不存在,将当前状态加入Q表中
            self.q_table = self.q_table.append(
                pd.Series(
                    [0] * len(self.actions),
                    index = self.q_table.columns,
                    name = state,
                )
            )
    def choose_action(self, observation):
        self.check_state_exist(observation)
        # 从均匀分布的[0,1)中随机采样,当小于阈值时采用选择最优行为的方式,当大于阈值时
        # 采用选择随机行为的方式,这样人为增加随机性是为了解决陷入局部最优
        if np.random.rand() < self.epsilon:
            # 选择最优行为
            state_action = self.q_table.ix[observation, :]
            # 因为一个状态下最优行为可能会有多个,所以在碰到这种情况时,需要随机选择一
            # 个行为进行
            state_action = state_action.reindex(np.random.permutation(state_action.index))
            action = state_action.idxmax()
        else:
            # # 选择随机行为
            action = np.random.choice(self.actions)
        return action
    def learn(self, *args):
        pass
# 离线策略Q-learning
class QLearningTable(RL):
    def __init__(self, actions, learning_rate = 0.01, reward_decay = 0.9, e_greedy = 0.9):
        super(QLearningTable, self).__init__(actions, learning_rate, reward_decay, e_
greedy)
    def learn(self, s, a, r, s_):
        self.check_state_exist(s_)
        q_predict = self.q_table.ix[s, a]
        if s_ != 'terminal':
            # 如果下一个状态为非终止状态使用公式:Q_target = r + γ maxQ(s',a')计算
            q_target = r + self.gamma * self.q_table.ix[s_, :].max()
        else:
            q_target = r
        # 更新公式:Q(s,a)←Q(s,a) + α(r + γ maxQ(s',a') - Q(s,a))
```

```
            self.q_table.ix[s, a] += self.lr * (q_target - q_predict)
    def get_policy(q_table,rows = 5,cols = 5,pixels = 40,origin = 20):
        policy = []
        for i in range(rows):
            for j in range(cols):
                # 求出每个格子的状态
                item_center_x, item_center_y = (j * pixels + origin), (i * pixels + origin)
                item_state = [item_center_x - 15.0, item_center_y - 15.0, item_center_x +
    15.0, item_center_y + 15.0]
                if item_state in [env.canvas.coords(env.hell1), env.canvas.coords(env.hell2),
                          env.canvas.coords(env.hell3),
                         env.canvas.coords(env.hell4),
                          env.canvas.coords(env.hell5),
                         env.canvas.coords(env.hell6),
                          env.canvas.coords(env.hell7),
                         env.canvas.coords(env.oval)]:
                    policy.append(-1)
                    continue
                if str(item_state) not in q_table.index:
                    policy.append((0, 1, 2, 3))
                    continue
                # 选择最优行为
                item_action_max = get_action(q_table,str(item_state))
                policy.append(item_action_max)
        return policy
```

5.5.4 实例小结

通过对 6×6 的迷宫寻宝游戏多次运行 Sarsa 和 Q-learning 的代码发现，Sarsa 方法和 Q-learning 方法在经过 100 多条轨迹后可以得到最优策略。Sarsa 得到的最优策略带有 ε 的随机性，而 Q-learning 得到的策略是一个确定的策略。

因此在一个完全可观测的马尔可夫决策过程模型中，如果确定会有一个最优策略的话，这个策略一定是确定性策略，这时候应该选择 Q-learning 方法。

5.6 小结

时序差分是本书介绍的第二个用来解决未知模型强化学习问题的基础方法，也是通过学习采样数据获得最优策略。与蒙特卡罗不同的是，时序差分方法不要求轨迹完整。

时序差分通过学习后继状态的值函数，实现对当前状态值函数的更新。整个时序差分强化学习也遵循了广义策略迭代框架，由策略评估和策略改进两部分组成。根据产生采样的策略和评估改进的策略是否相同，时序差分方法分为在线策略时序差分方法和离线策略时序差分方法。对应的比较著名的方法分别为 Sarsa 和 Q-learning。

本章最后以迷宫寻宝游戏为例，分别对 Sarsa 和 Q-learning 进行详细介绍，并给出了核心代码。相比较而言，离线策略产生的轨迹数据更为丰富，且获得的结果是一个确定性策略，因此在实际中比较常用。

5.7 习题

1. DP、MC、TD、CV 中，哪个不属于强化学习方法？

2. 简述时序差分算法。

3. MC 和 TD 分别是无偏估计吗？MC 和 TD 谁的方差大？

4. 简述 Sarsa，写出其 $Q(s,a)$ 更新公式。它是 on-policy 还是 off-policy？为什么？

5. 简述 Q-learning，写出其 $Q(s,a)$ 更新公式。它是 on-policy 还是 off-policy？为什么？