DDPG(Deep Deterministic Policy Gradient)

Paper Link: https://arxiv.org/pdf/1509.02971.pdf

Key Features

- DDPG(Deep Deterministic Policy Gradient) is a deterministic policy gradient based actor-critic algorithm.
- DDPG introduces an off-policy algorithm learns by samples which are extracted from behavior policy to ensure exploration.
- DDPG is an algorithm that has strengths in solving tasks with continuous action space.
- DDPG's performance can significantly exceed stochastic policy in high-dimensional action space.

Background

Limit of prior algorithms with respect to continuous action space

DQN(Deep Q-network) has poor performance in continuous action space, contrary to its performance in discrete action space. As a solution to this, a 'discretization of action space' method has been proposed, but this is also not a good method because of the curse of the dimension. The actor-critic algorithm has been suggested as an algorithm presented to solve these problems. However, naive application of actor-critic algorithm

with neural network on the task is not sufficient to solve the problem, because this type of training is unstable.

DDPG = DQN + DPG(Deterministic Policy Gradient)

DDPG is an algorithm that combines DQN and DPG. Then, what is DPG? DPG is a policy gradient based algorithm that solves problems using deterministic policy on task which has continuous action space. Originally, policy gradient algorithms are widely used in RL problems with continuous action space, so that are mainly learned based on stochastic policy. However, stochastic policy has a problem that the cost of the computation is higher than that of deterministic policy. Thus, usage of deterministic policy has an advantage in this computational amount. The disadvantage of using deterministic policy compared to stochastic policy is that it is less exploration. To solve this problem, DPG algorithm introduces stochastic behavior policy to sample various samples.

Deep Deterministic Policy Gradient

In summary, DDPG combines deterministic policy based actor-critic framework and off-policy learning technique which are replay buffer and target network. Due to this structure, it is possible to learn the task whose action space is continuous with a smaller amount of computation. In addition, sample bias can decrease because of using replay buffer.

Method

Policy mapping by DPG based actor-critic algorithm

As with Q-learning, introducing nonlinear function approximation means that convergence guarantee is no longer possible. But, it is an essential element for learning about environments with large-scale state space and large-scale action space. Accordingly, the DDPG uses neural networks to learn these environment. DDPG algorithm maintains a parameterized actor μ which specifies the current policy by

deterministically mapping states to a specific action. The critic Q is learned using the Bellman equation as in Q-learning. The actor is updated by chain rule to the expected return from the start distribution J with respect to the actor parameters.

$$\nabla_{\theta\mu} J \approx \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_{\theta\mu} Q(s, a|\theta^Q)|s = s_t, a = \mu(s_t|\theta^\mu)] \tag{1}$$
$$= \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_a Q(s, a|\theta^Q)|s = s_t, a = \mu(s_t) \nabla_{\theta_\mu} \mu(s|\theta^\mu)|s = s_t] \tag{2}$$

One thing to keep in mind in the process of learning with neural networks is, It is assumed that the samples used during learning are IID(Independent and Identically Distributed). Therefore, mini-batch learning is more essential than on-policy learning. Therefore, DDPG introduces replay buffer to conduct offline learning like DQN.

Additional technique on DDPG

DDPG uses the following techniques to improve learning stability and performance.

The first method is that DDPG performs soft target update by setting a ratio τ. Because periodically updating the entire parameters of target network can destabilize learning. This slows the learning speed but increases the learning stability.

The second method is to use a noise policy in which noise is added to the actor policy to increase exploration.

Algorithm

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Implementation on JORLDY

- DDPG JORLDY Implementation

```
### act function ###
# extract action by usage of OU_Noise for exploration

@torch.no_grad()
def act(self, state, training=True):
    self.actor.train(training)
    mu = self.actor(self.as_tensor(state))
    mu = mu.cpu().numpy()
    action = mu + self.OU.sample().clip(-1.0, 1.0) if training else mu
    return {"action": action}
```

```
### learn function ###
# 1. calculate target Q
# 2. calculate critic_loss and optimize critic_network
# 3. calculate actor_loss and optimize actor_network

def learn(self):
...

    # Critic Update
    with torch.no_grad():
        next_action = self.target_actor(next_state)
        next_q = self.target_critic(next_state, next_action)
        target_q = reward + (1 - done) * self.gamma * next_q
    q = self.critic(state, action)
    critic_loss = F.mse_loss(target_q, q)

    self.critic_optimizer.zero_grad()
    critic_loss.backward()
    self.critic_optimizer.step()

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

    # Actor Update
    action_pred = self.actor(state)
    actor_loss = -self.critic(state, action_pred).mean()

    self.actor_optimizer.zero_grad()
    actor_loss.backward()
    self.actor_optimizer.step()

...
```

```
### target update function ###
# new policy = tau * new_policy + (1-tau) * old_policy

def update_target_soft(self):
    for t_p, p in zip(self.target_critic.parameters(), self.critic.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

- Deterministic Policy Gradient Algorithms
  (D. Silver et al, 2014)

Public implementations

- Baselines
- rllab
- rllib (Ray)
- TD3 release repo