Proximal Policy Optimization - PPO

This is a PyTorch implementation of Proximal Policy Optimization - PPO.

PPO is a policy gradient method for reinforcement learning. Simple policy gradient methods do a single gradient update per sample (or a set of samples). Doing multiple gradient steps for a single sample causes problems because the policy deviates too much, producing a bad policy. PPO lets us do multiple gradient updates per sample by trying to keep the policy close to the policy that was used to sample data. It does so by clipping gradient flow if the updated policy is not close to the policy used to sample the data.

You can find an experiment that uses it here. The experiment uses Generalized Advantage Estimation.

Open in Colab

```python
import torch

from labml_helpers.module import Module
from labml_nn.rl.ppo.gae import GAE
```

PPO Loss

Here's how the PPO update rule is derived.

We want to maximize policy reward

$$\max_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

where r is the reward, π is the policy, τ is a trajectory sampled from policy, and γ is the discount factor between $[0, 1]$.

$$\mathbb{E}_{\tau\sim\pi\theta}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi OLD}(s_t,a_t)\right] =$$

$$\mathbb{E}_{\tau\sim\pi\theta}\left[\sum_{t=0}^{\infty}\gamma^t \Big(Q^{\pi OLD}(s_t,a_t) - V^{\pi OLD}(s_t)\Big)\right] =$$

$$\mathbb{E}_{\tau\sim\pi\theta}\left[\sum_{t=0}^{\infty}\gamma^t \Big(r_t + V^{\pi OLD}(s_{t+1}) - V^{\pi OLD}(s_t)\Big)\right] =$$

$$\mathbb{E}_{\tau\sim\pi\theta}\left[\sum_{t=0}^{\infty}\gamma^t \Big(r_t\Big)\right] - \mathbb{E}_{\tau\sim\pi\theta}\Big[V^{\pi OLD}(s_0)\Big] = J(\pi_\theta) - J(\pi_{\theta_{OLD}})$$

So,

$$\max_\theta J(\pi_\theta) = \max_\theta \mathbb{E}_{\tau\sim\pi\theta}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi OLD}(s_t,a_t)\right]$$

Define discounted-future state distribution,

$$d^\pi(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P(s_t = s|\pi)$$

Then,

$$\begin{aligned} J(\pi_\theta) - J(\pi_{\theta_{OLD}}) &= \mathbb{E}_{\tau\sim\pi\theta}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi OLD}(s_t,a_t)\right] \\ &= \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^{\pi\theta}, a\sim\pi\theta}\Big[A^{\pi OLD}(s,a)\Big] \end{aligned}$$

Importance sampling a from $\pi_{\theta_{OLD}}$,

$$\begin{aligned} J(\pi_\theta) - J(\pi_{\theta_{OLD}}) &= \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^{\pi\theta}, a\sim\pi\theta}\Big[A^{\pi OLD}(s,a)\Big] \\ &= \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^{\pi\theta}, a\sim\pi\theta_{OLD}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{OLD}}(a|s)}A^{\pi OLD}(s,a)\right] \end{aligned}$$

Then we assume $d^{\pi\theta}(s)$ and $d^{\pi\theta_{OLD}}(s)$ are similar. The error we introduce to $J(\pi_\theta) - J(\pi_{\theta_{OLD}})$ by this assumption is bound by the KL divergence between π_θ and $\pi_{\theta_{OLD}}$. Constrained Policy Optimization shows the proof of this. I haven't read it.

$$\begin{aligned}
J(\pi_\theta) - J(\pi_{\theta OLD}) &= \frac{1}{1-\gamma} \mathop{\mathbb{E}}_{\substack{s \sim d^{\pi_\theta} \\ a \sim \pi_{\theta OLD}}} \Bigg[\frac{\pi_\theta(a|s)}{\pi_{\theta OLD}(a|s)} A^{\pi OLD}(s,a) \Bigg] \\
&\approx \frac{1}{1-\gamma} \mathop{\mathbb{E}}_{\substack{s \sim d^{\pi_{\theta OLD}} \\ a \sim \pi_{\theta OLD}}} \Bigg[\frac{\pi_\theta(a|s)}{\pi_{\theta OLD}(a|s)} A^{\pi OLD}(s,a) \Bigg] \\
&= \frac{1}{1-\gamma} \mathcal{L}^{CPI}
\end{aligned}$$

```python
class ClippedPPOLoss(Module):
```

```python
    def __init__(self):
        super().__init__()
```

```python
    def forward(self, log_pi: torch.Tensor, sampled_log_pi: torch.Tensor,
                advantage: torch.Tensor, clip: float) -> torch.Tensor:
```

ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta OLD}(a_t|s_t)}$; *this is different from rewards* r_t.

```python
        ratio = torch.exp(log_pi - sampled_log_pi)
```

Cliping the policy ratio

$$\mathcal{L}^{CLIP}(\theta) = \mathbb{E}_{a_t, s_t \sim \pi_{\theta OLD}} \Bigg[min \Big(r_t(\theta) \bar{A}_t, clip\big(r_t(\theta), 1-\epsilon, 1+\epsilon \big) \bar{A}_t \Big) \Bigg]$$

The ratio is clipped to be close to 1. We take the minimum so that the gradient will only pull π_θ towards $\pi_{\theta OLD}$ if the ratio is not between $1-\epsilon$ and $1+\epsilon$. This keeps the KL divergence between π_θ and $\pi_{\theta OLD}$ constrained. Large deviation can cause performance collapse; where the policy performance drops and doesn't recover because we are sampling from a bad policy.

Using the normalized advantage $\bar{A}_t = \frac{\hat{A}_t - \mu(\hat{A}_t)}{\sigma(\hat{A}_t)}$ introduces a bias to the policy gradient estimator, but it reduces variance a lot.

```python
        clipped_ratio = ratio.clamp(min=1.0 - clip,
                                    max=1.0 + clip)
        policy_reward = torch.min(ratio * advantage,
                                  clipped_ratio * advantage)

        self.clip_fraction = (abs((ratio - 1.0)) > clip).to(torch.float).mean()

        return -policy_reward.mean()
```

Clipped Value Function Loss

Similarly we clip the value function update also.

$$V^{\pi_\theta}_{CLIP}(s_t) = clip\Bigl(V^{\pi_\theta}(s_t) - \hat{V}_t, -\epsilon, +\epsilon\Bigr)$$
$$\mathcal{L}^{VF}(\theta) = \frac{1}{2}\mathbb{E}\biggl[max\Bigl(\bigl(V^{\pi_\theta}(s_t) - R_t\bigr)^2, \bigl(V^{\pi_\theta}_{CLIP}(s_t) - R_t\bigr)^2\Bigr)\biggr]$$

Clipping makes sure the value function V_θ doesn't deviate significantly from $V_{\theta_{OLD}}$.

```python
class ClippedValueFunctionLoss(Module):
```

```python
    def forward(self, value: torch.Tensor, sampled_value: torch.Tensor, sampled_return: torch.Tensor, clip: float):
        clipped_value = sampled_value + (value - sampled_value).clamp(min=-clip, max=clip)
        vf_loss = torch.max((value - sampled_return) ** 2, (clipped_value - sampled_return) ** 2)
        return 0.5 * vf_loss.mean()
```

labml.ai