深度强化学习方法第三次作业说明文件：**TD(0), Sarsa, Q-learning**

一、问题重述

基于强化学习的课程背景，我们学习了动态规划、蒙特卡洛方法，并基于此探讨了时序差分问题，在时序差分方法中，通过综合前两种方法的自举特性和采样特性，组成了新的一种强化学习方法，并且通过同轨策略和离轨策略划分为 Sarsa 方法和 Q-learning 方法。

TD(0) 是 Monte Carlo 的改进，无物理模型的采样，无需等待路径终点，根据已学习的下一状态值，预测当前状态。其迭代更新公式及算法如下所示：

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

表格型 TD(0) 算法，用于估计 v_π

输入：待评估的策略 π
算法参数：步长 $\alpha \in (0,1]$
对于所有 $s \in \mathcal{S}^+$，任意初始化 $V(s)$，其中 $V(\text{终止状态}) = 0$
对每幕循环：
  初始化 S
  对幕中的每一步循环：
    $A \leftarrow$ 策略 π 在状态 S 下做出的决策动作
    执行动作 A，观察到 R，S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  直到 S 为终止状态

图 1：TD(0)算法

Sarsa 方法是一类同轨策略的时序差分方法，是指采样策略与目标策略相同。此处是学习动作值，不是状态值。按照动作策略π，评估所有动作价值，并且根据q值，通过ε贪婪法生成新的策略。其迭代更新公式及算法如下所示：

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

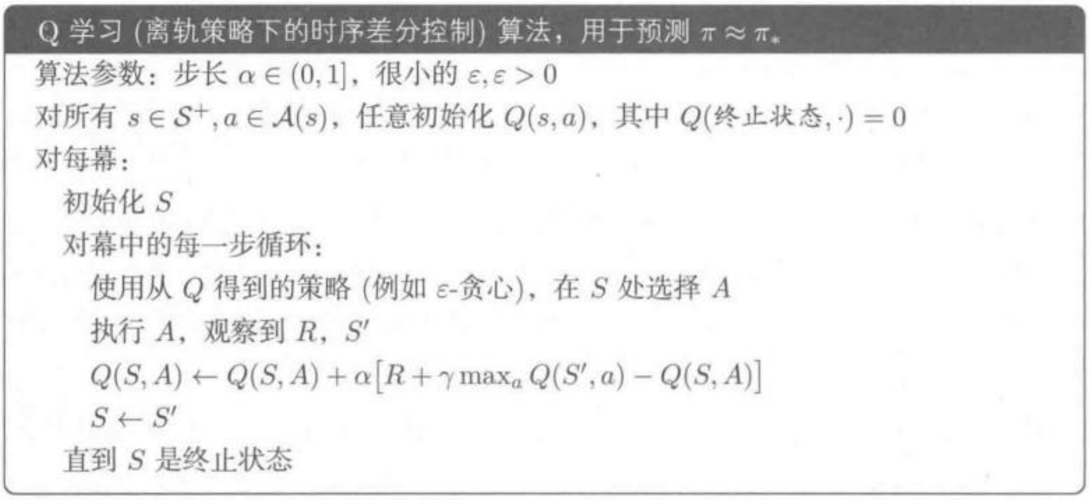
Sarsa (同轨策略下的 TD 控制) 算法，用于估计 $Q \approx q_*$

算法参数：步长 $\alpha \in (0,1]$，很小的 $\varepsilon, \varepsilon > 0$
对所有 $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$，任意初始化 $Q(s,a)$，其中 $Q(\text{终止状态}, \cdot) = 0$
对每幕循环：
  初始化 S
  使用从 Q 得到的策略 (例如 ε-贪心)，在 S 处选择 A
    对幕中的每一步循环：
    执行动作 A，观察到 R，S'
    使用从 Q 得到的策略 (例如 ε-贪心)，在 S' 处选择 A'
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
    $S \leftarrow S'$；$A \leftarrow A'$；
  直到 S 是终止状态

图 2：Sarsa 算法

Q-learning 方法是一类离轨策略的时序差分方法，是指采样策略与目标策略不同，采样策略使用ε贪婪法，目标策略通过采用贪婪法，计算下一个状态最优值V^*，其迭代更新公式及算法如下所示：

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



Q 学习（离轨策略下的时序差分控制）算法，用于预测 $\pi \approx \pi_*$

算法参数：步长 $\alpha \in (0,1]$，很小的 $\varepsilon, \varepsilon > 0$

对所有 $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$，任意初始化 $Q(s,a)$，其中 $Q(终止状态, \cdot) = 0$

对每幕：

  初始化 S

  对幕中的每一步循环：

    使用从 Q 得到的策略（例如 ε-贪心），在 S 处选择 A

    执行 A，观察到 R，S'

    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$

    $S \leftarrow S'$

  直到 S 是终止状态

图 3：Q-learning 算法

二、实验环境

硬件环境 PC 机,CPU Intel(R)Core (TM) I7-9750H@2.60GHz 内存 16GB

软件环境 Visual Studio Code，python 版本 3.7.6

三、实验方法

1、 Cliffwalk 悬崖行走问题建模：

在本次实验中，我们将悬崖地图看作一个有（4 rows×12 columns）48 个状态，4 个动作的有限状态马尔可夫过程，其中地图如下图所示，模拟在悬崖边散步，agent 目标是从起点 S 到终点 G，agent 的动作有上下左右四个，agent 每一步的动作奖励为-1，掉下悬崖是-100，走到终点是 0。采样策略都是ε贪婪法。

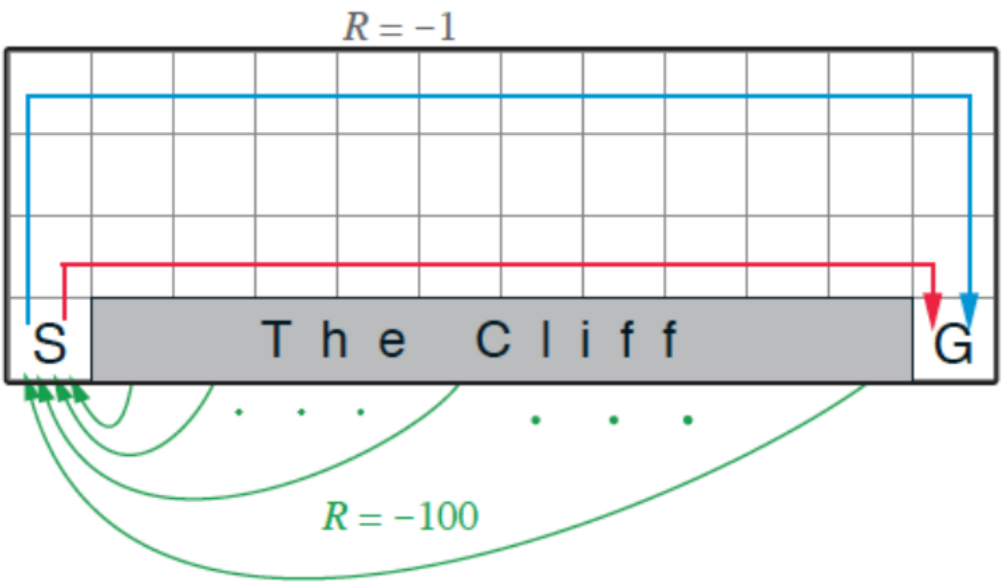


图 4：悬崖行走地图

因此在实验编程中会初始化一个有着（4, rows*columns）个状态-动作对的动作价值函数/（rows*columns）个状态的状态价值函数，并且对此价值 Q 函数的 list 构建相关的函数（如ε贪婪策略函数、策略更新函数等等）。函数的参数及功能说明在代码中也有呈现。然后分别通过 TD(0), Sarsa, Q-learning 的迭代方程分别实现迭代更新。

2、 问题解决与结果展示：

1). TD(0)算法

相对于 Sarsa 和 Q-learning 初始化动作价值函数而言，TD(0)初始化状态价值函数，因此其代码单独列于 cliff_walk_td.py 中，其中关键的 TD(0)迭代函数如下所示

```
def td_0(num_episodes = 500, gamma_discount = 0.9, alpha = 0.5, epsilon = 0.1):
"""
        Implementation of td(0) algorithm. (Sutton's book), adapted from qlearning

        Args:
            num_episodes -- type(int), 500 acts' number of games to train agent
            gamma_discount -- type(float) discount factor determines importance of
future rewards
            alpha -- type(float) determines convergence rate of the algorithm (can think
as updating states fast or slow)
            epsilon -- type(float) explore/ exploit ratio (exe: default value 0.1
indicates %10 exploration)

        Returns:
            v_table -- type(np.array) Determines state value
            reward_cache -- type(list) contains cumulative_reward, which is used to plot
a draw
"""
        # initialize all states to 0
        # Terminal state cliff_walking ends
        reward_cache = list()
        step_cache = list()
        v_table = createV_table()
        agent = (3, 0) # 1. starting from left down corner
        # start iterating through the episodes
        for episode in range(0, num_episodes):
            env = np.zeros((4, 12))
            env = visited_env(agent, env)
            agent = (3, 0) # starting from left down corner
            game_end = False
            reward_cum = 0 # cumulative reward of the episode
            step_cum = 0 # keeps number of iterations untill the end of the game
            while(game_end == False):
                # get the state from agent's position
                state = get_state(agent)
                state_value=v_table[state]
                # choose action using epsilon-greedy policy
                action = epsilon_greedy_policy(agent,v_table)
                # move agent to the next state
                agent = move_agent(agent, action)
```

```
                env = visited_env(agent, env) # mark the visited path
                step_cum += 1
                # observe next state value
                next_state = get_state(agent)
                max_next_state_value = v_table[state]
                # observe reward and determine whether game ends
                reward, game_end = get_reward(next_state)
                reward_cum += reward
                # update q_table
                v_table = update_vTable(v_table, state, reward, max_next_state_value,
gamma_discount, alpha)
                # update the state
                state = next_state
            reward_cache.append(reward_cum)
            if(episode > 498):
                print("Agent trained with td(0) after 500 iterations")
                print(env) # display the last 2 path agent takes
            step_cache.append(step_cum)
    return v_table, reward_cache, step_cache
```

除了更新函数外，代码中还包括了初始化算法**`createV_table`**创建了一个 rows×columns 的状态价值函数表，**`epsilon_greedy_policy`**创建了ε贪婪算法的函数，**`vis_env`**提供了路径可视化的方法，**`get_state`,`move_agent`,`get_reward`**提供了状态转移中的必要函数，而**`update_vTable`**给出了更新状态函数的方法，如下所示：

```
def update_vTable(v_table, state,   reward, next_state_value, gamma_discount = 0.9, alpha
= 0.5):"""
        Update the q_table based on observed rewards and maximum next state value
        Sutton's Book pseudocode:   V(S) <- V(S) + [alpha * (reward + (gamma * V(S')) -
V(S) ]
        Args:
            v_table -- type(np.array) Determines state value
            state -- type(int) state value between [0,47]
            reward -- type(int) reward in the corresponding state
            next_state_value -- type(float) maximum state value at next state
            gamma_discount -- type(float) discount factor determines importance of
future rewards
            alpha -- type(float) controls learning convergence

        Returns:
            v_table -- type(np.array) Determines state value
        """
        update_v_value = v_table[state] + alpha * (reward + (gamma_discount *
next_state_value) - v_table[state])
        v_table[state] = update_v_value
    return v_table
```

相对于 Sarsa 和 Q-learning 算法,TD(0)的收敛性相对较差，最后一个 epoch（500）迭代中的路线图在随机性的引导下容易得到不同的路线，得到的路线和每个状态的价值函数热力图如下所示，可以看见在三次独立的迭代中，路线产生了差异：

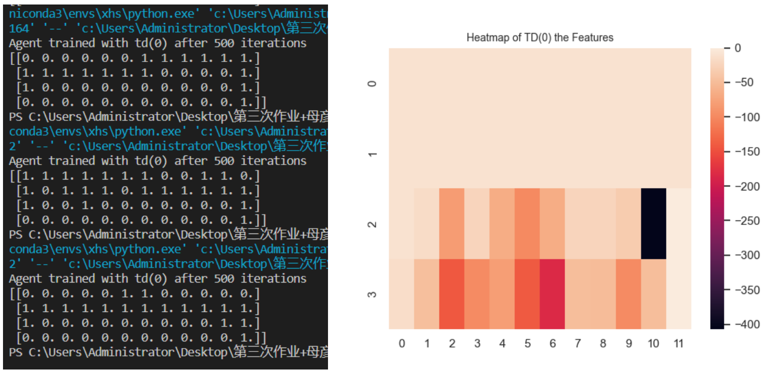


图 5：TD(0)方法下路径选择与状态价值函数热力图

2). Sarsa 算法

Sarsa 算法是一类同轨策略的时间差分方法，即是目标策略和采样策略一致，其中关键的 Sarsa 迭代函数如下所示：

```
def sarsa(num_episodes = 500, gamma_discount = 0.9, alpha = 0.5, epsilon = 0.1):
    reward_cache = list()
    step_cache = list()
    q_table = createQ_table()
    # start iterating through the episodes
    for episode in range(0, num_episodes):
        agent = (3, 0) # starting from left down corner
        game_end = False
        env = np.zeros((4, 12))
        env = visited_env(agent, env)
        reward_cum = 0 # cumulative reward of the episode
        step_cum = 0 # keeps number of iterations untill the end of the game
        # choose action using policy
        state, _ = get_state(agent, q_table)
        action = epsilon_greedy_policy(state, q_table)
        while(game_end == False):
            # move agent to the next state
            agent = move_agent(agent, action)
            env = visited_env(agent, env)
            step_cum += 1
            # observe next state value
            next_state, _ = get_state(agent, q_table)
            # observe reward and determine whether game ends
            reward, game_end = get_reward(next_state)
            reward_cum += reward
            # choose next_action using policy and next state, which is the need of sarsa
            next_action = epsilon_greedy_policy(next_state, q_table)
            # update q_table
```

```
            next_state_value = q_table[next_action][next_state] # differs from q-learning uses the next action determined by policy
            q_table = update_qTable(q_table, state, action, reward, next_state_value, gamma_discount, alpha)
            # update the state and action
            state = next_state
            action = next_action # differs q_learning both state and action must updated
        reward_cache.append(reward_cum) # everyepoch's reward record
        step_cache.append(step_cum)
        if(episode > 498):
            print("Agent trained with SARSA after 500 iterations")
            print(env) # display the last 2 path agent takes
    return q_table, reward_cache, step_cache
```

Sarsa 算法的最后一个 epoch 迭代得到的路线和每个状态的价值函数热力图如下所示：

```
Agent trained with SARSA after 500 iterations
[[0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

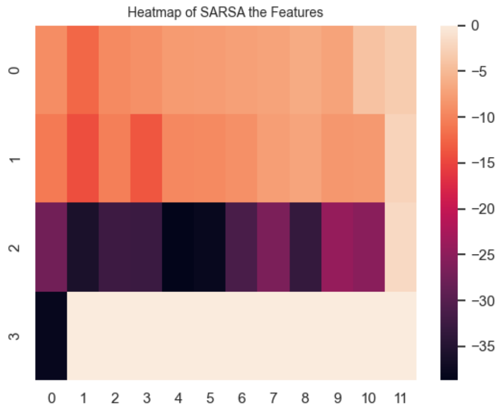


图 6：Sarsa 方法下路径选择与动作价值函数热力图

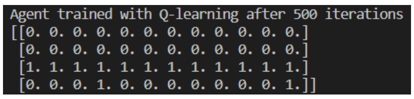
3). Q-learning 算法

Q-learning 算法是一类离轨策略的时间差分方法，即是目标策略和采样策略不一致，其中关键的 Q-learning 迭代函数如下所示：

```
def qlearning(num_episodes = 500, gamma_discount = 0.9, alpha = 0.5, epsilon = 0.1):
    # initialize all states to 0
    # Terminal state cliff_walking ends
    reward_cache = list()
    step_cache = list()
    q_table = createQ_table()
    agent = (3, 0) # 1. starting from left down corner
    # start iterating through the episodes
    for episode in range(0, num_episodes):
        env = np.zeros((4, 12))
        env = visited_env(agent, env)
```

```
        agent = (3, 0) # starting from left down corner
        game_end = False
        reward_cum = 0 # cumulative reward of the episode
        step_cum = 0 # keeps number of iterations untill the end of the game
        while(game_end == False):
            # get the state from agent's position
            state, _ = get_state(agent, q_table)
            # choose action using epsilon-greedy policy
            action = epsilon_greedy_policy(state, q_table)
            # move agent to the next state
            agent = move_agent(agent, action)
            env = visited_env(agent, env) # mark the visited path
            step_cum += 1
            # observe next state value
            next_state, max_next_state_value = get_state(agent, q_table)
            # observe reward and determine whether game ends
            reward, game_end = get_reward(next_state)
            reward_cum += reward
            # update q_table
            q_table = update_qTable(q_table, state, action, reward, max_next_state_value, gamma_discount, alpha)
            # update the state
            state = next_state
        reward_cache.append(reward_cum)
        if(episode > 498):
            print("Agent trained with Q-learning after 500 iterations")
            print(env) # display the last 2 path agent takes
        step_cache.append(step_cum)
    return q_table, reward_cache, step_cache
```

Q-learning 算法的最后一个 epoch 迭代得到的路线和每个状态的价值函数热力图如下所示：



```
Agent trained with Q-learning after 500 iterations
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

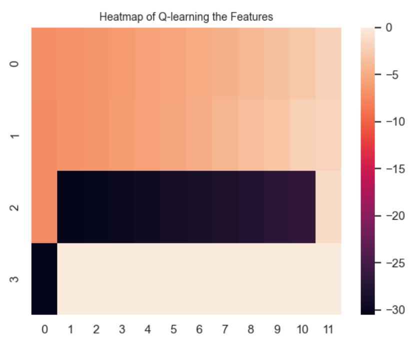


图 7：Q-learning 方法下路径选择与动作价值函数热力图

3、 结果比较

对比 TD（0）与 Sarsa，Q-learning 的步数以及到终点时的收益积累，可以发现 TD（0）的效果相对较差，并且经历的 epoch 更多，而 Q-learning 相对于 Sarsa 性能更好，在下两图中可以看见 Q-learning 的收敛相对更稳定，可能与其使用了离轨策略有关，Q-learning 和 Sarsa 的性能比较也是强化学习中一直被关注的话题，值得进一步研究

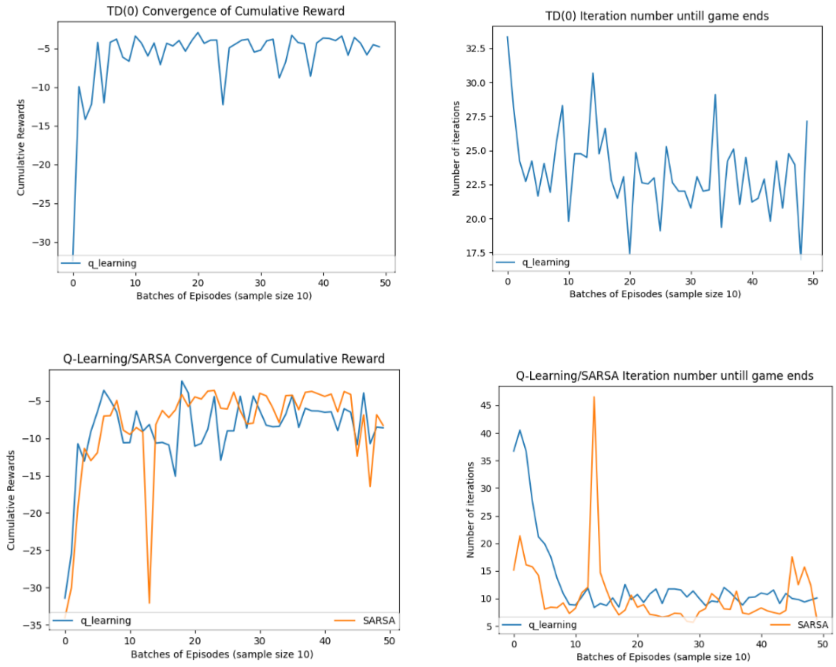


图 8：三类方法每个 epoch 下到达终点时累计收益及步数可视化

4、 额外说明

本次的代码由于 TD(0)运用状态价值函数，因此单独写在 cliff_walk_td.py 文件中，Sarsa 和 Q-learning 用动作价值函数，卸载 cliff_walk.py 中，代码仓库亦可见 https://github.com/Mu-Yanchen/rl_hw，代码参考 https://github.com/zeynepCankara/Cliff-Walking-Solution