Hands-On Reinforcement Learning for Games

Ch9, Optimizing for Continuous Control

최우진

2022/02/04

Importance Sampling

- **중요도 샘플링**

$$
\begin{aligned}
\mathbb{E}_{X\sim P}[f(X)] &= \sum P(X)f(X) \\
&= \sum Q(X)\left[\frac{P(X)}{Q(X)}f(X)\right] \\
&= \mathbb{E}_{X\sim Q}\left[\frac{P(X)}{Q(X)}f(X)\right]
\end{aligned}
$$

Introducing proximal policy optimization(PPO)

- 이전에 배운 TRPO는 복잡하여 구현하기 어렵다.

$$\begin{aligned}
&\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right] \\
&\text{subject to} \quad \hat{\mathbb{E}}_t[\mathrm{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]] \le \delta.
\end{aligned}$$

$$\underset{\theta}{\text{maximize}}\,\hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t - \beta\,\mathrm{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]\right]$$

- PPO는 first-order optimization을 사용하여 구현이 쉬우며 TRPO와 비슷한 성능이다.

Introducing proximal policy optimization(PPO)

- PPO의 Actor의 Loss

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

- Clipped loss function

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

PPO and clipped objectives

```python
for iteration in range(iterations):
    s = env.reset()
    done = False
    while not done:
        for t in range(T_horizon):
            prob = model.pi(torch.from_numpy(s).float())
            m = Categorical(prob)
            a = m.sample().item()
            s_prime, r, done, info = env.step(a)

            model.put_data((s, a, r/100.0, s_prime, prob[a].item(), done))
            s = s_prime

            score += r
            if done:
                if score / print_interval > min_play_reward:
                    play_game()
                break

        model.train_net()

    if iteration % print_interval == 0 and iteration != 0:
        print("# of episode :{}, avg score : {:.1f}".format(iteration, score/print_interval))
        score = 0.0

env.close()
```

PPO and clipped objectives

```python
def train_net(self):
    s, a, r, s_prime, done_mask, prob_a = self.make_batch()

    for i in range(K_epoch):
        td_target = r + gamma * self.v(s_prime) * done_mask
        delta = td_target - self.v(s)
        delta = delta.detach().numpy()

        advantage_lst = []                                         GAE
        advantage = 0.0
        for delta_t in delta[::-1]:
            advantage = gamma * lmbda * advantage + delta_t[0]
            advantage_lst.append([advantage])
        advantage_lst.reverse()

        advantage = torch.tensor(advantage_lst, dtype=torch.float)

        pi = self.pi(s, softmax_dim=1)
        pi_a = pi.gather(1, a)
        ratio = torch.exp(torch.log(pi_a) - torch.log(prob_a))  # a/b == exp(log(a)-log(b))

        surr1 = ratio * advantage
        surr2 = torch.clamp(ratio, 1 - eps_clip, 1 + eps_clip) * advantage

        loss = -torch.min(surr1, surr2) + F.smooth_l1_loss(self.v(s), td_target.detach().float())

        self.optimizer.zero_grad()
        loss.mean().backward()
        self.optimizer.step()
```

Generalized Advantage Estimation

- **TD(λ)**

- Consider the following n-step returns for $n = 1, 2, \infty$:

$$
\begin{array}{lll}
n=1 & (TD) & G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1}) \\
n=2 & & G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \\
\vdots & & \vdots \\
n=\infty & (MC) & G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T
\end{array}
$$

- Define the n-step return

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

- The λ-*return* G_t^λ combines all n-step returns $G_t^{(n)}$
- Using weight $(1-\lambda)\lambda^{n-1}$

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

Generalized Advantage Estimation

$$\hat{A}_t^{(1)} := \delta_t^V \qquad\qquad = -V(s_t) + r_t + \gamma V(s_{t+1}) \tag{11}$$

$$\hat{A}_t^{(2)} := \delta_t^V + \gamma \delta_{t+1}^V \qquad = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \tag{12}$$

$$\hat{A}_t^{(3)} := \delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+2}^V = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+3}) \tag{13}$$

$$\hat{A}_t^{(k)} := \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k}) \tag{14}$$

$$\begin{aligned}
\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} &:= (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \ldots\right) \\
&= (1-\lambda)\left(\delta_t^V + \lambda(\delta_t^V + \gamma \delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+2}^V) + \ldots\right) \\
&= (1-\lambda)(\delta_t^V (1 + \lambda + \lambda^2 + \ldots) + \gamma \delta_{t+1}^V (\lambda + \lambda^2 + \lambda^3 + \ldots) \\
&\qquad + \gamma^2 \delta_{t+2}^V (\lambda^2 + \lambda^3 + \lambda^4 + \ldots) + \ldots) \\
&= (1-\lambda)\left(\delta_t^V \left(\frac{1}{1-\lambda}\right) + \gamma \delta_{t+1}^V \left(\frac{\lambda}{1-\lambda}\right) + \gamma^2 \delta_{t+2}^V \left(\frac{\lambda^2}{1-\lambda}\right) + \ldots\right) \\
&= \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V
\end{aligned} \tag{16}$$

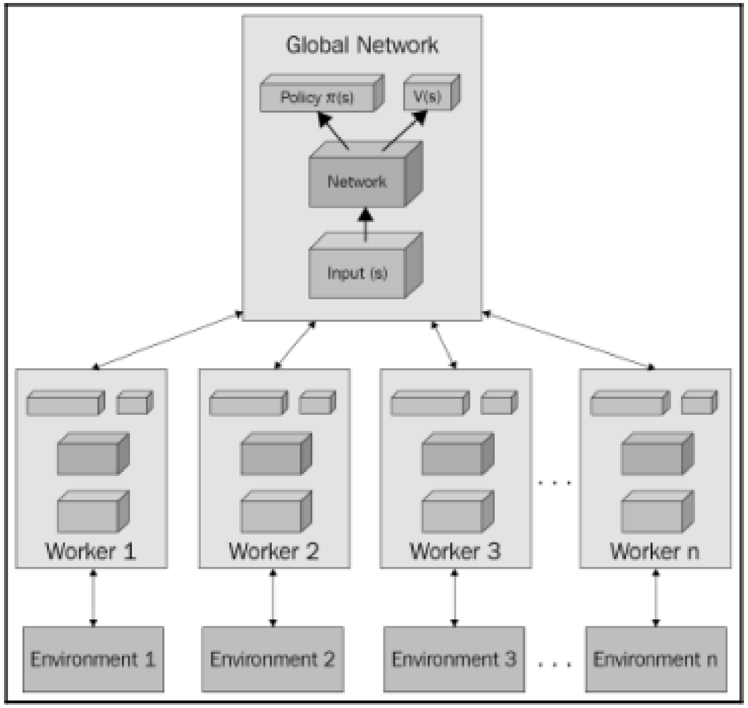
Using PPO with recurrent networks

- LSTM 적용

```
class PPO(nn.Module):
    def __init__(self, input_shape, num_actions):
        super(PPO, self).__init__()
        self.data = []

        self.fc1   = nn.Linear(input_shape,64)
        self.lstm  = nn.LSTM(64,32)
        self.fc_pi = nn.Linear(32,num_actions)
        self.fc_v  = nn.Linear(32,1)
        self.optimizer = optim.Adam(self.parameters(), lr=learning_rate)

    def pi(self, x, hidden):
        x = F.relu(self.fc1(x))
        x = x.view(-1, 1, 64)
        x, lstm_hidden = self.lstm(x, hidden)
        x = self.fc_pi(x)
        prob = F.softmax(x, dim=2)
        return prob, lstm_hidden

    def v(self, x, hidden):
        x = F.relu(self.fc1(x))
        x = x.view(-1, 1, 64)
        x, lstm_hidden = self.lstm(x, hidden)
        v = self.fc_v(x)
        return v
```

Using A3C (Asynchronous Advantage Actor-Critic)

- 여러 개의 에이전트를 실행하며 비동기적으로 공유 네트워크를 업데이트
- 다양한 환경에서 얻은 다양한 데이터로 학습 가능
- DQN에서는 replay buffer를 사용하여
  다양한 데이터로 학습이 가능했지만,
  오래된 데이터 또한 학습에 사용하는 단점 발생
- A3C는 항상 최신 데이터로만 학습



Reference

- Hands-On Reinforcement Learning for Games, Ch 9.
- https://yonghyuc.wordpress.com/2019/07/26/generalized-advantage-estimation-gae/
- https://daeson.tistory.com/334

Ch9, Optimizing for Continuous Control

감사합니다