TPRO算法

策略梯度的缺点

TPRO全称算法为信任区域策略优化，它是深度强化学习当中，深度策略方法种非常具有代表性的算法，在引入该算法前先来探讨一个问题。

策略梯度的更新是存在一定问题的，尤其当我们$\pi_\theta(a|s)$如果是一个神经网络构建的策略的话，就会遇到很多问题，训练不稳定等，其实这些问题主要来自利用REINFORCE利用似然比更新的时候，计算得到的梯度的陡度会非常大，以至于在进行迭代时，当前策略的改变也非常大。

如果改变非常大的话，由于是深度神经网络，就类似一步就直接走到了悬崖下这种感觉，一步梯度更新绕过了很多最小值，结果就导致变化量非常大，而这种变化量非常大往往是灾难性的，因为可能会错过很多最优值，使我们的策略变得很差。

当策略变得很差，所以在新一轮更新时，又会和环境交互得到的数据，而这些数据离最优的优化策略很远，所以在进行优化时又会变得很差，这样形成了一个恶性的循环！

所以针对这个问题，我们考虑在更新时找到一块**信任区域**（trust region），在这个区域上更新策略时能够得到某种策略性能的安全性保证，这就是**信任区域策略优化**（trust region policy optimization，TRPO）算法的主要思想。

策略优化目标

策略的优化目标是策略的价值期望，$J(\theta) = E\pi_\theta[r]$，训练θ使得$E\pi_\theta[r]$更大，越来越高

策略优化目标$J(\theta)$有两种等价形式

第一种为：

$$J(\theta) = E_{\tau\sim p_\theta(\tau)}\left[\sum_t \gamma^t \cdot r(s_t, a_t)\right]$$

对于一个策略采样得到一个序列 τ 基于这个序列去计算累计奖励的期望来作为策略价值的期望

另一种形式为：

$$J(\theta) = E_{s_0\sim p_\theta(s_0)}\left[V^{\pi_\theta}(s_0)\right]$$

直接通过评估初始状态的价值期望来作为策略的价值期望，因为初始状态存在一定分布，并且这种分布往往与环境本身有关，初始状态只和环境相关和策略没有什么关系，但故且暂时用$P_\theta(s_0)$去表示，但与θ是无关的，通过环境采样反馈出的初始状态s_0我们从初始状态下follow我们的策略，然后会得到一个或多个序列（策略可能为确定策略或随机策略），通过这些序列来计算得到的状态价值期望$E[V(s_0)]$来衡量这个策略的价值

因为策略就是从s_0开始的选择一系列将要执行的动作和奖励

按照定义，这两种形式都是策略与环境交互的价值期望，是一回事，所以这两个值是完全恒等的

有了这个概念，我们可以推导优化目标的优化量，即当θ发生改变，$\theta \rightarrow \theta'$，$J(\theta') - J(\theta)$的差值是怎样的

优化目标差值：

首先介绍公式的相关定义，以及部分等式的转换，防止公式推导时忘记什么意思：

$$V^{\pi_\theta}(s) = E_{a\sim\pi_\theta(s)}\left[Q^{\pi_\theta}(s,a)\right]$$

$$= E_{a\sim\pi_\theta(s)}\left[E_{\tau\sim p_\theta(\tau)}\left[\sum_{s_k=s,a_k=a}^{\infty}\sum_{t=k}^{\infty}\gamma^{t-k}\cdot r(s_t,a_t)\right]\right]$$

- τ：轨迹
- s_0：初始状态
- s_t， a_t，$r(s_t,a_t)$： t 时刻的状态，动作和奖励
- π_θ：使用的策略
- θ：表示策略所使用的参数
- Q^{π_θ} 和 V^{π_θ}：策略 π_θ 下的 Q 值与状态值函数

其实应该也不会忘记，不过长时间不看可能忘记，毕竟符号太多了

下面就是对优化目标差值的推导：

$$J(\theta')-J(\theta) = J(\theta') - \mathbb{E}_{s_0\sim p(s_0)}[V^{\pi_\theta}(s_0)]$$
$$= J(\theta') - \mathbb{E}_{\tau\sim p_{\theta'}(\tau)}[V^{\pi_\theta}(s_0)]$$

初始状态的分布与θ无关

$$= J(\theta') - \mathbb{E}_{\tau\sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty}\gamma^t V^{\pi_\theta}(s_t) - \sum_{t=1}^{\infty}\gamma^t V^{\pi_\theta}(s_t)\right]$$

$\because s_0\sim p(s_0)$与$\tau\sim p_{\theta'}(\tau)$等价

$$= J(\theta') + \mathbb{E}_{\tau\sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty}\gamma^t(\gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t))\right]$$

$J(\theta')$的定义

$$= \mathbb{E}_{\tau\sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t)\right] + \mathbb{E}_{\tau\sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty}\gamma^t(\gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t))\right]$$

$$= \mathbb{E}_{\tau\sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty}\gamma^t(r(s_t,a_t) + \gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t))\right] = A^{\pi_\theta}(s_t,a_t)$$

$$= \mathbb{E}_{\tau\sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi_\theta}(s_t,a_t)\right]$$

$Q^{\pi_\theta}(s_t,a_t)$

$\because s_t,a_t$来自$\tau\sim p_{\theta'}(\tau)$

$A^{\pi_\theta}(s_t,a_t)$
$= Q^{\pi_\theta}(s_t,a_t) - V^{\pi_\theta}(s_t)$

不方便采样

$\pi_{\theta'}$未与环境充分交互，得不到大量的τ

但对于这个期望的序列 τ 是新策略**θ'**，而新策略**$\pi_{\theta'}$** 还没有与环境充分交互，得不大很多的经验片段以及数据，以至于我们的积分是拿不到的，所以我们需要对其进行重要性采样

$$J(\theta') - J(\theta) = E_{\tau\sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi_\theta}(s_t,a_t)\right]$$

最后计算分为两部分，首先采样到了状态s_t，为某一时刻t的状态。基于当前s_t，我们follow新策略**$\pi_{\theta'}$**去采样得到了对应的动作a_t，这样我们就可以得到一个状态动作对<st,at>，然后通过这个状态follow老的策略**π_θ**，去计算：

$$\tau^t A^{\pi\theta}(s_t, a_t)$$

在这个计算过程中上面简单的分析是可行的，但实际情况中，新策略$\pi_{\theta'}$只能得到一步状态动作对<st,at>，但我们需要的是多个序列 τ 去求均值（期望），而刚刚的新策略所得只是一个序列中的一个结点，想得到大量的这种结点是不可行的，因为新策略还没有与环境进行充分交互，没有那么多的经验数据，大量的序列 τ 是拿不到的，但我们可以老策略π_θ已经很成熟了，并且与环境已经进行了充分的交互，所以有大量的经验数据序列，所以一个想法，我们希望用老π_θ采样的数据去代替，通过重要性采样这个方法就可以实现

重要性采样：

前面已经解释过重要性采样，但是之前的重要性采样好像只是数理统计方面的知识，和强化学习的重要性采样还是有些出入，这里再重新补一下

总之长话短说，就是把重要性采样的基本作用概括一下

现在我们有两个分布，一个分布A，一个分布B，而我们只有另一个分布A的数据，已知分布A的数据所以我们直接可以通过数据来计算A分布的期望，但是分布B也想计算期望的话又没有数据怎么办呢？这就用到了重要性采样，而这个例子可以举一反三对号入座😼，这个分布A就是老策略π_θ，而分布B就是新策略$\pi_{\theta'}$，我们通过老策略的数据来对新策略进行计算

数学上的本质原理：

重要性采样：

x 服从A分布中采样出的 $f(x)$ 期望，记作 $E_{x\sim A}[f(x)]$

$$E_{x\sim A}[f(x)] = \int_x A(x) f(x)\, dx$$

$$= \int_x B(x) \frac{A(x)}{B(x)} \cdot f(x)\, dx$$

$$= E_{x\sim B}\left[\frac{A(x)}{B(x)} \cdot f(x)\right]$$

于是得到 x 服从 B 分布中采样出的 $\frac{A(x)}{B(x)} \cdot f(x)$ 期望，

引入 $\beta(x) = \frac{A(x)}{B(x)}$ 相当于一个权重分配，使得 $\beta(x)$ 相当于每个 x 有对应的权重，就可以记作 x 是从 $q(x)$ 中采样出的

所以，通过重要性采样，我们可以将优化目标的差值记作：

$$A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)$$

$$\begin{aligned} & J(\theta') - J(\theta) \\ &= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)} \left[\sum_{t=0}^{\infty} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \\ &= \sum_t \mathbb{E}_{s_t \sim p_{\theta'}(s_t)} [\mathbb{E}_{a_t \sim \pi_{\theta'}(a_t|s_t)} [\gamma^t A^{\pi_\theta}(s_t, a_t)]] \\ &= \sum_t \mathbb{E}_{s_t \sim p_{\theta'}(s_t)} [\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} [\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t)]] \end{aligned}$$

仍然是 $p_{\theta'}$ (近似操作)

重要性采样

但实际上通过重要性采样这里实际上是 p_θ，而不是 $p_{\theta'}$，近似操作

虽然我们通过了老策略采样的数据代替了新策略的采样数据，但这种方法仍然是近似操作，但我们尽可能的希望：

$$p_\theta(s) \simeq p_{\theta'}(s)$$

而满足这种情况时，说明我们新老策略更新前后的变化是比较小的，所以当θ → θ' 更新时，得到的新策略，就会变得平滑从而训练效果就会比较稳定，于是就避免了**策略梯度的缺点**：*梯度的陡度会非常大，以至于在进行迭代时，当前策略的改变也非常大*

约束策略变化：

所以我们需要尽可能的减小两个策略的差异，来约束这个变化，使我们更新时变化幅度较小，从而使我们的新策略$\pi_{\theta'}$变得不会太大！我们可以通过**KL散度**的方式来约束衡量这个差异

使用KL散度来约束策略更新的幅度，使新策略和旧策略的KL散度小于等于一个具体的值，这个值是我们规定的，一般是一个较小的值，即：

$$\mathbb{E}_{s_t \sim p(s_t)} [D_{KL}(\pi_{\theta'}(a_t|s_t) \| \pi_\theta(a_t|s_t))] \leq \epsilon$$

然后就是我看不懂的具体对θ' 是如何更新的，反正通过KL散度然后使用共轭梯度的方法，还涉及一个黑塞矩阵H，总之好多我看不懂的名词，不过反正PPT上说的就是更多的使用违反约束作为惩罚来更新θ' 🐬

constraint violate as penalty

$$\begin{aligned} \theta' \leftarrow \arg\max_{\theta'} & \sum_t \mathbb{E}_{s_t \sim p_\theta(s_t)} [\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} [\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t)]] \\ & -\lambda(D_{KL}(\pi_{\theta'}(a_t|s_t) \| \pi_\theta(a_t|s_t)) - \epsilon) \end{aligned}$$

1. 优化上式，更新θ'
2. 更新 $\lambda \leftarrow \lambda + \alpha(D_{KL}(\pi_{\theta'}(a_t|s_t) \| \pi_\theta(a_t|s_t)) - \epsilon)$

这个意思就是将KL的惩罚项加入到更新公式，如果当KL散度大于规定的值，那么就在更新中的惩罚项就是正数，如上图更新所示，减去一个正数就会使我们更新的参数θ' 变小，从而达到了约束效果，其中λ因子类似于学习率，来控制惩罚力度

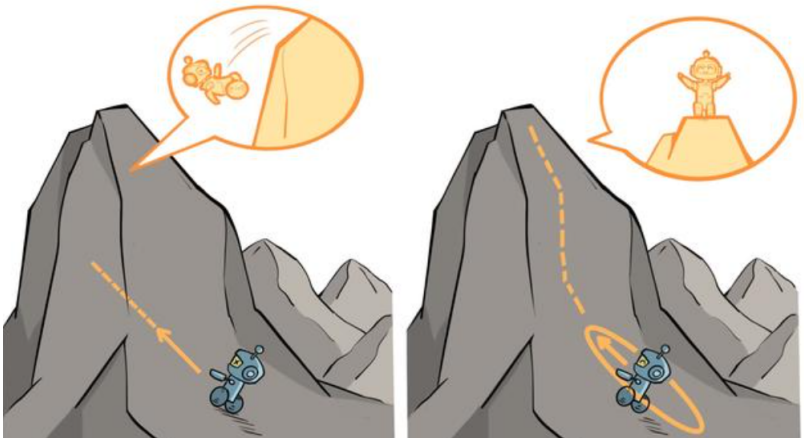
这个λ就类似于书上的α，每次迭代更新时需要搜索，选择满足要求的数

TRPO：

概念性解释：

ok，有了前置知识和为什么引入TPRO算法的前因后果，那么终于可以概括的说一说TPRO算法的概念了

我们用神经网络构造了一个策略，该策略通过策略梯度的方法进行更新，但策略梯度更新时是不稳定的，它可能迈的步子太大甚至跌落悬崖，导致性能暴跌。所以我们通过KL散度给当前策略划定了一个圈，一个范围，在这个圈是值得信任的，即Trust region，在这个可信任的区域更新参数是比较稳定的。通过这个可信任区域对策略进行优化即TRPO算法（**trust region policy optimization，TRPO**



左图表示当完全不设置信任区域时，策略的梯度更新可能导致策略的性能骤降；右图表示当设置了信任区域时，可以保证每次策略的梯度更新都能来带性能的提升

算法流程：

- 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$\{s_1, a_1, r_1, s_2, a_2, r_2, \dots\}$
- 根据收集到的数据和价值网络估计每个状态动作对的优势$A(s_t, a_t)$
- 计算策略目标函数的梯度g
- 用共轭梯度法计算$x = H^{-1}g$
- 用线性搜索找到一个i值，并更新策略网络参数$\theta_{k+1} = \theta_k + \alpha^i \sqrt{\frac{2\delta}{x^T H x}} x$，其中$i \in \{1, 2, \dots, K\}$为能提升策略并满足 KL 距离限制的最小整数
- 更新价值网络参数（与 Actor-Critic 中的更新方法相同）
- **end for**

其中共轭梯度计算的**x**表示参数θ更新的方向，α^i 相当于惩罚项中的λ，使用的算法原型仍然是**A2C**，只是在该算法基础上策略梯度的更新划分了一个可信任区域**TR**，来控制策略梯度的更新幅度

代码实践：

```
class TRPO:
    """ TRPO算法 """
    def __init__(self, hidden_dim, state_space, action_space, lmbda,
                 kl_constraint, alpha, critic_lr, gamma, device):
        state_dim = state_space.shape[0]
        action_dim = action_space.n
        # 策略网络参数不需要优化器更新
        self.actor = PolicyNet(state_dim, hidden_dim,
action_dim).to(device)
        self.critic = ValueNet(state_dim, hidden_dim).to(device)
        self.critic_optimizer = torch.optim.Adam(self.critic.parameters(),
                                                 lr=critic_lr)
        self.gamma = gamma
        self.lmbda = lmbda  # GAE参数
        self.kl_constraint = kl_constraint  # KL距离最大限制
        self.alpha = alpha  # 线性搜索参数
        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 hessian_matrix_vector_product(self, states, old_action_dists,
vector):
        # 计算黑塞矩阵和一个向量的乘积
        new_action_dists =
torch.distributions.Categorical(self.actor(states))
        kl = torch.mean(
            torch.distributions.kl.kl_divergence(old_action_dists,
                                                 new_action_dists))  # 计算
平均KL距离
        kl_grad = torch.autograd.grad(kl,
                                      self.actor.parameters(),
                                      create_graph=True)
        kl_grad_vector = torch.cat([grad.view(-1) for grad in kl_grad])
        # KL距离的梯度先和向量进行点积运算
        kl_grad_vector_product = torch.dot(kl_grad_vector, vector)
        grad2 = torch.autograd.grad(kl_grad_vector_product,
                                    self.actor.parameters())
        grad2_vector = torch.cat([grad.view(-1) for grad in grad2])
        return grad2_vector

    def conjugate_gradient(self, grad, states, old_action_dists):  # 共轭梯
度法求解方程
        x = torch.zeros_like(grad)
        r = grad.clone()
        p = grad.clone()
        rdotr = torch.dot(r, r)
        for i in range(10):  # 共轭梯度主循环
```

```
            Hp = self.hessian_matrix_vector_product(states,
old_action_dists,
                                                    p)
            alpha = rdotr / torch.dot(p, Hp)
            x += alpha * p
            r -= alpha * Hp
            new_rdotr = torch.dot(r, r)
            if new_rdotr < 1e-10:
                break
            beta = new_rdotr / rdotr
            p = r + beta * p
            rdotr = new_rdotr
        return x

    def compute_surrogate_obj(self, states, actions, advantage,
old_log_probs,
                              actor):  # 计算策略目标
        log_probs = torch.log(actor(states).gather(1, actions))
        ratio = torch.exp(log_probs - old_log_probs)
        return torch.mean(ratio * advantage)

    def line_search(self, states, actions, advantage, old_log_probs,
                    old_action_dists, max_vec):  # 线性搜索
        old_para = torch.nn.utils.convert_parameters.parameters_to_vector(
            self.actor.parameters())
        old_obj = self.compute_surrogate_obj(states, actions, advantage,
                                             old_log_probs, self.actor)
        for i in range(15):  # 线性搜索主循环
            coef = self.alpha**i
            new_para = old_para + coef * max_vec
            new_actor = copy.deepcopy(self.actor)
            torch.nn.utils.convert_parameters.vector_to_parameters(
                new_para, new_actor.parameters())
            new_action_dists = torch.distributions.Categorical(
                new_actor(states))
            kl_div = torch.mean(
                torch.distributions.kl.kl_divergence(old_action_dists,
                                                     new_action_dists))
            new_obj = self.compute_surrogate_obj(states, actions,
advantage,
                                                 old_log_probs, new_actor)
            if new_obj > old_obj and kl_div < self.kl_constraint:
                return new_para
        return old_para

    def policy_learn(self, states, actions, old_action_dists,
old_log_probs,
                     advantage):  # 更新策略函数
        surrogate_obj = self.compute_surrogate_obj(states, actions,
advantage,
                                                   old_log_probs,
self.actor)
        grads = torch.autograd.grad(surrogate_obj, self.actor.parameters())
        obj_grad = torch.cat([grad.view(-1) for grad in grads]).detach()
        # 用共轭梯度法计算x = H^(-1)g
```

```
        descent_direction = self.conjugate_gradient(obj_grad, states,
                                                    old_action_dists)

        Hd = self.hessian_matrix_vector_product(states, old_action_dists,
                                                descent_direction)
        max_coef = torch.sqrt(2 * self.kl_constraint /
                              (torch.dot(descent_direction, Hd) + 1e-8))
        new_para = self.line_search(states, actions, advantage,
old_log_probs,
                                    old_action_dists,
                                    descent_direction * max_coef)  # 线性搜
索
        torch.nn.utils.convert_parameters.vector_to_parameters(
            new_para, self.actor.parameters())  # 用线性搜索后的参数更新策略

    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)
        td_target = rewards + self.gamma * self.critic(next_states) * (1 -

dones)
        td_delta = td_target - self.critic(states)
        advantage = compute_advantage(self.gamma, self.lmbda,
                                      td_delta.cpu()).to(self.device)
        old_log_probs = torch.log(self.actor(states).gather(1,

 actions)).detach()
        old_action_dists = torch.distributions.Categorical(
            self.actor(states).detach())
        critic_loss = torch.mean(
            F.mse_loss(self.critic(states), td_target.detach()))
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()  # 更新价值函数
        # 更新策略函数
        self.policy_learn(states, actions, old_action_dists, old_log_probs,
                          advantage)
```