值迭代和策略迭代
以GYMNASIUM冰湖环境为例

强化学习实验报告

张恒硕

2025 年 3 月 16 日

南開大學
Nankai University

目 录

图 片

表 格

1 实验目的

1. 解析gymnasium包中的一个离散状态环境的马尔可夫过程元素。
2. 对该环境进行值迭代和策略迭代，获得最优策略和价值矩阵。
3. 比较分析值迭代和策略迭代的区别和特点。

2 实验原理

2.1 值迭代

迭代更新状态的价值函数，选择当前能带来最大长期回报的动作，逐步逼近最优策略。

步骤

1. 初始化：初始化所有状态的值函数$V(s)$为0。
2. 迭代更新：重复以下公式直至收敛：

$$V_{k+1}(s) = \max_a [R(s,a) + \gamma \sum_{s'} P(s'|s,a) V_k(s')]$$

3. 最优策略：按以下公式选取策略：

$$\pi^*(s) = \arg\max_a [R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s')]$$

特点

- 优点：保证收敛到全局最优策略，尤其适用于复杂策略空间。
- 缺点：每次迭代需遍历所有动作，计算成本较高。

2.2 策略迭代

交替执行策略评估和策略改进，先固定策略优化值函数，再基于新值函数优化策略。

步骤

1. 初始化：选择任意策略π，如随机策略。
2. 策略评估：固定策略，迭代计算值函数直至收敛：

$$V_{k+1}(s) = \sum_{a} \pi(a|s)[R(s,a) + \gamma \sum_{s'} P(s'|s,a)V_k(s')]$$

3. 策略改进：生成新策略：

$$\pi'(s) = \arg\max_{a}[R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^{\pi}(s')]$$

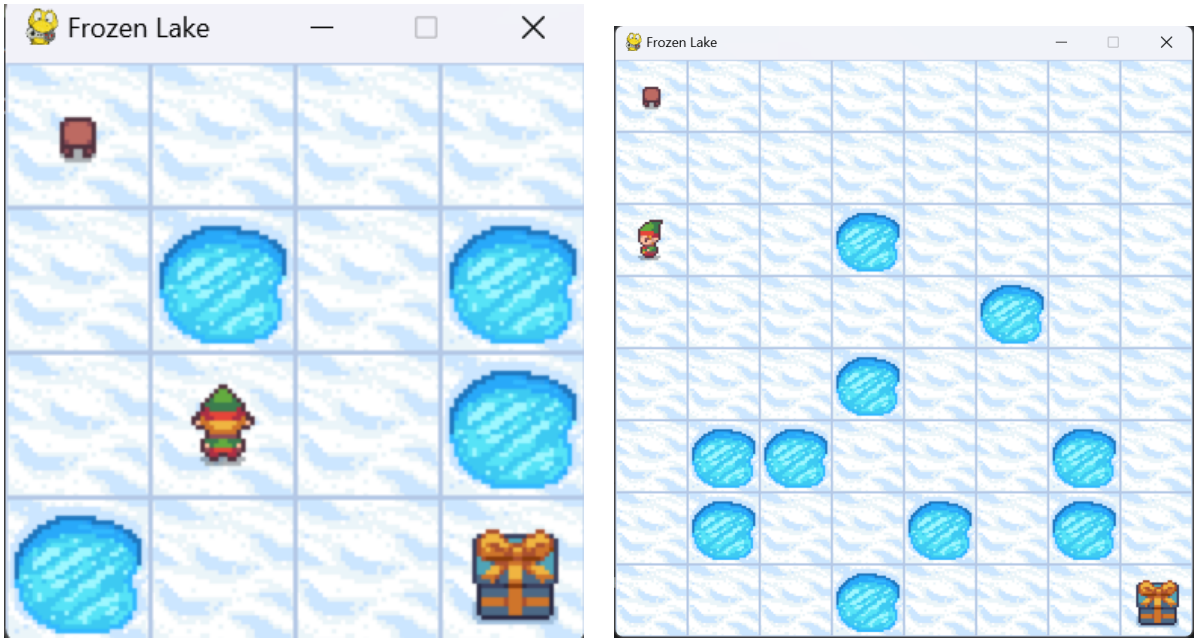
4. 终止条件：若新策略与原策略一致，停止迭代得到最优策略，否则回到第二步。

特点

- 优点：通常比值迭代更快收敛，尤其在策略空间较小时。
- 缺点：依赖初始策略质量，可能陷入局部最优。

3 环境分析

上次实验中选取的环境MountainCarContinuous-v0是连续状态的，不利于迭代操作的实现，本次更换为FrozenLake-v1环境，即冰湖环境，其是一个离散状态的网格化环境，目标是在有洞的冰面抵达目标点。图1 on the following page给出了两个尺寸的默认地图。



(a) 4*4地图　　(b) 8*8地图

图 1: 默认地图

以下结合源码对其马尔可夫过程元素进行分析。

状态空间

Listing 1: 状态空间

```
self.nrow, self.ncol = nrow, ncol = desc.shape
self.initial_state_distrib = np.array(desc == b"S").astype("float64").ravel()
self.initial_state_distrib /= self.initial_state_distrib.sum()
self.observation_space = spaces.Discrete(nrow * ncol)
```

状态空间大小是格子数量，其将二维网格拼接成一维长向量来编码状态。

动作空间

Listing 2: 动作空间

```
nA = 4
self.action_space = spaces.Discrete(nA)
```

四个动作$\{0,1,2,3\}$依次是向左、向下、向右、向上移动。

奖励函数

Listing 3: 奖励函数

```
def update_probability_matrix(row, col, action):
    new_row, new_col = inc(row, col, action)
    new_state = to_s(new_row, new_col)
    new_letter = desc[new_row, new_col]
    terminated = bytes(new_letter) in b"GH"
    reward = float(new_letter == b"G")
    return new_state, reward, terminated
```

只有到达目标才能获得奖励，其他状态没有奖励。

状态转移

Listing 4: 状态转移

```
for row in range(nrow):
    for col in range(ncol):
        s = to_s(row, col)
        for a in range(4):
            li = self.P[s][a]
            letter = desc[row, col]
            if letter in b"GH":
                li.append((1.0, s, 0, True))
            else:
                if is_slippery:
                    for b in [(a - 1) % 4, a, (a + 1) % 4]:
                        li.append(
                            (1.0 / 3.0, *update_probability_matrix(row, col, b))
                        )
                else:
                    li.append((1.0, *update_probability_matrix(row, col, a)))
```

小人在移动时，如果进洞或到达目标将不再运动。其他状况下，根据滑性参数进行移动。如果不滑则向指定方向移动，滑则概率偏移。

折扣率　代码中未显式定义。

4 关键代码解析

以下给出针对这个环境的值迭代和策略迭代的核心函数。

4.1 值迭代

Listing 5: 值迭代

```
# 值迭代
def value_iteration():
    V = np.zeros(obs_n)
    iters = 0
    while True:
        delta = 0
        # 遍历所有状态
        for s in range(obs_n):
            v = V[s]
            action_values = []
            # 计算动作价值
            for a in range(act_n):
                action_values.append(sum(p * (r + GAMMA * V[s_]) for p, s_, r, _ in P[
                    s][a]))
            # 更新值函数
            V[s] = max(action_values) if action_values else 0
            # 更新本轮最大变化量
            delta = max(delta, abs(v - V[s]))
        iters += 1
        if delta < THETA:
            break
    # 最优策略
```

```
    policy = np.zeros(obs_n)
    for s in range(obs_n):
        action_values = []
        for a in range(act_n):
            action_values.append(sum(p * (r + GAMMA * V[s_]) for p, s_, r, _ in P[s][a
                ]))
        best_action = np.argmax(action_values) if action_values else 0
        policy[s] = best_action
    return policy, V, iters
```

值迭代过程中，专注于更新值函数，提高对动作价值的评估精确度，最终用收敛的值函数来获取最优策略。

4.2 策略迭代

Listing 6: 策略迭代

```
# 策略评估
def policy_evaluation(policy):
    V = np.zeros(obs_n)
    eval_iters = 0
    while True:
        delta = 0
        # 遍历所有状态
        for s in range(obs_n):
            v = V[s]
            # 计算值函数
            V[s] = sum(p * (r + GAMMA * V[s_]) for p, s_, r, _ in P[s][policy[s]])
            # 更新本轮最大变化量
            delta = max(delta, abs(v - V[s]))
        eval_iters += 1
        if delta < THETA:
            break
    return V, eval_iters
```

```
76
77 # 策略改进
78 def policy_improvement(V):
79     policy = np.zeros(obs_n)
80     for s in range(obs_n):
81         action_values = []
82         # 计算动作价值
83         for a in range(act_n):
84             action_values.append(sum(p * (r + GAMMA * V[s_]) for p, s_, r, _ in P[s][a
                ]))
85         # 贪婪策略
86         best_action = np.argmax(action_values) if action_values else 0
87         policy[s] = best_action
88     return policy
89
90 # 策略迭代
91 def policy_iteration():
92     iters = 0
93     total_eval_iters = 0
94     # 初始化策略为随机动作
95     policy = np.zeros(obs_n)
96     for s in range(obs_n):
97         policy[s] = env_unwrapped.action_space.sample()
98     while True:
99         # 策略评估
100         V, eval_iters = policy_evaluation(policy)
101         total_eval_iters += eval_iters
102         # 策略改进
103         new_policy = policy_improvement(V)
104         iters += 1
105         # 不再更新则停止迭代
106         if np.array_equal(policy, new_policy):
107             break
108         policy = new_policy
109     return policy, V, iters, total_eval_iters
```

策略迭代过程中，评估与改进是同步的。每轮迭代中，有预先设定的策略，按其将值函数更新到收敛，然后利用结果进行贪婪选择，获得新的策略。在策略更新的过程中，对动作的价值评估也会同步收敛，但速度与初始化的策略有关。

5 实验结果与分析

5.1 实验结果

为了比较两种迭代方法的性能，在以下对比实验中，固定了折扣率GAMMA $= 0.9$和迭代终止阈值THETA $= 1e-6$，前者越大、后者越小，都会增加迭代次数，延长迭代时间，同时提高价值评估精度。这里选取的参数已经能保障结果收敛。

在两个尺寸的默认地图上测试时，两种方法均获得了一致的最优策略和价值矩阵，可参见下图图2。

```
最优策略：[0. 3. 0. 3. 0. 0. 0. 0. 3. 1. 0. 0. 0. 2. 1. 0.]
最优值函数：[0.06888624 0.06141117 0.07440763 0.05580502 0.09185097 0.
 0.11220727 0.         0.14543392 0.24749561 0.29961676 0.
 0.         0.37993504 0.63901974 0.         ]
```

(a) 4*4地图

```
最优策略：[3. 2. 2. 2. 2. 2. 2. 2. 3. 3. 3. 3. 2. 2. 2. 1. 3. 3. 0. 0. 2. 3. 2. 1.
 3. 3. 3. 1. 0. 0. 2. 1. 3. 3. 0. 0. 2. 1. 3. 2. 0. 0. 0. 1. 3. 0. 0. 2.
 0. 0. 1. 0. 0. 0. 0. 2. 0. 1. 0. 0. 1. 1. 1. 0.]
最优值函数：[0.00640709 0.00854531 0.01229829 0.01778774 0.02508078 0.03246966
 0.03957022 0.04297739 0.00602067 0.00764255 0.01090966 0.01642506
 0.02605305 0.03619313 0.04935386 0.05730388 0.00508728 0.00585089
 0.00677358 0.         0.02557004 0.03882063 0.0676391  0.08435552
 0.00422306 0.0047676  0.00581836 0.00785349 0.0203601  0.
 0.09175454 0.12919069 0.00317883 0.00319507 0.00270403 0.
 0.03444354 0.06195115 0.10901886 0.2096906  0.00186623 0.
 0.         0.01085063 0.0325007  0.06304155 0.         0.36008752
 0.00117704 0.         0.00137657 0.00366816 0.         0.11568666
 0.         0.63051369 0.00088166 0.00077211 0.00092074 0.
 0.13824885 0.32258065 0.61443932 0.         ]
```

(b) 8*8地图

图 2: 迭代结果

下表表1给出了两种方法的迭代轮数比较，这能在一定程度上反应二者的运行速度。在比较时，值迭代轮数应乘以可行动作数，策略迭代数应选取括号中的评估次数。以4*4地图数据为例，值迭代是$4 \times 60 = 240$（轮），策略迭代则是231（轮），二者相当。

表 1: 两种方法迭代轮数对比

尺寸	值迭代	策略迭代
4*4	60	6(231)
8*8	63	5(281)

值得注意的是，值迭代是固定的，其过程不会在重复执行中变化。而策略迭代由于初始策略的随机性，造成迭代轮数浮动较大，以上选取的是反复实验中的中位数。

5.2 实验分析

根据以上结果与数据，结合原理，得到以下结论：

1. 处理简单问题时，值迭代和策略迭代的结果精度和速度基本相当。
2. 处理复杂问题时，策略迭代要比值迭代更快收敛，但可能得到局部最优解，而后者可以稳定获得全局最优解。一般认为值迭代更适合复杂问题。
3. 代码实现上，值迭代只需要维护值函数，策略迭代则要同时维护值函数和策略，后者复杂度高于前者。

6 实验总结

本次实验聚焦值迭代和策略迭代两种方法，探索了解决强化学习问题的基本思路。对于简单问题，两种办法殊途同归，但在面对复杂问题时，可能会显示出明显的差距。