Actor-Critic算法

简介：

一个同时基于价值函数和策略梯度的算法。上一章讲述了策略梯度最典型的算法——**REINFORCE**，但该算法会存在一定的问题：

- 基于片段式数据的任务

这就需要把整个片段全部遍历完之后，REINFORCE才能直接计算累计折扣奖励，所以任务具有终止态，否则数据是采不完的

- 低数据利用效率

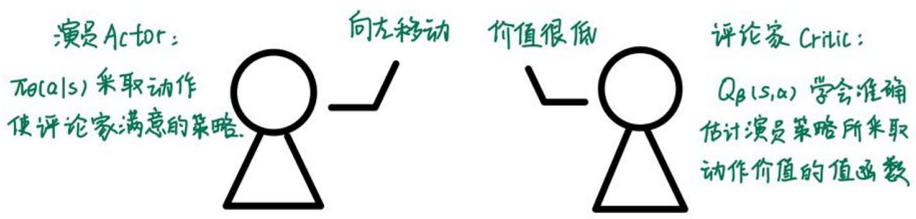
实际中，REINFORCE需要大量的训练数据，并且该数据是必须运行完才能计算累计回报

- 高训练方差（最重要的缺陷）

首先片段中间的过程是高度不确定的，所以单个或多个片段中采样到的值函数具有很高的方差

所以我们思考有没有一种办法直接去估计Q(s,a)，而不是使用G_t来估算Q。所以可以去建立一个可训练的值函数Q_β来实现这个过程，（β为Q函数的参数化建模的参数，是可训练的）

而创建了这个函数，Actor-Critic的架构就体现出了，该算法框架就如它的名字一般，演员是在台上去做动作。每个状态采取对应的动作，而评论家就是专门用来评估这个策略好还是不好，所以当评论家估计的比较准确时，演员就会只在乎评论家反馈的结果，而不在意真实世界的奖励是多少。而评论家就去收集真实世界的反馈，来得到一套自我比较好的评价体系，使得对Q_β能够估计的越准确越好



所以Actor-Critic就是一个双模型的训练框架，评论家去收集演员产生的数据然后去估计Q函数，演员执行使评论家满意的动作

算法流程：

Actor-Critic 分为两个部分：Actor（策略网络）和 Critic（价值网络）

- Actor 要做的是与环境交互，并在 Critic 价值函数的指导下用策略梯度学习一个更好的策略。
- Critic 要做的是通过 Actor 与环境交互收集的数据学习一个价值函数，这个价值函数会用于判断在当前状态什么动作是好的，什么动作不是好的，进而帮助 Actor 进行策略更新。

Actor 的更新采用策略梯度的原则，Critic 更新利用时序差分的方法，与DQN一样对值函数进行建模训练，然后使用梯度下降方法来更新 Critic 价值网络参数即可

- 初始化策略网络参数θ，价值网络参数ω
- **for** 序列 $e = 1 \to E$ **do**：
- 用当前策略π_θ采样轨迹$\{s_1, a_1, r_1, s_2, a_2, r_2, \ldots\}$
- 为每一步数据计算: $\delta_t = r_t + \gamma V_\omega(s_{t+1}) - V_\omega(s_t)$
- 更新价值参数$w = w + \alpha_\omega \sum_t \delta_t \nabla_\omega V_\omega(s_t)$
- 更新策略参数$\theta = \theta + \alpha_\theta \sum_t \delta_t \nabla_\theta \log \pi_\theta(a_t|s_t)$
- **end for**

代码实践：

首先定义策略网络`PolicyNet`（与 REINFORCE 算法一样）。

```
class PolicyNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim, action_dim):
        super(PolicyNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return F.softmax(self.fc2(x), dim=1)
```

Actor-Critic 算法中额外引入一个价值网络，接下来的代码定义价值网络`ValueNet`，其输入是某个状态，输出则是状态的价值。

```
class ValueNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim):
        super(ValueNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, 1)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

义`ActorCritic`算法，主要包含采取动作（`take_action()`）和更新网络参数（`update()`）两个函数

```
class ActorCritic:
    def __init__(self, state_dim, hidden_dim, action_dim, actor_lr,
critic_lr,
                 gamma, device):
        # 策略网络
        self.actor = PolicyNet(state_dim, hidden_dim, action_dim).to(device)
        self.critic = ValueNet(state_dim, hidden_dim).to(device)  # 价值网络
        # 策略网络优化器
        self.actor_optimizer = torch.optim.Adam(self.actor.parameters(),
                                                lr=actor_lr)
        self.critic_optimizer = torch.optim.Adam(self.critic.parameters(),
```

```
                                                lr=critic_lr)  # 价值网络优化器
        self.gamma = gamma
        self.device = device

    def take_action(self, state):
        state = torch.tensor([state], dtype=torch.float).to(self.device)
        probs = self.actor(state)
        action_dist = torch.distributions.Categorical(probs)
        action = action_dist.sample()
        return action.item()

    def update(self, transition_dict):
        states = torch.tensor(transition_dict['states'],
                              dtype=torch.float).to(self.device)
        actions = torch.tensor(transition_dict['actions']).view(-1, 1).to(
            self.device)
        rewards = torch.tensor(transition_dict['rewards'],
                               dtype=torch.float).view(-1, 1).to(self.device)
        next_states = torch.tensor(transition_dict['next_states'],
                                   dtype=torch.float).to(self.device)
        dones = torch.tensor(transition_dict['dones'],
                             dtype=torch.float).view(-1, 1).to(self.device)

        # 时序差分目标
        td_target = rewards + self.gamma * self.critic(next_states) * (1 - dones)
        td_delta = td_target - self.critic(states)  # 时序差分误差
        log_probs = torch.log(self.actor(states).gather(1, actions))
        actor_loss = torch.mean(-log_probs * td_delta.detach())
        # 均方误差损失函数
        critic_loss = torch.mean(
            F.mse_loss(self.critic(states), td_target.detach()))
        self.actor_optimizer.zero_grad()
        self.critic_optimizer.zero_grad()
        actor_loss.backward()  # 计算策略网络的梯度
        critic_loss.backward()  # 计算价值网络的梯度
        self.actor_optimizer.step()  # 更新策略网络的参数
        self.critic_optimizer.step()  # 更新价值网络的参数
```