TD3(Twin Delayed DDPG)

Paper Link: https://arxiv.org/pdf/1802.09477.pdf

Key Features

- Clipping double Q-learning improves learning performance by limiting the upper limit of bias
- Delayed and smooth update minimizes the accumulation of TD errors
- Adding noise on calculating update target after action choice prevent network overfitting and causes agent to explore within the state space

Background

TD3 is an algorithm based on DDPG, designed to solve the problem of **Overestimation bias** which is a weakness of Value-based algorithms, **Accumulated error** occurred by TD algorithms, and **Overfitting by convergence to deterministic policy** when the policy is local optimum.

Overestimation bias

In the process of designating update target as next state's Q-value, **Value-based algorithms have proven to be larger than the actual value if it is biased.** Although it is a policy-based algorithm Actor-Critic, which is an algorithm that uses function approximation by critic-network, also has a problem like this.

When these over-estimation bias accumulates, it converges into a sub-optimal policy. Existing value-based algorithms resolved these problem through usage of double network(doubling), however, policy-based algorithms can’t solve the problem simply by applying doubling. When doubling is used in a policy network (Actor), the effect is invalidated during slow convergence. In addition, when doubling is used in both of the networks(Actor and Critic), there is a problem that the sample can't be extracted from the independent identically distribution(iid). To solve this problem, TD3 use clipping double Q-learning algorithm as following.

$$y_1(r, s', d) = r + \gamma \underset{i=1,2}{min} Q_{\theta'}^{i}(s', \pi_{\phi_1}(s'))$$

Clipping double Q-learning learns by setting update target based on estimates with fewer errors using two independent Critic networks. This learning method reduces the variance of learning and thus helps stable learning. In summary, Clipping double Q-learning improves learning performance by limiting the upper limit of bias.

Accumulated error

It is essential to deal with update target variance in training. Two factors that lower the learning variance in Actor-Critic are **Overestimation bias** and **Slow learning speed**, which are the problems raised above. However, despite these advantages, the overestimation bias is a problem to be solved in order to converge into a global optimum and Slow learning speed has a bad effect on improving performance.

Overestimation bias accumulation has the potential to make the learning direction to maximization estimation error accumulation like as following equation.

$$\begin{aligned} Q_\theta(s_t, a_t) &= r_t + \gamma \mathbb{E}[Q_\theta(s_{t+1}, a_{t+1})] - \delta_t && (1) \\ &= r_t + \gamma \mathbb{E}[r_{t+1} + \gamma \mathbb{E}[Q_\theta(s_{t+2}, a_{t+2}) - \delta_{t+1}]] - \delta_t && (2) \\ &= \mathbb{E} s_i \sim p_\pi, a_i \sim \pi && (3) \end{aligned}$$

Due to the method of the mini-batch update, the update-target of Temporal Difference(TD) has no guarantee of error reduction accumulated in full-batch because the learning slope tries to reduce only by the error of mini-batch.

If this problem is not resolved, biased value estimation and poor policy can be mutually affected, and as a result, policy can converge into sub-optimal policy, so two methods are proposed to reduce the variance of target network-based value estimation.

The first method is to delay(d) the target network(policy and value estimator) update until its errors for behavior network(value estimator) that are continuously updated are minimized.

$$
\begin{aligned}
&Update\ behavior\ value\ estimators: && (4)\\
&\quad \theta_i \leftarrow argmin_{\theta_i} N^{-1}\Sigma(y - Q_{\theta_i}(s,a))^2 && (5)\\
&if \quad t\ mod\ d\ then, && (6)\\
&\quad Update\ \phi\ by\ the\ deterministic\ policy\ gradient: && (7)\\
&\quad\quad \nabla_\phi J(\phi) = N^{-1}\Sigma\nabla_a Q_{\theta_1}(s,a)|_{a=\pi_\phi(s)}\nabla_\phi\pi_\phi(s) && (8)\\
&\quad Update\ target\ \phi^{'}\ and\ \theta^{'}_{i=1,2}: && (9)\\
&\quad\quad \theta^{'}_i \leftarrow \tau\theta_i + (1-\tau)\theta^{'}_i && (10)\\
&\quad\quad \phi^{'} \leftarrow \tau\phi + (1-\tau)\phi^{'} && (11)
\end{aligned}
$$

Since the target policy is updated after the value estimation becomes more accurate, the poor policy can be avoided(biased sample extraction due to poor policy can also be prevented).

The second method is to use the Momentum technique. In order to reduce the TD error, update of the target network is performed based on the following equation.

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$

When updating the policy, the previous policy θ is reflected as a ratio of τ and the next policy θ' is reflected as a ratio of $(1-\tau)$. If target policy update in this way, variance can be reduced because it prevents radical updates from the previous policy to the next policy.

Overfitting by convergence to deterministic policy

The greedy action choice by the deterministic policy extracts a biased sample, which causes network overfitting. In order to solve this problem, a solution of **adding noise(ϵ) on calculating update target after action choice** is proposed as follows.

$$y = r + \gamma Q_{\theta'}(s', \pi_{\phi'}(s') + \epsilon),$$
$$\epsilon \sim clip(\mathcal{N}(0, \sigma), -c, c)$$

Adding noise reduces the number of transitions created by certain preferred update target for learners to prevent overfitting.

Method

Algorithm

Algorithm 1 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ
with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ **to** T **do**
  Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
  $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'
  Store transition tuple (s, a, r, s') in $\mathcal{B}$

  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
  $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  if t mod d **then**
    Update ϕ by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks:
    $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
    $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
  end if
end for

Implementation on JORLDY

- TD3 JORLDY Implementation

```
### learn function ###
# 1. Calculate Q-values from two Q-networks
# 2. Set target Q-value based on the smaller of the two Q-networks.
# 3. Two Q-networks update respectively
# 4. Update policy-network every delay period

def learn(self):
...

    # Critic Update
    with torch.no_grad():
```

```
        noise = (torch.randn_like(action) * self.target_noise_std).clamp(
            -self.target_noise_clip, self.target_noise_clip
        )
        next_action = (self.target_actor(next_state) + noise).clamp(-1.0, 1.0)
        next_q1 = self.target_critic1(next_state, next_action)
        next_q2 = self.target_critic2(next_state, next_action)
        min_next_q = torch.min(next_q1, next_q2)
        target_q = reward + (1 - done) * self.gamma * min_next_q

    critic_loss1 = F.mse_loss(target_q, self.critic1(state, action))
    self.critic_optimizer1.zero_grad()
    critic_loss1.backward()
    self.critic_optimizer1.step()

    critic_loss2 = F.mse_loss(target_q, self.critic2(state, action))
    self.critic_optimizer2.zero_grad()
    critic_loss2.backward()
    self.critic_optimizer2.step()

    max_Q = torch.max(target_q, axis=0).values.cpu().numpy()[0]

    # Delayed Actor Update
    if self.num_learn % self.update_delay == 0:
        action_pred = self.actor(state)
        actor_loss = -self.critic1(state, action_pred).mean()

        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()
        self.actor_loss = actor_loss.item()
        if self.num_learn > 0:
            self.update_target_soft()

...
```

```
### Soft update function or networks ###

def update_target_soft(self):
    for t_p, p in zip(self.target_critic1.parameters(), self.critic1.parameters()):
        t_p.data.copy_(self.tau * p.data + (1 - self.tau) * t_p.data)
    for t_p, p in zip(self.target_critic2.parameters(), self.critic2.parameters()):
        t_p.data.copy_(self.tau * p.data + (1 - self.tau) * t_p.data)
    for t_p, p in zip(self.target_actor.parameters(), self.actor.parameters()):
        t_p.data.copy_(self.tau * p.data + (1 - self.tau) * t_p.data)
```

References

Relevant papers

- Addressing Function Approximation Error in Actor-Critic Methods
  (Fujimoto et al, 2018)

Public implementations

- TD3 release repository
  (sfujim)