Deep Deterministic Policy Gradient

> Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D., 2015. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971.

1. Motivation

- While DQN solves the problem with high-dimensional **observation spaces**, it can only handle **discrete** and low-dimensional **action spaces**.
- However, interesting problems, such as physical control tasks, have continous and high dimensiona action spaces.
- An obvious approach to adop DQN to such continous domain is to simply discritize the action space. However, there are many drawbacks:
    - The number of actions increases exponentially with the number of degrees of freedom (DOF). For example, for 7-DOF system, if each action is discretized with 10-bins, then we have 10^7 number of actions. The situation is even worse if we want to do fine-grained control.
    - Such large action spaces are difficult to explore efficiently, such training DQN is intractable.
    - Naive discritization throws away information about the structure of action domain, which may be essential for solving the problems.

2. Deep Deterministic Policy Gradient

- Recall that in DQN, we build an action-value function: $Q(s, a|\theta) : R^n \rightarrow R^k$ with n is the number of state and k number of discrete action, to reflect the quality of each action a at the state s. Since we only have k actions, the optimal one can be found by computing the value for all actions, and then select one with highest value:

$$a_{opt} = \backslash\mathrm{argmax}_a Q(s, a|\theta)$$

- However, when a is continuous, solving the maximization is no longer easy. A possible way is by taking derivative and set to 0, i.e:

$$\nabla_a Q(s, a_{opt}) = 0$$

- In order to solve this, Q must be differentiable w.r.t a, and this is exactly how DDPG extends the DQN:
    - Instead of solving a that maximizes Q, we create another network $A(s|\phi)$ to directly predict the optimal action a. For a system with n states, and the agent has m DOF, network $A(s|\phi)$ maps from R^n to R^m.
    - The network $A(s|\phi)$ is trained to maximize $Q(s, A(s|\phi)|\theta)$ simply by gradient descent:

$$\phi_{t+1} = \phi_t + \alpha \nabla_\phi Expected[-Q(s, A(s|\phi))]$$

      Note, we use gradient descent on **negative** $Q(s, A(s|\phi)$ since our objective is to **maximize** the expected rewards in Q.
- Note that, $Q(s, A(s|\phi)|\theta)$ can be written as $Q(s|\theta, \phi)$ to express that it depends on two sets of network parameters θ and ϕ. However, they are decoupled, such that:

- The network $Q(s,a|\theta)$ is called `Critic` since it evaluates the quality of action a. It is updated exactly as in DQN by treating action a (and so on for ϕ) as constant. Different with DQN, in DDPG, $Q(s,a|\theta) : R^{n+m} \rightarrow R$ has $concat([s,a])$ as input, and **outputs a scalar**.
- The network $A(s|\phi)$ is called `Actor` since it directly makes the decision. It is updated by treating θ as constant.
- In training implementation, we used two separated optimizers, one for each network.
- In inference, only network $A(a|\phi)$ is used to predict action, and $Q(s,a|\theta)$ can be safely discarded.

3. What makes DDPG work:

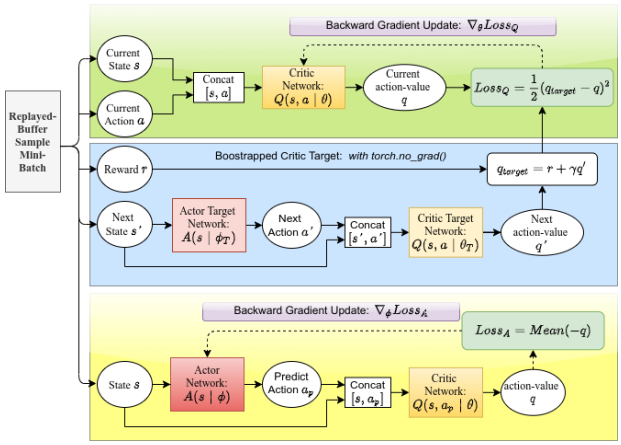
- Since DDPG is an extension of DQN, it inherits the keys of DQN, namely `TargetNetwork` and `Replayed Buffer`.
- Here, we need two target networks A' and Q' for the main networks A and Q, respectively. Slightly different with DQN, instead of update the target networks by snapshot of the main networks once every τ steps, we update it once per main network update by momentum (a.k.a `polyak`) average

$$\theta_{target} \leftarrow \rho\theta_{target} + (1-\rho)\theta$$

$$\phi_{target} \leftarrow \rho\phi_{target} + (1-\rho)\phi$$

  where $\rho \in [0,1]$ is called `polyak`, and often close to 1, .e.g 0.99.
- An important key of RL is the balance of Exploration vs Exploitation.
  - Since the actor network A outputs the action directly, we add a small Gaussian noise $N(0,\sigma)$ to the action a for random exploration during training.
  - In addition, to increase the exploration level, at the first T steps, we can choose the action randomly by uniform sampling from action space.
  - For testing, we stricly follow the policy, and do not add noise to the actions.

4. DDPG Pseudo Code



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:

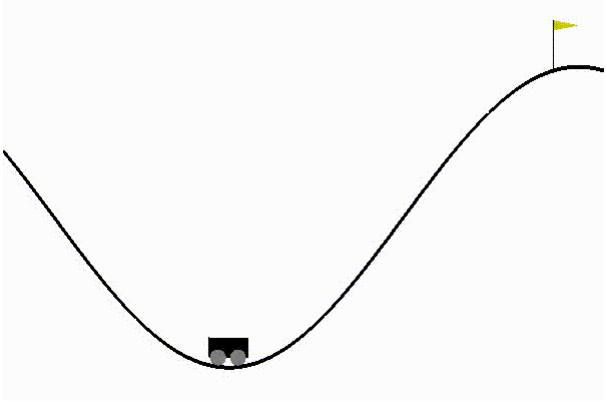
$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    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

To train DDPG agent for Mountain Car Continous problem, do:

```
python tools/train.py configs/DDPG/ddpg_mountaincar_continuous.py
```

Result after training 200 episodes: