南開大學
Nankai University

人工智能学院

强化学习实验报告

实验十　DDPG

姓名：石若川

学号：2111381

专业：智能科学与技术

2024 年 5 月 27 日

1 实验目的

- 学习 DDPG 强化学习算法，比较 DDPG 算法和 PG 算法、PPO 算法的区别。
- 利用 DDPG 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 Deterministic Policy Gradient(DPG)

David Silver 等人与 2014 年提出了 DPG 算法，将连续动作空间中的确定性策略梯度算法用于强化学习。一般的策略梯度算法基本思想中利用参数化的概率 $\pi_\theta(a|s) = \mathbb{P}[a|s;\theta]$ 来表示随机策略。而 DPG 中，作者使用确定性的策略 $a = \mu_\theta(s)$。随机策略在实际使用中需要更大的样本，需要更多的计算资源，而确定性策略能够避免这一问题。然而确定性的策略只采样一个动作，难以保证探索。为了保证算法进行探索，作者引入了 off-policy 的算法，使用随机策略选作为行为策略，使用确定性策略作为目标策略，算法使用 actor-critic 架构。

2.1.1 随机策略

对于带折扣的 MDP 问题，作者对折扣状态分布进行如下定义：

$$\int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} p_1(s) p(s \to s', t, \pi) \mathrm{d}s$$

因此可以将目标函数写为：

$$\begin{aligned} J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \pi_\theta(s,a) r(s,a) \mathrm{d}a\mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}\left[r(s,a)\right] \end{aligned} \tag{1}$$

随机策略的梯度可以写为

$$\begin{aligned} \nabla_\theta J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) \mathrm{d}a\mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a)\right] \end{aligned} \tag{2}$$

Actor-Critic 是广泛应用的一种基于策略梯度架构。式2中的 $Q^\pi(s,a)$ 为未知的动作值函数，Critic 中使用 $Q^w(s,a)$ 去近似 $Q^\pi(s,a)$。当满足以下条件时，$Q^w(s,a)$ 为无偏估计：

- $Q^w(s,a) = \nabla_\theta \log \pi_\theta(a \mid s)^T w$
- w 使得 $\varepsilon^2(w) = E_{s\sim\rho^\pi, a\sim\pi_\theta}\left[\left(Q^{\mathrm{w}}(s,a) - Q^{\pi}(s,a)\right)^2\right]$ 最小

所以，式2可以写为：

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^w(s,a)\right] \tag{3}$$

对于 off-policy 的 Actor-Critic 框架而言，由行为策略 $\beta(a|s) \neq \pi_\theta(a|s)$ 采样动作，目标函数写为：

$$\begin{aligned} J_\beta(\pi_\theta) &= \int_{\mathcal{S}} \rho^\beta(s) V^\pi(s) \mathrm{d}s \\ &= \int_{\mathcal{S}} \int_{\mathcal{A}} \rho^\beta(s) \pi_\theta(a|s) Q^\pi(s,a) \mathrm{d}a\mathrm{d}s \end{aligned}$$

目标函数的梯度写为：

$$\nabla_\theta J_\beta(\pi_\theta) \approx \int_{\mathcal{S}} \int_{\mathcal{A}} \rho^\beta(s) \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) \mathrm{d}a\mathrm{d}s \tag{4}$$

$$= \mathbb{E}_{s\sim\rho^\beta, a\sim\beta} \left[\frac{\pi_\theta(a|s)}{\beta_\theta(a|s)} \nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a) \right] \tag{5}$$

2.1.2 确定性策略

考虑确定性策略 $\mu_\theta : \mathcal{S} \to \mathcal{A}$，目标函数为：

$$\begin{aligned} J(\mu_\theta) &= \int_{\mathcal{S}} \rho^\mu(s) r(s, \mu_\theta(s)) \mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho^\mu} \left[r(s, \mu_\theta(s)) \right] \end{aligned} \tag{6}$$

梯度为：

$$\begin{aligned} \nabla_\theta J(\mu_\theta) &= \int_{\mathcal{S}} \rho^\mu(s) \nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)} \mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho^\mu} \left[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)} \right] \end{aligned} \tag{7}$$

对于确定性策略的 Actor-Critic 架构，利用 $\mu_\theta(s)$ 替换随机策略中的 $\pi(s,a)$，目标函数写为：

$$\begin{aligned} J_\beta(\mu_\theta) &= \int_{\mathcal{S}} \rho^\beta(s) V^\mu(s) \mathrm{d}s \\ &= \int_{\mathcal{S}} \rho^\beta(s) Q^\mu(s, \mu_\theta(s)) \mathrm{d}s \end{aligned} \tag{8}$$

梯度为：

$$\begin{aligned} \nabla_\theta J_\beta(\mu_\theta) &\approx \int_{\mathcal{S}} \rho^\beta(s) \nabla_\theta \mu_\theta(a|s) Q^\mu(s,a) \mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho^\beta} \left[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)} \right] \end{aligned} \tag{9}$$

策略的更新过程如下，其中 Critic 使用 Q-learning 的方法进行更新。

$$\delta_t = r_t + \gamma Q^w(s_{t+1}, \mu_\theta(s_{t+1})) - Q^w(s_t, a_t) \tag{10}$$

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w Q^w(s_t, a_t) \tag{11}$$

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \mu_\theta(s_t) \nabla_a Q^w(s_t, a_t)|_{a=\mu_\theta(s)} \tag{12}$$

一般的 off-policy 型 Actor-Critic 算法均需要进行重要性采样，但是由于确定性策略去掉了对于动作的积分，所以在 Actor 中不需要进行重要性采样。

2.1.3 相容函数逼近

作者提出了在不影响确定性策略梯度下，使得 $\nabla_a Q^\mu(s,a)$ 可以被 $\nabla_a Q^w(s,a)$ 替代的相容条件。当满足以下条件时，对于确定性策略 $\mu_\theta(s)$，$\nabla_\theta J_\beta(\theta) = \mathbb{E}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q^w(s,a)|_{a=\mu_\theta(s)}\right]$

1. $\nabla_a Q^w(s,a)|_{a=\mu_\theta(s)} = \nabla_\theta \mu_\theta(s)^\top w$
2. w 使得均方误差 $MSE(\theta, w) = \mathbb{E}\left[\epsilon(s;\theta,w)^T \epsilon(s;\theta,w)\right]$ 最小，其中 $\epsilon(s;\theta,w) = \nabla_a Q^w(s,a)|_{a=\mu_\theta(s)} - \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}$

2.2 Deep Deterministic Policy Gradient(DDPG)

DPG 所提出的相容函数条件过于苛刻，在实际使用中很难满足。DDPG 提出了一种结合 DQN 和 DPG 的算法。

DPG 使用确定性策略选择动作，Critic 利用 Q-learning 形式的贝尔曼方程更新行为值函数 $Q(s,a)$，Actor 的利用以下公式进行更新：

$$\begin{aligned}\nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_{\theta^\mu} Q(s,a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}\right] \\ &= \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_a Q(s,a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_{s=s_t}\right]\end{aligned}$$

DQN 利用神经网络拟合行为值函数，解决具有连续状态空间和动作空间的强化学习问题。在 DQN 出现之前，利用神经网络拟合行为值函数进行训练时会出现不稳定的情况。这是因为训练神经网络时，需要满足样本是独立同分布的假设。然而，采样的样本轨迹会依赖前一时刻的状态-动作，因此并不满足独立同分布。DQN 使用经验回放机制来训练神经网络。它将代理与环境的交互经验存储在一个经验池中，然后随机抽样这些经验进行训练。这种随机抽样有助于打破数据之间的相关性，并使得训练更加稳定。另外，DQN 引入了独立的目标网络，它与在线网络具有相同的架构，但是参数更新频率较低。这种延迟更新的策略有助于减小 TD 学习中的偏差，提高算法的稳定性。

DDPG 结合了 DQN 和 DPG 的思想，将 DPG 中的行为值函数利用神经网络表示。DDPG 同样使用经验回放，将采样的 (s_t, a_t, r_t, s_{t+1}) 存入回放缓冲区中，每个时间步 Actor 和 Critic 从经验缓冲区中均匀采样一个 minibatch 进行更新。不同于 DQN 通过直接复制权重更新网络参数，DDPG 利用滤波的方法更新网络参数 θ：

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \quad \tau \ll 1$$

这是一种软更新的方式，可以极大地提高学习的稳定性。

由于观测空间中不同物理量的单位不同，所以神经网络难以找到合适的超参数进行有效学习。因此，DDPG 引入了批归一化的方法，使得样本具有相同的均值和方差。借助批归一化，网络可以在具有不同单位的任务重学习，无需手动调整单位在一定范围之内

最后，为促进探索，DDPG 对 Actor 策略添加了 Ornstein-Uhlenbeck 噪音过程 $\mathcal{N}$，构造出一个探索策略 μ'：

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

DDPG 的伪代码如下所示：

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
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 Actor(nn.Module):
    def __init__(self, input_size, output_size, hidden_size):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return torch.tanh(self.fc3(x))

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

```
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size + output_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        xs = F.relu(self.fc1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)

def hidden_init(layer):
    fan_in = layer.weight.data.size()[0]
    lim = 1. / np.sqrt(fan_in)
    return (-lim, lim)
```

DDPG 算法部分的代码如下代码所示：

```
class DDPGAgent:
    def __init__(self, input_size, action_size, hidden_size, lr_actor, lr_critic,
        gamma, tau, buffer_size, batch_size):
        self.gamma = gamma
        self.tau = tau
        self.memory = ReplayBuffer(buffer_size, batch_size)

        self.actor_local = Actor(input_size, action_size, hidden_size).to(device)
        self.actor_target = Actor(input_size, action_size, hidden_size).to(device)
        self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=lr_actor)

        self.critic_local = Critic(input_size, action_size, hidden_size).to(device)
        self.critic_target = Critic(input_size, action_size, hidden_size).to(device)
        self.critic_optimizer = optim.Adam(self.critic_local.parameters(),
            lr=lr_critic, weight_decay=0.0001)

        self.noise = OUNoise(action_size)

        self.update_targets(1.0) # 硬更新目标网络
```

```
18
19      def act(self, state, add_noise=True):
20          state = torch.from_numpy(state).float().to(device)
21          self.actor_local.eval()
22          with torch.no_grad():
23              action = self.actor_local(state).cpu().data.numpy()
24          self.actor_local.train()
25          if add_noise:
26              action += self.noise.sample()
27          return np.clip(action, -action_bound, action_bound)
28
29      def step(self, state, action, reward, next_state, done):
30          self.memory.add(state, action, reward, next_state, done)
31
32          if len(self.memory) > self.memory.batch_size:
33              experiences = self.memory.sample()
34              self.learn(experiences)
35
36      def learn(self, experiences):
37          states, actions, rewards, next_states, dones = experiences
38
39          # 更新Critic
40          actions_next = self.actor_target(next_states)
41          Q_targets_next = self.critic_target(next_states, actions_next)
42          Q_targets = rewards + (self.gamma * Q_targets_next * (1 - dones))
43          Q_expected = self.critic_local(states, actions)
44          critic_loss = F.mse_loss(Q_expected, Q_targets)
45          self.critic_optimizer.zero_grad()
46          critic_loss.backward()
47          self.critic_optimizer.step()
48
49          # 更新Actor
50          actions_pred = self.actor_local(states)
51          actor_loss = -self.critic_local(states, actions_pred).mean()
52          self.actor_optimizer.zero_grad()
53          actor_loss.backward()
54          self.actor_optimizer.step()
55
56          # 软更新目标网络
57          self.update_targets(self.tau)
58
59      def update_targets(self, tau):
```

```
60          for target_param, local_param in zip(self.actor_target.parameters(),
                self.actor_local.parameters()):
61              target_param.data.copy_(tau * local_param.data + (1.0 - tau) *
                    target_param.data)
62          for target_param, local_param in zip(self.critic_target.parameters(),
                self.critic_local.parameters()):
63              target_param.data.copy_(tau * local_param.data + (1.0 - tau) *
                    target_param.data)
64
65      def save_checkpoint(self, directory, episode):
66          if not os.path.exists(directory):
67              os.makedirs(directory)
68          filename = os.path.join(directory, 'checkpoint_{}.pth'.format(episode))
69          torch.save({
70              'actor_state_dict': self.actor_local.state_dict(),
71              'critic_state_dict': self.critic_local.state_dict()
72          }, filename)
73          print('保存当前模型至 \'{}\''.format(filename))
74
75      def load_checkpoint(self, directory, filename):
76          checkpoint = torch.load(os.path.join(directory, filename))
77          self.actor_local.load_state_dict(checkpoint['actor_state_dict'])
78          self.critic_local.load_state_dict(checkpoint['critic_state_dict'])
79          print('重新开始训练 checkpoint \'{}\'.'.format(filename))
80          return int(filename.split('_')[1].split('.')[0])
81
82  class OUNoise:
83      def __init__(self, size, mu=0., theta=0.15, sigma=0.2):
84          self.mu = mu * np.ones(size)
85          self.theta = theta
86          self.sigma = sigma
87          self.size = size
88          self.reset()
89
90      def reset(self):
91          self.state = np.ones(self.size) * self.mu
92
93      def sample(self):
94          x = self.state
95          dx = self.theta * (self.mu - x) + self.sigma * np.random.randn(len(x))
96          self.state = x + dx
97          return self.state
```

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
Actor 学习率 lr	10^{-4}
Critic 学习率	10^{-3}
折扣因子 γ	0.99
软更新参数 τ	10^{-3}
clip 因子 ϵ	0.2
batch size	128
迭代次数	200

图4.1展示了 DDPG、PPO 和 PG 方法在 LunarLanderContinuous 环境下的训练迭代曲线。结果表明，DDPG 方法的奖励可以在 25 轮的迭代后上升至 250 左右，可以很快地寻找到较优的策略。而 PPO 方法的奖励上升较缓慢，经过 200 轮的迭代只能达到 0 左右。PG 方法对于连续环境的解决能力远不如 DDPG 和 PPO 方法，甚至会出现奖励下降的情况。

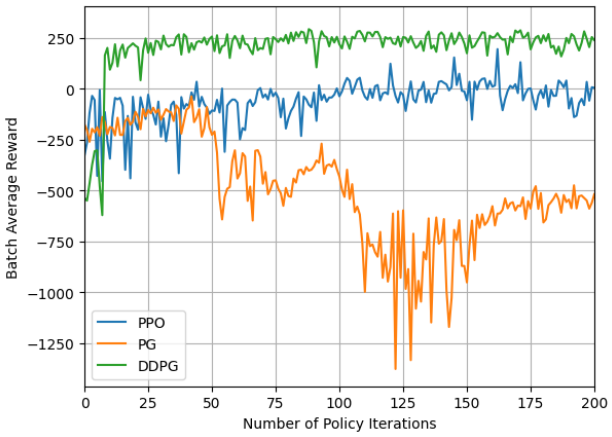

图 4.1: DDPG、PPO 和 PG 的迭代曲线

利用训练得到的模型进行测试，图4.2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。

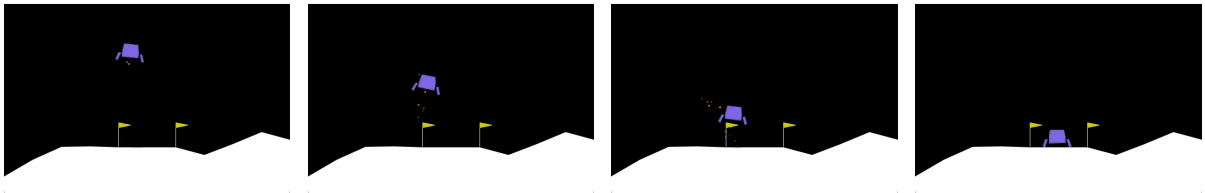

图 4.2: 一次测试的结果

5 实验总结

通过本次实验，我学习了 DDPG 方法的原理，利用 DDPG、PPO 和 PG 方法解决了 Gym 库中的 LunarLanderContinuous 的问题，并比较了 DDPG、PPO 和 PG 的区别。

DDPG 是一种使用确定性策略的策略梯度下降方法。确定性策略直接输出一个具体的动作，而不是动作的概率分布。这种方法在高维连续动作空间中尤为有效。这是因为策略的输出动作空间可能非常大，概率分布方法可能需要很长时间才能收敛到一个有效的策略，而使用确定性策略只对状态空间积分，计算简单。DDPG 给出了确定性策略梯度理论和确定性 Actor-Critic 算法。

实验中在 LunarLanderContinuous 环境上对 DDPG 方法进行了测试。测试结果表明，经过 200 次的迭代，DDPG 方法可以成功解决 LunarLanderContinuous 问题。