时序差分算法
以GYMNASIUM冰湖环境为例

强化学习实验报告

张恒硕

2025 年 4 月 7 日

南開大學
Nankai University

目 录

图 片

表 格

1 实验目的

1. 了解Sarsa和Q-learning两种时序差分学习算法的原理。
2. 编写冰湖离散状态环境时序差分学习算法代码，加强对其理解。
3. 比较分析两种算法的区别和特点，并联系先前蒙特卡洛算法进行分析。

2 实验原理

2.1 时序差分（Temporal Difference, TD）学习

时序差分学习是一种结合了蒙特卡罗方法（MC）和动态规划方法（DP）优点的强化学习方法。其可以直接从与环境的互动中获取信息，不需要构建环境动态特性模型；同时运用自举思想，可以基于已得到的其他状态的估计来更新当前状态的价值函数。

MC必须等到一个幕的结尾才能确定增量，更新的是G_t；而TD只需要等到下一时刻即可更新，更新的是$R_{t+1}+\gamma V(S_{t+1})$。

TD(0)（单步TD）的更新公式为：

$$V_{t+1}(S_t)=V_t(S_t)+\alpha_t(S_t)[R_{t+1}+\gamma V_t(S_{t+1})-V_t(S_t)]$$

其中$[R_{t+1}+\gamma V_t(S_{t+1})-V_t(S_t)]$称为TD误差$\delta_t$， $R_{t+1}+\gamma V_t(S_{t+1})$称为TD目标。

2.2 Sarsa

Sarsa（State-Action-Reward-State-Action）是on-policy-TD方法，其更新公式是：

$$Q_{t+1}(S_t,A_t)=Q_t(S_t,A_t)+\alpha_t[R_{t+1}+\gamma Q_t(S_{t+1},A_{t+1})-Q_{t+1}(S_t,A_t)]$$

其根据执行的策略直接从经历中学习，更新基于实际采取的动作序列。

2.3 Q-learning

Q-learning是off-policy-TD方法，其更新公式是：

$$Q_{t+1}(S_t,A_t)=Q_t(S_t,A_t)+\alpha_t[R_{t+1}+\gamma \max_a Q_t(S_{t+1},a)-Q_t(S_t,A_t)]$$

其能够使用行为策略生成的数据学习目标策略，学习过程不依赖实际采取的动作，总朝向可能的最大未来奖励更新动作价值估计。

3 关键代码解析

代码的主体部分和上次（以下称为源代码）一致，变化的主要是采样（迭代）的部分。

3.1 主体

Listing 1: 主体

```
class FrozenLake:
    def __init__(self, env):
        self.env = env
        self.obs_n = env.observation_space.n  # 状态空间大小
        self.act_n = env.action_space.n  # 动作空间大小
        self.qvalue = np.zeros((self.obs_n, self.act_n))  # 状态-动作价值函数
        self.gamma = 0.9  # 折扣率
        self.alpha = 0.1  # 学习率
        self.epsilon = 0.1  # 探索概率

    # epsilon-greedy（多最优随机选择）
    def sample_action(self, state):
        if np.random.rand() < self.epsilon:
            action = np.random.choice(self.act_n)
        else:
            max_actions = np.where(self.qvalue[state] == np.max(self.qvalue[state]))
                [0]
            action = np.random.choice(max_actions)
        return action

    # 贪婪决策（多最优随机选择）
    def get_greedy_policy(self):
        greedy_policy = np.zeros(self.obs_n)
        for i in range(self.obs_n):
```

```
            max_actions = np.where(self.qvalue[i] == np.max(self.qvalue[i]))[0]
            greedy_policy[i] = np.random.choice(max_actions)
        return greedy_policy

    # 算法
    def algorithm(self, episodes):
        start_time = time.time()
        ~
        end_time = time.time()
        print(f"总运行时间:{end_time - start_time:.2f}秒")
```

- 动作选取采用ϵ-greedy策略。
- 动作选取函数和最终决策函数都在源代码基础上添加了多最优随机选择。

3.2 Sarsa

Listing 2: Sarsa

```
    for episode in range(episodes):
    state = self.env.reset()[0]
    action = self.sample_action(state)
    flag = False  # 终止或截断标签
    while not flag:
        # 选取动作
        next_state, reward, flag, _, _ = self.env.step(action)
        next_action = self.sample_action(next_state)
        # q值更新
        td_target = reward + self.gamma * self.qvalue[next_state, next_action]
        td_error = td_target - self.qvalue[state, action]
        self.qvalue[state, action] += self.alpha * td_error
        state = next_state
        action = next_action
```

- 使用下一状态的状态-动作二元组更新值函数，并采用该二元组作为下一时刻的选择。

3.3 Q-learning

Listing 3: Q-learning

```
    for episode in range(episodes):
    state = self.env.reset()[0]
    flag = False  # 终止或截断标签
    while not flag:
        # 选取动作
        action = self.sample_action(state)
        next_state, reward, flag, _, _ = self.env.step(action)
        best_next_action = np.argmax(self.qvalue[next_state])
        # q值更新
        td_target = reward + self.gamma * self.qvalue[next_state, best_next_action]
        td_error = td_target - self.qvalue[state, action]
        self.qvalue[state, action] += self.alpha * td_error
        state = next_state
```

- 使用下一状态的最佳动作更新值函数，但在下一状态的实际决策时，使用ϵ-greedy策略选择动作。

3.4 可视化

与源代码一致，略。

3.5 评估策略

Listing 4: 评估策略

```
    # 评估策略
    def evaluate_policy(self, policy, num_episodes):
        success_count = 0
        for _ in range(num_episodes):
            state = self.env.reset()[0]
            flag = False
            while not flag:
```

```
                action = policy[state]
                next_state, reward, flag, _, _ = self.env.step(action)
                state = next_state
                if reward == 1:
                    success_count += 1
                    break
        success_rate = success_count / num_episodes
        print(f"策略成功率: {success_rate * 100:.2f}%")
        return success_rate
```

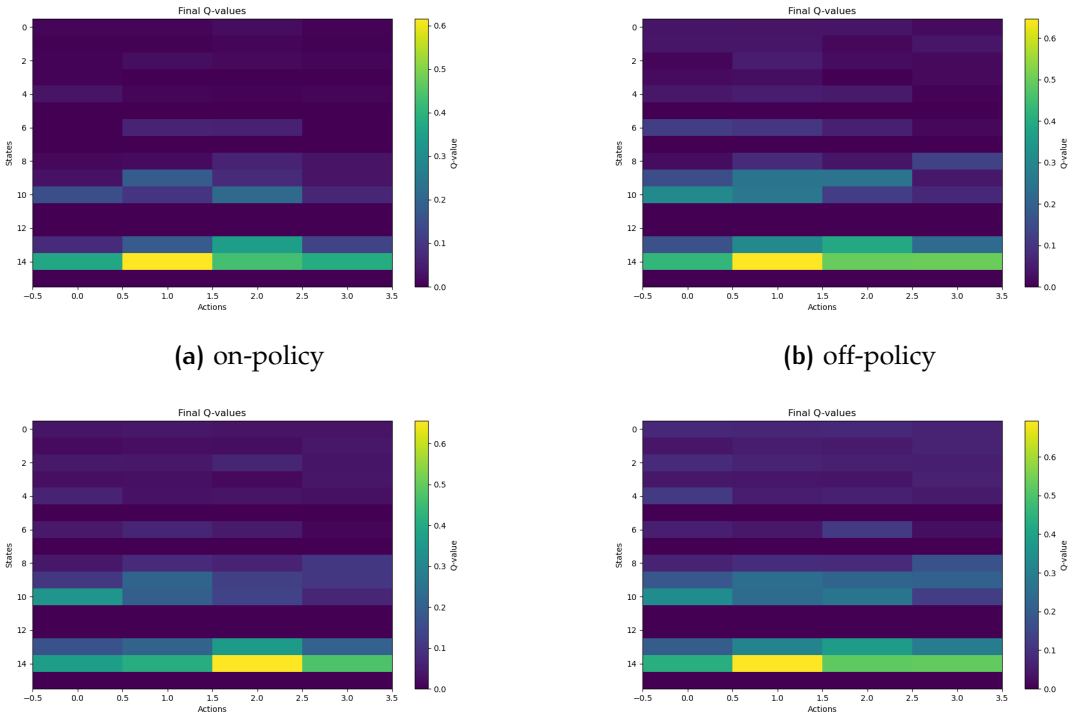
- 在源代码基础上添加了评估策略函数，其根据大量（默认1000次）实际实验反馈最终策略的性能，包括运行时间和实验成功率。
- 当前的10000次迭代并不能使结果完全收敛，比较时进行平均。

4 实验结果与分析

4.1 实验结果

在有滑动的4*4冰湖环境中，针对上次实现的MC算法和本次实现的TD算法进行对比，统一采样（迭代）10000次。

以下图组1 on the following page展示了最终的价值函数，可以发现，各种算法的价值函数有着相同的收敛趋势。



(a) on-policy

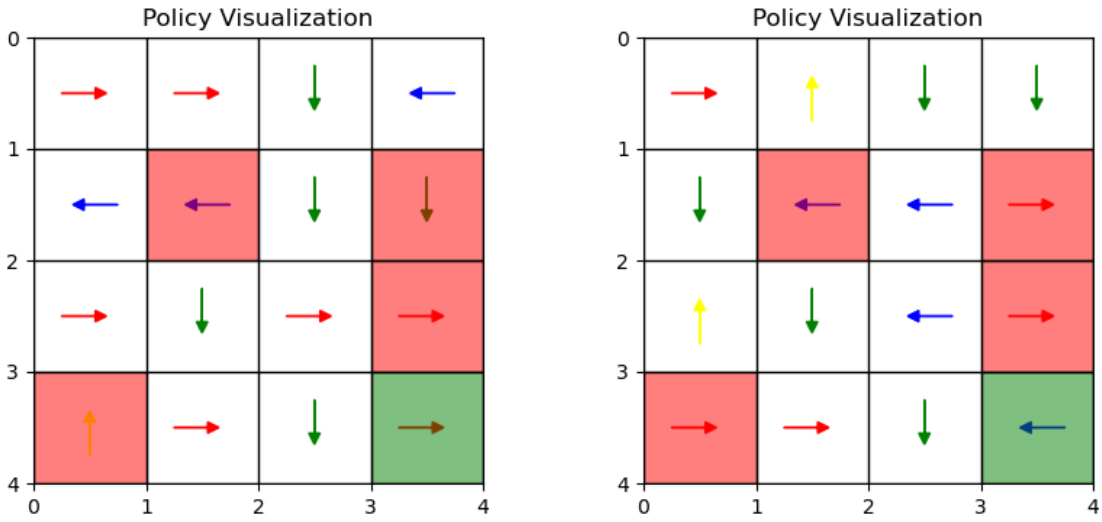
(b) off-policy

(c) Sarsa

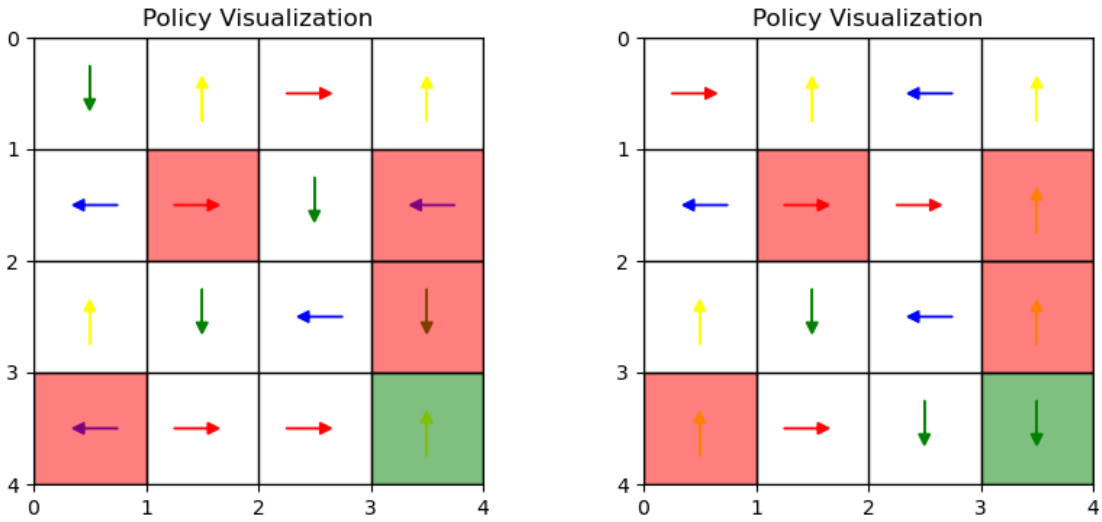
(d) Q-learning

图 1: 价值函数对比

以下图组2 on the next page展示了由价值函数贪婪决策的结果，两个TD算法的结果大致一样，在多次重复试验中也获得了大体一致的结果。而两个MC算法的结果虽然趋于前两者，但仍有很多不同之处，且在当前迭代次数下并不稳定。



(a) on-policy **(b)** off-policy



(c) Sarsa **(d)** Q-learning

图 2: 策略对比

下表1 on the following page则展示了各算法的性能对比，值得注意的是，为获得代表性的结果，对多次结果进行了平均（运行时间只包括获得最优策略的耗时，不包括评估）。

可以发现，TD算法的耗时都要高于MC算法，结果也要更优，这说明经过更多的计算，价值函数的收敛效果更好，由其产生的策略也更优。表中的数据是经过平均的，真实的策略成功率为当前数据上下浮动50%。

表 1: 各算法性能对比

算法	耗时（s）	策略成功率（%）
MC(on-policy)	1.80	17.19
MC(off-policy)	2.09	18.30
TD(Sarsa)	3.18	49.37
TD(Q-learning)	3.45	42.88

4.2 综合分析

- MC(on-policy)：专注于当前策略，使用自己生成的数据来改进自己，使策略和价值函数紧密相连。
- MC(off-policy)：更灵活，可以学习其他策略，但是需要更多样本。
- TD(Sarsa)：关注策略的实际表现而非理论上的最优表现。
- TD(Q-learning)：旨在找到最佳可能的动作价值函数，无论其是否由当前策略选择，可以在不改变现有策略的情况下评估各种行动的效果。

结合上述结果数据和以上罗列的算法特征，产生以下结论：

与整幕结束后更新的MC算法相比，即时更新的TD算法虽然计算量高，但是方差更小，可以在更少的次数下得到准确的价值估计。

而两个TD算法中，Sarsa较为保守，在存在风险的任务中，会避开低回报的动作； Q-learning较为乐观，更倾向于探索并找到最优解。因此，在冰湖这个存在陷阱的任务中，Sarsa会比Q-learning取得更好的结果。

这一点可以参见sutton的教材，在131页（汉译版）悬崖行走的例子中， Sarsa会走远离悬崖的较长安全路径，而选择悬崖边较短路径的Q-learning则有更大风险，如下图3 on the next page。

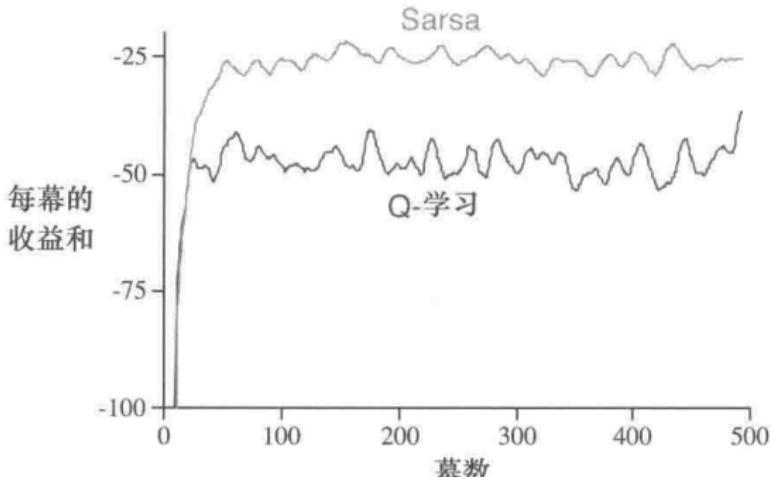


图 3: 悬崖行走例子

5 实验总结

本次实验着重尝试了时序差分学习的Sarsa和Q-learning算法，其也分别对应 on-policy和off-policy两种策略。在代码的实现过程中，加深了对它们的理解，并通过比较了解了MC、TD各算法的特点和侧重，验证了书本和课堂的内容。