策略梯度算法

基于策略的强化学习

 Q-learning、DQN 及 DQN 改进算法都是基于价值的强化学习，其中 Q-learning 是处理有限状态的算法，而 DQN 可以用来解决连续状态的问题。之前所有讨论求梯度，都是对价值函数求梯度，例如V(s)或Q(s,a)，用参数化建模$V_\theta(s)$或$Q_\theta(s,a)$去逼近$V^\pi(s)$或 $Q^\pi(s,a)$

所以我们可以从另一个角度去思考，策略本身也可以参数化表示，这就是在强化学习中，除了基于值函数的方法的另一支非常经典的方法，那就是**基于策略**的方法。

我们可以将策略参数化——$\boldsymbol{\pi_\theta(a|s)}$，用来表示在某个状态下的动作分布

如果策略是确定性策略（在一个状态下只有一个确定的动作）那么毫无疑问，$a = \pi_\theta(a|s)$，通过这种方式直接映射动作a；如果是随机性策略，则$\pi_\theta(a|s) = P(a|s;\theta)$用参数化的形式表示一个分布，通过构建策略函数我们希望将可见的已知状态泛化到未知的状态上

对比两者，基于值函数的方法主要是学习值函数，然后根据值函数导出一个策略，学习过程中并不存在一个显式的策略；而基于策略的方法则是直接显式地学习一个目标策略。基于策略的强化学习与基于价值的强化学习对比有以下优缺点：

优点：

- 具有更好的收敛性质和学习的稳定性

因为策略的参数化学习，它在策略空间中的学习是一个近乎连续的学习，它没有非常陡峭的跳变，而如果在Q-learning中学习，改变其中某个值的话，最后得到的策略就会是一个很明显的跳变策略，但基于策略的强化学习在策略空间是平缓的

- 在高维度或连续动作空间中更有效

对于在高纬度或连续动作空间的值函数学习是很困难的，通常连续的动作空间的每一个动作都应该去求对应的价值函数然后并取最大值，学习效率并不高

- 能够学习出随机策略

缺点：

- 通常会收敛到局部最优而非全局最优

对于这个问题，价值函数一般也不能避免

- 评估一个策略通常不够高效并具有较大的方差

因为评估一个策略，就是一个很难的问题，所以评估的方法使用不当加上不够高效的策略，评估时的偏差相对来说很高

策略梯度是基于策略的方法的基础，下面介绍策略梯度算法

策略梯度

策略梯度是策略学习中最典型的算法，对于一个随机策略，我们用参数化的方式表示：$\boldsymbol{\pi_\theta(a|s) = P(a|s;\theta)}$

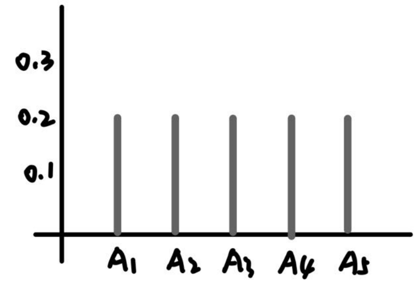
对于参数化的建模，直觉上我们应该：

- 降低带来较低价值/奖励的动作出现的概率
- 提高带来较高价值/奖励的动作出现的概率

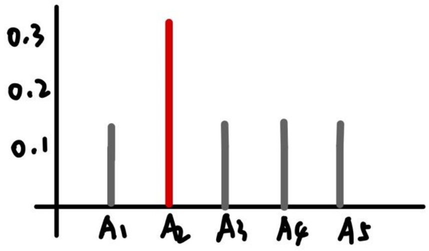
举个例子：

以一个状态的离散动作空间维度为5举例（PPT上）

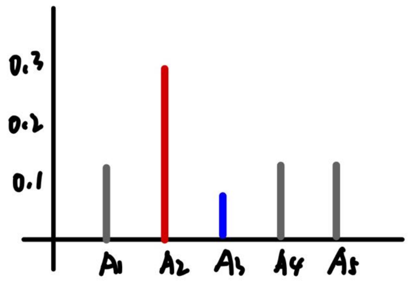
①首先初始化参数θ，一般初始时对于每个动作以等概率的方式去采样



②当采样采取到了动作A_2，将A_2作用于环境中，观察到了正奖励，所以在以后遇到这个状态时，我们更希望采样到A_2这个动作，所以我们希望去更新θ使得我们的策略在以后碰到这个状态之后更到的概率去采取动作A_2



③采取动作A_3观察到得到了夫奖励，所以我们希望调整θ，来降低采取动作A_3的概率



所以我们应该明白我们的输出应该如何改变，以至于我们可以用梯度回传的方式改变我们策略里面的参数θ

单步MDP的策略梯度：

现在来计算一下策略梯度，考虑一个最简单的MDP，首先我们对于一个MDP某个状态s，该状态是由一个distribution（分布）采样出的（d(s)），在一步决策后采取动作a，这个MDP结束，它可以获得奖励为r_{sa}，而由于单步决策，所以这个r_{sa}就是对应的价值函数（动作价值函数），因为它只有一步

这个决策用策略π_θ表示，那么该策略的价值期望为， 这也是我们策略学习的目标函数记作J(θ)

$$J(\theta)=E_{\pi_\theta}[r]=\sum_{s\in S}d(s)\sum_{a\in A}\pi_\theta(a|s)r_{sa}$$

毫无疑问，我们希望训练的θ使得策略的价值期望更大，越来越高，对于θ而言我们对其求梯度有：

$$\frac{\partial J(\theta)}{\partial \theta} = \sum_{s \in S} d(s) \sum_{a \in A} \left(\frac{\partial \pi_\theta(a|s)}{\partial \theta} \right) \cdot r_{sa}$$

对θ求导，所以穿过∑求和符号

对于这种分布的参数求导的方法，在数学上有一个技巧叫做**似然比**（likelihood ratio）

$$\frac{\partial \pi_\theta(a|s)}{\partial \theta} = \pi_\theta(a|s) \frac{1}{\pi_\theta(a|s)} \cdot \frac{\partial \pi_\theta(a|s)}{\partial \theta}$$

$$\approx \pi_\theta(a|s) \frac{\partial \log \pi_\theta(a|s)}{\partial \theta}$$

利用似然比，则对J(θ)求导可写作：

$$\frac{\partial J(\theta)}{\partial \theta} = \sum_{s \in S} d(s) \sum_{a \in A} \frac{\partial \pi_\theta(a|s)}{\partial \theta} \cdot r_{sa}$$

$$= \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(a|s) \cdot \frac{\partial \log \pi_\theta(a|s)}{\partial \theta} \cdot r_{sa}$$

$$= E_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \cdot r_{sa} \right]$$

通过这种方式可以有效的去求**J(θ)**对**θ**的梯度值，虽然最后的结果仍然是期望expectation，但我们可以通过用大量的经验采样（如d(s)的状态s，和从π_θ中采样的动作a）来计算logπ_θ的梯度以及r_{sa}，二者相乘之后再求平均值，就可以近似的估算出**J(θ)**对**θ**的梯度值

❓ logπ_θ的梯度如何利用具体数据计算存疑

策略梯度定理：

上节讲的是一步MDP的策略梯度的推导，如果是长期的、多步的MDP时，我们希望用长期的价值函数Q^{π_θ}(s,a)来代替之前的瞬时奖励r_{sa}

而非常巧合的一点在于，有大佬已经证明：将r_{sa}换成Q^{π_θ}(s,a)，就可以去做整个关于价值函数的策略梯度。具体的证明并不是很严格，但是很清晰，这里也就不在展开啦，知道又这回事就行

所以r_{sa}是一步的MDP，而对于多步的MDP我们就可以拿Q^{π_θ}(s,a)去指导我们当前的策略，去做相应的更新，使得我们的**J(θ)**能够提升

策略梯度定理涉及：起始状态目标函数 J_1，平均奖励目标函数 J_{avR}，和平均价值目标函数 J_{avV}

策略梯度定理：对任意可微的策略 $\pi_\theta(a|s)$，任意策略的目标函数 $J = J_1, J_{avR}, J_{avV}$，其策略梯度是：

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\pi_\theta}\left[\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} Q^{\pi_\theta}(s,a)\right]$$

蒙特卡罗策略梯度：

但是我们需要去计算这个$\mathbf{Q^{\pi\theta}(s,a)}$，而对于$Q^{\pi\theta}(s,a)$的计算在强化学习中有很多方法，我们可以利用蒙特卡罗的方法来计算Q函数，通过一个片段的累计奖励值G_t来作为对应$Q^{\pi\theta}(s,a)$的无偏采样

$$\Delta\theta_t = \alpha \frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta} G_t$$

这个方法叫做蒙特卡罗策略梯度（简称REINFORCE，之所以全大写是用来特指蒙特卡罗的策略梯度），使用蒙特卡罗的方法去计算G_t，利用G_t来估算Q，然后代入去更新**J(θ)**对**θ**的梯度，其中α为学习率，二者相乘就是步长

简单复习一下蒙特卡罗如何估计Q的：

在一个片段中，蒙特卡洛估计方法会在该状态每一次出现都计算它的回报，并记录该状态出现的次数和回报，并进行累加求出总次数和总汇报，然后求平均值作为该状态的价值

所以蒙特卡洛策略梯度算法很简单：

算法流程：

REINFORCE算法

initialize θ arbitrarily

通过当前策略采样片段

for each episode $\{s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do

根据片段利用蒙特卡罗方法可以计算每个状态相应的价值

  for $t = 1$ to $T-1$ do

    $\theta \leftarrow \theta + \alpha \boxed{\frac{\partial}{\partial \theta} \log \pi_\theta(a_t|s_t) G_t}$

学习率

$\times$ $\frac{\partial J(\theta)}{\partial \theta}$ = 步长

  end for

end for

return θ

该算法是典型的在线策略算法，因为它需要通过当前学好的策略去进行采样经验片段，然后通过片段去计算G_t，然后再进行梯度计算，进行参数迭代

其次该算法也不需要知道环境的模型如何，只需要采样数据来进行学习，然后根据直觉来增大或减少概率

Softmax随机策略：

那么最后一个环节，$\boldsymbol{\pi_\theta(a|s)}$该如何去设计搭建，一般我们的动作是离散的状态空间，所以一种方法就是利用softmax回归来设计策略，即softmax策略

$$\pi_\theta(a|s) = \frac{e^{f_\theta(s,a)}}{\sum_{a'} e^{f_\theta(s,a')}}$$

然后将打分函数指数化,使之变为正数

- 式中，$f_\theta(s,a)$是用θ参数化的状态-动作对得分函数，可以预先定义

然后除以打分之和,用这个占比来表示π_θ,其中π_θ中的θ便为$f_\theta(s,a)$中的θ

该方法在机器学习中使用的特别特别多,所以我们可以放心的去使用

其对数似然的梯度是

通过打分函数可将对策略的梯度,转换为该函数的梯度公式

$$\begin{aligned}\frac{\partial \log \pi_\theta(a|s)}{\partial\theta} &= \frac{\partial f_\theta(s,a)}{\partial\theta} - \frac{1}{\sum_{a'} e^{f_\theta(s,a')}} \sum_{a''} e^{f_\theta(s,a'')} \frac{\partial f_\theta(s,a'')}{\partial\theta} \\ &= \frac{\partial f_\theta(s,a)}{\partial\theta} - \mathbb{E}_{a'\sim\pi_\theta(a'|s)}\left[\frac{\partial f_\theta(s,a')}{\partial\theta}\right]\end{aligned}$$

当$f_\theta(s,a)$已知,通过采样就可以计算,

通过该方式，就可以对策略梯度进行具体的求解，以线性得分函数为例，其策略梯度为：

$$f_\theta(s,a) = \theta^{\mathrm{T}} x(s,a)$$

$$\begin{aligned}\frac{\partial \log \pi_\theta(a|s)}{\partial\theta} &= \frac{\partial f_\theta(s,a)}{\partial\theta} - \mathbb{E}_{a'\sim\pi_\theta(a'|s)}\left[\frac{\partial f_\theta(s,a')}{\partial\theta}\right] \\ &= x(s,a) - \mathbb{E}_{a'\sim\pi_\theta(a'|s)}[x(s,a')]\end{aligned}$$

REINFORCE实践：

首先定义策略网络`PolicyNet`，其输入是某个状态，输出则是该状态下的动作概率分布，这里采用在离散动作空间上的`softmax()`函数来实现一个可学习的**多项分布**（multinomial distribution）。

```
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)
```

再定义我们的 REINFORCE 算法。在函数`take_action()`函数中，我们通过动作概率分布对离散的动作进行采样。在更新过程中，我们按照算法将损失函数写为策略回报的负数，这样求最小值即我们所需策略回报的最大值，对θ求导后就可以通过梯度下降来更新策略

```
class REINFORCE:
    def __init__(self, state_dim, hidden_dim, action_dim, learning_rate,
gamma,
                 device):
        self.policy_net = PolicyNet(state_dim, hidden_dim,
                                    action_dim).to(device)
        self.optimizer = torch.optim.Adam(self.policy_net.parameters(),
```

```python
                                          lr=learning_rate)  # 使用Adam优化器
        self.gamma = gamma  # 折扣因子
        self.device = device

    def take_action(self, state):  # 根据动作概率分布随机采样
        state = torch.tensor([state], dtype=torch.float).to(self.device)
        probs = self.policy_net(state)
        action_dist = torch.distributions.Categorical(probs)
        action = action_dist.sample()
        return action.item()

    def update(self, transition_dict):
        reward_list = transition_dict['rewards']
        state_list = transition_dict['states']
        action_list = transition_dict['actions']

        G = 0
        self.optimizer.zero_grad()
        for i in reversed(range(len(reward_list))):  # 从最后一步算起
            reward = reward_list[i]
            state = torch.tensor([state_list[i]],
                                 dtype=torch.float).to(self.device)
            action = torch.tensor([action_list[i]]).view(-1, 1).to(self.device)
            log_prob = torch.log(self.policy_net(state).gather(1, action))
            G = self.gamma * G + reward
            loss = -log_prob * G  # 每一步的损失函数
            loss.backward()  # 反向传播计算梯度
        self.optimizer.step()  # 梯度下降
```