第3章 动态规划

3.1 动态规划简介

动态规划(Dynamic Programming)是运筹学的一个分支,是求解决策过程最优化的数学方法。20世纪50年代初,美国数学家R. E. Bellman(贝尔曼)等在研究多阶段决策过程的优化问题时提出了著名的最优化原理,把多阶段过程转化为一系列单阶段问题,利用各阶段之间的关系逐个求解,创立了解决这类过程优化问题的新方法——动态规划。

其基本思想是将待求解问题分解成若干个子问题,先求解子问题,然后从这些子问题的解得到原问题的解。适合于用动态规划求解的问题,经分解得到的子问题往往不是互相独立的,因此在解决子问题的时候,其结果通常需要存储起来用以解决后续复杂问题。这样就可以避免大量的重复计算,节省时间。

当问题具有下列特性时,通常可以考虑使用动态规划来求解:第一个特性是一个复杂问题的最优解由数个小问题的最优解构成,可以通过寻找子问题的最优解来得到复杂问题的最优解;第二个特性是子问题在复杂问题内重复出现,使得子问题的解可以被存储起来重复利用。

马尔可夫决策过程(MDP)具有上述两个特性。贝尔曼方程把问题递归为求解子问题,价值函数相当于存储了一些子问题的解,可以复用。因此可以使用动态规划来求解马尔可夫决策过程(MDP)。

使用动态规划算法求解马尔可夫决策过程(MDP)模型,也就是在清楚模型结构(包括状态转移概率、回报等)的基础上,用规划方法来进行策略评估和策略改进,最终获得最优策略。

策略评估(预测):给定一个马尔可夫决策过程模型MDP:$<S,A,P,R,\gamma>$和一个策略π,要求输出基于当前策略π的所有状态的值函数V_π。

策略改进(控制):给定一个马尔可夫决策过程模型MDP:$<S,A,P,R,\gamma>$和一个策略π,要求确定最优值函数V^*和最优策略π^*。

3.2 策略评估

策略评估要解决问题是，给定一个策略 π，如何计算在该策略下的值函数 V_π。

因为实际中涉及的马尔可夫模型规模一般比较大，直接求解效率低，因此可使用迭代法进行求解。考虑应用贝尔曼(Bellman)期望方程进行迭代，公式如下：

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

可见，状态 s 处的值函数 $V_\pi(s)$，可以利用后继状态 s' 的值函数 $V_\pi(s')$ 来表示，依此类推，这种求取值函数的方法称为自举法(Bootstrapping)。

如图 3-1 所示，初始所有状态值函数全部为 0。第 $k+1$ 次迭代求解 $V_\pi(s)$ 时，使用第 k 次计算出来的值函数 $V_k(s')$ 更新计算 $V_{k+1}(s)$。迭代时使用的公式如下：

$$V_{k+1}(s)=\sum_{a\in A}\pi(a\mid s)\left(R_s^a+\gamma\sum_{s'\in S}P_{ss'}^aV_k(s')\right)$$

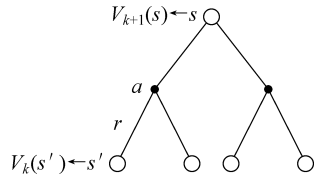


图 3-1 迭代法

对于模型已知的强化学习算法，上式中，$P_{ss'}^a$、$\pi(a\mid s)$、R_s^a 都是已知数，唯一的未知数是值函数，因此该方法通过反复迭代最终将收敛。

3.3 策略改进

计算值函数的目的是利用值函数找到最优策略。既然值函数已经获得，接下来要解决的问题是如何利用值函数进行策略改善，从而得到最优策略。

一个很自然的方法是针对每个状态采用贪心策略对当前策略进行改进，即

$$\pi_{l+1}(s)\in\operatorname*{argmax}_a Q_{\pi_l}(s,a)$$

其中，$Q_\pi(s,a)$可由下式获得：

$$Q_\pi(s,a)=R_s^a+\gamma\sum_{s'\in S}P_{ss'}^aV_\pi(s')$$

在当前策略 π 的基础上，利用贪心算法选取行为，直接将所选择的动作改变为当前最优的动作。

改变之后，策略是不是变得更好了呢？

令动作改变后对应的策略为 π'，a'为在状态 s 下遵循策略 π' 选取的动作，同理 a''为在状态 s'下遵循策略 π' 选取的动作。改变动作的条件是 $Q_\pi(s,a')\geqslant V_\pi(s)$，则可得

$$V_\pi(s)\leqslant Q_\pi(s,a')=\sum_{s'\in S}R_s^{a'}+\gamma P_{ss'}^{a'}V_\pi(s')\leqslant\sum_{s'\in S}R_s^{a'}+\gamma P_{ss'}^{a'}Q_\pi(s',a'')=\cdots=V_{\pi'}(s)$$

可见，值函数对于策略的每一点改进都是单调递增的，因此对于当前策略 π，可以放心地将其改进为

$$\pi' = \max_{a \in A} Q_{\pi}(s,a)$$

直到 π' 与 π 一致，不再变化，收敛至最优策略。

贪心这个词在计算机领域是指在对问题求解时，总是做出在当前看来是最好的选择，不从整体最优上加以考虑，仅针对短期行为（即一步搜索）求得局部最优解。将贪心算法应用在强化学习中求解最优策略实际上可以获得全局最优解，因为贪心策略公式中的值函数 $V_{\pi}(s)$、$Q_{\pi}(s,a)$ 已经考虑了未来的回报。因此在策略改进时，可以放心使用贪心算法求得全局最优解。

3.4 策略迭代

将策略评估算法和策略改进算法合起来便有了策略迭代算法。策略迭代算法通常由策略评估和策略改进两部分构成。在策略评估中，根据当前策略计算值函数。在策略改进中，通过贪心算法选择最大值函数对应的行为。策略评估和策略改进两部分交替进行不断迭代。算法整体流程如图 3-2 所示。

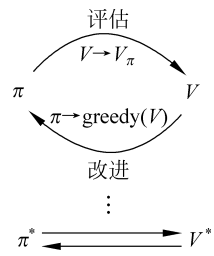


图 3-2　策略迭代

假设我们有一个初始策略 π_1，策略迭代算法首先评估该策略的价值（用 E 表示），得到该策略的值函数 V_{π_1} 或 Q_{π_1}。下一步，策略迭代算法会借助贪心算法对初始策略 π_1 进行改进（用 I 表示），得到 π_2。接着，对改进后的策略 π_2 进行评估，再进一步改进当前策略，如此循环迭代，直到策略收敛至最优。

$$\pi_1 \xrightarrow{E} V_{\pi_1}, \quad Q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} V_{\pi_2}, \quad Q_{\pi_2} \xrightarrow{I} \cdots \pi^* \xrightarrow{E} V^*, \quad Q^* \xrightarrow{I} \pi^*$$

其中，π_1 为初始策略，E 表示策略评估，I 表示策略改进。策略评估过程中，对于任意的策略 π_k，通过贝尔曼期望方程进行迭代计算得到 V_{π_k} 和 Q_{π_k}。例如：

$$V_{\pi_k}(s) = V_k^{\pi_k}(s) = \sum_{a \in A} \pi_k(a \mid s)\left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{k-1}^{\pi_k}(s')\right)$$

策略改进部分，用贪心算法得到更新的策略：

$$\pi'(s) = \arg\max_{a \in A} Q_{\pi_k}(s,a)$$

或者

$$\pi'(s) = \arg\max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{\pi_k}(s')$$

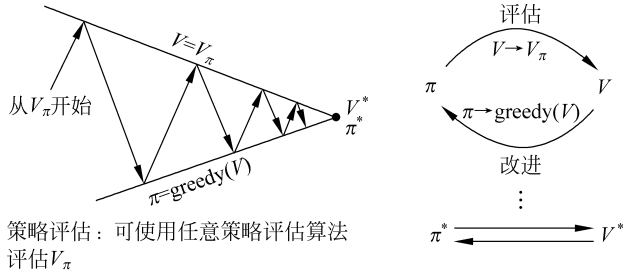
算法流程如下。

算法：策略迭代算法
输入：MDP 五元组 $M = < S, A, P, R, \gamma >$ 初始化值函数 $V(s) = 0$，初始化策略 π_1 为随机策略

Loop
  For $k=0,1,2,3\cdots$ do
    $\forall s\in S'$: $V'(s)=\sum_{a\in A}\pi(a\mid s)\left(R_s^a+\gamma\sum_{s'\in S}P_{ss'}^aV(s')\right)$
    if $V'=V$
    end if
  end for
  $\forall s\in S'$: $\pi'(s)=\underset{a\in A}{\operatorname{argmax}}Q(s,a)$
    or $\pi'(s)=\underset{a\in A}{\operatorname{argmax}}R_s^a+\gamma\sum_{s'\in S}P_{ss'}^aV(s')$
  if $\forall s$: $\pi'(s)=\pi(s)$ then
    break
  else $\pi=\pi'$
  end if
end loop

输出：最优策略 π

在策略评估过程中，往往需要等到值函数收敛之后才能进行策略改进，这其实是没有必要的。可以在进行一次策略评估之后就开始策略改进，如此循环往复执行这两个过程，最终会收敛到最优值函数和最优策略，如图 3-3 所示，这便是广义策略迭代的思想，很多强化学习方法都用到了这种思想。



策略评估：可使用任意策略评估算法评估V_π

策略改进：可使用任意策略改进算法，使新生成的策略$\pi'\geqslant\pi$

图 3-3 广义策略迭代

3.5 值迭代

策略迭代算法在每次进行策略评估时，采用贝尔曼期望方程更新值函数。而值迭代算法借助的是贝尔曼最优方程，直接使用行为回报的最大值更新原来的值，如图 3-4 所示。

$$V_{k+1}(s)=\max_{a\in A}\left(R_s^a+\gamma\sum_{s'\in S}P_{ss'}^aV_k(s')\right)$$

值迭代算法将策略改进视为值函数的改善，每一步都求取最大的值函数，即

$$V_1\rightarrow V_2\rightarrow V_3\rightarrow\cdots\rightarrow V^*$$

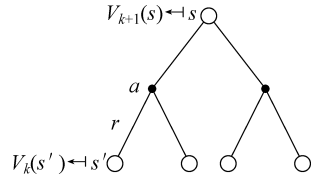


图 3-4　求取 $V_{k+1}(s)$

假设在状态 s 下，我们有一个初始值函数 $V_1(s)$，基于当前状态，我们有多个可选行为 a。每个行为 a 会引发一个立即回报 R_s^a，一个或多个状态转移，如从状态 s 转换至状态 s'。不同状态 s' 对应有不同的值函数 $V_1(s')$ 整个的 $R_s^a+\gamma\sum_{s'\in S}P_{ss'}^a$ $V_1(s)$ 称为 a 的行为回报。值迭代算法直接使用所有行为引发的行为回报中取值最大的那个值来更新原来的值，得到 $V_2(s)$。如此迭代计算，直至值函数收敛，整个过程没有遵循任何策略。

虽然算法中没有给出明确的策略，但是根据公式

$$V_{t+1}(s)\leftarrow\max_{a\in A}Q_{t+1}(s,a)$$

可以看出策略改进是隐含在值迭代过程中执行的。

算法流程如下。

算法：值迭代算法

输入：MDP 五元组 $M=<S,A,P,R,\gamma>$

初始化值函数 $V(s)=0$，收敛阈值 θ

For $k=0,1,2,3\cdots$ do

$\forall s\in S'$：$V'(s)=\max_{a\in A}R_s^a+\gamma\sum_{s'\in S}P_{ss'}^aV(s')$

if $\max_{s\in S}|V'(s)-V(s')|<\theta$ then

break

else $V=V'$

end if

end for

$\forall s\in S'$：$\pi'(s)=\arg\max_{a\in A}Q(s,a)$

or $\pi'(s)=\arg\max_{a\in A}R_s^a+\gamma\sum_{s'\in S}P_{ss'}^aV(s')$

输出：最优策略 π'

3.6 实例讲解

本节以在5×5的网格世界中寻找宝藏为例，分别使用策略迭代和值迭代算法寻找最优策略，并比较两种算法在处理上的异同，以及两者的运行效率。

3.6.1 “找宝藏”环境描述

1. 环境描述

首先构建寻宝环境的马尔可夫决策模型 $M=<S,A,P,R,\gamma>$。

如图3-5所示为5×5的网格世界，其状态空间为 $S=\{0,1,\cdots,24\}$，其中状态8为宝藏区。动作空间 $A=\{\text{UP},\text{RIGHT},\text{DOWN},\text{LEFT}\}$，宝藏区回报为 $r=0$，其他位置回报为 $r=-1$。

0	1	2	3	4
5	6	7	宝藏	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

图3-5 5×5网格世界

当智能体位于网格世界边缘位置时，任何使其离开网格世界的行为都会使其停留在当前位置。例如，当位于0位置时，采取向上的行为，$a=\text{UP}$，智能体获取−1的回报后，重新回到位置0。当智能体位于宝藏区时，则无论采取何种行为，均会产生0回报，且位置不变。当智能体位于其他位置时，采取任何一个行为，则会向相应的方向移动一格。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

接下来基于gym构建寻找宝藏的环境，以下代码主要定义了寻宝藏MDP模型的状态空间、行为空间、状态转移概率和回报等信息。此环境代码较为简单，读者可结合注释，加深对此代码的理解。

```
import numpy as np
import sys
from gym.envs.toy_text import discrete
  #定义动作空间
UP = 0
RIGHT = 1
DOWN = 2
LEFT = 3
  #定义宝藏区(宝藏所在的状态)
DONE_LOCATION = 8
  #定义网格世界环境模型
class GridworldEnv(discrete.DiscreteEnv):
```

```
    """定义了一个 5×5 网格,如下所示:
                                        o o o o o
                                        o o o T o
                                        o o o o o
                                        o x o o o
                                        o o o o o
    x 是智能体的位置,T 是宝藏位置,智能体的目标是到达宝藏位置。在每个状态都可以采取 4 种
行为(UP = 0,RIGHT = 1,DOWN = 2,LEFT = 3)。每一步移动都会收到 - 1 回报,直到达到宝藏状
态"""
    def __init__(self, shape=[5,5]):
        if not isinstance(shape, (list, tuple)) or not len(shape) == 2:
            raise ValueError('shape argument must be a list/tuple of length 2')
        self.shape = shape
    #np.prod 函数是乘法,如传进去[4,5,6],那么就是 120。而这里传进去的就是 5,5,
    #得到 25 就代表元素的个数。Max_y 和 Max_x 得到横排和竖排的个数
        nS = np.prod(shape) #状态个数
        nA = 4 #动作个数
        MAX_Y = shape[0] #5
        MAX_X = shape[1] #5
    #grid 是创建一个 5x5 的表格,而 np.nditer 是为这个表格索引排序
    #flags=['multi_index']表示对 grid 进行多重索引,如(1,3)
        P = {}
        grid = np.arange(nS).reshape(shape)
        it = np.nditer(grid, flags=['multi_index'])
    #s 得到的是一个索引值。p[s][a]记录的是对于每一个格子采用四种走法,即上右下左之后产
    #生的回报、下一个状态、是否达到宝藏区等信息。
    #is_done 记录是否到达宝藏区,如果到达则返回 true,否则返回 false
    #Reward 为回报,除了宝藏区,其他状态都是 - 1 回报
        while not it.finished:
            s = it.iterindex
            y, x = it.multi_index
            P[s] = {a: [] for a in range(nA)}
            is_done = lambda s: s == DONE_LOCATION
            reward = 0.0 if is_done(s) else -1.0
    #p[s][a]中第一个参数是状态转移概率,为 1。第二个参数代表到达下一个的位置。
    #第三个参数 reward 表示回报。最后一个参数为 True 表示达到宝藏区,为 false 表示未到达宝
    #藏区
             if is_done(s):
    #如果位于宝藏区,则采取上下左右之后,依然会转换到宝藏区
                P[s][UP] = [(1, s, reward, True)]
                P[s][RIGHT] = [(1, s, reward, True)]
                P[s][DOWN] = [(1, s, reward, True)]
                P[s][LEFT] = [(1, s, reward, True)]
            else:
    #宝藏区之外的状态,采取上下左右之后,所进行的状态转换
```

```
                ns_up = s if y == 0 else s - MAX_X
                ns_right = s if x == (MAX_X - 1) else s + 1
                ns_down = s if y == (MAX_Y - 1) else s + MAX_X
                ns_left = s if x == 0 else s - 1
                P[s][UP] = [(1, ns_up, reward, is_done(ns_up))]
                P[s][RIGHT] = [(1, ns_right, reward, is_done(ns_right))]
                P[s][DOWN] = [(1, ns_down, reward, is_done(ns_down))]
                P[s][LEFT] = [(1, ns_left, reward, is_done(ns_left))]
            it.iternext()
        isd = np.ones(nS) / nS
        self.P = P
        super(GridworldEnv, self).__init__(nS, nA, P, isd)
```

3.6.2 策略迭代

1. 算法详情

使用策略迭代法对此问题进行求解。假设初始策略为均匀随机策略：

$\pi(\mathrm{UP}\mid\cdot)=0.25,\pi(\mathrm{RIGHT}\mid\cdot)=0.25,\pi(\mathrm{DOWN}\mid\cdot)=0.25,\pi(\mathrm{LEFT}\mid\cdot)=0.25$

(1) 首先评估给定随机策略下的值函数，使用贝尔曼期望方程迭代计算直至值函数收敛。初始所有状态值函数全部为 0。在对值函数进行迭代计算时，使用如下公式：

$$V_{k+1}(s)=\sum_{a\in A}\pi(a\mid s)\left(R_s^a+\gamma\sum_{s'\in S}P_{ss'}^aV_k(s')\right)$$

$k=0$ 时，随机策略下的值函数 $V_0^\pi(s)$ 为：

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

$k=1$ 时，随机策略下的值函数 $V_1^\pi(s)$ 为：

−1.	−1.25	−1.3125	−1.328 125	−1.332 031 25
−1.25	−1.625	−1.734 375	0	−1.333 007 81
−1.3125	−1.734 375	−1.867 187 5	−1.466 796 88	−1.699 951 17
−1.328 125	−1.765 625	−1.908 203 12	−1.843 75	−1.885 925 29
−1.332 031 25	−1.774 414 06	−1.920 654 3	−1.941 101 07	−1.956 756 59

$k=2$ 时，随机策略下的值函数 $V_2^{\pi}(s)$ 为：

−2.125	−2.578 125	−2.738 281 25	−2.349 609 38	−2.586 669 92
−2.578 125	−3.156 25	−2.940 429 69	0	−2.404 907 23
−2.738 281 25	−3.381 835 94	−3.424 316 41	−2.742 004 39	−3.183 197 02
−2.791 015 62	−3.463 867 19	−3.663 146 97	−3.558 044 43	−3.645 980 83
−2.807 373 05	−3.491 577 15	−3.754 119 87	−3.802 505 49	−3.840 499 88

$k=3$ 时，随机策略下的值函数 $V_3^{\pi}(s)$ 为：

−3.351 562 5	−3.956 054 69	−3.996 093 75	−3.233 093 26	−3.702 835 08
−3.956 054 69	−4.558 593 75	−3.994 750 98	0	−3.322 734 83
−4.216 796 88	−4.915 893 55	−4.828 948 97	−3.892 547 61	−4.511 115 07
−4.319 763 18	−5.097 595 21	−5.309 677 12	−5.162 677 76	−5.290 068 39
−4.356 521 61	−5.174 953 46	−5.510 313 99	−5.578 999 28	−5.637 516 86

$k=4$ 时，随机策略下的值函数 $V_4^{\pi}(s)$ 为：

−4.653 808 59	−5.291 137 7	−5.128 768 92	−4.016 174 32	−4.686 144 83
−5.346 313 48	−5.887 023 93	−4.961 185 46	0	−4.129 998 68
−5.699 691 77	−6.378 314 97	−6.135 431 29	−4.952 306 03	−5.720 872 04
−5.868 392 94	−6.682 834 63	−6.872 814 42	−6.673 547 03	−6.830 501 08
−5.939 097 4	−6.826 799 87	−7.197 231 89	−7.271 823 76	−7.344 339 64

$k=41$ 时，随机策略下的值函数 $V_{41}^{\pi}(s)$ 为：

−35.744 947 27	−32.126 418 85	−24.538 604	−15.064 957 31	−16.926 123 27
−36.995 083 77	−33.063 404 23	−23.041 452 41	0	−15.190 355 34
−39.424 276 68	−36.729 162 81	−30.883 369 16	−23.265 333 84	−25.005 309 54
−41.894 676 92	−40.310 332 99	−37.114 272 53	−33.742 770 78	−33.113 846 48
−43.387 487 16	−42.362 064 91	−40.283 327 44	−38.184 851 34	−37.281 958 44

$k=338$ 时，随机策略下的值函数 $V_{338}^{\pi}(s)$ 收敛，收敛结果为：

−47.136 143 06	−41.727 086 85	−31.242 294 47	−18.621 143 29	−20.621 140 63
−48.545 230 94	−42.802 841 77	−29.378 665 22	0	−18.621 145 76
−51.696 732 16	−47.560 396 44	−39.469 530 83	−29.378 669 62	−31.242 303 61
−54.984 595 17	−52.272 495 81	−47.560 403 97	−42.802 855 06	−41.727 106 15
−56.984 585 38	−54.984 604 26	−51.696 748 9	−48.545 254 16	−47.136 173 06

对应代码如下。

```
V = np.zeros(env.nS)
i = 0

while True:
    value_delta = 0
    # 遍历各状态
    for s in range(env.nS):
        v = 0
        # 遍历各行为的概率(上,右,下,左)
        for a, action_prob in enumerate(policy[s]):
    # 对于每个行为确认下个状态
    # 四个参数: prob:概率, next_state: 下一个状态的索引, reward: 回报,
    #done: 是否是终止状态
            for prob, next_state, reward, done in env.P[s][a]:
                # 使用贝尔曼期望方程进行状态值函数的求解
                v += action_prob * prob * (reward + discount_factor * V[next_state])
        # 求出各状态和上一次求得状态的最大差值
        value_delta = max(value_delta, np.abs(v - V[s]))
        V[s] = v
```

(2) 使用如下公式：

$$\pi^*(a \mid s) = \underset{a \in A}{\operatorname{argmax}} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V^*(s')$$

对收敛的值函数 $V_{338}^{\pi}(s)$，使用贪心算法进行策略改进，求取 $V_{338}^{\pi}(s)$ 对应的改进后的策略 π_1，则有 π_1：

→	→	→	↓	←
→	→	→	宝藏	←
→	→	↑	↑	↑
↑	↑	↑	↑	↑
↑	→	↑	↑	↑

对应代码如下：

```
# 评估当前的策略,输出为各状态的当前的状态值函数
V = policy_eval_fn(policy, env, discount_factor)
# 定义一个当前策略是否改变的标识
policy_stable = True

# 遍历各状态
for s in range(env.nS):
    # 取出当前状态下最优行为的索引值
    chosen_a = np.argmax(policy[s])

    # 初始化行为数组[0,0,0,0]
    action_values = np.zeros(env.nA)
    for a in range(env.nA):
        # 遍历各行为
        for prob, next_state, reward, done in env.P[s][a]:
            # 根据各状态值函数求出行为值函数
            action_values[a] += prob * (reward + discount_factor * V[next_state])

    # 输出一个状态下所有的可能性
    best_a_arr, policy_arr = get_max_index(action_values)
    if chosen_a not in best_a_arr:
        policy_stable = False
    policy[s] = policy_arr
```

(3) 继续使用贝尔曼期望方程求取当前策略 π_1 下的值函数，直至值函数收敛。计算过程与(1)相同。

经过 5 轮迭代，值函数收敛，得到 $V_5^{\pi_1}(s)$为：

−4	−3	−2	−1	−2
−3	−2	−1	0	−1
−4	−3	−2	−1	−2
−5	−4	−3	−2	−3
−6	−5	−4	−3	−4

(4) 针对值函数 $V_5^{\pi_1}(s)$，进行第二次策略改善，得到 π_2。

(5) 继续求取 π_2 对应的值函数(策略评估)，针对收敛的值函数 $V_5^{\pi_2}(s)$进行第三次策略改进，得到 π_3。经过三次策略改进，策略收敛。

对应最优值函数 $V^*(V_5^{\pi_2}(s))$为：

−4	−3	−2	−1	−2
−3	−2	−1	0	−1
−4	−3	−2	−1	−2
−5	−4	−3	−2	−3
−6	−5	−4	−3	−4

最优策略 $\pi^*(\pi_3)$为：

↓→	↓→	↓→	↓	←↓
→	→	→	宝藏	←
↑→	↑→	↑→	↑	←↑
↑→	↑→	↑→	↑	←↑
↑→	↑→	↑→	↑	←↑

代码如下。

```
if policy_stable:
      return policy, V
```

policy_stable 初始为 True，策略未达到收敛时不会输出策略和值函数，只有达到最优收敛之后结束迭代时，才会输出最优策略和值函数。

2. 核心代码

策略迭代算法主要包含两步，即策略评估和策略改进，重点包含了两个函数 policy_eval 方法和 policy_improvement 方法。

policy_eval 方法是策略评估方法，输入要评估的策略 policy，环境 env，折扣因子 discount_factor，阈值 threshold。输出当前策略下收敛的值函数 V。

policy_improvement 是策略改进方法，输入为环境 env，策略评估函数 policy_eval，折扣因子 discount_factor。输出为最优值函数和最优策略。

(1) 首先展示策略评估方法：

```
def policy_eval(policy, env, discount_factor = 1, threshold = 0.00001): #
    # 初始化各状态的状态值函数
    V = np.zeros(env.nS)
    i = 0
```

```
    while True:
        value_delta = 0
        # 遍历各状态
        for s in range(env.nS):
            v = 0
            # 遍历各行为的概率(上,右,下,左)
            for a, action_prob in enumerate(policy[s]):
                # 对于每个行为确认下个状态
                # P[s][a]含四个参数。prob:状态转换概率, next_state: 下一个状态, reward:
                #回报, done: 是否是终止状态(宝藏区)
                for prob, next_state, reward, done in env.P[s][a]:
                    # 使用贝尔曼期望方程进行状态值函数的求解
                    v += action_prob * prob * (reward + discount_factor * V[next_state])
            # 求出各状态和上一次求得状态的最大差值
            value_delta = max(value_delta, np.abs(v - V[s]))
            V[s] = v
        i += 1
        # 当前循环得出的各状态和上一次状态的最大差值小于阈值,则收敛,停止运算
        if value_delta < threshold:
            break
    return np.array(V)
```

(2) 策略改进方法通过调用策略评估方法实现。

```
# 定义两个全局变量用来记录运算的次数
v_num = 1
i_num = 1
# 根据传入的四个行为选择值函数最大的索引,返回的是一个索引数组和一个行为策略
def get_max_index(action_values):
    indexs = []
    policy_arr = np.zeros(len(action_values))
    action_max_value = np.max(action_values)

    for i in range(len(action_values)):
        action_value = action_values[i]

        if action_value == action_max_value:
            indexs.append(i)
            policy_arr[i] = 1
    return indexs,policy_arr

# 将策略中的每行可能行为改成元组形式,方便对多个方向的表示
def change_policy(policys):
    action_tuple = []
```

```
    for policy in policys:
        indexs, policy_arr = get_max_index(policy)
        action_tuple.append(tuple(indexs))

    return action_tuple
#policy_improvement 是策略改进方法,输入为环境 env,策略评估函数 policy_eval,折扣因子
#输出为最优值函数和最优策略
def policy_improvement(env, policy_eval_fn = PolicyEvaluationSolution.policy_eval, discount_
factor = 1.0)
      # 初始化一个随机策略
    policy = np.ones([env.nS, env.nA]) / env.nA

    while True:
        global i_num
        global v_num

        v_num = 1
        # 评估当前的策略,输出为各状态的当前的状态值函数
        V = policy_eval_fn(policy, env, discount_factor)
        # 定义一个当前策略是否改变的标识
        policy_stable = True

        # 遍历各状态
        for s in range(env.nS):
            # 取出当前状态下最优行为的索引值
            chosen_a = np.argmax(policy[s])

            # 初始化行为数组[0,0,0,0]
            action_values = np.zeros(env.nA)
            for a in range(env.nA):
                # 遍历各行为
                for prob, next_state, reward, done in env.P[s][a]:
                    # 根据各状态值函数求出行为值函数
                    action_values[a] += prob * (reward + discount_factor * V[next_state])

            #因为 np.argmax(action_values)只会选取第一个最大值出现的索引,会丢掉其他方
            #向的可能性,所以这个方法会输出一个状态下所有的可能性
            best_a_arr, policy_arr = get_max_index(action_values)

            # 如果求出的最大行为值函数的索引(方向)没有改变,则定义当前策略未改变,收敛输出,
            # 否则将当前状态中将有最大行为值函数的方向置 1,其余方向置 0
            if chosen_a not in best_a_arr:
                policy_stable = False
            policy[s] = policy_arr
        i_num = i_num + 1
```

```
        # 如果当前策略没有发生改变,即已经到了最优策略,返回
        if policy_stable:
            return policy, V
env = GridworldEnv()
policy, v = policy_improvement(env)
update_policy_type = change_policy(policy)
```

3.6.3 值迭代

1. 算法详情

在进行一次策略评估(即求取当前策略下的值函数)之后就进行策略改进,这种方法称为值函数迭代算法。即,在每次进行值函数计算时,直接选择那个使得值函数最大的行为。

(1) 初始值函数 $V_0^{\pi}(s)$为($k=0$):

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

代码如下。

```
V = np.zeros(env.nS)
```

(2) 使用贝尔曼最优方程,直接使用当前状态的最大行为值函数更新当前状态值函数。

$$V_{k+1}(s)=\max_{a\in A}\left(R_s^a+\gamma\sum_{s'\in S}P_{ss'}^aV_k(s')\right)$$

第一次迭代($k=1$),得 $V_1^{\pi}(s)$为:

−1	−1	−1	−1	−1
−1	−1	−1	0	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1
−1	−1	−1	−1	−1

第二次迭代($k=2$),得 $V_2^{\pi}(s)$ 为:

−2	−2	−2	−1	−2
−2	−2	−1	0	−1
−2	−2	−2	−1	−2
−2	−2	−2	−2	−2
−2	−2	−2	−2	−2

第三次迭代($k=3$),得 $V_3^{\pi}(s)$ 为:

−3	−3	−2	−1	−2
−3	−2	−1	0	−1
−3	−3	−2	−1	−2
−3	−3	−3	−2	−3
−3	−3	−3	−3	−3

第七次之后各状态的最优行为值函数已经收敛,$V_7^{\pi}(s)$ 为:

−4	−3	−2	−1	−2
−3	−2	−1	0	−1
−4	−3	−2	−1	−2
−5	−4	−3	−2	−3
−6	−5	−4	−3	−4

对应的最优策略为:

↓ →	↓ →	↓ →	↓	← ↓
→	→	→	宝藏	←
↑ →	↑ →	↑ →	↑	← ↑
↑ →	↑ →	↑ →	↑	← ↑
↑ →	↑ →	↑ →	↑	← ↑

求取各状态下最大行为值函数代码如下：

```
V = np.zeros(env.nS)

while True:
    # 停止条件
    delta = 0
    # 遍历每个状态
    for s in range(env.nS):
        # 计算当前状态的各行为值函数
        q = one_step_lookahead(s, V)
        # 找到最大行为值函数
        best_action_value = np.max(q)
        # 值函数更新前后求差
        delta = max(delta, np.abs(best_action_value - V[s]))
        # 更新当前状态的值函数,即将最大的行为值函数赋值给当前状态,用以更新当前状态的
        # 值函数
        V[s] = best_action_value

    i_num += 1
    # 如果当前状态值函数更新前后相差小于阈值,则说明已经收敛,结束循环
    if delta < threshold:
        break
```

求取最优策略的代码如下。

```
# 初始化策略
policy = np.zeros([env.nS, env.nA])
# 遍历各状态
for s in range(env.nS):
    # 根据已经计算出的 V,计算当前状态的各行为值函数
    q = one_step_lookahead(s, V)
    # 求出当前最大行为值函数对应的动作索引
    # 输出一个状态下所有的可能性
    best_a_arr, policy_arr = get_max_index(q)
    # 将当前所有最优行为赋值给当前状态
    policy[s] = policy_arr
```

2. 核心代码

值迭代方法将策略改进视为值函数的改进，每一步都求取最大行为值函数，用最大行为值函数更新当前状态值函数。值迭代方法为 value_iteration，输入为环境 env，阈值 threshold，折扣因子 discount_factor。输出为最优值函数和最优策略。在进行值函数更新时，使用了最优贝尔曼方程。其中方法 one_step_lookahead 用以求取当前状态的所有行为

值函数。

代码如下。

```
from lib.envs.gridworld import GridworldEnv
# 定义1个全局变量用来记录运算的次数
i_num = 1
# 根据传入的四个行为选择值函数最大的索引,返回的是一个索引数组和一个行为策略
def get_max_index(action_values):
    indexs = []
    policy_arr = np.zeros(len(action_values))
    action_max_value = np.max(action_values)
    for i in range(len(action_values)):
        action_value = action_values[i]
        if action_value == action_max_value:
            indexs.append(i)
            policy_arr[i] = 1
    return indexs,policy_arr
# 将策略中的每行可能行为改成元组形式,方便对多个方向的表示
def change_policy(policys):
    action_tuple = []
    for policy in policys:
        indexs, policy_arr = get_max_index(policy)
        action_tuple.append(tuple(indexs))
    return action_tuple

def value_iteration(env, threshold=0.0001, discount_factor=1.0):
    """
     值迭代算法
          env表示环境
            env.P[s][a](prob,next_state,reward,done)记录状态转移概率、下一个状态、回报、
是否结束
            env.nS是环境状态空间
            env.nA是环境动作空间
            discount_factor:折扣因子
            返回:最优策略和最优值函数
    """
    global i_num

    def one_step_lookahead(state, V)
    #求取当前状态的所有行为值函数
        q = np.zeros(env.nA)
        for a in range(env.nA):
            for prob, next_state, reward, done in env.P[state][a]:
                q[a] += prob * (reward + discount_factor * V[next_state])
        return q
```

```
    V = np.zeros(env.nS)

    while True:
        # 停止条件
        delta = 0
        # 遍历每个状态
        for s in range(env.nS):
            # 计算当前状态的各行为值函数
            q = one_step_lookahead(s, V)
            # 找到最大行为值函数
            best_action_value = np.max(q)
            # 值函数更新前后求差
            delta = max(delta, np.abs(best_action_value - V[s]))
            # 更新当前状态的值函数,即将最大的行为值函数赋值给当前状态,用以更新当前
            # 状态的值函数
            V[s] = best_action_value
               i_num += 1
        # 如果当前状态值函数更新前后相差小于阈值,则说明已经收敛,结束循环
        if delta < threshold:
              break

    # 初始化策略
    policy = np.zeros([env.nS, env.nA])
    # 遍历各状态
    for s in range(env.nS):
        # 根据已经计算出的 V,计算当前状态的各行为值函数
        q = one_step_lookahead(s, V)
        # 求出当前最大行为值函数对应的动作索引
        # 在初始策略中对应的状态上将最大行为值函数方向置 1,其余方向保持不变,仍为 0
        # 因为 np.argmax(action_values)只会选取第一个最大值出现的索引,
        # 会丢掉其他方向的可能性,所以以下方法会输出一个状态下所有的可能性
        best_a_arr, policy_arr = get_max_index(q)
        # 将当前所有最优行为赋值给当前状态
        policy[s] = policy_arr
    return policy, V
env = GridworldEnv()
policy, v = value_iteration(env)
update_policy_type = change_policy(policy)
```

3.6.4 实例小结

本节分别使用策略迭代和值迭代解决同一个网格世界寻找宝藏的问题,均给出了最优策略。

从策略迭代的代码可以看到,进行策略改进之前需要得到收敛的值函数。值函数的收

敛往往需要很多次迭代。例如，在第一次策略改进之前，进行了 338 次迭代。进行策略改进之前一定要等到策略值函数收敛吗？答案是不需要。依然以第一次策略改进之前策略评估为例，策略评估迭代 41 次和迭代 338 次所得到的贪心策略是一样的，可见策略迭代方法在效率方面存在问题。

相比而言，值迭代方法就高效得多，每一次都直接用最大的行为值函数更新当前状态的值函数，直至值函数收敛，输出最优策略，其解决同样的问题仅仅需要 7 次迭代。

3.7 小结

动态规划是用来解决已知模型强化学习问题的基础方法。具体来说，它包括策略迭代和值迭代两类算法。策略迭代通过策略评估和策略改进交替进行求取最优策略。值迭代在进行每一步的策略评估时，直接求取最优值函数，值函数收敛后求取最优策略。本章通过网格世界寻找宝藏的实例，详细介绍了两种算法的求解过程和代码，并对两者的效率进行了比较，相比而言，值迭代算法具有更高的效率。

3.8 习题

1. 动态规划是什么？它适合解决什么类型的问题？
2. 什么是策略评估和策略改进？
3. 简述策略迭代算法和值迭代算法。
4. 策略迭代和值迭代这两个算法的区别和联系是什么？
5. 假设在本章案例的迷宫问题中，对每个状态的立即奖赏加上一个常量 C，这样对最终结果是否有影响？请给出解释。