实验一

姓 名: 任 永 文
学 号: SA23011253

强化学习
(秋季, 2023)

中国科学技术大学
计算机科学与技术学院

2023 年 11 月 12 日

1 实验目的

- 理解、学习蒙特卡罗算法原理，并编码实现 first-visit 版本
- 理解、学习 TD 算法原理，并编码实现 SARSA、Q-learning

2 实验原理

MC 原理

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$
Initialize:
  $\pi \leftarrow$ an arbitrary ε-soft policy
  $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
  $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
      $A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)
      For all $a \in \mathcal{A}(S_t)$:
$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Sarsa 原理

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S', A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Q-learning 原理

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

3 实验内容

MC 方法实现

```
# 步骤 1: 生成一个回合。
    # 一个回合是由（状态，动作，奖励）元组组成的数组
episode = []
state = env.reset()
while True:
    # 根据状态选择动作的概率分布
    probs = [0.8, 0.2] if state[0] > 18 else [0.2, 0.8]  # 概率分布
    action = np.random.choice(np.arange(2), p=probs)  # 根据概率选择动作
    next_state, reward, done, info = env.step(action)  # 执行动作，获取下一个
    ↪  状态、奖励等信息
    episode.append((state, action, reward))  # 记录状态、动作和奖励
    state = next_state
    if done:
        break

# 步骤 2: 找出我们在这个回合中访问过的所有（状态，动作）对
states, actions, rewards = zip(*episode)  # 解压回合元组，获取状态、动作和奖励
↪  信息
```

```
# 步骤 3: 计算所有采样回合中该状态的平均回报
discounts = np.array([discount_factor**i for i in range(len(rewards) + 1)])
↪ # 计算折扣因子
for i, state in enumerate(states):
    # 更新动作值函数
    returns_sum[state][actions[i]] += sum(rewards[i:] * discounts[:-(i + 1)])
    ↪ # 计算回报总和
    returns_count[state][actions[i]] += 1  # 记录访问次数
    Q[state][actions[i]] = returns_sum[state][actions[i]] /
    ↪ returns_count[state][actions[i]]  # 计算平均值
```

Sarsa 方法实现

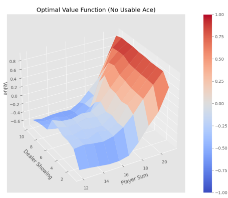
```
# 步骤 1: 执行一步
next_state, reward, done, _ = env.step(action)  # 执行选定的动作，获取下一个状
↪ 态、奖励等信息
# 步骤 2: 选择下一个动作
next_action_probs = policy(next_state)  # 获取下一个状态下的动作概率分布
next_action = np.random.choice(np.arange(len(next_action_probs)),
↪ p=next_action_probs)  # 根据概率选择下一个动作
# 更新统计信息
stats.episode_rewards[i_episode] += reward  # 更新回合奖励
stats.episode_lengths[i_episode] = t  # 更新回合长度

# 步骤 3: 时序差分更新
td_target = reward + discount_factor * Q[next_state][next_action]  # 计算时序
↪ 差分目标值
td_delta = td_target - Q[state][action]  # 计算时序差分误差
Q[state][action] += alpha * td_delta  # 更新动作值函数
```

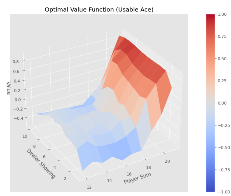
```
if done:  # 如果当前回合结束
    break  # 跳出循环，结束回合

action = next_action  # 更新当前动作为下一步的动作
state = next_state  # 更新当前状态为下一步的状态
```

Q-learning 方法实现

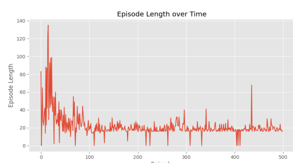
```
# 步骤 1: 执行一步
action_probs = policy(state)  # 获取当前状态下的动作概率分布
action = np.random.choice(np.arange(len(action_probs)), p=action_probs)  # 根
↪ 据概率选择动作
next_state, reward, done, _ = env.step(action)  # 执行选定的动作，获取下一个状
↪ 态、奖励等信息

# 更新统计信息
stats.episode_rewards[i_episode] += reward  # 更新回合奖励
stats.episode_lengths[i_episode] = t  # 更新回合长度

# 步骤 2: 时序差分更新
best_next_action = np.argmax(Q[next_state])  # 选择下一个状态中具有最大动作值的
↪ 动作
td_target = reward + discount_factor * Q[next_state][best_next_action]  # 计
↪ 算时序差分目标值
td_delta = td_target - Q[state][action]  # 计算时序差分误差
Q[state][action] += alpha * td_delta  # 更新动作值函数

if done:  # 如果当前回合结束
    break  # 跳出循环，结束回合

state = next_state  # 更新当前状态为下一步的状态
```

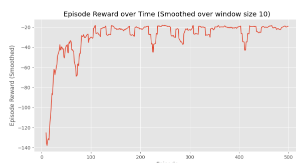
4 实验结果

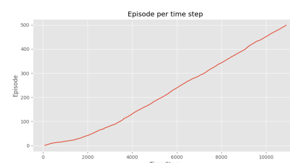


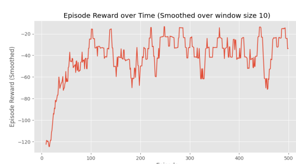


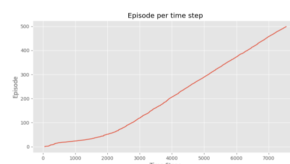
经过 5000000 个 episode 后，greedy policy 依然没有找到最优策略，但是 ϵ-greedy policy 所找到的最优策略要比单纯的 greedy policy 好的多。











用 Sarsa 和 Q-learning 两种算法求解最佳策略。Sarsa 产生数据的策略和更新 Q 值的策略相同，即属于 on-policy 算法；而 Q-learning 更新 Q 值的策略为贪婪策略，其产生数据的策略和更新 Q 值的策略不同，即属于 off-policy 算法；对于 Sarsa 算法而言，它的迭代速度较慢，它选择的路径较长但是相对比较安全，因此每次迭代的累积奖励也比较多，对于 Q-leaning 而言，它的迭代速度较快，由于它每次迭代选择的是贪婪策略因此它更有可能选择最短路径，不过这样更容易掉入悬崖，因此每次迭代的累积奖励也比较少。