A2C 算法实战

定义模型

通常来讲，Critic 的输入是状态，输出则是一个维度的价值，而 Actor 输入的也会状态，但输出的是概率分布，因此我们可以定义两个网络，如代码 1 所示。

代码 1: 实现 Actor 和 Critic

```
class Critic(nn.Module):
    def __init__(self,state_dim):
        self.fc1 = nn.Linear(state_dim, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 1)
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        value = self.fc3(x)
        return value

class Actor(nn.Module):
    def __init__(self, state_dim, action_dim):
        self.fc1 = nn.Linear(state_dim, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, action_dim)
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        logits_p = F.softmax(self.fc3(x), dim=1)
        return logits_p
```

这里由于是离散的动作空间，根据在策略梯度章节中设计的策略函数，我们使用了 softmax 函数来输出概率分布。另外，实践上来看，由于 Actor 和 Critic 的输入是一样的，因此我们可以将两个网络合并成一个网络，以便于加速训练。这有点类似于 Duelling DQN 算法中的做法，如代码 2 所示。

代码 2: 实现合并的 Actor 和 Critic

```
class ActorCritic(nn.Module):
    def __init__(self, state_dim, action_dim):
        self.fc1 = nn.Linear(state_dim, 256)
        self.fc2 = nn.Linear(256, 256)
        self.action_layer = nn.Linear(256, action_dim)
        self.value_layer = nn.Linear(256, 1)
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        logits_p = F.softmax(self.action_layer(x), dim=1)
        value = self.value_layer(x)
        return logits_p, value
```

注意当我们使用分开的网络时，我们需要在训练时分别更新两个网络的参数，即需要两个优化，而使用合并的网络时则只需要更新一个网络的参数即可。

动作采样

与 DQN 算法不同等确定性策略不同，A2C 的动作输出不再是 Q 值最大对应的动作，而是从概率分布中采样动作，这意味着即使是很小的概率，也有可能被采样到，这样就能保证探索性，如代码 3 所示。

代码 3: 采样动作

```
from torch.distributions import Categorical
class Agent:
    def __init__(self):
        self.model = ActorCritic(state_dim, action_dim)
    def sample_action(self,state):
        '''动作采样函数
        '''
        state = torch.tensor(state, device=self.device, dtype=torch.float32)
        logits_p, value = self.model(state)
        dist = Categorical(logits_p)
        action = dist.sample()
        return action
```

注意这里直接利用了 `PyTorch` 中的 `Categorical` 分布函数，这样就能直接从概率分布中采样动作了。

策略更新

我们首先需要计算出优势函数，一般先计算出回报，然后减去网络输出的值即可，如代码 4 所示。

代码清 4: 计算优势函数

```
class Agent:
    def _compute_returns(self, rewards, dones):
        returns = []
        discounted_sum = 0
```

```
        for reward, done in zip(reversed(rewards), reversed(dones)):
            if done:
                discounted_sum = 0
            discounted_sum = reward + (self.gamma * discounted_sum)
            returns.insert(0, discounted_sum)
        # 归一化
        returns = torch.tensor(returns, device=self.device,
dtype=torch.float32).unsqueeze(dim=1)
        returns = (returns - returns.mean()) / (returns.std() + 1e-5) # 1e-5 to avoid
division by zero
        return returns
    def compute_advantage(self):
        '''计算优势函数
        '''
        logits_p, states, rewards, dones = self.memory.sample()
        returns = self._compute_returns(rewards, dones)
        states = torch.tensor(states, device=self.device, dtype=torch.float32)
        logits_p, values = self.model(states)
        advantages = returns - values
        return advantages
```

这里我们使用了一个技巧，即将回报归一化，这样可以让优势函数的值域在 $[-1, 1]$ 之间，这样可以让优势函数更稳定，从而减少方差。计算优势之后就可以分别计算 Actor 和 Critic 的损失函数了，如代码 5 所示。

代码 5: 计算损失函数

```
class Agent:
    def compute_loss(self):
        '''计算损失函数
        '''
        logits_p, states, rewards, dones = self.memory.sample()
        returns = self._compute_returns(rewards, dones)
        states = torch.tensor(states, device=self.device, dtype=torch.float32)
        logits_p, values = self.model(states)
        advantages = returns - values
        dist = Categorical(logits_p)
        log_probs = dist.log_prob(actions)
        # 注意这里策略损失反向传播时不需要优化优势函数，因此需要将其 detach 掉
        actor_loss = -(log_probs * advantages.detach()).mean()
        critic_loss = advantages.pow(2).mean()
        return actor_loss, critic_loss
```

到这里，我们就实现了 A2C 算法的所有核心代码，完整代码请读者参考本书的代码仓库。最后展示一下训练的效果，如图 1 所示。

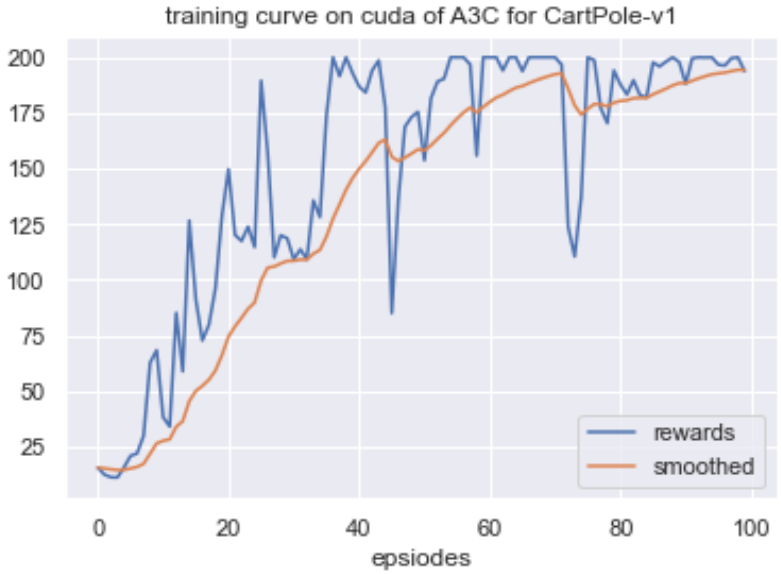

图 1: CartPole 环境 A2C 算法训练曲线