```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim


class PPO():
    def __init__(self,
                 actor_critic,
                 clip_param,
                 ppo_epoch,
                 num_mini_batch,
                 value_loss_coef,
                 entropy_coef,
                 lr=None,
                 eps=None,
                 max_grad_norm=None,
                 use_clipped_value_loss=True):

        self.actor_critic = actor_critic

        self.clip_param = clip_param
        self.ppo_epoch = ppo_epoch
        self.num_mini_batch = num_mini_batch

        self.value_loss_coef = value_loss_coef
        self.entropy_coef = entropy_coef

        self.max_grad_norm = max_grad_norm
        self.use_clipped_value_loss = use_clipped_value_loss

        self.optimizer = optim.Adam(actor_critic.parameters(), lr=lr, eps=eps)

    def update(self, rollouts):
        advantages = rollouts.returns[:-1] - rollouts.value_preds[:-1]
        advantages = (advantages - advantages.mean()) / (
            advantages.std() + 1e-5)

        value_loss_epoch = 0
        action_loss_epoch = 0
        dist_entropy_epoch = 0

        for e in range(self.ppo_epoch):
            if self.actor_critic.is_recurrent:
                data_generator = rollouts.recurrent_generator(
                    advantages, self.num_mini_batch)
            else:
                data_generator = rollouts.feed_forward_generator(
                    advantages, self.num_mini_batch)

            for sample in data_generator:
                obs_batch, recurrent_hidden_states_batch, actions_batch, \
                   value_preds_batch, return_batch, masks_batch,
old_action_log_probs_batch, \
                        adv_targ = sample

                # Reshape to do in a single forward pass for all steps
                values, action_log_probs, dist_entropy, _ =
self.actor_critic.evaluate_actions(
                    obs_batch, recurrent_hidden_states_batch, masks_batch,
                    actions_batch)

                ratio = torch.exp(action_log_probs -
                                  old_action_log_probs_batch)
                surr1 = ratio * adv_targ
                surr2 = torch.clamp(ratio, 1.0 - self.clip_param,
```

$r(t) = \frac{\pi}{\pi_{old}}$

$\min(r(t)A_t, \text{clip}(1-\epsilon, r(t)A_t, 1+\epsilon))$

```
                                1.0 + self.clip_param) * adv_targ
                action_loss = -torch.min(surr1, surr2).mean()

                if self.use_clipped_value_loss:
                    value_pred_clipped = value_preds_batch + \
                        (values - value_preds_batch).clamp(-self.clip_param,
self.clip_param)
                    value_losses = (values - return_batch).pow(2)
                    value_losses_clipped = (
                        value_pred_clipped - return_batch).pow(2)
                    value_loss = 0.5 * torch.max(value_losses,
                                                 value_losses_clipped).mean()
                else:
                    value_loss = 0.5 * (return_batch - values).pow(2).mean()

                self.optimizer.zero_grad()
                (value_loss * self.value_loss_coef + action_loss -
                 dist_entropy * self.entropy_coef).backward()
                nn.utils.clip_grad_norm_(self.actor_critic.parameters(),
                                         self.max_grad_norm)
                self.optimizer.step()

                value_loss_epoch += value_loss.item()
                action_loss_epoch += action_loss.item()
                dist_entropy_epoch += dist_entropy.item()

        num_updates = self.ppo_epoch * self.num_mini_batch

        value_loss_epoch /= num_updates
        action_loss_epoch /= num_updates
        dist_entropy_epoch /= num_updates

        return value_loss_epoch, action_loss_epoch, dist_entropy_epoch
```

use larger of two loss options

→ usual three terms for actor critics: value loss, actor loss, entropy

much simpler than TRPO?