PPO

July 7, 2025

1 PPO

https://arxiv.org/pdf/1707.06347

1.1 1. Policy Gradient

$$E(R(\tau))_{\tau \sim P_\theta(\tau)} = \sum_{\tau} R(\tau) P_\theta(\tau)$$

We want expected reward to be as large as possible.
So we need to find its gradient

$$\begin{aligned}
\nabla E(R(\tau))_{\tau \sim P_\theta(\tau)} &= \nabla \sum_{\tau} R(\tau) P_\theta(\tau) \\
&= \sum_{\tau} R(\tau) \nabla P_\theta(\tau) \\
&= \sum_{\tau} R(\tau) \nabla P_\theta(\tau) \frac{P_\theta(\tau)}{P_\theta(\tau)} \\
&= \sum_{\tau} R(\tau) P_\theta(\tau) \frac{\nabla P_\theta(\tau)}{P_\theta(\tau)} \\
&\approx \frac{1}{N} \sum_{n=1}^{N} R(\tau^n) \frac{\nabla P_\theta(\tau)}{P_\theta(\tau)} \text{ Rough Average} \\
&= \frac{1}{N} \sum_{n=1}^{N} R(\tau^n) \nabla \log P_\theta(\tau^n) \\
&= \frac{1}{N} \sum_{n=1}^{N} R(\tau^n) \nabla \log \prod_{t=1}^{T_n} P_\theta(a_n^t \mid s_n^t) \\
&= \frac{1}{N} \sum_{n=1}^{N} R(\tau^n) \sum_{t=1}^{T_n} \nabla \log P_\theta(a_n^t \mid s_n^t) \\
&= \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} R(\tau^n) \nabla \log P_\theta(a_n^t \mid s_n^t) \\
&= \nabla_\theta J(\theta)
\end{aligned}$$

$$Loss = -\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla \log P_\theta(a_n^t \mid s_n^t)$$

Basic Gradient Update:

$$1.\text{Sample } (\tau)^i \text{ from } \pi_\theta(a_t \mid s_t)$$

$$2.\text{Compute gradient } \nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla \log P_\theta(a_n^t \mid s_n^t)$$

$$3.\text{Take one step along gradient } \theta = \theta + \alpha\nabla_\theta J(\theta)$$

On Policy: Use the same model to do data collection and train model

Problem 1:

$$\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla \log P_\theta(a_n^t \mid s_n^t)$$

This means if the trajectory has **positive reward sum**, we would **raise all probabilities of actions** along this trajectory, which is obviously inefficient.

So we should find the actual impact of an action.
We'd like to find how current action affects future rewards

Compute Future Reward:

$$R(\tau^n) \rightarrow \sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n = R_t^n \ (\gamma \text{ is called discount factor})$$

$$\text{replacement: } J(\theta) = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R_t^n \nabla \log P_\theta(a_n^t \mid s_n^t)$$

$$\gamma < 1$$

$$\gamma \text{ decreases as t gets bigger(futher == less impact on future)}$$

Problem 2:
Let's say we have a good situation where all actions increases future reward, so we **increases all their probabilities**.This could be quite inefficient as probability sum remain one, meaning increment could only be subtle.(The same for all negative situation)

So what we actually want is to significantly **increases probabilities of the actions that has "big" positive reward** and decrease those who has small positive reward.(Focus on **reletive reward**)

So we need to choose a proper **baseline** and reletive_reward = reward - baseline

$$\text{replacement: } J(\theta) = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}(R_t^n - B(s_n^t))\nabla \log P_\theta(a_n^t \mid s_n^t)$$

1.2 2. Actor-Critic

To further optimize our algorithm, we introduce three fuctions:

Action-Value Function(Q-function):

$$Q_\theta(s_t, a_t) = \sum_{t'=t}^{T} E_{\pi_\theta}[(r(s_t', a_t') \mid s_t, a_t)]$$

Expected total future reward by taking a_t in s_t

State-Value Function(V-function):

$$V_\theta(s_t) = E_{a_t \sim \pi_\theta(a_t|s_t)}[Q^\pi(s_t, a_t)]$$

Expected total future reward in s_t

Advantage Function:

$$A_\theta(s_t, a_t) = Q_\theta(s_t, a_t) - V_\theta(s_t)$$

Gained advantage by taking a_t in s_t compared with other actions

Noted that we need to fit two neural networks, one for Q-function and one for V-function

$$\text{replacement: } J(\theta) = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} A_\theta(s_t, a_t)\nabla \log P_\theta(a_n^t \mid s_n^t)$$

Then what's the relation between Q and V?

We can make inference based on definition, then we have:

$$Q_\theta(s_t, a_t) = r_t + \gamma * V_\theta(s_{t+1})$$

$$\text{replacement: } A_\theta(s_t, a_t) = r_t + \gamma * V_\theta(s_{t+1}) - V_\theta(s_t)$$

So we only need to fit one neural network which is V-fuction!

Now we can do recursive sampling on V-function:

$$\text{recursive formula: } V_\theta(s_{t+1}) \approx r_{t+1} + \gamma * V_\theta(s_t + 2)$$

$$
\begin{aligned}
&A_\theta^1(s_t,a_t) = r_t + \gamma * V_\theta(s_{t+1}) - V_\theta(s_t)\\
&A_\theta^2(s_t,a_t) = r_t + \gamma * r_{t+1} + \gamma^2 * V_\theta(s_{t+2}) - V_\theta(s_t)\\
&A_\theta^3(s_t,a_t) = r_t + \gamma * r_{t+1} + \gamma^2 * V_\theta(s_{t+2}) + \gamma^3 * V_\theta(s_{t+3}) - V_\theta(s_t)\\
&A_\theta^T(s_t,a_t) = r_t + \gamma * r_{t+1} + \gamma^2 * V_\theta(s_{t+2}) + \gamma^3 * V_\theta(s_{t+3}) + ... + \gamma^T * r_T - V_\theta(s_t)
\end{aligned}
$$

With more terms(bigger T), the bias goes down and the variance goes up

Simplified version:

$$
\begin{aligned}
\sigma_t^V &= r_t + \gamma * V_\theta(s_{t+1}) - V_\theta(s_t)\\
\sigma_{t+1}^V &= r_t + \gamma * r_{t+1} + \gamma^2 * V_\theta(s_{t+2}) - V_\theta(s_t + 1)
\end{aligned}
$$

$$
\text{replacement: } \begin{aligned}
&A_\theta^1(s_t,a_t) = \sigma_t^V\\
&A_\theta^2(s_t,a_t) = \sigma_t^V + \gamma\sigma_{t+1}^V\\
&A_\theta^3(s_t,a_t) = \sigma_t^V + \gamma\sigma_{t+1}^V + \gamma^2\sigma_{t+2}^2
\end{aligned}
$$

1.3 3.Generalized Advantage Estimation(GAE)

$$
A_\theta^{GAE}(s_t,a_t) = (1-\lambda)(A_\theta^1 + \lambda A_\theta^2 + \lambda^2 A_\theta^3 + ...) = \sum_{b=0}^{\infty}(\gamma\lambda)^b \sigma_{t+b}^V
$$

$$
\text{replacement: } J(\theta) = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} A_\theta^{GAE}(s_t,a_t)\nabla \log P_\theta(a_n^t \mid s_n^t)
$$

1.4 4.Proximal Policy Optimization (PPO)

Off Policy: Use one model for collecting trajectories and use those data to train other models

Importance sampling:

$$
\begin{aligned}
E(f(x))_{x\sim p(x)} &= \sum_x f(x) * p(x)\\
&= \sum_x f(x) * p(x)\frac{q(x)}{q(x)}\\
&= \sum_x f(x)\frac{p(x)}{q(x)} * q(x)\\
&= E(f(x)\frac{p(x)}{q(x)})_{x\sim q(x)}\\
&= \frac{1}{N}\sum_{n=1}^{N} f(x)\frac{p(x)}{q(x)}_{x\sim q(x)}
\end{aligned}
$$

$$
\text{replacement: } J(\theta) = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} A_{\theta'}^{GAE}(s_t,a_t)\frac{P_\theta(a_n^t \mid s_n^t)}{P_{\theta'}(a_n^t \mid s_n^t)}\nabla \log P_\theta(a_n^t \mid s_n^t)
$$

$$\text{replacement: } J(\theta) = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} A_{\theta}'^{GAE}(s_t, a_t)\frac{\nabla P_\theta(a_n^t \mid s_n^t)}{P_\theta'(a_n^t \mid s_n^t)}$$

$$Loss = -\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} A_{\theta'}^{GAE}(s_t, a_t)\frac{\nabla P_\theta(a_n^t \mid s_n^t)}{P_{\theta'}(a_n^t \mid s_n^t)}$$

Now we can sample with policy $\theta'$and updata our policy θ

Intuition for the ratio: Bigger probability to take the action means making bigger change

Problem 3:
If the two policies have too distinctive distribution then the information can be useless.

Solution1:
We use **kl-divergence** to evaluate their differences. (kl=0 if the two distributions are the same)

$$Loss = -\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} A_{\theta'}^{GAE}(s_t, a_t)\frac{\nabla P_\theta(a_n^t \mid s_n^t)}{P_{\theta'}(a_n^t \mid s_n^t)} + \beta KL(P_\theta, P_{\theta'}) \text{ (} \beta \text{ is used for managing constraint)}$$

Solution2:
We use clip method.

$$Loss = -\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} min(A_{\theta'}^{GAE}(s_t, a_t)\frac{P_\theta(a_n^t \mid s_n^t)}{P_{\theta'}(a_n^t \mid s_n^t)}, clip(\frac{P_\theta(a_n^t \mid s_n^t)}{P_{\theta'}(a_n^t \mid s_n^t)}, 1-\epsilon, 1+\epsilon)A_{\theta'}^{GAE}(s_n^t, a_n^t))$$

1.5 Coding

[2]:
```python
import os
import torch
import torch.nn as nn
from torch.distributions.categorical import Categorical
import numpy as np
import gymnasium as gym
```

Write the rollout buffer

[3]:
```python
class RolloutBuffer:
    def __init__(self):
        self.states = []
        self.actions = []
```

```
        self.rewards = []
        self.log_probs = []
        self.values = []
        self.dones = []

    def generate_batches(self,batch_size):
        batches = []
        n_states = len(self.states)
        batch_start = np.arange(0,n_states,batch_size)
        # indices = np.arange(n_states, dtype=np.int64)
        #np.random.shuffle(indices) #diorder the indices for the sake of␣
↪stochastic gradient descent
        for i in batch_start:
            c_i = i
            t_batch = []
            for j in range(batch_size):
                t_batch.append(self.states[c_i])
                c_i+=1
            batches.append(t_batch)
        return batches

    def store(self, state, action, reward, value, log_prob, done):
        self.states.append(state)
        self.actions.append(action)
        self.rewards.append(reward)
        self.log_prob.append(log_prob)
        self.dons.append(done)

    def clear(self):
        self.states = []
        self.actions = []
        self.rewards = []
        self.log_probs = []
        self.values = []
        self.dones = []
```

Build Actor-Critic Networks

[4]:
```
class ActorNetwork(nn.Module):
    def __init__(self, n_actions, n_states, lr):
        super(ActorNetwork, self).__init__()
        self.fc1 = 32
        self.fc2 = 32
        self.lr = lr
        self.actor = nn.Sequential(nn.Linear(n_states, self.fc1),
                                   nn.ReLU(),
                                   nn.Linear(self.fc1, self.fc2),
```

```
                                    nn.ReLU(),
                                    nn.Linear(self.fc2, n_actions),
                                    nn.Softmax(dim=-1))

        self.optimizer = optim.Adam(self.parameters(), lr=self.lr)
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, states):
        act_dis = self.actor(states)
        act_dis = Categorical(act_dist)
        return act_dis
```

[5]:
```
class CriticNetwork(nn.Module):
    def __init__(self, n_states, lr):
        super(CriticNetwork, self).__init__()
        self.fc1 = 32
        self.fc2 = 32
        self.lr = lr
        self.critic = nn.Sequential(nn.Linear(n_states, self.fc1),
                                     nn.ReLU(),
                                     nn.Linear(self.fc1, self.fc2),
                                     nn.ReLU(),
                                     nn.Linear(self.fc2, 1))
        self.optimizer = optim.Adam(self.parameters(), lr = self.lr)
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, states):
        value = self.critic(states)
        return value
```

Implement the agent

[8]:
```
class Agent:
    def __init__(self, n_actions, gamma=0.99, lr=0.0003, policy_clip=0.1,␣
↪batch_size=32, N=2048, n_epochs=10, gae_lambda=0.95):
        self.gamma = gamma
        self.policy_clip = policy_clip
        self.n_epochs = n_epochs
        self.gae_lambda = gae_lambda

        self.actor = ActorNetwork(n_actions, n_states, lr)
        self.critic = CriticNetwork(n_states, lr)
```

[ ]: