蒙特卡洛算法
以GYMNASIUM冰湖环境为例

强化学习实验报告

张恒硕

2025 年 3 月 23 日

南開大學
Nankai University

目 录

图 片

表 格

1 实验目的

1. 了解on-policy和off-policy两种蒙特卡洛算法的原理。
2. 通过对冰湖的离散状态环境进行蒙特卡洛算法编写，加强对其的理解。
3. 比较分析两种蒙特卡洛算法的区别和特点，并联系迭代算法进行分析。

2 实验原理

2.1 蒙特卡洛（Monte Carlo, MC）方法

通过随机采样和经验平均来估计值函数，无需环境模型（即状态转移率），适用于无模型场景。其通过多次对完整轨迹的采样，计算累积奖励的期望值来逼近真实值函数。

2.2 on-policy

以策略为中心，评估和改进同一策略，依赖策略自身随机性探索并优化。
算法步骤：

1. 策略生成：使用当前策略（如$\varepsilon - greedy$）与环境交互，生成完整轨迹。
2. 值函数估计：对每个状态-动作对记录首次访问的累积回报，计算平均值作为Q值估计。
   更新公式为：
$$Q(s,a) = Q(s,a) + \alpha[G_t - Q(s,a)]$$
3. 策略改进：根据当前Q值，贪婪地更新策略，逐步减少ε以减少探索性。

on-policy实现简单，收敛速度较快，但可能陷入局部最优，适用于简单任务或需要快速收敛的场景，典型算法有SARSA。

2.3 off-policy

以数据为中心，使用行为策略（Behavior Policy）生成数据进行探索，目标策略（Target Policy）进行优化，借他山之石进行优化。
算法步骤：

1. 数据生成：使用行为策略（如随机策略）生成轨迹数据。

2. 重要性采样：通过重要性采样率（Importance Sampling Ratio）$W_t = \prod_{k=0}^{t-1} \frac{\pi(a_k|s_k)}{b(a_k|s_k)}$调整回报，将行为策略数据转换为对目标策略（如贪婪策略）的估计。更新公式为：

$$Q(s,a) = Q(s,a) + \alpha W_t[G_t - Q(s,a)]$$

3. 值函数更新：基于调整后的回报计算目标策略的Q值，逐步优化。

off-policy因重要性采样可能引入高方差，需结合截断或加权方法。其收敛速度较慢，但更通用，适用于复杂任务、需利用外部数据的任务，典型算法有Q-Learning。

3 关键代码解析

以下仅就蒙特卡洛采样的主体部分进行分析，其它的参数设定和初始化、函数不作展示，请见附件源码。

3.1 采样主体

Listing 1: 采样主体

```
# 主体算法
def policy(self, num_episodes):
    start_time = time.time()
    for episode in range(num_episodes):
        flag = False # 终止或截断标签
        # 采样一条轨迹
        state_traj = []
        action_traj = []
        reward_traj = []
        g = 0
        W = 1 # 重要性采样权重
        self.env.reset()
        cur_state = 0
        while not flag:
            # 选择下一动作
```

```
            cur_action = self.sample_action(cur_state)
            state_traj.append(cur_state)
            action_traj.append(cur_action)
            # 更新状态
            next_state, reward, flag, _, _ = self.env.step(cur_action)
            cur_state = next_state
            reward_traj.append(reward)
        # 利用轨迹更新行为值函数
        for i in reversed(range(len(state_traj))):
            # 计算状态-动作对(s,a)的访问频次
            self.n[state_traj[i], action_traj[i]] += W
            # 更新状态动作值
            ......
        # 更新策略
        if episode % 100 == 0:
            ......
            self.epsilon = max(self.epsilon * self.epsilon_decay, self.epsilon_min)
    end_time = time.time()
    print(f"总运行时间:{end_time - start_time:.2f}秒")
```

本部分定义了蒙特卡洛算法中采样一个轨迹的循环部分，适用于两种算法。在一次轨迹探索过程中，不断地在未结束状态下选取动作，更新状态，利用生成轨迹的结果反馈到价值矩阵和策略上，最终使采样对决策产生积极影响。

值得注意的点：

- 默认构建的冰湖环境是有滑动的。
- 每次采样前reset部分变量，其会在采样过程中变化。
- 过程中ε进行退火，但在达到最小值后不再变化。
- 最后利用值函数贪婪得到最优策略。

3.2 on-policy

Listing 2: on-policy

```
while not flag:
```

```
    # 根据策略pi采样一个动作
    cur_action = np.random.choice(self.act_n, p=self.Pi[cur_state, :])
    ……
for i in reversed(range(len(state_traj))):
    ……
    # 增量更新当前状态动作值
    g = reward_traj[i] + self.gamma * g
    # 新Q值=[当前Q值*(访问次数-1)（代表累积经验）+新折扣累积奖励g]/访问次数
    self.qvalue[state_traj[i], action_traj[i]] = (
        self.qvalue[state_traj[i], action_traj[i]] * (self.n[state_traj[i],
            action_traj[i]] - 1) + g
    ) / self.n[state_traj[i], action_traj[i]]
# 更新策略
if episode % 100 == 0:
    self.update_epsilon_greedy()
    self.epsilon = max(self.epsilon * self.epsilon_decay, self.epsilon_min)
```

- 均匀初始化策略。
- 根据策略pi采样动作，按概率随机抽取。
- 增量更新值函数。
- $\varepsilon-greedy$更新策略。

3.3 off-policy

Listing 3: off-policy

```
while not flag:
# 选择下一动作
    cur_action = self.sample_action(cur_state)
    ……
for i in reversed(range(len(state_traj))):
    ……
    # 增量更新当前状态动作值
    g = reward_traj[i] + self.gamma * g
```

```
    # 增量=重要性采样权重/访问频次
    delta = (W / (self.n[state_traj[i], action_traj[i]] + 1e-6)) * (g - self.qvalue[
        state_traj[i], action_traj[i]])
    self.qvalue[state_traj[i], action_traj[i]] += delta
    # 重要性采样权重=目标策略在当前状态采取当前动作的概率/行为策略在当前状态采取当前动作的概率
    W *= self.target_Pi[state_traj[i], action_traj[i]] / self.behaviour_Pi[state_traj[
        i], action_traj[i]]
# 更新策略
if episode % 100 == 0:
    self.behaviour_Pi = self.update_policy(self.epsilon)
    self.target_Pi = self.update_policy(self.epsilon / 10) # 低探索性
    self.epsilon = max(self.epsilon * self.epsilon_decay, self.epsilon_min)
```

- 均匀初始化行为策略，之后使用$\varepsilon-greedy$更新策略。
- 贪婪初始化目标策略。
- 使用$\varepsilon-greedy$选取动作。
- 利用重要性采样权重增量更新值函数。
- $\varepsilon-greedy$更新两个策略，更新目标策略时使用小ε以减小探索性。

3.4 可视化

Listing 4: 可视化

```
# 价值函数热力图
def visualize_q(self):
    plt.figure(figsize=(10, 6))
    plt.title("Final Q-values")
    plt.xlabel("Actions")
    plt.ylabel("States")
    plt.imshow(self.qvalue, cmap='viridis', aspect='auto')
    plt.colorbar(label='Q-value')
    plt.show()

```

```
# 策略可视化
def visualize_policy(self, policy):
    # 环境信息
    desc = env.unwrapped.desc.astype(str)
    n_rows = desc.shape[0]
    n_cols = desc.shape[1]
    # 绘图
    fig, ax = plt.subplots(figsize=(n_cols, n_rows))
    ax.set_xlim(0, n_cols)
    ax.set_ylim(n_rows, 0)
    ax.set_xticks(range(n_cols + 1))
    ax.set_yticks(range(n_rows + 1))
    ax.grid(which='major', axis='both', linestyle='-', color='black', linewidth=1)
    # 绘制策略箭头
    for row in range(n_rows):
        for col in range(n_cols):
            state = row * n_cols + col
            policy_action = policy[state]
            x_center = col + 0.5
            y_center = row + 0.5
            if policy_action == 0:  # left
                ax.arrow(x_center + 0.25, y_center, -0.3, 0, head_width=0.1,
                    head_length=0.1, fc='blue', ec='blue')
            elif policy_action == 1:  # down
                ax.arrow(x_center, y_center - 0.25, 0, 0.3, head_width=0.1,
                    head_length=0.1, fc='green', ec='green')
            elif policy_action == 2:  # right
                ax.arrow(x_center - 0.25, y_center, 0.3, 0, head_width=0.1,
                    head_length=0.1, fc='red', ec='red')
            elif policy_action == 3:  # up
                ax.arrow(x_center, y_center + 0.25, 0, -0.3, head_width=0.1,
                    head_length=0.1, fc='yellow', ec='yellow')
    # 标注陷阱 (红色背景)、目标 (绿色背景)
    holes = [(r, c) for r in range(desc.shape[0]) for c in range(desc.shape[1]) if
        desc[r, c] == 'H']
```

```
    goal = [(r, c) for r in range(desc.shape[0]) for c in range(desc.shape[1]) if desc
        [r, c] == 'G']
    for hole in holes:
        row, col = hole
        rect = Rectangle((col, row), 1, 1, facecolor="red", alpha=0.5, edgecolor="
            black")
        ax.add_patch(rect)
    for g in goal:
        row, col = g
        rect = Rectangle((col, row), 1, 1, facecolor="green", alpha=0.5, edgecolor="
            black")
        ax.add_patch(rect)
    plt.title("Policy Visualization")
    plt.show()
```

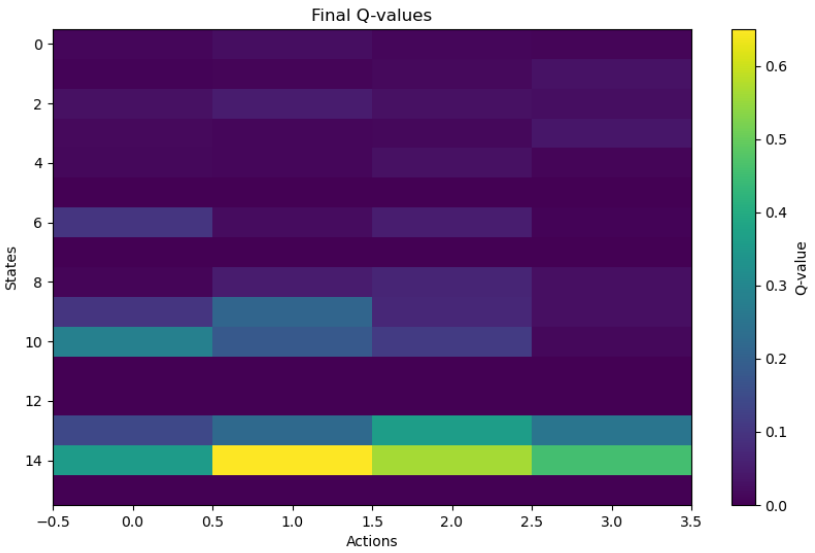
4 实验结果与分析

4.1 实验结果

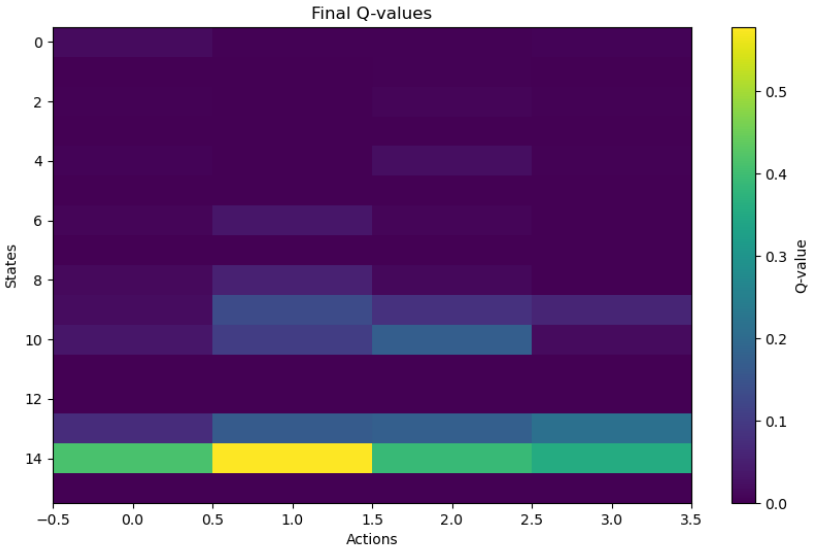
由于蒙特卡洛的采样具有随机性，每次运行的结果不同，以下分条目进行展示。

价值函数

在4*4地图上，分别选取一次运行的价值函数结果绘制热力图作展示，如下图组引用图1 on the following page。



(a) on-policy



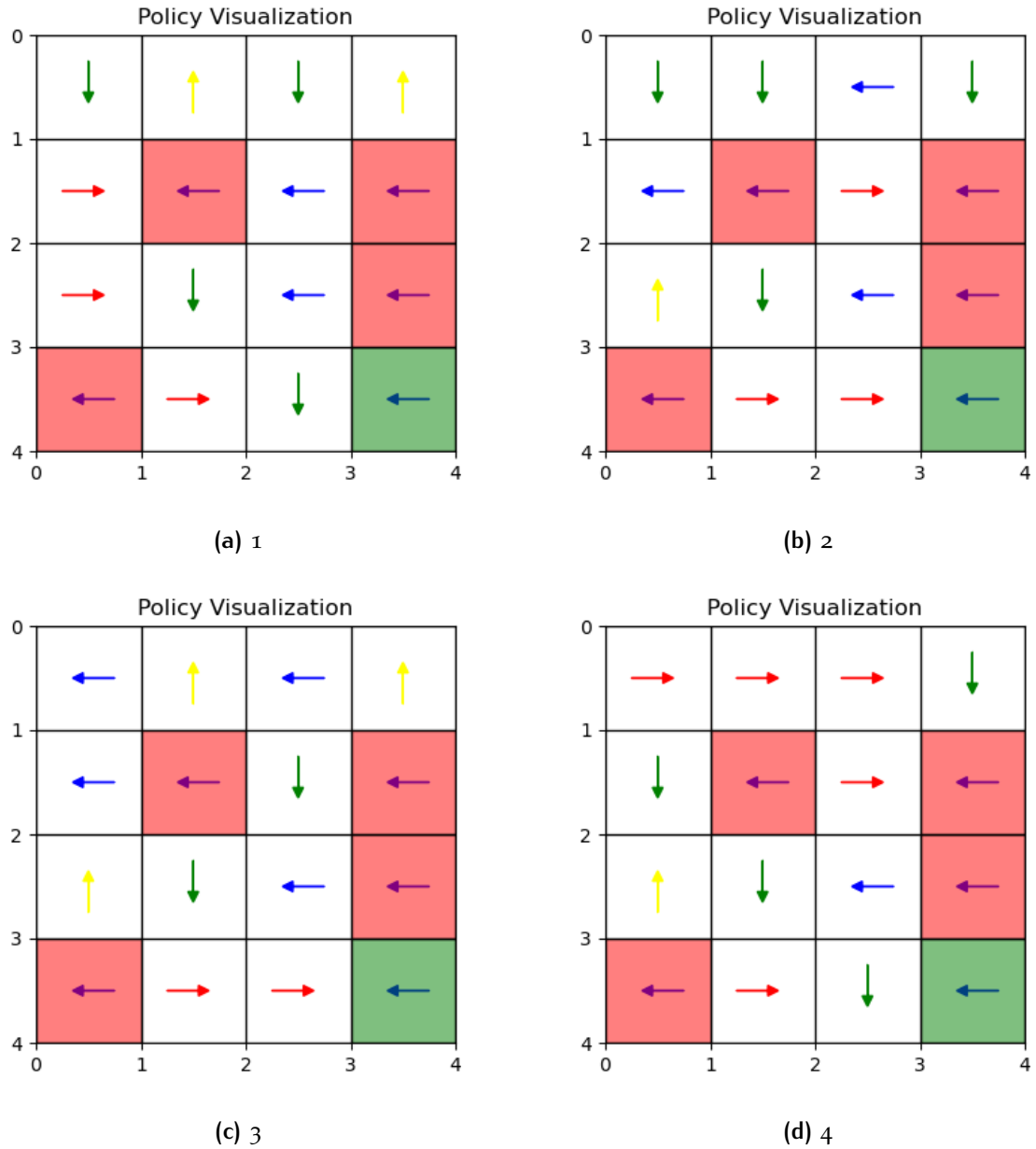
(b) off-policy

图 1: 价值函数

两个值函数的分布大同小异，大部分动作的价值较低，最鲜明的是在目标左侧点（序号14）处采取向下动作。由于滑动的存在，向下的动作可能会带来向左、右移动的结果，前者无伤大雅，只是回到了前置步骤，后者将抵达终点。相比之下，如果采取直接的向右移动策略，可能会向上移动（到序号10处），在这个位置采取动作可能会陷入陷阱。

最优策略

在4*4地图上，当前运行轮数的结果不会收敛，最终通过贪婪选取得到的策略并不固定，但是可以发现一些端倪。以下就两种算法分别给出几个最优策略结果，如图组2图组3 on the following page。



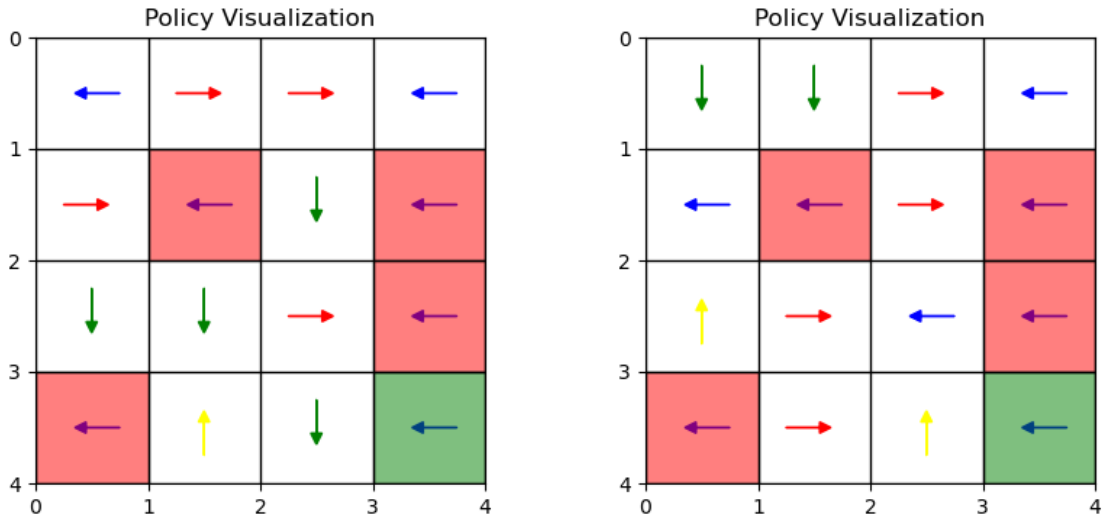
(a) 1

(b) 2

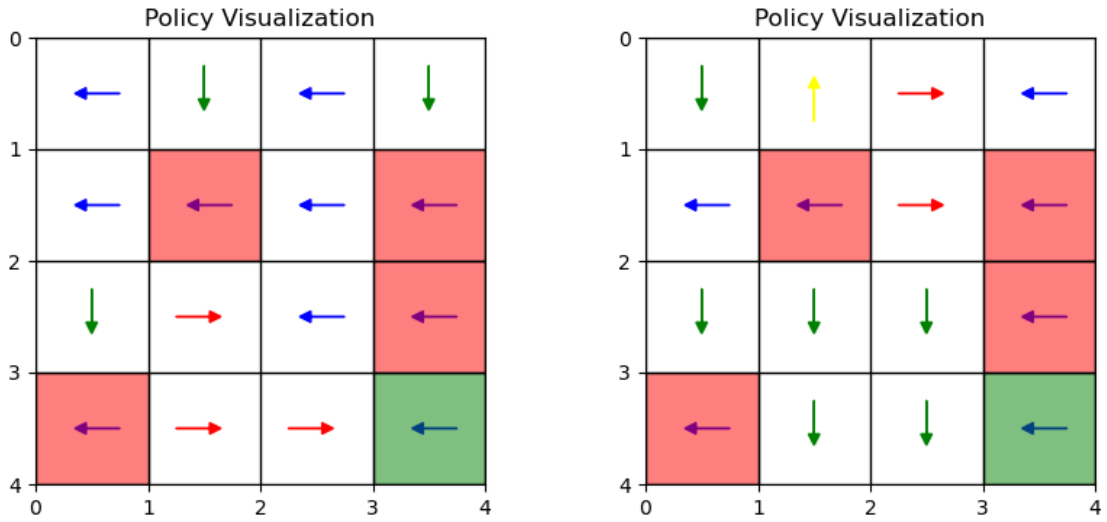
(c) 3

(d) 4

图 2: on-policy最优策略



(a) 1 (b) 2



(c) 3 (d) 4

图 3: off-policy最优策略

两种方法得到的策略都具有可执行性，并且充分考虑了滑动带来的干扰。

运行时间

以下表1 on the next page给出了两种算法在不同尺寸地图上的运行时间比较，选取的时间值不是精确值，而是多次运行的中位数。

表 1: 运行时间(s)

地图尺寸	4*4	8*8
on-policy	1.7	100
off-policy	1	60

PS：针对8*8地图，采样10000条轨迹并不能获得合理的结果，这里仅比较二者运行效率。

- 相同采样次数下，由于轨迹变长，8*8地图的耗时是4*4的幂级数倍。
- on-policy通常具有更高的稳定性，但可能需要更多样本才能收敛，因为每次策略更新后都需要新的数据。
- off-policy虽然可能更快地找到好的解决方案，但由于使用了不同的行为策略，学习过程可能不太稳定。

4.2 与迭代方法对比

蒙特卡洛方法采样一个轨迹的过程与迭代方法一次策略评估过程是相似的，但是二者需要的次数相差很多。蒙特卡洛方法由于采样的随机性，即使大量采样也无法收敛，而迭代方法在极少次评估中便可以收敛。在使用的冰湖4*4例子中，前者是10000次，后者是200次左右。这种巨大的差距来源于二者的根本原理。蒙特卡洛方法是在解空间中采样，提高估计水平，来优化策略，但每次采样的作用不一定是积极的或有作用的。迭代方法注重值函数的收敛，除了策略迭代的初始策略具有随机性以外，一直向最优（可能是局部最优）收敛。在任务复杂化的过程中，蒙特卡洛方法的复杂度增长也要远快于迭代方法。然而，虽然蒙特卡洛方法的效率较差，但是它是一种不需要预知模型，即状态转移率的方法，其可用范围更加宽广。在迭代方法不适用的情况下，蒙特卡洛方法也具有着较高的可行性。

4.3 综合分析

蒙特卡洛方法的思想较为简单，不需要状态转移率，是针对无模型强化学习任务的优秀方法。on-Policy 方法自我优化，off-policy方法利用其它策略优化自己，二者都是有效的方法。在面对维数灾难时，蒙特卡洛方法可能需要极大的算力，因为采样复杂度和采样次数在同步增加，然而相较于线性规划和直接搜索，其明显是更快的。

5 实验总结

本次实验着重尝试了蒙特卡洛方法的on-policy和off-policy两种算法，在实现代码的过程中，加深了对它们的理解。在两种算法对比和与迭代方法对比的过程中，体会到了不同方法的优劣。