PPO算法：

回顾TRPO

我们用老策略π_θ采样的数据来训练新策略$\pi_{\theta'}$，所以新策略$\pi_{\theta'}$距离不应该与老策略距离太远，如果差距过大，那么老策略采样出的数据会越来越没有价值，所以使用KL散度来约束策略更新的幅度

在经过约束后，我们可以近似认为前后两个策略的状态分布是一样的，但是在对动作上的选择不同的策略还是有区别的，因此使用**重要性采样**来让我们老策略采样出的历史的action，用来评估我们新的策略。从而去优化我们新的策略目标值

但TRPO在求解过程中存在一定的**不足**：

- 近似带来的误差

这也是重要性采样的通病，虽然我们使两个策略的KL散度保证不超过某个上界值，但即使如此我们会仍然碰到重要性采样估计不准，甚至过大的情况，导致学习的误差会比较大

- 求解约束优化问题的困难

另外一方面，就是近似的去求解约束优化问题本身的一些困难。TRPO 使用很多复杂的方法来进行求解，比如泰勒展开近似、共轭梯度、线性搜索等方法，这使计算过程非常复杂，每一步更新的运算量非常大。

在TRPO这些痛点的基础上进行改进——一种新的算法PPO算法

PPO全称**Proxmial Policy Optimization**

PPO中的第一个P为Proxmial，也就是近似替代的意思，它在TRPO上有3个改进

PPO—截断

第一项改进，就是针对**重要性采样**来进行改进，PPO提出了一种clip（截断;修建）的方式，将重要性采样的权重，ratio本身去做了一个截断，即下式子中的$\mathbf{r_t(\theta)}$：

$$L^{CPI}(\theta)=\hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right]=\hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right]$$

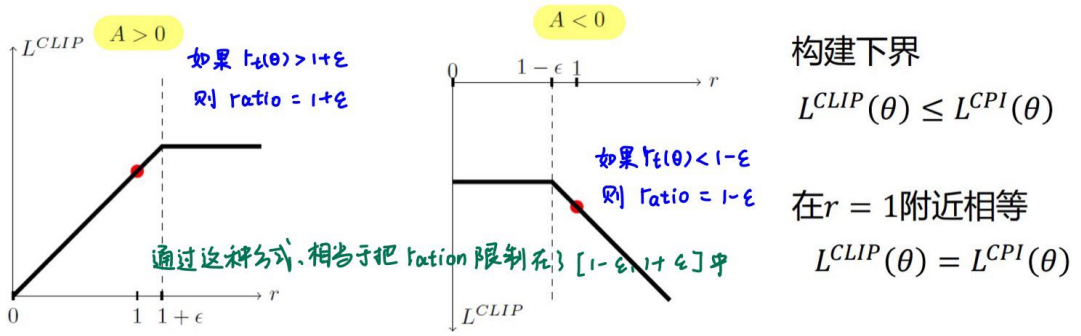
具体截断的目标值为：

$$L^{CLIP}(\theta)=\hat{\mathbb{E}}_t\left[\min\left(r_t(\theta)\hat{A}_t,\text{clip}(r_t(\theta),1-\epsilon,1+\epsilon)\hat{A}_t\right)\right]$$

它是两项值，然后取最小值的关系。

- 第一个是原来值，TRPO中的ratio乘以优势函数
- 第二项，也就是该算法提出的基于截断式的优化

重点看第二项，第二项将原来的rtion，与1+ε和1-ε做一个上下界的截断



如果不进行截断，ration会变得非常的不确定，比如我们的ratio可能会存在大于1+ε的情况，即老策略采样到的动作的概率作太小，但是它作为分母就会导致ration变得很大，这样就会使得权重变得过大，由于我们限制了两个策略前后的**KL divergence**，所以这两个策略的ration就不应该有过大或者过下的情况出现

所以通过截断的方式，就使TRPO的鲁棒性或者说是稳定性获得了提升

PPO—优势函数A_t

PPO在TRPO上的第二个改进，即优势函数使用**多步时序差分**

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$$

在一般的优势函数中，动作价值函数直接就是当前奖励加上一步的下一个状态价值乘以折扣然后与当前状态价值函数做差来作为优势函数，这是一步的时序差分计算出来的优势函数

而PPO中，往前看一定数量的步数，使得我们获得了T步的经验数据作为整个优势函数的信息量，显然是比一步带来的信息量会更大，使得我们目标的估计会更加靠谱一些

- 在每次迭代中，并行N个actor收集T步经验数据
- 计算每步的$\hat{A}_t$和$L^{CLIP}(\theta)$，构成mini-batch
- 更新参数θ，并更新$\theta_{\text{old}} \leftarrow \theta$

PPO—惩罚

PPO相比TRPO上的第三个改进为KL散度惩罚项参数自适应的方法

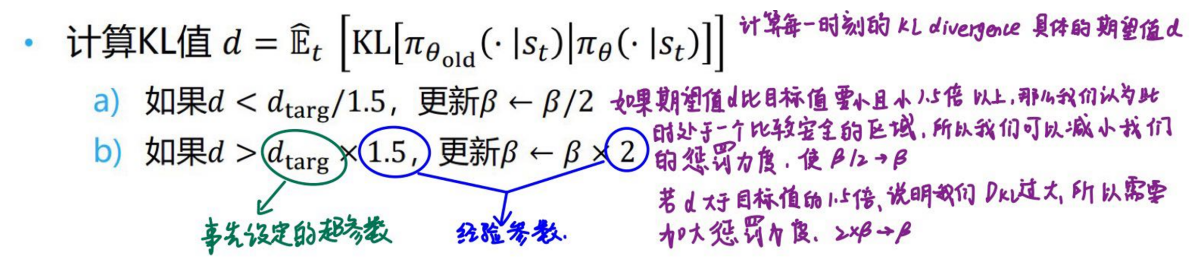
在TRPO中的惩罚项系数为λ，如果策略的更新幅度增大，λ也会增大，从而增加惩罚力度，来约束策略的变化幅度

λ的更新是通过KL散度进行更新的，如果这个KL散度超过了ε，即$D_{KL} - \varepsilon > 0$，就加上这个超出的部分$D_{KL} - \varepsilon$ 然后乘以某个比率来更新这个λ，即λ的更新为：

$$\lambda \leftarrow \lambda + \alpha(D_{KL}(\pi_{\theta'}(a_t|s_t) \parallel \pi_\theta(a_t|s_t)) - \epsilon)$$

而这个加法型的惩罚项更新可能会比较慢，以至于这个约束并不能够及时满足目标更新（❓为啥加法就慢啊

所以PPO利用**乘和除的方法**，来改进惩罚项参数使得PPO中的β值，也就是TRPO中的λ值。它的改进速度会快很多，具体的改进方法为：



通过这种方式，使得我们对β的管控，是否超过当前目标值（用于限制学习策略和之前一轮策略的差距）来使我们的β更加敏感，其作用更加明显一些

这就是PPO相比TRPO的第三点改进

代码实践：

```
class PPO:
    ''' PPO算法,采用截断方式 '''
    def __init__(self, state_dim, hidden_dim, action_dim, actor_lr,
critic_lr,
                 lmbda, epochs, eps, gamma, device):
        self.actor = PolicyNet(state_dim, hidden_dim, action_dim).to(device)
        self.critic = ValueNet(state_dim, hidden_dim).to(device)
        self.actor_optimizer = torch.optim.Adam(self.actor.parameters(),
                                                lr=actor_lr)
        self.critic_optimizer = torch.optim.Adam(self.critic.parameters(),
                                                 lr=critic_lr)
        self.gamma = gamma
        self.lmbda = lmbda
        self.epochs = epochs  # 一条序列的数据用来训练轮数
        self.eps = eps  # PPO中截断范围的参数
        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 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 = rl_utils.compute_advantage(self.gamma, self.lmbda,
```

```
td_delta.cpu()).to(self.device)
        old_log_probs = torch.log(self.actor(states).gather(1,
 actions)).detach()
        for _ in range(self.epochs):
            log_probs = torch.log(self.actor(states).gather(1, actions))
            ratio = torch.exp(log_probs - old_log_probs)
            surr1 = ratio * advantage
            surr2 = torch.clamp(ratio, 1 - self.eps,
                                1 + self.eps) * advantage  # 截断
            actor_loss = torch.mean(-torch.min(surr1, surr2))  # PPO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()
            critic_loss.backward()
            self.actor_optimizer.step()
            self.critic_optimizer.step()
```

深度策略梯度算法小结

- 相比价值函数学习最小化TD误差的目标，策略梯度方法直接优化策略价值的目标 更加贴合强化学习本质目标
- 基于神经网络的策略在优化时容易因为一步走得太大而变得很差，进而下一轮产生 很低质量的经验数据，进一步无法学习好
- Trust Region一类方法限制一步更新前后策略的差距（用KL散度），进而对策略 价值做稳步地提升
- PPO在TRPO的基础上进一步通过限制importance ratio的range，构建优化目标 的下界，进一步保证优化的稳定效果，是目前最常用的深度策略梯度算法
- 针对连续动作的决定性策略，可以从构建的critic中直接回传梯度到动作上，然后 通过链式法则进一步将梯度回传到策略网络中
- 分布式的actor-critic算法能够充分利用多核CPU资源采样环境的经验数据，利用 GPU资源异步地更新网络，这有效提升了DRL的训练效率