Reinforcement Learning Lab01 Report

申长硕 PB22020518
AI&DS
stephen_shen@mail.ustc.edu.cn

2024 年 11 月 13 日

摘要

研究和实现强化学习中的关键算法,包括蒙特卡罗方法和时序差分学习(TD)算法,应用于经典游戏环境 Blackjack 和 Cliffwalking。

- 蒙特卡罗算法的实现:(First-Visit Monte-Carlo, Every-Visit Monte-Carlo)
- 时序差分学习算法的实现:(SARSA, Q-Learning)

1 实验背景与目标

本实验旨在通过比较不同的强化学习算法——Monte Carlo、SARSA 和 Q-Learning——在两个经典环境中的表现，分析它们的优缺点，并探讨它们在实际应用中的适用性和效率。

1.1 Blackjack

Blackjack 是一种经典的卡牌游戏，目标是在不超过 21 点的情况下获得比庄家更高的点数。玩家可以选择“要牌”或“停牌”，并根据手中的牌和庄家的展示牌做出决策。该游戏的状态空间包括玩家的点数、庄家的展示牌和玩家手中是否有可用的王牌（Ace）。在该环境中，强化学习算法的目标是通过学习最优策略来最大化长期奖励。通过实现 First-Visit Monte Carlo 和 Every-Visit Monte Carlo，我们将评估这些算法在学习 Blackjack 策略时的收敛速度和稳定性。

1.1.1 Cliff Walking

Cliff Walking 是一个典型的强化学习环境，通常用于展示价值迭代和策略迭代的效果。该环境呈现为一个网格，Agent 位于起始位置，导航到目标位置，同时避免走到悬崖边缘。每一步的奖励通常是 -1，而落入悬崖则会给予较大的负奖励。目标是找到一条最优路径，以最小化所需的步数和避免惩罚。通过观察 SARSA, Q-Learning, Double Q-Learning 在 Cliff Walking 环境中的表现，我们希望揭示这些算法在策略学习和探索-利用平衡方面的差异。

通过这两个环境的实验，更深入地理解不同强化学习算法的特性及其在解决实际问题中的应用潜力。

2 实验方法与实现

本实验采用三种强化学习算法：Monte Carlo、SARSA 和 Q-Learning，以分别解决 Blackjack 和 Cliff Walking 环境中的决策问题。以下是每种方法的原理及其实现。

2.1 Monte Carlo 方法

Monte Carlo 方法是一种基于样本的强化学习技术，通过对多次实验的结果进行统计分析来估计状态的价值和策略。该方法的核心思想是通过生成完整的 episode，并在回合结束后更新状态-动作值函数（Q 值）。Monte Carlo 方法可以分为两类：**First-Visit Monte Carlo** 和 **Every-Visit Monte Carlo**。在 First-Visit 版本中，仅在状态-动作对首次出现时更新其值，而 Every-Visit 版本则在每次访问时更新。

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

Algorithm parameter: small $\varepsilon > 0$
Initialize:
    $\pi \leftarrow$ an arbitrary $\varepsilon$-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 $\pi$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    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 $a \in \mathcal{A}(S_t)$:
                $\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$
```

图 1: Pseudo Code Monte Carlo

2.1.1 First-Visit Monte Carlo

我在实现 First-Visit MC 的时候扫了两遍当前 episode，第一遍正向扫标记各个状态动作对第一次出现的时间步，第二边反向扫计算 **G** 并只将第一遍扫描时标记的时间步进行更新

具体执行结果请见后面结果分析

```python
def mc(env, num_episodes, discount_factor=1.0, epsilon=0.1):
    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)

    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

    for i_episode in range(1, num_episodes + 1):
        # Print out which episode we're on, useful for debugging.
        if i_episode % 1000 == 0:
            print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
            sys.stdout.flush()
        #  Generate an episode.
            # An episode is an array of (state, action, reward) tuples
            episode = [] # 之后用于存储(state, action, value)元组
            state = env.reset() # 重置环境, 获得初始状态
            done = False

            while not done:
                action_probs = policy(state) # 根据当前策略选择动作概率
                action = np.random.choice(np.arange(len(action_probs)), p=action_probs) # 根据action_probs来选择动作
                next_state, reward, done, _ = env.step(action) # 执行动作并获得下一步的状态、奖励等
                episode.append((state, action, reward)) # 存储`状态-动作-奖励`
                state = next_state

            # Calculate average return for this state over all sampled episodes
            G = 0
            first_visit_idx = {} # 记录一下每个状态-动作对出现的第一个位置
            first_visit_t = set()
            for t, (state, action, _) in enumerate(episode):
                if (state, action) not in first_visit_idx:
                    first_visit_idx[(state, action)] = t
                    first_visit_t.add(t)

            for t, (state, action, reward) in enumerate(reversed(episode)):
                G = reward + discount_factor * G  # 计算回报
                if t in first_visit_t:
                    returns_sum[(state, action)] += G  # 累加回报
                returns_count[(state, action)] += 1  # 更新计数
                Q[state][action] = returns_sum[(state, action)] / returns_count[(state, action)]  # 更新 Q 值
                policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)  # 更新策略
    return Q, policy
```

2.1.2 Every-Visit Monte Carlo

实现 Every-Visit MC 方法只需要逆序扫描一遍即可，每次都更新的当前状态动作对的 Q 值

```python
def mc_every_visit(env, num_episodes, discount_factor=1.0, epsilon=0.1):
    """
    Monte Carlo Control using Epsilon-Greedy policies.
    Finds an optimal epsilon-greedy policy.
    """

    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)
    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

    for i_episode in range(1, num_episodes + 1):
        if i_episode % 1000 == 0:
            print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
            sys.stdout.flush()

        # Step 1: Generate an episode
        episode = []
        state = env.reset()
        done = False

        while not done:
            action_probs = policy(state)
            action = np.random.choice(np.arange(len(action_probs)), p=action_probs)
            next_state, reward, done, _ = env.step(action)
            episode.append((state, action, reward))
            state = next_state

        # Step 2: Calculate returns for each (state, action) pair
        G = 0
        # Reverse iterate over the episode
        for state, action, reward in reversed(episode):
            G = reward + discount_factor * G
            returns_sum[(state, action)] += G
            returns_count[(state, action)] += 1
            Q[state][action] = returns_sum[(state, action)] / returns_count[(state, action)]

            # Update policy
            policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

    return Q, policy
```

2.2 TD: SARSA

SARSA 方法是一种基于时间差（Temporal Difference, TD）的强化学习算法，它是 on-policy 控制算法，通过与环境的交互学习最优策略。SARSA 即 **State-Action-Reward-State-Action**，核心思想是在每一步中使用当前策略进行学习，更新状态-动作值函数（Q 值）以寻找最优策略。

```
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., $\varepsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-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
```

图 2: Pseudo Code SARSA

代码实现

```python
def sarsa(env, num_episodes, discount_factor=1.0, alpha=0.5, epsilon=0.1):
    # The final action-value function.
    # A nested dictionary that maps state -> (action -> action-value).
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    stats = plotting.EpisodeStats(
        episode_lengths=np.zeros(num_episodes),
        episode_rewards=np.zeros(num_episodes))

    policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

    for i_episode in range(num_episodes):
        # Print out which episode we're on, useful for debugging.
        if (i_episode + 1) % 100 == 0:
            print("\rEpisode {}/{}.".format(i_episode + 1, num_episodes), end="")
            sys.stdout.flush()
        # Reset the environment
        state = env.reset()

        # One step in the environment
        for t in itertools.count():
            # step 1 : Take a step( 1 line code, tips : env.step() )
            action_probs = policy(state)
            action = np.random.choice(
                np.arange(len(action_probs)), p=action_probs
            )
            next_state, reward, done, _ = env.step(action)

            # step 2 : Pick the next action
            # Update statistics
            stats.episode_rewards[i_episode] += reward
            stats.episode_lengths[i_episode] = t

            next_action_probs = policy(next_state)
            next_action = np.random.choice(
                np.arange(len(next_action_probs)), p=next_action_probs
            )

            # step 3 : TD Update
            # compute Q value
            if next_state not in Q or next_action not in Q[next_state]:
                Q[next_state][next_action] = 0

            Q[state][action] += alpha * (
                reward + discount_factor * Q[next_state][next_action] - Q[state][action]
            )

            state, action = next_state, next_action
            if done:
                break

    return Q, stats
```

2.3 TD: Q-Learning

Q-Learning 是一种基于时间差（Temporal Difference, TD）的强化学习算法，属于 off-policy 控制算法。Q-Learning 旨在通过学习最优的状态-动作值函数（Q 值）来找到最优策略，即使是在探索阶段也能获得最优的行为策略。Q-Learning 的核心是利用当前的 Q 值更新来指导学习，而不依赖于当前的策略。而 Double Q-Learning 则通过学习两个状态动作值函数，双方使用彼此的当前最优策略来做学习更新。

2.3.1 Q-Learning

```
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 $Q(terminal, \cdot) = 0$

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

图 3: Pseudo Code Q-Learning

```python
def q_learning(env, num_episodes, discount_factor=1.0, alpha=0.5, epsilon=0.1):
    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    stats = plotting.EpisodeStats(
        episode_lengths=np.zeros(num_episodes),
        episode_rewards=np.zeros(num_episodes))

    policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

    for i_episode in range(num_episodes):
        if (i_episode + 1) % 100 == 0:
            print("\rEpisode {}/{}.".format(i_episode + 1, num_episodes), end="")

        state = env.reset()

        for t in itertools.count():
            action_probs = policy(state)
            action = np.random.choice(np.arange(len(action_probs)), p=action_probs)

            next_state, reward, done, _ = env.step(action)
            stats.episode_rewards[i_episode] += reward
            stats.episode_lengths[i_episode] = t

            best_next_action = np.argmax(Q[next_state])
            Q[state][action] += alpha * (reward + discount_factor * Q[next_state][
                best_next_action] - Q[state][action])

            state = next_state

            if done:
                break
    return Q, stats
```

2.3.2 Double Q-Learning

```
Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q_1(s,a)$ and $Q_2(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\mathit{terminal}, \cdot) = 0$
Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using the policy $\varepsilon$-greedy in $Q_1 + Q_2$
        Take action $A$, observe $R$, $S'$
        With 0.5 probabilility:
            $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S',a)\big) - Q_1(S,A)\Big)$
        else:
            $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S',a)\big) - Q_2(S,A)\Big)$
        $S \leftarrow S'$
    until $S$ is terminal
```

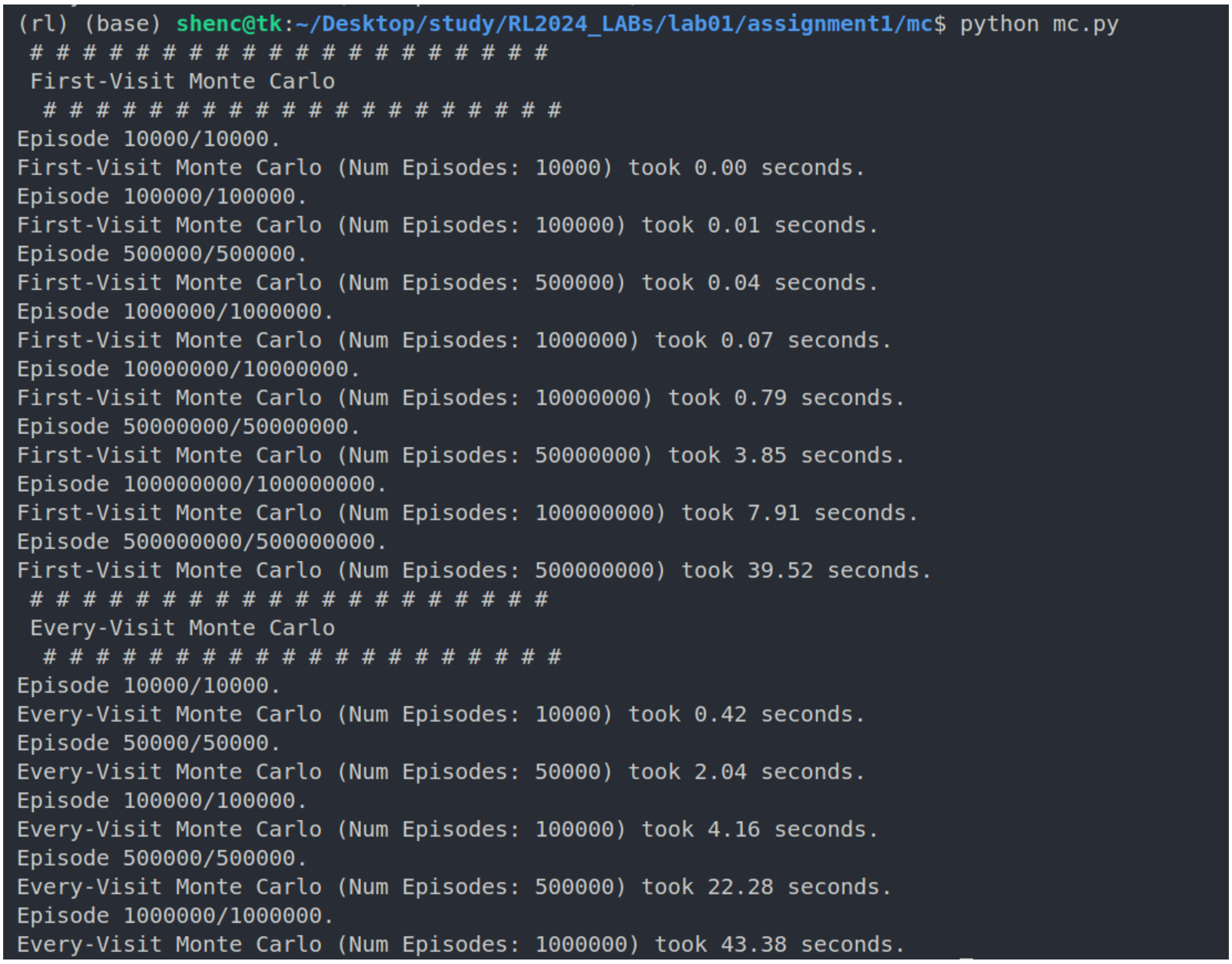
图 4: Pseudo Code Double Q-Learning

```python
def double_q_learning(env, num_episodes, discount_factor=1.0, alpha=0.5, epsilon=0.1):
    Q1 = defaultdict(lambda: np.zeros(env.action_space.n))
    Q2 = defaultdict(lambda: np.zeros(env.action_space.n))
    stats = plotting.EpisodeStats(
        episode_lengths=np.zeros(num_episodes),
        episode_rewards=np.zeros(num_episodes))

    for i_episode in range(num_episodes):
        if (i_episode + 1) % 100 == 0:
            print("\rEpisode {}/{}.".format(i_episode + 1, num_episodes), end="")

        state = env.reset()

        for t in itertools.count():
            action_probs = make_epsilon_greedy_policy(Q1, epsilon, env.action_space.n)(state)
            action = np.random.choice(np.arange(len(action_probs)), p=action_probs)
            next_state, reward, done, _ = env.step(action)

            stats.episode_rewards[i_episode] += reward
            stats.episode_lengths[i_episode] = t

            if np.random.rand() < 0.5:
                best_next_action = np.argmax(Q2[next_state])
                Q1[state][action] += alpha * (reward + discount_factor * Q2[next_state][
                    ↪ best_next_action] - Q1[state][action])
            else:
                best_next_action = np.argmax(Q1[next_state])
                Q2[state][action] += alpha * (reward + discount_factor * Q1[next_state][
                    ↪ best_next_action] - Q2[state][action])

            state = next_state

            if done:
                break
    return Q1, Q2, stats
```

3 实验结果与分析

3.1 Monte Carlo for Blackjack

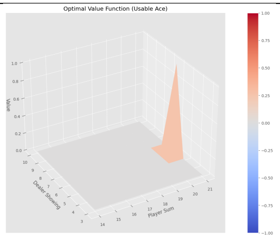
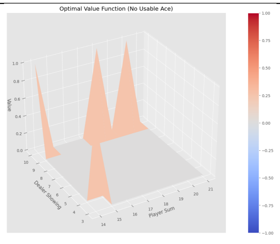
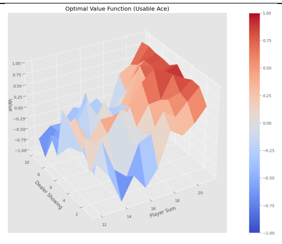
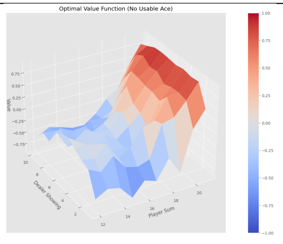
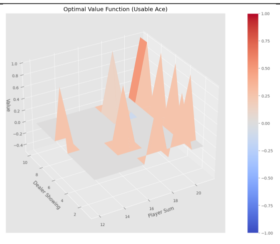
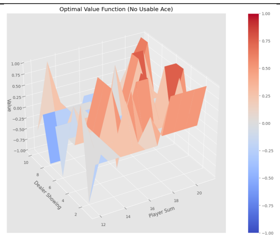
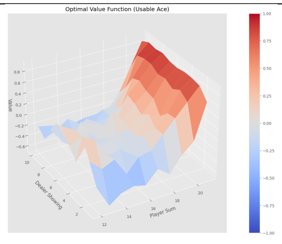
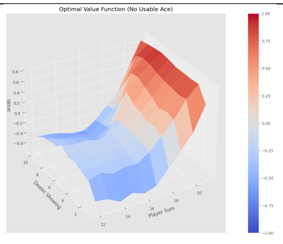
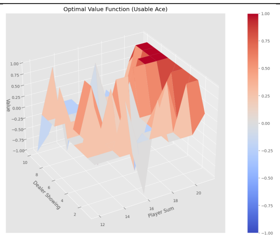
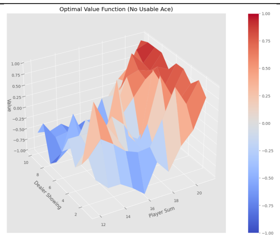
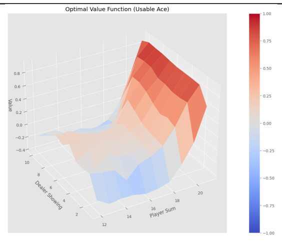
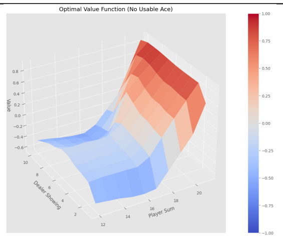
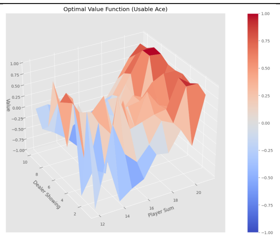
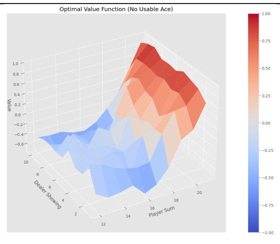
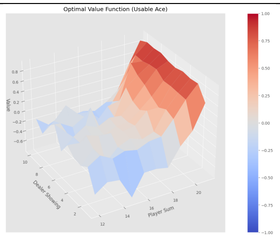
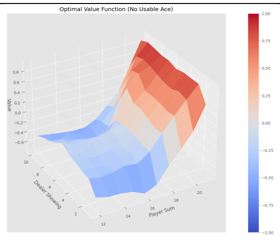
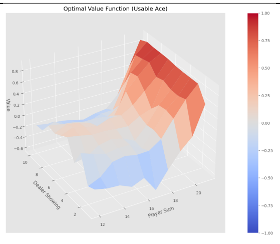
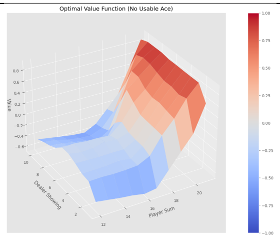
3.1.1 输出



```
(rl) (base) shenc@tk:~/Desktop/study/RL2024_LABs/lab01/assignment1/mc$ python mc.py
 # # # # # # # # # # # # # # # # # # # #
 First-Visit Monte Carlo
  # # # # # # # # # # # # # # # # # # # #
Episode 10000/10000.
First-Visit Monte Carlo (Num Episodes: 10000) took 0.00 seconds.
Episode 100000/100000.
First-Visit Monte Carlo (Num Episodes: 100000) took 0.01 seconds.
Episode 500000/500000.
First-Visit Monte Carlo (Num Episodes: 500000) took 0.04 seconds.
Episode 1000000/1000000.
First-Visit Monte Carlo (Num Episodes: 1000000) took 0.07 seconds.
Episode 10000000/10000000.
First-Visit Monte Carlo (Num Episodes: 10000000) took 0.79 seconds.
Episode 50000000/50000000.
First-Visit Monte Carlo (Num Episodes: 50000000) took 3.85 seconds.
Episode 100000000/100000000.
First-Visit Monte Carlo (Num Episodes: 100000000) took 7.91 seconds.
Episode 500000000/500000000.
First-Visit Monte Carlo (Num Episodes: 500000000) took 39.52 seconds.
 # # # # # # # # # # # # # # # # # # # #
 Every-Visit Monte Carlo
  # # # # # # # # # # # # # # # # # # # #
Episode 10000/10000.
Every-Visit Monte Carlo (Num Episodes: 10000) took 0.42 seconds.
Episode 50000/50000.
Every-Visit Monte Carlo (Num Episodes: 50000) took 2.04 seconds.
Episode 100000/100000.
Every-Visit Monte Carlo (Num Episodes: 100000) took 4.16 seconds.
Episode 500000/500000.
Every-Visit Monte Carlo (Num Episodes: 500000) took 22.28 seconds.
Episode 1000000/1000000.
Every-Visit Monte Carlo (Num Episodes: 1000000) took 43.38 seconds.
```

图 5: 对不同 num_episodes 进行训练

表 1: 不同 num_episodes 下的最优值函数（选其中几个）

num_epsds	First Visit (Usable Ace)	First Visit (No Usable Ace)	Every Visit (Usable Ace)	Every Visit (No Usable Ace)
10,000				
100,000				
1,000,000				
10,000,000			无需再练	无需再练
100,000,000			无需再练	无需再练
500,000,000			无需再练	无需再练

3.1.2 结果分析

在 Blackjack 游戏中，我们使用了 First-Visit 和 Every-Visit Monte Carlo 方法进行训练。通过不同的 num_episodes，我们观察到以下几点：

- **收敛性**：随着 num_episodes 的增加，算法的收敛速度显著提高。特别是在 100,000 次和 1,000,000 次训练后，策略的性能有了明显改善，展示了更高的平均回报。

- **可用 Ace 和不可用 Ace 的表现**：在使用可用王牌的情况下，算法学习了更优的策略，回报普遍高于不可用王牌的情况。这表明可用王牌在 Blackjack 中是一个重要的策略因素。
- **算法比较**：First-Visit 和 Every-Visit 方法在收敛速度和稳定性上表现相似，但在某些 num_episodes 下，Every-Visit 方法的学习速度稍快。这可能与每次访问都更新 Q 值有关，使得值函数更新更加频繁。
- 但是两种方法在这个任务上执行相同时间得到的效果是差不多的，毕竟 first-visit 虽单 episode 没有 every-visit 带来的训练收益大，但是它快呀。

3.2 Temporal Difference for Cliff Walking

3.2.1 输出

表 2: 三种方法 (**SARSA, Q-Learning, Double Q-Learning**) 在 Cliff Walking 中的表现

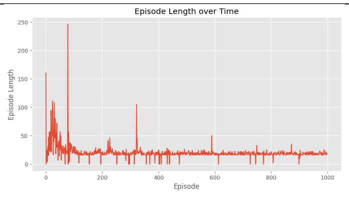
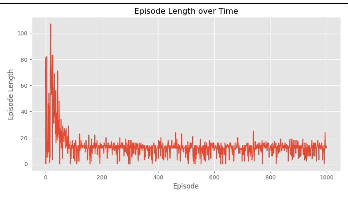
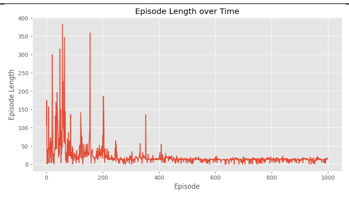
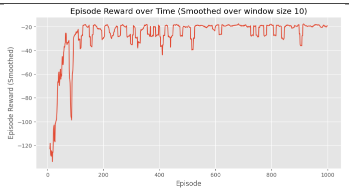
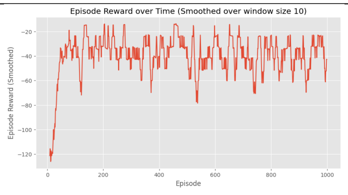
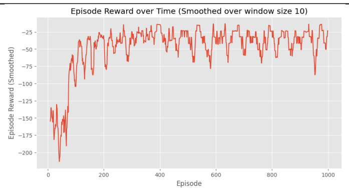
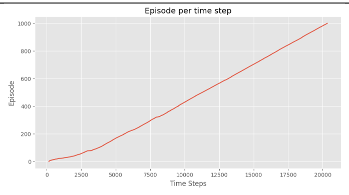
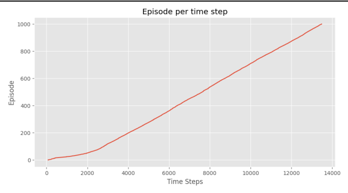
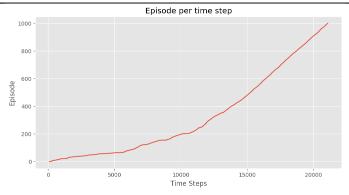
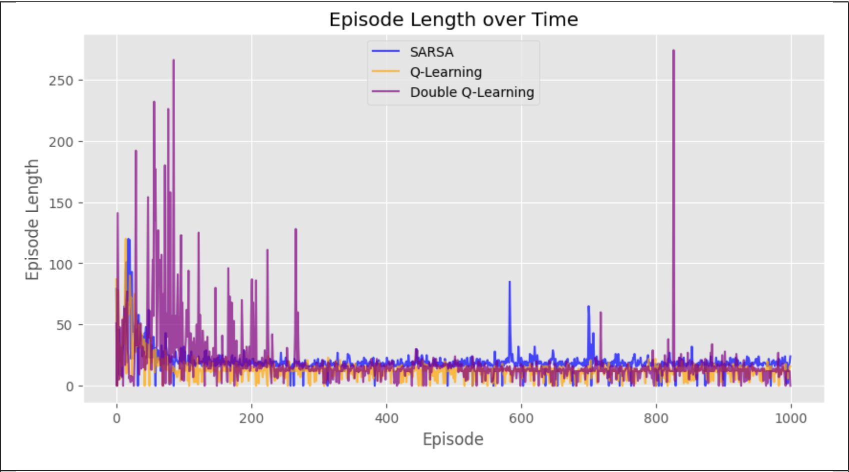
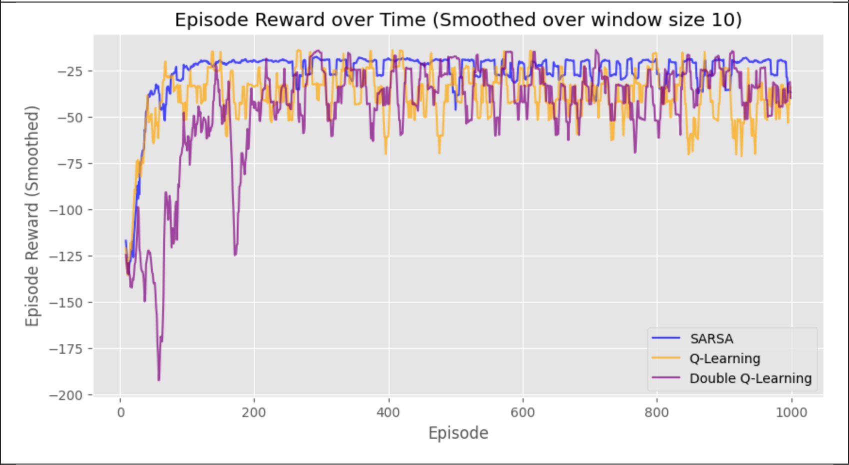
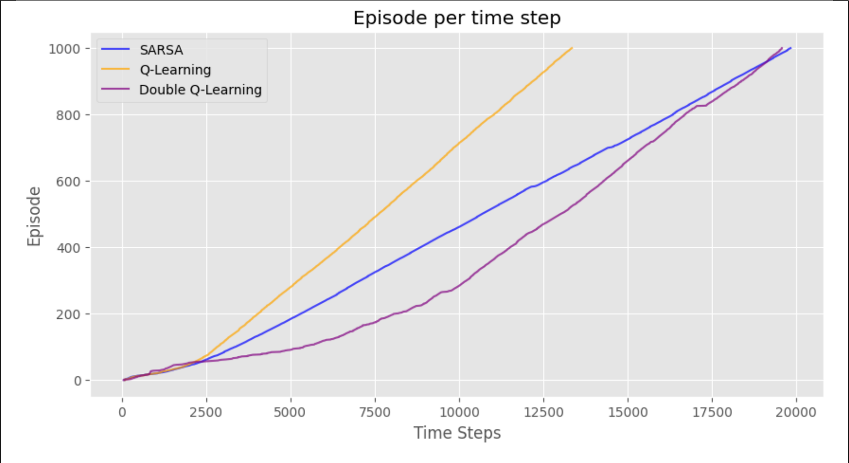
Matrix	**SARSA**	**Q-Learning**	**Double Q-Learning**
length			
reward			
timesteps			

表 3: 放在同一图中比较

length	
reward	
timesteps	

3.2.2 结果分析

针对 Cliff Walking 环境，我们分别应用了 SARSA、Q-Learning 和 Double Q-Learning 方法进行比较。通过实验结果，我们得出以下结论：

- **学习效率**：三种方法的学习效果都还不错，如果比较 reward 的话，发现 sarsa 的效果最好，直观上来讲，sarsa 通过当前策略选择下一步，由于有 epsilon-greedy 的贪心在，所以方便更好的 exploration，实际训练出来的效果也确实不错。
- **稳定性**：Double Q-Learning 在学习过程中表现出更好的稳定性，尤其是在环境较复杂时。通过学习两个 Q 值函数，Double Q-Learning 减少了由于过估计导致的学习波动。
- **策略探索**：所有算法在 Exploration-Exploitation 平衡方面表现出不同的特性。SARSA 更倾向于遵循当前

策略，而 Q-Learning 更积极地探索，这使得它在长期学习中能找到更优的策略。

4 总结与反馈

4.1 总结

本实验通过比较 Monte Carlo 方法和时序差分学习算法在 Blackjack 和 Cliff Walking 环境中的表现，深入分析了不同算法的优缺点。发现：

- Monte Carlo 方法在 Blackjack 中能够有效学习到最优策略，但需要大量的训练回合。
- 在 Cliff Walking 环境中，Q-Learning 和 Double Q-Learning 提供了比 SARSA 更优的学习效率和稳定性。
- 这些发现为未来的 RL 课程的学习有价值的参考，尤其是在选择合适的算法解决具体问题时。

4.2 反馈

实验环境有点老，和当前较新版本的 setuptools 有冲突，在配置环境上有较大阻力（可能只有我一个人有这种问题 hhh）。