

DDPG 算法实战

同之前章节一样，本书在实战中将演示一些核心的代码，完整的代码请参考 JoyRL 代码仓库。

算法流程

如图 1 所示，DDPG 算法的训练方式其实更像 DQN 算法。注意在第 15 步中 DDPG 算法将当前网络参数复制到目标网络的方式是软更新，即每次一点点地将参数复制到目标网络中，与之对应的是 DQN 算法中的硬更新。软更新的好处是更加平滑缓慢，可以避免因权重更新过于迅速而导致的震荡，同时降低训练发散的风险。

DDPG 算法

- 1: 初始化 critic 网络 $Q(s, a | \theta^Q)$ 和 actor 网络 $\mu(s | \theta^\mu)$ 的参数 θ^Q 和 θ^μ
- 2: 初始化对应的目标网络参数，即 $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
- 3: 初始化经验回放 D
- 4: **for** 回合数 $= 1, M$ **do**
- 5: **交互采样：**
- 6: 选择动作 $a_t = \mu(s_t | \theta^\mu) + \mathcal{N}_t$, \mathcal{N}_t 为探索噪声
- 7: 环境根据 a_t 反馈奖励 s_t 和下一个状态 s_{t+1}
- 8: 存储样本 (s_t, a_t, r_t, s_{t+1}) 到经验回放 D 中
- 9: 更新环境状态 $s_{t+1} \leftarrow s_t$
- 10: **策略更新：**
- 11: 从 D 中取出一个随机批量的 (s_i, a_i, r_i, s_{i+1})
- 12: 求得 $y_i = r_i + \gamma Q' \left(s_{i+1}, \mu' \left(s_{i+1} | \theta^{\mu'} \right) | \theta^{Q'} \right)$
- 13: 更新 critic 参数，其损失为： $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i | \theta^Q))^2$
- 14: 更新 actor 参数： $\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a | \theta^Q) \Big|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s | \theta^\mu) \Big|_{s_i}$
- 15: 软更新目标网络： $\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$, $\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$
- 16: **end for**

图 1: DDPG 算法流程

定义模型

如代码 1 所示，DDPG 算法的模型结构跟 Actor-Critic 算法几乎是一样的，只是由于 DDPG 算法的 Critic 是 Q 函数，因此也需要将动作作为输入。除了模型之外，目标网络和经验回放的定义方式跟 DQN 算法一样，这里不做展开。

代码 1: 实现 DDPG 算法的 \$text{Actor}\$ 和 \$text{Critic}\$

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim = 256, init_w=3e-3):
        super(Actor, self).__init__()
        self.linear1 = nn.Linear(state_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, hidden_dim)
        self.linear3 = nn.Linear(hidden_dim, action_dim)
```

```

        self.linear3.weight.data.uniform_(-init_w, init_w)
        self.linear3.bias.data.uniform_(-init_w, init_w)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = torch.tanh(self.linear3(x)) # 输入0到1之间的值
        return x

class Critic(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=256, init_w=3e-3):
        super(Critic, self).__init__()

        self.linear1 = nn.Linear(state_dim + action_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, hidden_dim)
        self.linear3 = nn.Linear(hidden_dim, 1)
        # 随机初始化为较小的值
        self.linear3.weight.data.uniform_(-init_w, init_w)
        self.linear3.bias.data.uniform_(-init_w, init_w)

    def forward(self, state, action):
        # 按维数1拼接
        x = torch.cat([state, action], 1)
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = self.linear3(x)
        return x

```

动作采样

由于 DDPG 算法输出的是确定性策略，因此不需要像其他策略梯度算法那样，通过借助高斯分布来采样动作的概率分布，直接输出 Actor 的值即可，如代码 2 所示。

代码 2: DDPG 算法的动作采样

```

class Agent:
    def __init__(self):
        pass
    def sample_action(self, state):
        state = torch.FloatTensor(state).unsqueeze(0).to(self.device)
        action = self.actor(state)
        return action.detach().cpu().numpy()[0, 0]

```

策略更新

如代码 3 所示，DDPG 算法的策略更新则更像 Actor-Critic 算法。

代码 3: DDPG 算法的策略更新

```

class Agent:
    def __init__(self):

```

```

    pass
def update(self):
    # 从经验回放中中随机采样一个批量的样本
    state, action, reward, next_state, done = self.memory.sample(self.batch_size)
    actor_loss = self.critic(state, self.actor(state))
    actor_loss = - actor_loss.mean()

    next_action = self.target_actor(next_state)
    target_value = self.target_critic(next_state, next_action.detach())
    expected_value = reward + (1.0 - done) * self.gamma * target_value
    expected_value = torch.clamp(expected_value, -np.inf, np.inf)

    actual_value = self.critic(state, action)
    critic_loss = nn.MSELoss()(actual_value, expected_value.detach())

    self.actor_optimizer.zero_grad()
    actor_loss.backward()
    self.actor_optimizer.step()
    self.critic_optimizer.zero_grad()
    critic_loss.backward()
    self.critic_optimizer.step()

    # 各自目标网络的参数软更新
    for target_param, param in zip(self.target_critic.parameters(),
self.critic.parameters()):
        target_param.data.copy_(
            target_param.data * (1.0 - self.tau) +
            param.data * self.tau
        )
    for target_param, param in zip(self.target_actor.parameters(),
self.actor.parameters()):
        target_param.data.copy_(
            target_param.data * (1.0 - self.tau) +
            param.data * self.tau
        )

```

核心代码到这里全部实现了，我们展示一下训练效果，如图 2 所示。

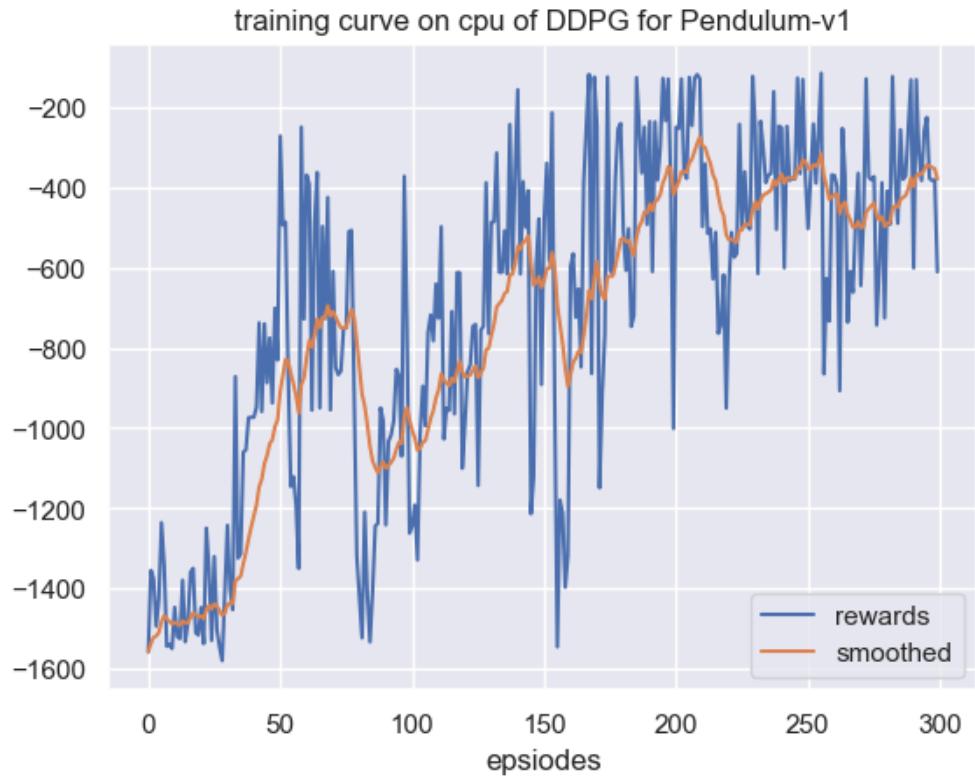


图 2: Pendulum 环境 DDPG 算法训练曲线

这里我们使用了一个具有连续动作空间的环境 Pendulum，如图 3 所示。在该环境中，钟摆以随机位置开始，我们的目标是将其向上摆动，使其保持直立。

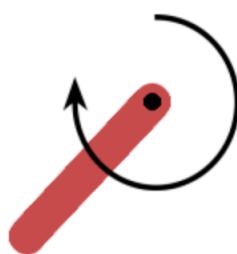


图 3: Pendulum 环境演示