机器学习 HW4: 强化学习

姓名：聂礼昂
学号：2022012097

2024 年 12 月 12 日

1 介绍

本次作业是清华大学软件学院，机器学习, 2024 年秋季学期课程第四次作业。

2 Bellman Equation

问题 1

在折扣系数 $\gamma = 0.5$ 的马尔可夫决策过程 (MDP) 中，策略 π 的状态值函数 $V^\pi(s)$ 的定义为：

$$V^\pi(s) = \mathbb{E}^\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s \right], \tag{1}$$

其中：

- $S_0 = s$ 表示初始状态为 s。
- R_{t+1} 表示从时间步 t 到时间步 $t+1$ 所获得的奖励。
- γ 是折扣因子，用于权衡短期与长期奖励。

问题 2

状态值函数 $V^\pi(s)$ 所符合的贝尔曼 (Bellman) 期望方程为：

$$v^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v^\pi(s') \right) \tag{2}$$

其中：

- $\mathcal{A}$ 为动作集合。

- $\pi(a \mid s)$ 表示在状态 s 下采取动作 a 的概率。
- $\mathcal{P}_{ss'}^a$ 表示从状态 s 执行动作 a 后转移到状态 s' 的概率。
- $\mathcal{R}_s^a$ 表示从状态 s 执行动作 a 转移到状态 s' 所获得的奖励。
- $V^\pi(s')$ 是状态 s' 的值函数。

问题 3

假设均匀随机策略 π_0 (即所有动作被均等选择)，初始状态值函数为:

$$V_0^{\pi_0}(A) = V_0^{\pi_0}(B) = V_0^{\pi_0}(C) = 0.$$

根据贝尔曼期望方程，我们可以逐步更新 V^{π_0}。

对于状态 A:

$$V_1^{\pi_0}(A) = R_A^{ab} + \gamma V_0^{\pi_0}(B) = -4$$

对于状态 B:

$$V_1^{\pi_0}(B) = \frac{1}{2}\left[R_B^{ba} + \gamma V_0^{\pi_0}(A)\right] + \frac{1}{2}\left[R_B^{bc} + \gamma V_0^{\pi_0}(C)\right] = 1.5$$

对于状态 C:

$$V_1^{\pi_0}(C) = \frac{1}{2}\left[R_C^{cb} + \gamma V_0^{\pi_0}(B)\right] + \frac{1}{2}\left[R_C^{ca} + \gamma(\mathcal{P}_{CA}^{ca} V_0^{\pi_0}(A) + \mathcal{P}_{CC}^{ca} V_0^{\pi_0}(C))\right] = 4$$

问题 4

在求得 V^{π_0} 之后，我们可以利用贪心策略 π_1:

$$q_\pi(s,a) = R(s,a) + \gamma \sum_{s'} \mathcal{P}_{ss'}^a v^{\pi_0}(s')$$

$$\pi_1(s) = \arg\max_a \left[q_\pi(s,a)\right],$$

其中 $q_\pi(s,a)$ 是 state-action value 函数。对于每个状态 s，我们选择使得 $q_\pi(s,a)$ 最大的动作 a，从而确定新的最优策略 π_1。

$$\pi_1(a) = ab$$

$$\pi_1(b) = bc$$

$$\pi_1(c) = ca$$

3 TD(λ) & Eligibility Trace

3.1 后向视角

累积资格跟踪可以写为

$$e_t(s) = \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \mathcal{I}(s, S). \tag{3}$$

因此，后向视角状态 s 的价值更新量可以写为

$$\begin{aligned}
\Delta V_{all}^{back}(s) = \sum_{t=0}^{T-1} \Delta V_t^{back}(s) &= \alpha \sum_{t=0}^{T-1} \delta_t \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \mathcal{I}(s, s_t) &(4)\\
&= \alpha \sum_{k=0}^{T-1} \sum_{t=0}^{k} (\gamma\lambda)^{k-t} \mathcal{I}(s, s_t) \delta_k &(5)\\
&= \alpha \sum_{t=0}^{T-1} \sum_{k=t}^{T-1} (\gamma\lambda)^{k-t} \mathcal{I}(s, s_t) \delta_k &(6)\\
&= \alpha \sum_{t=0}^{T-1} \mathcal{I}(s, s_t) \sum_{k=t}^{T-1} (\gamma\lambda)^{k-t} \delta_k. &(7)
\end{aligned}$$

3.2 前向视角

由前向视角的更新公式可以得出以下表达式

$$\Delta V_t^{for}(s_t) = \alpha(R_t^{\lambda} - V_t(s_t))$$

3.3 等价性

首先计算在前向视角中，状态 s 整条轨迹的价值更新量

$$
\begin{aligned}
\frac{1}{\alpha}\Delta V_t^{for}(s_t) &= R_t^{\lambda} - V_t(s_t) \\
&= -V_t(s_t) + (1-\lambda)\lambda^0[r_{t+1} + \gamma V_t(s_{t+1})] \\
&\quad + (1-\lambda)\lambda^1[r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2})] \\
&\quad + (1-\lambda)\lambda^2[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V_t(s_{t+3})] \\
&\quad \vdots \\
&= -V_t(s_t) \\
&\quad + (\gamma\lambda)^0[r_{t+1} + \gamma V_t(s_{t+1}) - \gamma\lambda V_t(s_{t+1})] \\
&\quad + (\gamma\lambda)^1[r_{t+2} + \gamma V_t(s_{t+2}) - \gamma\lambda V_t(s_{t+2})] \\
&\quad + (\gamma\lambda)^2[r_{t+3} + \gamma V_t(s_{t+3}) - \gamma\lambda V_t(s_{t+3})] \\
&\quad \vdots \\
&= (\gamma\lambda)^0[r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)] \\
&\quad + (\gamma\lambda)^1[r_{t+2} + \gamma V_t(s_{t+2}) - V_t(s_{t+1})] \\
&\quad + (\gamma\lambda)^2[r_{t+3} + \gamma V_t(s_{t+3}) - V_t(s_{t+2})] \\
&\quad \vdots \\
&\approx \sum_{k=t}^{\infty}(\gamma\lambda)^{k-t}\delta_k \\
&\approx \sum_{k=t}^{T-1}(\gamma\lambda)^{k-t}\delta_k.
\end{aligned}
$$

因此，对于前向视角，整条轨迹，状态 s 的价值更新量为

$$
\Delta V_{all}^{for}(s) = \sum_{t=0}^{T-1}\Delta V_t^{for}(s_t)\mathcal{I}(s,s_t) = \alpha\sum_{t=0}^{T-1}\mathcal{I}(s,s_t)\sum_{k=t}^{T-1}(\lambda\gamma)^{k-t}\delta_k, \tag{8}
$$

此处结果与 3.1 公式 (7) 所计算出的后向视角状态 s 的价值更新量相等。

因此，我们可以得出

$$
\Delta V_{all}^{for}(s) = \Delta V_{all}^{back}(s) \tag{9}
$$

4 Q-Learning & Sarsa

4.1 Q-learning 算法

增加一行代码即可

```
Q[s][a] = Q[s][a] + lr * (reward + gamma * np.max(Q[nexts]) - Q[s][a])
```

4.2 Sarsa 算法

参照之前的 Qlearning 即可实现 Sarsa 算法

```python
def Sarsa(
        env: gym.Env,
        num_episodes: int = 5000,
        gamma: float = 0.95,
        lr: float = 0.1,
        e: float = 1,
        decay_rate: float = 0.99
    ):


    # Initialize Q-table with zeros
    Q = np.zeros((env.observation_space.n, env.action_space.n))
    episode_reward = []

    for i in range(num_episodes):
        tmp_episode_reward = 0
        s, info = env.reset()

        # Choose action using epsilon-greedy strategy
        if np.random.rand() > e:
            a = np.argmax(Q[s])
        else:
            a = np.random.randint(env.action_space.n)

        while True:
            # Take action and observe next state and reward
            nexts, reward, terminated, truncated, info = env.step(a)
            done = terminated or truncated

            # Choose next action using epsilon-greedy strategy
            if np.random.rand() > e:
                nexta = np.argmax(Q[nexts])
            else:
                nexta = np.random.randint(env.action_space.n)

            # Update Q[s][a] using SARSA formula
            Q[s][a] = Q[s][a] + lr * (reward + gamma * Q[nexts][nexta] - Q[s][a])

            tmp_episode_reward += reward
            s, a = nexts, nexta  # Move to the next state and action

```

```
            if done:
                break

        # Record total reward for this episode
        episode_reward.append(tmp_episode_reward)
        print("Total reward until episode", i + 1, ":", tmp_episode_reward)
        sys.stdout.flush()

        # Decay epsilon
        if i % 10 == 0:
            e = e * decay_rate

    return Q, episode_reward
```

4.3 不同步长下的学习曲线

以下是在 reward=-0.03, 步长为 0.1 和 0.5 的情况下两种算法的的学习曲线

如图所示，当学习率较高时，曲线的波动性更大，导致算法在最优策略附近震荡，而不是稳定收敛。

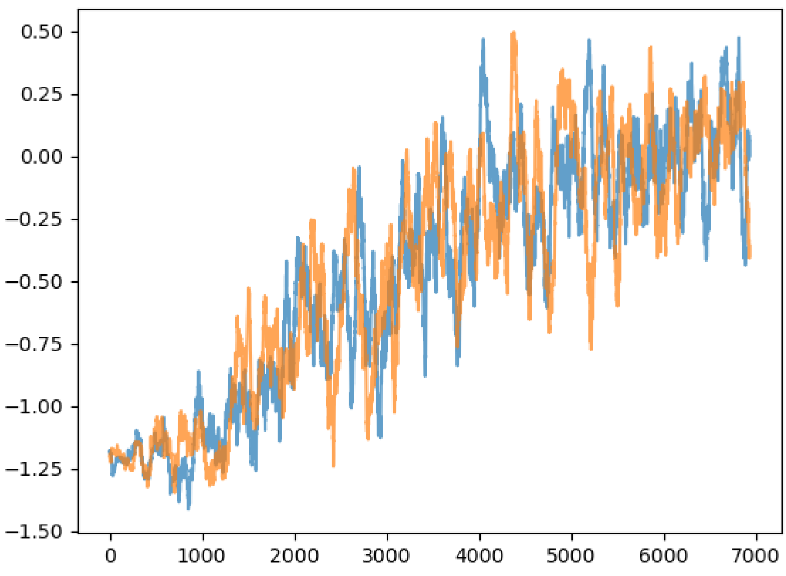

图 1: lr=0.1 reward=-0.03

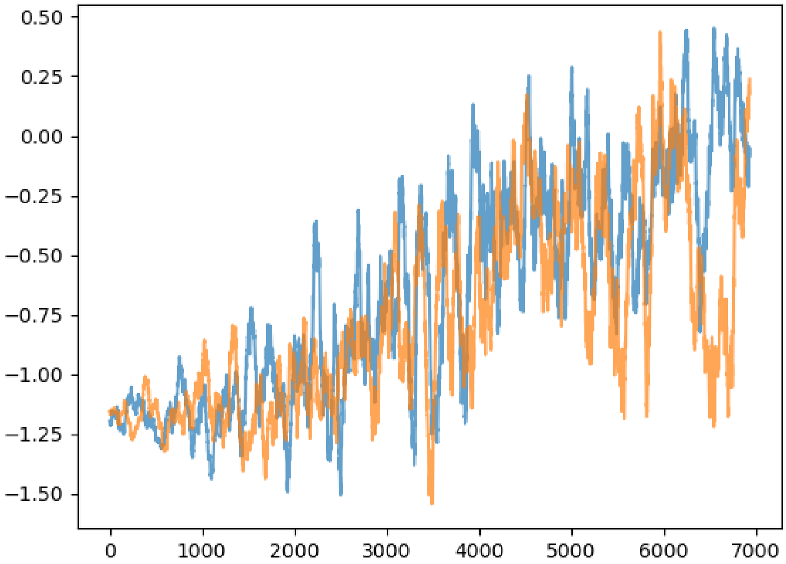

图 2: lr=0.5 reward=-0.03

4.4 增大惩罚值的策略分析

以下是在步长为 0.1，reward=-0.03 和-0.3 的情况下两种算法的的学习曲线

增大惩罚值会使得算法更加倾向于学习避免惩罚的策略。因为每一步的惩罚更大，算法会更快地认识到某些动作或状态转移是不利的，从而加速学习过程，使得算法更快地收敛到一个较好的策略。

在对不同的惩罚值，算法所学到的策略进行分析之后，可以发现，增大惩罚值导致算法学到的策略更加保守。因为每一步的损失更大，算法可能会倾向于选择那些即使不是最优但风险较小的动作，以减少总体的损失。

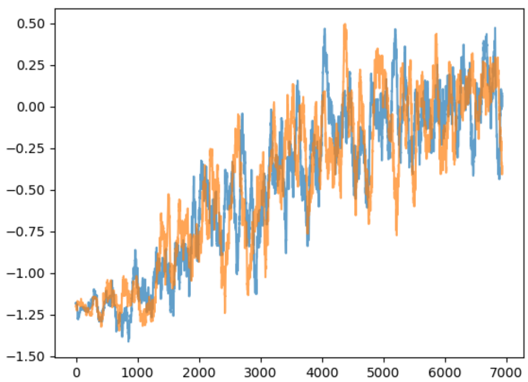

图 3: lr=0.1 reward=-0.03

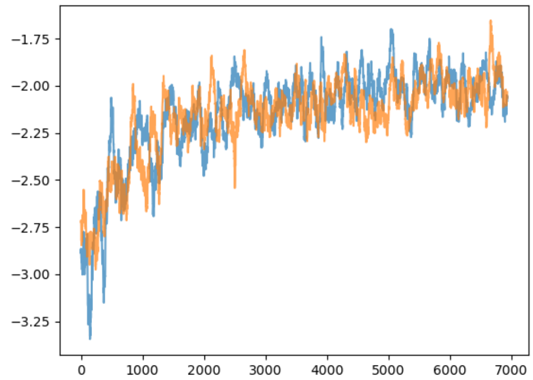

图 4: lr=0.1 reward=-0.3

5 REINFORCE & AC

5.1 REINFORCE 算法

```
def learn(self):

        loss = []

        # 计算回报
        discounted_rewards = []
        cumulative_reward = 0
        for r in reversed(self.rewards):
            cumulative_reward = r + GAMMA * cumulative_reward
            discounted_rewards.insert(0, cumulative_reward)

        # 标准化回报
        discounted_rewards = torch.tensor(discounted_rewards, dtype=torch.float32)
        discounted_rewards = (discounted_rewards - discounted_rewards.mean()) / (discounted_rewards.std() + 1e-7)

        # 计算每个步骤的损失
        loss = []
        for t in range(len(self.states)):
            # 获取当前状态下采取的动作概率
            action_prob = self.action_probs[t]
            action = self.actions[t]

            # 对数概率
            log_prob = torch.log(action_prob.squeeze(0)[action])

```

```
            # 计算损失: -log(P(a_t|s_t)) * R_t
            loss.append(-log_prob * discounted_rewards[t])


        # code for autograd and back propagation
        self.optim.zero_grad()
        loss = torch.cat(loss).sum()
        loss.backward()
        self.optim.step()

        self.states, self.actions, self.action_probs, self.rewards = [], [], [], []
        return loss.item()
```

5.2 TD Actor-Critic 算法

```
def learn(self):

        policy_loss = None
        td_target = None
        delta = None
        self.make_batch()


        # Calculate td_target and delta
        with torch.no_grad():
            # v(S_t+1) for all time steps except the last one
            next_state_values = self.ac.v(self.states_prime)  # v(S_t+1)
            td_target = self.rewards + GAMMA * next_state_values * (1 - self.done)  # R_t+1 +  v(S_t+1)

        # Compute the advantage delta_t = td_target - v(S_t)
        state_values = self.ac.v(self.states)  # v(S_t)
        delta = td_target - state_values  # Advantage function

        # Calculate the policy loss: -log( (a_t | s_t)) * delta_t
        # This requires action probabilities for each state-action pair
        log_probs = torch.log(self.action_probs.gather(1, self.actions))  # log( (a_t | s_t))
        policy_loss = -(log_probs * delta).mean()  # Policy gradient loss

        # compute value loss and total loss
        # td_target is used as a scalar here, and is detached to stop gradient
```

```python
        value_loss = F.smooth_l1_loss(self.ac.v(self.states), td_target.
            ↪ detach())
        loss = policy_loss + value_loss

        # code for autograd and back propagation
        self.optim.zero_grad()
        loss = loss.mean()
        loss.backward()
        self.optim.step()

        self.states, self.actions, self.action_probs, self.rewards = [],
            ↪ [], [], []
        return loss.item()
```

5.3 两个模型

以下是两个不同算法的 loss 曲线，和 Reward 曲线，分别取了三个不同的种子值。

可以看到 REINFORCE 的训练曲线通常比较波动。这是由于它是基于蒙特卡洛方法的，回报的估计通常是高方差的，因此损失函数和奖励的变化可能会表现出很大的波动。在初期，损失函数可能会下降得较慢，训练的稳定性较差。

并且 REINFORCE 的收敛速度较慢，主要是因为它需要依赖全轨迹的回报，且采样高方差会使得梯度估计不准确，收敛过程不稳定，尤其是在较大的状态空间下。

TD Actor-Critic 的训练曲线相较于 REINFORCE 更加平滑。由于 Critic 网络提供了对策略改进的更稳定的反馈（通过 TD 误差），Actor 网络能够更稳定地更新策略，训练过程中损失函数的变化较为平缓。

同时 TD Actor-Critic 通常收敛得比 REINFORCE 更快。Critic 的值函数提供了一个低方差的信号，能够更精确地指导策略更新，从而加速了学习过程。

本次实验由于计算时间太长，对于 episodes=3000 的时候，经常跑到一半电脑就出问题，因此实验数据中没有取到最高的 3000。但对于 reward 是已经达到 500 的了。

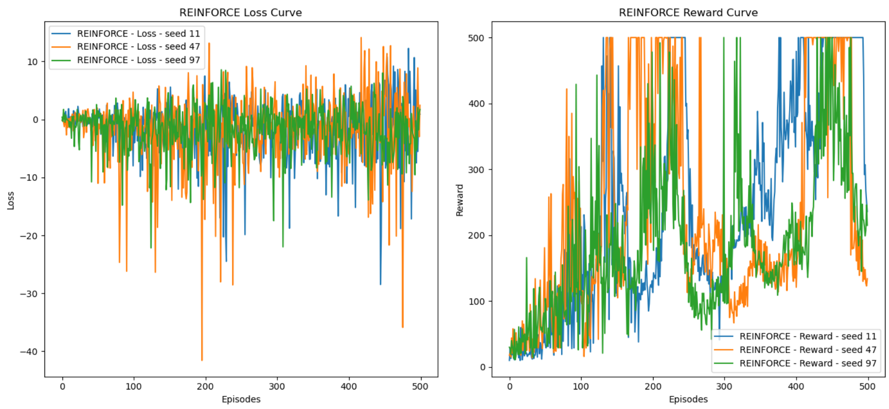

图 5: REINFORCE 算法

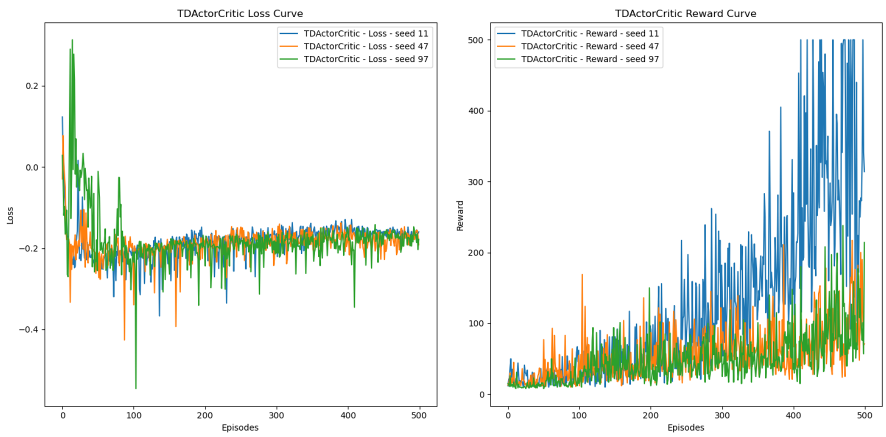

图 6: TD Actor-Critic 算法