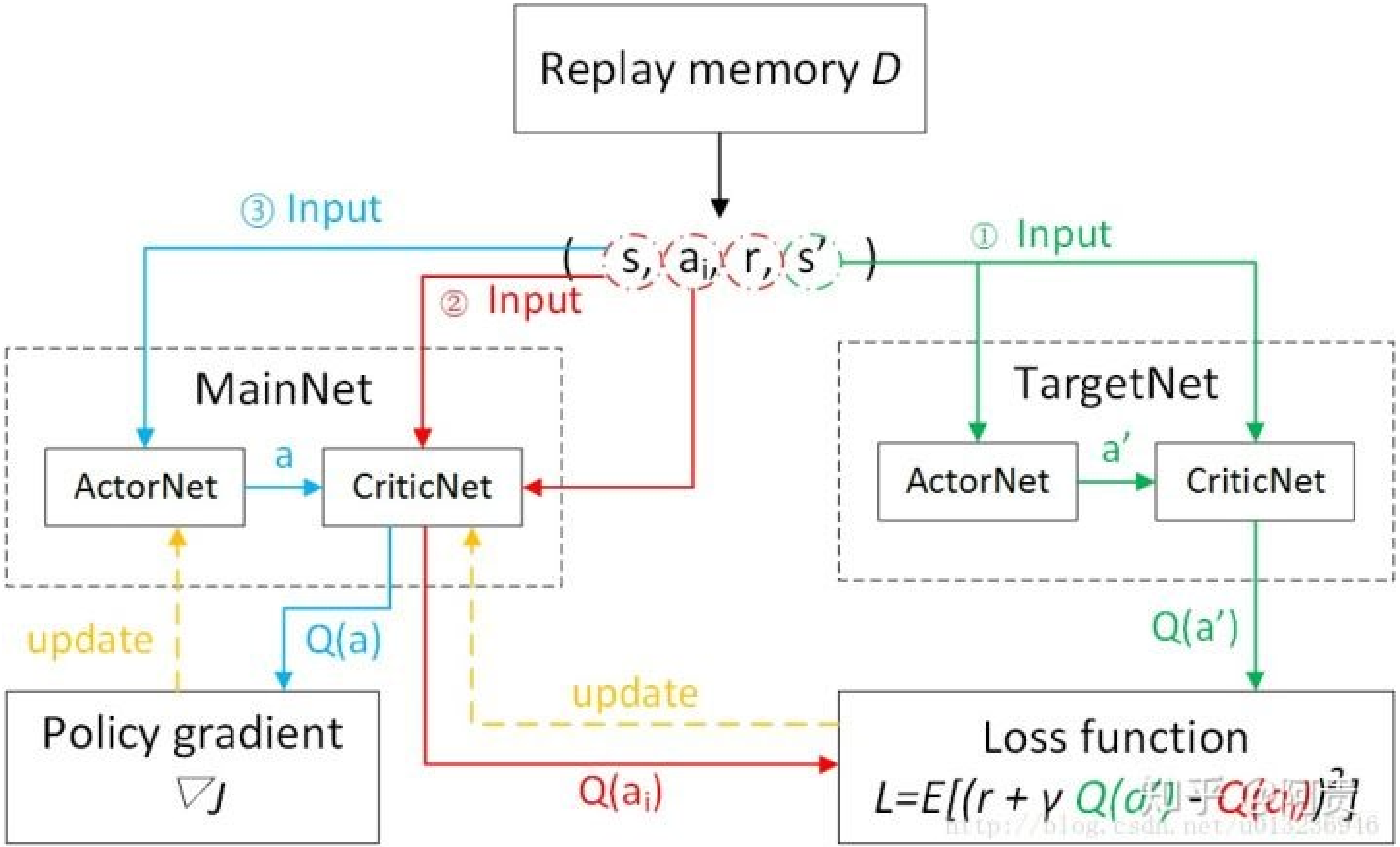
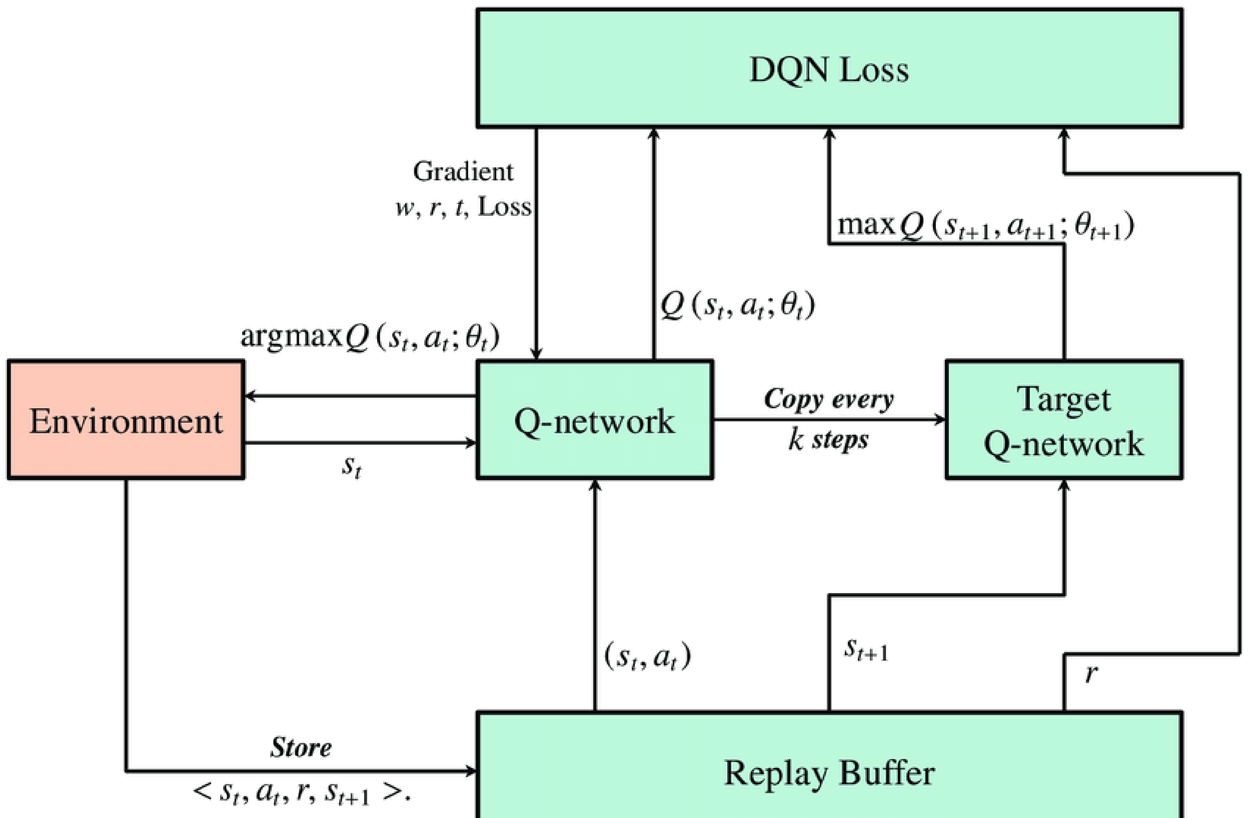
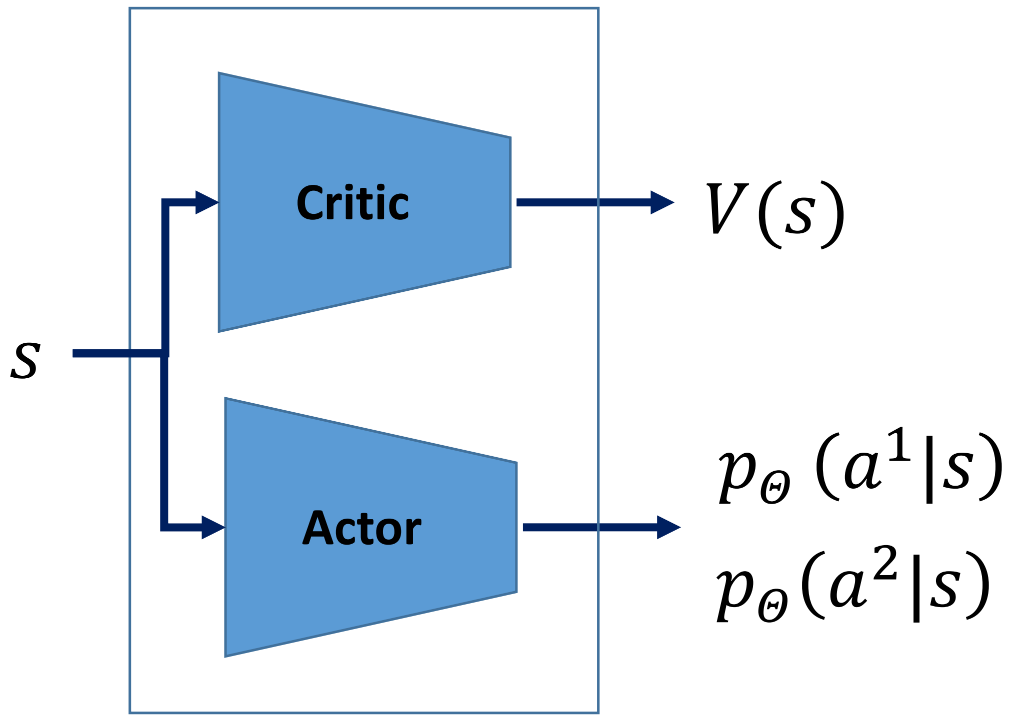
Deep deterministic policy gradient



圖片來源: https://zhuanlan.zhihu.com/p/47873624

Comparison – DQN and PPO-AC





Introduction

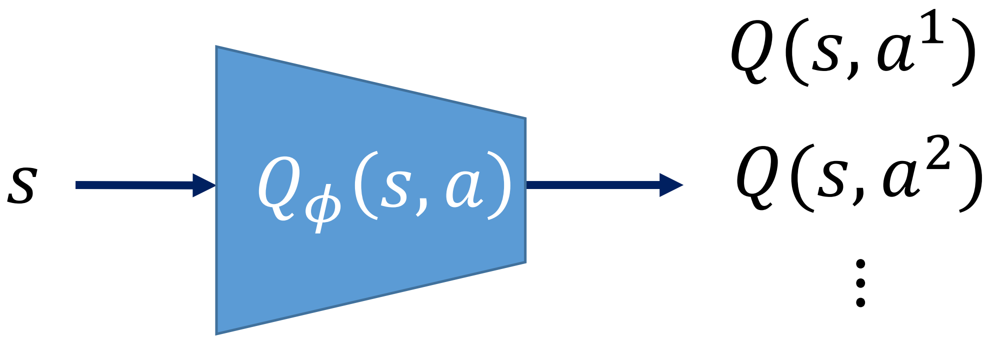
Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

$$Q^*(s,a) = \mathbb{E}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]$$

$$a^*(s) = arg \max_{a} Q^*(s,a)$$

Reference: https://spinningup.openai.com/en/latest/algorithms/ddpg.html#deep-deterministic-policy-gradient

The Q-learning side of DDPG – Critic Net



Critic Net $\quad Q_\phi(s,a), Q_{\phi_{targ}}(s',a')$

$$L(\phi,\mathcal{D}) = \underset{(s,a,r,s',a')\sim\mathcal{D}}{\mathbb{E}}\left[\left(Q_\phi(s,a) - \ (r_s^a + \gamma(1-d)\max_{a'} Q_{\phi_{targ}}(s',a')\right)^2\right]$$

Trick one – replay buffer $\mathcal{D}$

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',a')\boxed{\sim\mathcal{D}}}{\mathbb{E}}\left[\left(Q_\phi(s,a) - \left(r_s^a + \gamma(1-d)\max_{a'} Q_\phi(s',a')\right)\right)^2\right])$$

In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences, but it may not always be good to keep everything. If you only use the very-most recent data, you will overfit to that and things will break; if you use too much experience, you may slow down your learning.

9. DDPG_NN_and_MemoryBuffer.ipynb

```
class ReplayBuffer:
    """Fixed-size buffer to sto

    def __init__(self, action_s
        """Initialize a ReplayE
```

Trick two – Target Network ϕ_{targ}

$$L(\phi,\mathcal{D}) = \underset{(s,a,r,s',a')\sim\mathcal{D}}{\mathbb{E}}\left[\left(Q_\phi(s,a) - \left(r_s^a + \boxed{\gamma(1-d)\max_{a'} Q_{\phi_{targ}}(s',a')}\right)\right)^2\right])$$

Eval network ϕ

Target network ϕ_{targ}

9. DDPG_Agent.ipynb

```
self.critic_local = Critic(state_size, action_size, r
self.critic_target = Critic(state_size, action_size,
self.critic_optimizer = optim.Adam(self.critic_local.
```

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

```
self.soft_update(self.critic_local, self.critic_target, TAU)
self.soft_update(self.actor_local, self.actor_target, TAU)
```

DDPG: calculating Max over actions in the Target

Computing the maximum over actions in the target is a challenge in continuous action spaces. DDPG deals with this by using a target policy network to compute an action which approximately maximizes $Q_{\phi_{targ}}$. The target policy network is found the same way as the target Q-function: by polyak averaging the policy parameters over the course of training.

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',a') \sim \mathcal{D}}{\mathbb{E}} \left[\left(Q_{\phi}(s,a) - \ (r_s^a + \gamma(1-d) Q_{\phi_{targ}}(s', \mu_{\theta_{targ}}(s') \right)^2 \right])$$

$\mu_{\theta_{targ}}$: target policy network

DDPG: calculating Max over actions in the Target

$$L(\phi,\mathcal{D}) = \underset{(s,a,r,s',a')\sim\mathcal{D}}{\mathbb{E}}\left[\left(Q_\phi(s,a) - \ (r_s^a + \gamma(1-d)Q_{\phi_{targ}}(s',\mu_{\theta_{targ}}(s')\right)^2\right])$$

```
states, actions, rewards, next_states, dones = experiences

actions_next = self.actor_target(next_states)
Q_targets_next = self.critic_target(next_states, actions_next)

Q_targets = rewards + (GAMMA * Q_targets_next * (1 - dones))

Q_expected = self.critic_local(states, actions)
critic_loss = F.mse_loss(Q_expected, Q_targets)

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

$\mu_{\theta_{targ}}(s')$

$Q_{\phi_{targ}}(s',\mu_{\theta_{targ}}(s')$

$(r_s^a + \gamma(1-d)Q_{\phi_{targ}}(s',\mu_{\theta_{targ}}(s')$

$Q_\phi(s,a)$

9. DDPG_Agent.ipynb

The policy learning side of DDPG

$$\max_{\theta} \mathbb{E}_{s \backsim \mathcal{D}}\left[Q_{\phi}(s, \mu_{\theta}(s))\right]$$

actions_pred = self.actor_local(states) $\mu_{\theta}(s)$

actor_loss = -self.critic_local(states, actions_pred).mean() $Q_{\phi}(s, \mu_{\theta}(s))$

self.actor_optimizer.zero_grad()

actor_loss.backward()

self.actor_optimizer.step()

9. DDPG_Agent.ipynb

Exploration vs. Exploitation

Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals. To make DDPG policies explore better, we add noise to their actions at training time.

9. DDPG_Agent.ipynb

```
#adding noise for exploration!
if add_noise:
    acts += self.noise.sample()
```

DDPG algorithm

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$

Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$

repeat

    Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$

    Execute a in the environment

    Observe next state s', reward r, and done signal d to indicate whether s' is terminal

    Store (s, a, r, s', d) in replay buffer $\mathcal{D}$

    If s' is terminal, reset environment state.

    if it's time to update **then**

        for however many updates **do**

            Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$

            Compute targets

$$y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

DDPG algorithm

Update Q-function by one step of gradient descent using

$$\nabla_{\phi} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi}(s,a) - y(r,s',d)\right)^2$$

Update policy by one step of gradient ascent using

$$\nabla_{\theta} \frac{1}{|B|} \sum_{s \in B} Q_{\phi}(s, \mu_{\theta}(s))$$

Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1-\rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1-\rho)\theta$$

end for

end if

until convergence