人工智能学院

强化学习实验报告

实验九　PPO

姓名：石若川

学号：2111381

专业：智能科学与技术

2024 年 5 月 12 日

1 实验目的

- 学习 PPO 强化学习算法，比较 PPO 算法和策略梯度方法的区别。
- 利用 PPO 算法解决 Gym 库中的 Lunar Lander 环境问题，实现火箭在月球表面的降落。

2 实验原理

2.1 Trust Region Policy Optimization(TRPO)

TRPO 于 2015 年由 John Schulman 等人提出，是一种基于策略梯度方法的算法。一般的策略梯度算法存在以下缺点：

- 很难在整个优化过程选择一个时间步长，特别是由于状态和回报在改变统计特性
- 策略经常会过早地收敛到一个次优的几乎确定的策略

当学习步长不合适时，策略梯度方法很可能会崩溃。传统强化学习算法很难保证单调收敛，而 TRPO 却给出了一个能保证单调收敛的策略改善方法。

考虑一个带折扣的无限 MDP 过程 $(\mathcal{S}, \mathcal{A}, P, r, \rho_0, \gamma)$。其中 ρ_0 为初始状态 s_0 的分布。定义 π 为在状态-动作下的随机策略，$\pi \in [0,1]$。折扣回报函数 $\eta(\pi)$ 为：

$$\eta(\pi) = \mathbb{E}_{s_0, a_0, \ldots}\left[\sum_{t=0}^{\infty} \gamma^t r\left(s_t\right)\right]$$

其中，$s_0 \sim \rho_0\left(s_0\right), a_t \sim \pi\left(a_t \mid s_t\right), s_{t+1} \sim P\left(s_{t+1} \mid s_t, a_t\right)$。

值函数、动作值函数和优势值函数的定义如下：

$$\begin{aligned}
Q_\pi(s_t, a_t) &= \mathbb{E}_{s_{t+1}, a_{t+1}, \ldots}\left[\sum_{l=0}^{\infty} \gamma^l r(s_{t+l})\right], \\
V_\pi(s_t) &= \mathbb{E}_{a_t, s_{t+1}, \ldots}\left[\sum_{l=0}^{\infty} \gamma^l r(s_{t+l})\right], \\
A_\pi(s, a) &= Q_\pi(s, a) - V_\pi(s)
\end{aligned}$$

Sham Kakade 于 2002 年提出了以下替换回报函数：

$$\eta(\tilde{\pi}) = \eta(\pi) + \mathbb{E}_{s_0, a_0, \cdots \sim \tilde{\pi}}\left[\sum_{t=0}^{\infty} \gamma^t A_\pi(s_t, a_t)\right] \tag{1}$$

该替代回报函数将新的策略 $\tilde{\pi}$ 的回报函数拆分为旧的策略 π 的回报函数和新旧策略的回报差。若能保证新旧策略的回报差始终大于等于 0，则可以保证新的策略始终不比旧的策略差，进而保证策略单调

收敛。式1的证明如下：

$$
\begin{aligned}
\mathbb{E}_{\tau|\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t A_\pi\left(s_t,a_t\right)\right] &= \mathbb{E}_{\tau|\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t\left(r\left(s_t\right)+\gamma V_\pi\left(s_{t+1}\right)-V_\pi\left(s_t\right)\right)\right] \\
&= \mathbb{E}_{\tau|\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t\left(r\left(s_t\right)\right)+\sum_{t=0}^{\infty}\gamma^{t+1}V_\pi\left(s_{t+1}\right)-\sum_{t=0}^{\infty}\gamma^t V_\pi(s_t)\right] \\
&= \mathbb{E}_{\tau|\tilde{\pi}}\left[-V_\pi\left(s_0\right)+\sum_{t=0}^{\infty}\gamma^t r\left(s_t\right)\right] \\
&= -\mathbb{E}_{s_0}\left[V_\pi\left(s_0\right)\right]+\mathbb{E}_{\tau|\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t r\left(s_t\right)\right] \\
&= -\eta(\pi)+\eta(\tilde{\pi})
\end{aligned}
$$

将式1展开，可得：

$$
\begin{aligned}
\eta(\tilde{\pi}) &= \eta(\pi)+\sum_{t=0}^{\infty}\sum_s P(s_t=s|\tilde{\pi})\sum_a\tilde{\pi}(a|s)\gamma^t A_\pi(s,a) \\
&= \eta(\pi)+\sum_s\sum_{t=0}^{\infty}\gamma^t P(s_t=s|\tilde{\pi})\sum_a\tilde{\pi}(a|s)A_\pi(s,a) \\
&= \eta(\pi)+\sum_s\rho_{\tilde{\pi}}(s)\sum_a\tilde{\pi}(a|s)A_\pi(s,a) \qquad (2)
\end{aligned}
$$

使用确定性策略 $\tilde{\pi}(s)=\arg\max_a A_\pi(s,a)$ ，如果至少有一个状态-动作对的优势函数为正数，且对该状态有非零的访问概率，则还没有收敛到最优策略；否则，说明该策略已经达到最优。

在函数近似中，由于估计和近似误差，通常不可避免地会有一些状态的预期优势为负，即 $\sum_a\tilde{\pi}(a\mid s)A_\pi(s,a)<0$。$\rho_{\tilde{\pi}}(s)$ 的复杂性使得式2很难优化。因此，忽略状态分布的变化，依然采用旧的策略所对应的状态分布 $\rho_\pi(s)$，原来的代价函数变为：

$$
L_\pi(\tilde{\pi})=\eta(\pi)+\sum_s\rho_\pi(s)\sum_a\tilde{\pi}(a|s)A_\pi(s,a) \qquad (3)
$$

对任意模型参数 θ_0 有：

$$
L_{\pi_{\theta_0}}(\pi_{\theta_0})=\eta(\pi_{\theta_0}),\nabla_\theta L_{\pi_{\theta_0}}(\pi_\theta)\big|_{\theta=\theta_0}=\nabla_\theta\eta(\pi_\theta)\big|_{\theta=\theta_0}
$$

因此可以说明在小步长下，$L_\pi(\tilde{\pi})$ 与 $\eta(\tilde{\pi})$ 变化趋势一致，但我们无法确定这个小步长的取值范围 (信任区域)。

为解决这一问题，引入了 Kakade&Langford 的结论。新策略 π_{new} 可以写成包括旧策略 π_{old} 在内的混合策略：

$$
\pi_{\text{new}}\left(a\mid s\right)=(1-\alpha)\pi_{\text{old}}\left(a\mid s\right)+\alpha\pi'(a\mid s) \qquad (4)
$$

其中 $\pi'=\text{argmax}_\pi L_{\pi_{\text{old}}}(\pi)$。Kakade&Langford 证明了该新策略的回报函数有下界：

$$
\eta\left(\pi_{\text{new}}\right)\geq L_{\pi_{\text{old}}}\left(\pi_{\text{new}}\right)-\frac{2\epsilon\gamma}{(1-\gamma)^2}\alpha^2,\quad \text{where }\epsilon=\max_s\left|\mathbb{E}_{a\sim\pi'(a|s)}\left[A_\pi(s,a)\right]\right| \qquad (5)
$$

该结论限制了策略必须为式4的混合策略。因此作者将其拓展到了一般随机策略。

2.2 一般随机策略

作者将策略梯度的步长 α 修改为策略 π 和 $\tilde{\pi}$ 之间距离的一种度量 (Total Variation Divergence，总变量散度 D_{TV}^{max})。其中，$D_{TV}(p\|q)=\frac{1}{2}\sum_i|p_i-q_i|$, 定义了离散概率分布 p,q 之间的一种距离。定义：

$$D_{\mathrm{TV}}^{\max}(\pi,\tilde{\pi})=\max_s D_{TV}(\pi(\cdot|s)\parallel\tilde{\pi}(\cdot|s)) \tag{6}$$

作者对式5中 ϵ 进行修改，用最大状态动作优势 $\max_{s,a}|A_\pi(s,a)|$ 替代状态期望优势。

令 $\alpha=D_{\mathrm{TV}}^{\max}(\pi_{\mathrm{old}},\pi_{\mathrm{new}})$，将下界修改为：

$$\eta(\pi_{\mathrm{new}})\geq L_{\pi_{\mathrm{old}}}(\pi_{\mathrm{new}})-\frac{4\epsilon\gamma}{(1-\gamma)^2}\alpha^2,\quad where\ \epsilon=\max_{s,a}|A_\pi(s,a)| \tag{7}$$

总变量散度和 KL 散度之间存在关系：$D_{TV}(p\|q)^2\leq D_{\mathrm{KL}}(p\|q)$。

令 $D_{\mathrm{KL}}^{\max}(\pi,\tilde{\pi})=\max_s D_{\mathrm{KL}}(\pi(\cdot\mid s)\|\tilde{\pi}(\cdot\mid s))$ 将式7写为：

$$\eta(\tilde{\pi})\geq L_\pi(\tilde{\pi})-CD_{\mathrm{KL}}^{\max}(\pi,\tilde{\pi}),\quad where\ C=\frac{4\epsilon\gamma}{(1-\gamma)^2} \tag{8}$$

令 $M_i=L_{\pi_\xi}(\pi)-CD_{KL}^{\max}(\pi_i,\pi)$ 为下界函数。由于 $\pi_i+1=\operatorname{argmax}_\pi M_i(\pi)$, 则 $M_i(\pi_{i+1})=\max_\pi M_i(\pi)$，即可通过迭代下界函数来使 $\eta(\tilde{\pi})$ 单调上升。

数学推导如下：

$$\eta(\pi_{i+1})\geq M_i(\pi_{i+1})$$

$$\eta(\pi_i)=M_i(\pi_i)$$

$$therefore,\quad \eta(\pi_{i+1})-\eta(\pi_i)\geq M_i(\pi_{i+1})-M(\pi_i)$$

因此，通过在每轮迭代中最大化 M_i, 我们保证了 η 是非递减的。

2.3 参数化策略的优化

将策略参数化后，利用 θ 替代 π_θ，例如 $\eta(\theta):=\eta(\pi_\theta), L_\theta(\tilde{\theta}):=L_{\pi_\theta}(\pi_{\tilde{\theta}})$, 以及$D_{\mathrm{KL}}(\theta\|\hat{\theta}):=D_{\mathrm{KL}}(\pi_\theta\|\pi_{\bar{\theta}})$。根据式8，策略的更新方法为：

$$\theta_{new}=\arg\max_\theta[L_{\theta_{\mathrm{old}}}(\theta)-CD_{\mathrm{KL}}^{\max}(\theta_{\mathrm{old}},\theta)]$$

该方法受惩罚系数 C 的影响，更新步长会很小，导致更新很慢。一个解决方法是将 KL 散度作为惩罚项的极值问题，转化为 KL 散度作为约束条件的优化问题，即：

$$\arg\max_\theta L_{\theta_{\mathrm{old}}}(\theta),\quad \text{subject to } D_{\mathrm{KL}}^{\max}(\theta_{\mathrm{old}},\theta)\leq\delta \tag{9}$$

根据式9，KL 散度在状态空间的每一点上都有约束。在实际求解过程中，求解如此多的约束是不实际的。因此，作者采用了启发式逼近法，使用 KL 散度的均值：

$$\bar{D}_{\mathrm{KL}}^\rho(\theta_1,\theta_2):=\mathbb{E}_{s\sim\rho}[D_{\mathrm{KL}}(\pi_{\theta_1}(\cdot\mid s)\|\pi_{\theta_2}(\cdot\mid s))]$$

由此，得出了以下策略更新方法：

$$\arg\max_{\theta} L_{\theta_{\text{old}}}(\theta), \quad \text{subject to } \bar{D}_{\text{KL}}^{\rho_{\theta_{old}}}(\theta_{\text{old}}, \theta) \le \delta \tag{10}$$

2.4 基于样本的目标和约束

将 $L_{\theta_{\text{old}}}(\theta)$ 的表达式代入，可得：

$$\arg\max_{\theta} \sum_s \rho_{\theta_{\text{old}}}(s) \sum_a \pi_\theta(a \mid s) A_{\theta_{\text{old}}}(s,a) \quad \text{subject to } \bar{D}_{\text{KL}}^{\rho_{\theta_{\text{old}}}}(\theta_{\text{old}}, \theta) \le \delta \tag{11}$$

首先利用 $\frac{1}{1-\gamma}\mathbb{E}_{s\sim\rho_{\theta_{\text{old}}}}[\ldots]$ 替换 $\sum_s \rho_{\theta_{\text{old}}}(s)[\ldots]$。接着，将优势函数 A_{old} 替换为 Q_{old} 。由于 $A = Q - V$ ，且状态价值函数 V 在状态已知时不变，因此在求极值问题中该替换不影响结果。最后，用 q 表示采样分布，此时单个状态 s_n 对损失函数的贡献为：

$$\sum_a \pi_\theta(a \mid s_n) A_{\theta_{\text{old}}}(s_n, a) = \mathbb{E}_{a\sim q}\left[\frac{\pi_\theta(a \mid s_n)}{q(a \mid s_n)} A_{\theta_{\text{old}}}(s_n, a)\right]$$

最终优化问题转换为：

$$\begin{aligned} &\arg\max_{\theta} \mathbb{E}_{s\sim\rho_{\theta_{\text{old}}}, a\sim q}\left[\frac{\pi_\theta(a|s)}{q(a|s)} Q_{\theta_{\text{old}}}(s,a)\right] \\ &\text{subject to } \mathbb{E}_{s\sim\rho_{\theta_{\text{old}}}}[D_{\text{KL}}(\pi_{\theta_{\text{old}}}(\cdot \mid s) \| \pi_\theta(\cdot \mid s))] \le \delta \end{aligned} \tag{12}$$

对式中的目标函数进行一阶近似，对约束条件进行二阶近似：

$$\mathbb{E}_{s\sim\rho_{\theta_{\text{old}}}, a\sim q}\left[\frac{\pi_\theta(a|s)}{q(a|s)} Q_{\theta_{\text{old}}}(s,a)\right] \approx g^T(\theta - \theta_{old})$$

$$\mathbb{E}_{s\sim\rho_{\theta_{\text{old}}}}[D_{\text{KL}}(\pi_{\theta_{\text{old}}}(\cdot \mid s) \| \pi_\theta(\cdot \mid s))] \approx \frac{1}{2}(\theta - \theta_{old})^T \boldsymbol{F}(\theta - \theta_{old})$$

其中，g 表示目标函数的梯度

$$g = \nabla_\theta \mathbb{E}_{s\sim\rho_{\theta_{\text{old}}}, a\sim q}\left[\frac{\pi_\theta(a|s)}{q(a|s)} Q_{\theta_{\text{old}}}(s,a)\right]\Big|_{\theta=\theta_{\text{old}}}$$

$\boldsymbol{F}$ 表示策略之间平均 KL 散度的 Hessian 矩阵，即 θ_{old} 的 Fisher 信息矩阵

$$\boldsymbol{F} = H\left[\mathbb{E}_{s\sim\rho_{\theta_{\text{old}}}}[D_{\text{KL}}(\pi_{\theta_{\text{old}}}(\cdot \mid s) \| \pi_\theta(\cdot \mid s))]\right]$$

式12转换为：

$$\begin{aligned} \theta_{new} &= \arg\max_{\theta} g^T(\theta - \theta_{old}) \\ &\text{subject to } \frac{1}{2}(\theta - \theta_{old})^T \boldsymbol{F}(\theta - \theta_{old}) \le \delta \end{aligned}$$

利用 KKT 条件可以得到迭代公式：

$$\theta_{new} = \theta_{old} + \sqrt{\frac{2\delta}{g^T \boldsymbol{F}^{-1} g}} \boldsymbol{F}^{-1} g$$

2.5 Proximal Policy Optimization(PPO)

PPO 算法同样由 John Schulman 提出，和 TRPO 不同的是：TRPO 对目标函数采取一阶近似，约束条件采取二阶近似，然而 PPO 采用了一系列的一阶方法（Clip）。在效果相近的同时，PPO 在算法上更加简单。

定义 $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$，易得 $r(\theta_{\text{old}}) = 1$。TRPO 的目标函数可以写为：

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right]$$

当没有约束时，该目标函数会使得策略被过大更新。因此需要对 $r_t(\theta)$ 过于偏离 1 的情况进行惩罚。

作者提出的方法是将 $r_t(\theta)$ 限制在 $[1-\epsilon, 1+\epsilon]$ 范围内，其中 ϵ 为超参数，表示限制范围。PPO 的目标函数写为：

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \tag{13}$$

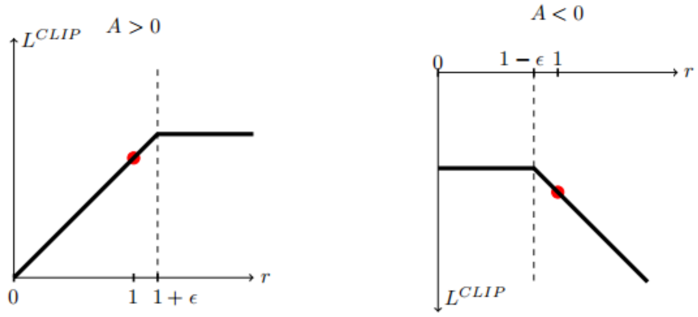

图 2.1: L^{CLIP}

2.6 PPO 算法

作者将 PPO 的目标函数定义为：

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right]$$

其中 c_1, c_2 为超参数，L_t^{VF} 为值函数的损失，S 为熵，用来促进探索。

在 T 个时间步中运行策略（其中 T 远远小于幕的长度），并利用收集到的样本进行更新。这种方法需要一个不超出时间步 T 的优势估计器：

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$$

在此基础上进行拓展，可以使用广义优势估计的截断版本：

$$\begin{aligned}
&\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \\
&\text{where} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)
\end{aligned}$$

使用固定长度轨迹段的 PPO 算法如下所示。每次迭代，N 个 Actor 中的每个 Actor 都会收集 T 个时间步的数据。然后，在这些 NT 个时间步的数据上构建损失，并在 K 幕内使用小批量 SGD 对其进行优化。

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

3 部分代码

实验总使用 Actor-Critic 架构，Actor 和 Critic 的网络设置如下代码所示：

```
class ActorCriticDiscrete(nn.Module):
    def __init__(self, input_size, output_size, hidden_size):
        super(ActorCriticDiscrete, self).__init__()

        # 定义actor网络的架构
        self.action_layer = nn.Sequential(
                nn.Linear(input_size, hidden_size), # 输入层
                nn.ReLU(),                        # 激活函数
                nn.Linear(hidden_size, output_size), # 输出层
                nn.Softmax(dim=-1)               # 对输出进行softmax激活
                )

        # 定义critic网络的架构
        self.value_layer = nn.Sequential(
               nn.Linear(input_size, hidden_size), # 输入层
               nn.ReLU(),                        # 激活函数
               nn.Linear(hidden_size, 1)       # 输出层
               )

    def act(self, state, memory):
        # 将状态转换为PyTorch张量
        state = torch.from_numpy(state).float()
        # 使用actor网络计算动作概率
        action_probs = self.action_layer(state)
        # 从概率分布中采样一个动作
        dist = Categorical(action_probs)
        action = dist.sample()

        # 将状态、动作和对数概率存储在内存中
        memory.states.append(state)
```

```
            memory.actions.append(action)
            memory.logprobs.append(dist.log_prob(action))

            return action.item()

        def evaluate(self, state, action):
            # 使用actor网络计算动作概率
            action_probs = self.action_layer(state)
            # 创建一个Categorical分布
            dist = Categorical(action_probs)

            # 计算所选动作的对数概率
            action_logprobs = dist.log_prob(action)
            # 计算动作分布的熵
            dist_entropy = dist.entropy()

            # 使用critic网络计算状态值
            state_value = self.value_layer(state)

            return action_logprobs, torch.squeeze(state_value), dist_entropy
```

PPO 算法部分的代码如下代码所示：

```
class PPOAgent:
    def __init__(self, input_size, output_size, hidden_size, lr, eps, gamma,
        K_epochs, eps_clip, update_timestep):
        # 使用超参数和内存初始化PPOAgent
        self.lr = lr
        self.gamma = gamma
        self.eps_clip = eps_clip
        self.K_epochs = K_epochs
        self.timestep = 0
        self.memory = Memory()
        self.update_timestep = update_timestep

        # 初始化actor-critic网络和优化器
        self.policy = ActorCriticDiscrete(input_size, output_size, hidden_size)
        self.optimizer = torch.optim.Adam(self.policy.parameters(), lr=lr, eps=eps)
        self.policy_old = ActorCriticDiscrete(input_size, output_size, hidden_size)
        self.policy_old.load_state_dict(self.policy.state_dict())

        # 定义均方误差损失函数
        self.MseLoss = nn.MSELoss()
```

```

    def update(self):
        # 使用蒙特卡洛方法估算状态回报
        rewards = []
        discounted_reward = 0
        for reward, is_terminal in zip(reversed(self.memory.rewards),
            reversed(self.memory.is_terminals)):
            if is_terminal:
                discounted_reward = 0
            discounted_reward = reward + (self.gamma * discounted_reward)
            rewards.insert(0, discounted_reward)

        # 对回报进行标准化
        rewards = torch.tensor(rewards, dtype=torch.float32)
        rewards = (rewards - rewards.mean()) / (rewards.std() + 1e-5)

        # 将列表转换为张量
        old_states = torch.stack(self.memory.states).detach()
        old_actions = torch.stack(self.memory.actions).detach()
        old_logprobs = torch.stack(self.memory.logprobs).detach()

        # 优化策略K次
        for _ in range(self.K_epochs):
            # 使用actor-critic网络评估旧动作和值
            logprobs, state_values, dist_entropy = self.policy.evaluate(old_states,
                old_actions)

            # 计算重要性采样系数 (pi_theta / pi_theta__old)
            ratios = torch.exp(logprobs - old_logprobs.detach())

            # 计算替代损失
            advantages = rewards - state_values.detach()
            surr1 = ratios * advantages
            surr2 = torch.clamp(ratios, 1 - self.eps_clip, 1 + self.eps_clip) *
                advantages
            loss = -torch.min(surr1, surr2) + 0.5 * self.MseLoss(state_values,
                rewards) - 0.01 * dist_entropy

            # 执行梯度下降
            self.optimizer.zero_grad()
            loss.mean().backward()
            self.optimizer.step()
```

```python

        # 将新权重复制到旧策略中
        self.policy_old.load_state_dict(self.policy.state_dict())

    def step(self, reward, done):
        self.timestep += 1
        # 存储奖励和终止标志
        self.memory.rewards.append(reward)
        self.memory.is_terminals.append(done)

        # 更新策略
        if self.timestep % self.update_timestep == 0:
            self.update()
            # 更新后清空内存
            self.memory.clear_memory()
            self.timstamp = 0

    def act(self, state):
        # 使用旧策略选择动作
        return self.policy_old.act(state, self.memory)

    def save_checkpoint(self, directory, episode):
        # 保存当前模型检查点
        if not os.path.exists(directory):
            os.makedirs(directory)
        filename = os.path.join(directory, 'checkpoint_{}.pth'.format(batch))
        torch.save(self.policy.state_dict(), f=filename)
        print('保存当前模型至 \'{}\''.format(filename))

    def load_checkpoint(self, directory, filename):
        # 加载模型检查点以便重新训练
        self.policy.load_state_dict(torch.load(os.path.join(directory, filename)))
        print('重新开始训练 checkpoint \'{}\'.'.format(filename))
        return int(filename[11:-4])
```

4 实验结果

实验中使用 Gym 库中的 Lunar Lander 环境，设定超参数如下表所示：

表 1: 超参数设置

超参数	设定值
学习率 lr	10^{-4}
折扣因子 γ	0.99
K	4
clip 因子 ϵ	0.2
batch size	10
迭代次数	2000

图4.2是使用 PPO 和策略梯度方法训练的曲线。结果表明 PPO 方法的训练速度更快，最终的奖励值高于策略梯度方法。但是训练过程中，PPO 曲线的震荡幅度大于策略梯度方法。这一现象的原因是式13中使用了重要性采样的方法，来更新策略中的 θ。重要性采样会引入较大的方差，这就会导致在训练过程中出现较大的震荡。而策略梯度的训练并未使用重要性采样，因此方差较小，训练过程更平稳。

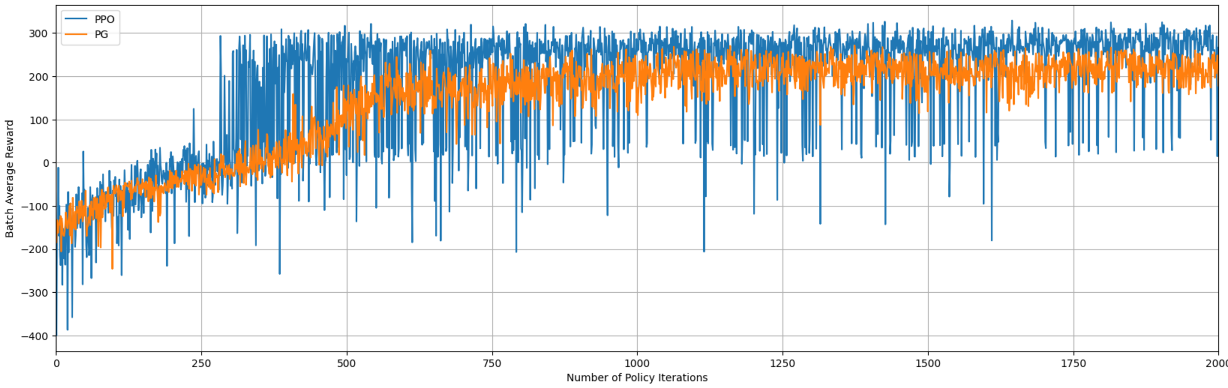

图 4.2: PPO 和策略梯度的迭代曲线

为减少训练过程的震荡幅度，实验中尝试增大 batch size。通过增大 batch size，采样更多的样本，从而得到进行更准确的更新。图4.3为 $batchsize = 10, 32, 64$ 的训练曲线。结果表明增大 batch size 确实可以减小曲线的震荡幅度，使更新更加平稳。但是当 batch size 过大时，会使得训练速度过慢。因此需要选择合适的 batch size 大小。

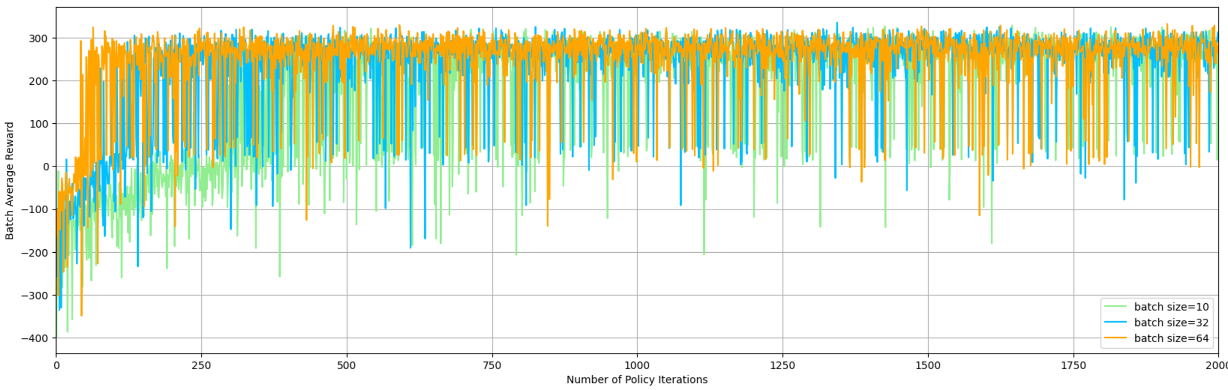

图 4.3: 不同 batch size 的迭代曲线

利用训练得到的模型进行测试，图4.4展示了一次测试结果，测试视频见附件 test.mp4。测试结果

说明，根据该模型着陆器能够准确平稳地在着陆点附近着陆。

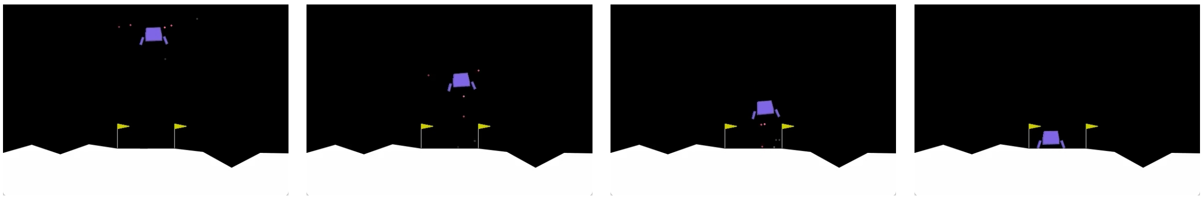

图 4.4: 一次测试的结果

5 实验总结

通过本次实验，我学习了 PPO 方法，利用 PPO 方法和策略梯度方法解决了 Gym 库中的 Lunar Lander 问题，并比较了 PPO 和策略梯度的区别。

PPO 是在一般策略梯度方法基础上提出的改进型算法。策略梯度方法存在核心问题在于采样的偏差。如果策略一次更新太远，那么下一次采样将完全偏离，导致策略更新到完全偏离的位置，从而形成恶性循环。PPO 的核心思想就是对当前 policy 和旧 policy 的偏差做裁剪。如果新旧策略的偏差超过 ϵ 就进行裁剪。利用裁剪，就可以控制更新的幅度。

实验中在 Lunar Lander 环境上对 PPO 方法进行了测试。测试结果表明，经过 2000 次的迭代，PPO 方法可以成功解决 Lunar Lander 问题。