# 强化学习 HW5

— 饶翔云 520030910366

## Problem 1

### Coding:

#### TRPO:

```
""" ------------- Programming 1: implement the linear search to find best
parameter for actor (you may refer to original TRPO paper,
Appendix C) ------------- """
""" YOUR CODE HERE """
# load new parameter to new actor
torch.nn.utils.convert_parameters.vector_to_parameters(new_para,
                                                       new_actor.parameters())
# compute kl divergence and new objective
new_action_dists = torch.distributions.Categorical(new_actor(states))
kl = 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)
improve = new_obj - old_obj
# check if kl is less than constraint and improve is large enough
if kl < self.kl_constraint and improve > 0:
    self.actor.load_state_dict(new_actor.state_dict())
    break
""" ------------- Programming 1 ------------- """
```

```
 """ ------------- Programming 2: implement the conjugate_gradient function,
 the linear search function, to update actor parameter (you may refer to original
 TRPO paper, Section 6) ------------- """
""" YOUR CODE HERE """
# compute gradient vector
grads = torch.cat([grad.view(-1) for grad in grads])
# compute search direction
step_dir = self.conjugate_gradient(grads.data, states, old_action_dists)
# get hessian vector product
hd = self.hessian_matrix_vector_product(states, old_action_dists, step_dir)
# get max coef
max_coef = torch.sqrt(2 * self.kl_constraint / (torch.dot(step_dir, hd) + 1e-8))
# get max vec
max_vec = max_coef * step_dir
# line search
self.line_search(states, actions, advantage, old_log_probs,
                 old_action_dists, max_vec)
""" ------------- Programming 2 ------------- """
```

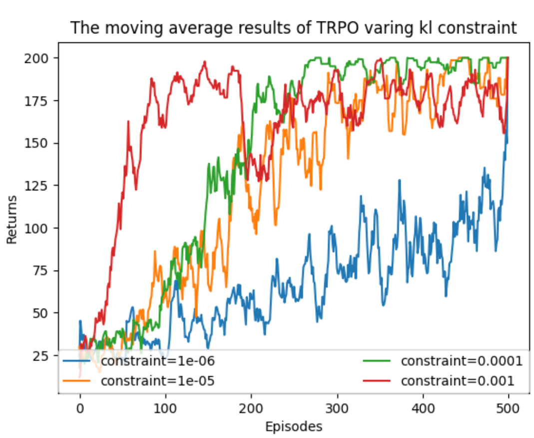
```
""" ------------- Programming 3: Compute GAE and update the parameter of actor and critic ------------- """
""" YOUR CODE HERE """
# compute GAE
values = self.critic(states)
next_values = self.critic(next_states)
td_target = rewards + self.gamma * next_values * (1 - dones)
td_delta = td_target - values
advantage = compute_advantage(self.gamma, self.lmbda, td_delta)
# normalize advantage if needed
advantage = (advantage - advantage.mean()) / (advantage.std() + 1e-10)
# update critic
critic_loss = torch.nn.functional.mse_loss(values, td_target.detach())
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()
# get gradient of loss and hessian vector product of kl divergence
old_action_dists = torch.distributions.Categorical(self.actor(states).detach())
old_log_probs = torch.log(old_action_dists.probs.gather(1, actions)).detach()
# update actor
self.policy_learn(states, actions, old_action_dists, old_log_probs, advantage)
""" ------------- Programming 3 ------------- """
```

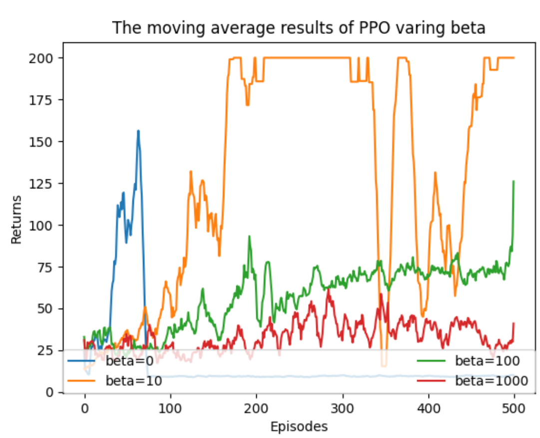
PPO:

```
""" ------------- Programming 4: Compute Advantage Function ------------- """
""" YOUR CODE HERE """
# compute td delta by using Bellman equation
td_target = rewards + self.gamma * self.critic(next_states) * (1 - dones)
td_delta = td_target - self.critic(states)
# compute advantage
advantage = compute_advantage(self.gamma, self.lmbda, td_delta)
# normalize advantage
# advantage = (advantage - advantage.mean()) / (advantage.std() + 1e-12)
# initalize old_log_probs for computing ratio
old_probs = self.actor(states)
dist = torch.distributions.Categorical(old_probs)
old_log_probs = dist.log_prob(actions.squeeze(1)).detach()
old_probs = old_probs.view((-1, self.action_dim)).detach()
""" ------------- Programming 4 ------------- """
```

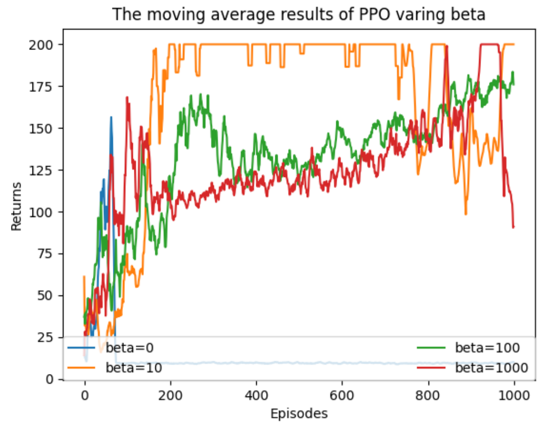
```
for _ in range(self.epochs):
    """ ------------- Programming 5: Update the parameter of actor and critic
    (you may refer to original PPO paper) ------------- """
    """ YOUR CODE HERE """
    from torch.distributions import Categorical
    from torch.nn.functional import kl_div
    # compute new_log_probs
    new_probs = self.actor(states)
    dist = Categorical(new_probs)
    new_log_probs = dist.log_prob(actions.squeeze(1))
    # compute ratio
    ratio = torch.exp(new_log_probs - old_log_probs.detach())
    # compute surrogate loss
    surrogate_loss = ratio * advantage
    # compute kl divergence(penalty)
    penalty = kl_div(input = new_probs.log(), target = old_probs.detach(),
                     reduction='batchmean')
    # update actor
    self.actor_optimizer.zero_grad()
    actor_loss = - (surrogate_loss - self.beta * penalty).mean()
    actor_loss.backward()
    self.actor_optimizer.step()
    # update critic
    self.critic_optimizer.zero_grad()
    critic_loss = F.mse_loss(self.critic(states), td_target.detach())
    critic_loss.backward()
    self.critic_optimizer.step()
# I want to update beta but it's not required.
# dl = kl_div(input=self.actor(states).log(),
#             target=old_probs.detach(), reduction='sum')
# print(dl)
#if abs(dl) >= 1.5 * 1e-2:
#    self.beta *= 2
#if abs(dl) <= 1e-2 / 1.5:
#    self.beta *= 0.5
# print(self.beta)
    """ ------------- Programming 5 ------------- """
```

## Results:




下图是自适应beta。




## Answers:

**(a)**

从图一我们可以看到，随着kl-constraint增大，每次进行游戏得到的总返回值是增大的。且kl-constraint越大，总返回值随着episode增大而增大的速度也越大。这是因为kl-constraint通过限制新的分布与旧的分布之间的KL散度，限制了actor梯度更新的范围。当kl-constraint增大，actor梯度更新的范围增大，随着训练轮数（episode）的增加，actor更新也就越快，所得到的总返回值也就越大。

**(b)**

从图二中我们可以看到，随着$\beta$的增大，总返回值随着episode增大的趋势开始减小。这是因为$\beta$和PPO中计算loss的KL-penalty惩罚项相关。当$\beta$大的时候，这个惩罚项就变得很大，减小了loss的值，从而减小了梯度更新范围，让更新速度变慢。当$\beta$为0时，我们从图中可以看到，PPO算法由于没有kl-penalty的约束而迅速发生了崩坏现象，导致总返回值变得很低。而$\beta$不为0的时候，可以看到PPO算法迅速收敛。除此之外，通过比较图二图三，我们可以发现，$\beta$基本用于防止初始时通过限制新分布和旧分布之间的距离来防止，PPO算法的崩坏，而通过观察输出，之后$\beta$基本降到一个接近于0的数。这意味着，崩坏现象通常在训练初期发生，这可以通过初始化一个相对大的$\beta$来避免。在训练中后期可以通过调整$\beta$的值来加速梯度更新。

**($c$)**

他们在某种程度上是相似的，比如都和梯度更新范围有关。因为kl-constraint限制了新的分布和旧的分布之间的距离从而限制了梯度更新范围，而在PPO中，继承了TRPO的思想，使用$\beta * KLD$的惩罚项来约束新旧分布之间的距离。所以他们是相似的。