Chapter 6
Combine Deep Q-Networks with Actor-Critic

Hongming Zhang, Tianyang Yu, and Ruitong Huang

Abstract The deep Q-network algorithm is one of the most well-known deep reinforcement learning algorithms, which combines reinforcement learning with deep neural networks to approximate the optimal action-value functions. It receives only the pixels as inputs and achieves human-level performance on Atari games. Actor-critic methods transform the Monte Carlo update of the REINFORCE algorithm into the temporal-difference update for learning the policy parameters. Recently, some algorithms that combine deep Q-networks with actor-critic methods such as the deep deterministic policy gradient algorithm are very popular. These algorithms take advantages of both methods and perform well in most environments especially with continuous action spaces. In this chapter, we give a brief introduction of the advantages and disadvantages of each kind of method, then introduce some classical algorithms that combine deep Q-networks and actor-critic like the deep deterministic policy gradient algorithm, the twin delayed deep deterministic policy gradient algorithm, and the soft actor-critic algorithm.

Keywords Deep Q-network · Actor-critic · Deep deterministic policy gradient · Twin delayed deep deterministic policy gradient · Soft actor-critic

6.1 Introduction

The deep Q-network (DQN) (Mnih et al. 2015) algorithm is a classical off-policy method. It combines the Q-learning algorithm with a deep neural network to realize end-to-end learning from visual inputs to decision outputs. This algorithm has

H. Zhang (✉)
Peking University, Beijing, China
e-mail: zhanghongming@pku.edu.cn

T. Yu
Nanchang University, Nanchang, China

R. Huang
Borealis AI, Toronto, ON, Canada

H. Dong et al. (eds.), *Deep Reinforcement Learning*,
https://doi.org/10.1007/978-981-15-4095-0_6

Table 6.1 Characteristics of DQN and actor-critic

Algorithm	On-policy/off-policy	Sample efficiency	Action space
DQN	Off-policy	High	Discrete
Actor-critic	On-policy	Low	Continuous
DQN+Actor-critic	Off-policy	High	Discrete and continuous

achieved human-level performance on Atari games using raw pixel inputs. However, while the inputs can be unprocessed and high-dimensional observation spaces, DQN can only handle discrete and low-dimensional action spaces. For continuous and high-dimensional action spaces, DQN fails to calculate the Q value of each action.

The actor-critic (AC) (Sutton and Barto 2018) method is an extension of the REINFORCE (Sutton and Barto 2018) algorithm. By incorporating a critic, this method transforms policy gradient's Monte Carlo update into a temporal-difference update. Through this multi-step method, the degree of bootstrapping can be flexibly selected, so the update of policy does not need to wait until the end of the game. Though some bias will be introduced in temporal-difference update, it can reduce variances and accelerate the learning. However, the original actor-critic method is still an on-policy algorithm, and the sample efficiency of on-policy methods is much lower than off-policy methods.

Combining DQN with actor-critic can take advantages of both algorithms. Because of DQN, actor-critic methods are transformed into off-policy methods. Networks can be trained with samples from a replay buffer that improves sample efficiency. Sampling from a replay buffer can also minimize correlations between samples, which can learn value functions in a stable and robust way. Also, due to the actor-critic method, we can easily handle problems with continuous action spaces by using a network to learn a policy π (Table 6.1).

Next, we introduce some classical algorithms: the deep deterministic policy gradient (DDPG) algorithm (Lillicrap et al. 2015), and its improvements: the twin delayed deep deterministic policy gradient (TD3) algorithm (Fujimoto et al. 2018) and the soft actor-critic (SAC) algorithm (Haarnoja et al. 2018a).

6.2 Deep Deterministic Policy Gradient (DDPG)

The deep deterministic policy gradient (DDPG) algorithm can be regarded as a combination of the deterministic policy gradient (DPG) algorithm (Silver et al. 2014) and deep neural networks; The DDPG algorithm can also be viewed as an extension of the DQN algorithm in continuous action space. It wants to tackle the problem with continuous action spaces that DQN cannot be straightforwardly applied to. DDPG establishes a Q function (critic) and a policy function (actor) simultaneously. The Q function (critic) is the same with DQN, temporal-difference methods (TD methods) are used to update it. The policy gradient algorithm is used to update the policy function (actor) through the value from Q function (critic).

In DDPG, the actor is a deterministic policy function, denoted as $\pi(s)$, and the parameter is denoted as θ^{π}. The action of each step is calculated directly by $A_t = \pi(S_t|\theta_t^{\pi})$, which does not need to sample from a stochastic policy.

A critical problem here is how to balance exploration and exploitation with this deterministic policy. In DDPG, noises sampled from a noise process N are added to actions when training. The action is $A_t = \pi(S_t|\theta_t^{\pi})+N_t$. N can be chosen according to the specific task; the original paper uses an Ornstein–Uhlenbeck process (O–U process) (Uhlenbeck and Ornstein 1930).

The O–U process satisfies the following stochastic differential equation:

$$\mathrm{d}X_t = \theta(\pi - X_t)\mathrm{d}t + \sigma \mathrm{d}W_t, \tag{6.1}$$

where X_t is a random variable, $\theta > 0, x, \sigma > 0$ are parameters. W_t is a Wiener process or named Brownian Motion (It and McKean 1965), which has the following properties:

- W_t is a process with independent increments, which means for times $T_0 < T_1 < \ldots < T_n$, random variables $W_{T_0}, W_{T_1} - W_{T_0}, \ldots, W_{T_n} - W_{T_{n-1}}$ are independent.
- For any time t and Δt, $W(t + \Delta_t) - W(t) \sim N(0, \sigma_W^2 \Delta t)$.
- W_t is a continuous function about t.

We know the Markov Decision Process (MDP) is based on Markov Chains; MDP satisfies the property $p(X_{t+1}|X_t, \ldots, X_1) = p(X_{t+1}|X_t)$, where X_t is a random variable at time step t. This means the random variable X_t is conditioned on the last time step's random variable X_{t-1}, which is time-correlated. The O–U noise is also time-correlated, which conforms to the property of Markov Decision Process (MDP). However, more recent results suggest that time-uncorrelated, mean-zero Gaussian noise also works well.

Back to the algorithm, the action-value function $Q(s, a|\theta^Q)$ is learned using the Bellman equation as in DQN.

In the state S_t, the next state S_{t+1} and the return R_t are obtained by executing action $A_t = \pi(S_t|\theta_t^{\pi})$ through the policy π. We have

$$Q^{\pi}(S_t, A_t) = \mathbb{E}[r(S_t, A_t) + \gamma Q^{\pi}(S_{t+1}, \pi(S_{t+1}))]. \tag{6.2}$$

Then we can compute the Q value:

$$Y_i = R_i + \gamma Q^{\pi}(S_{t+1}, \pi(S_{t+1})). \tag{6.3}$$

Using gradient descent to minimize the loss function:

$$L = \frac{1}{N}\sum_i \left(Y_i - Q(S_i, A_i|\theta^Q)\right)^2. \tag{6.4}$$

The policy function π is updated by applying the chain rule to the expected return from the start distribution J. Here, $J = \mathbb{E}_{R_i, S_i \sim E, A_i \sim \pi}[R_1]$ (E denotes the environment) and $R_t = \sum_{i=t}^{T} \gamma^{(i-t)} r(S_i, A_i)$. We have

$$\begin{aligned} \nabla_{\theta^\pi} J \approx & \mathbb{E}_{S_t \sim \rho^\beta} \left[\nabla_{\theta^\pi} Q \left(s, a | \theta^Q \right) |_{s=S_t, a=\pi(S_t|\theta^\pi)} \right], \\ = & \mathbb{E}_{S_t \sim \rho^\beta} \left[\nabla_a Q \left(s, a | \theta^Q \right) |_{s=S_t, a=\pi(S_t)} \nabla_{\theta_\pi} \pi \left(s | \theta^\pi \right) |_{s=S_t} \right]. \end{aligned} \tag{6.5}$$

Learning in mini-batches:

$$\nabla_{\theta^\pi} J \approx \frac{1}{N} \sum_i \nabla_a Q \left(s, a | \theta^Q \right) |_{s=S_i, a=\pi(S_i)} \nabla_{\theta^\pi} \pi \left(s | \theta^\pi \right) |_{S_i}. \tag{6.6}$$

In addition, DDPG adopts a similar way of the target network like DQN, but it updates network parameters by exponential smoothing rather than directly copying the parameters:

$$\theta^{Q'} \leftarrow \rho \theta^Q + (1-\rho) \theta^{Q'}, \tag{6.7}$$

$$\theta^{\pi'} \leftarrow \rho \theta^\pi + (1-\rho) \theta^{\pi'}. \tag{6.8}$$

Since the hyperparameter $\rho \ll 1$ here, the target network changes very slowly and smoothly, which improves the stability of learning.

The whole pseudocode shows in Algorithm 1.

6.3 Twin Delayed Deep Deterministic Policy Gradient (TD3)

The twin delayed deep deterministic policy gradient (TD3) algorithm is an improvement of DDPG, where three critical techniques are used:

1. Clipped double Q-learning for actor-critic: learn two Q-value functions, which is similar to double Q-learning.
2. Target networks and delayed policy updates: update the policy (and the target network) less frequently than the Q-value function.
3. Target policy smoothing regularization: Add noise to the target action to smooth the Q-value function and avoid overfitting.

For the first technique, we know that in DQN, there is an overestimation problem due to the existence of the max operation, this problem also exists in DDPG, because $Q(s, a)$ is updated in the same way as DQN

$$Q(s, a) \leftarrow R_s^a + \gamma \max_{\hat{a}} Q \left(s', \hat{a} \right). \tag{6.9}$$

Algorithm 1 DDPG

```
**Hyperparameters**: soft update factor $\rho$, reward discount factor $\gamma$
**Input** empty replay buffer $\mathcal{D}$, initialize parameters $\theta^Q$ of critic network $Q(s, a|\theta^Q)$ and parameters $\theta^\pi$ of actor network $\pi(s|\theta^\pi)$, target network $Q'$ and $\pi'$
Initialize target network $Q'$ and $\pi'$ with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\pi'} \leftarrow \theta^\pi$
**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**
        Selection action $A_t = \pi(S_t|\theta^\pi) + \mathcal{N}_t$
        Execute action $A_t$ and observe reward $R_t$, and observe new state $S_{t+1}$
        Store transion $(S_t, A_t, R_t, D_t, S_{t+1})$ in $\mathcal{D}$
        Set $Y_i = R_i + \gamma(1 - D_t)Q'(S_{t+1}, \pi'(S_{t+1}|\theta^{\pi'})|\theta^{Q'})$
        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:
            $$\nabla_{\theta^\pi} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=S_i, a=\pi(S_i)} \nabla_{\theta^\pi}\pi(s|\theta^\pi)|_{S_i}$$
        Update the target networks:
        $\theta^{Q'} \leftarrow \rho\theta^Q + (1-\rho)\theta^{Q'}$
        $\theta^{\pi'} \leftarrow \rho\theta^\pi + (1-\rho)\theta^{\pi'}$
    **end for**
**end for**
```

This is not a problem in the tabular case, because the Q-values are stored precisely. However, when we use a network as a function approximator in a more complex case, the estimation of Q-value will be noisy. That is to say:

$$Q^{approx}(s', \hat{a}) = Q^{target}\left(s', \hat{a}\right) + Y_{s'}^{\hat{a}}, \tag{6.10}$$

where $Y_{s'}^{\hat{a}}$ is a noise with zero mean. But with the max operator, the noise induces a difference between Q^{approx} and Q^{target}. Denote the difference as Z_s, we have

$$\begin{aligned} Z_s \overset{\text{def}}{=} & R_s^a + \gamma \max_{\hat{a}} Q^{approx}\left(s', \hat{a}\right) - \left(R_s^a + \gamma \max_{\hat{a}} Q^{target}(s', \hat{a})\right), \\ = & \gamma\left(\max_{\hat{a}} Q^{approx}(s', \hat{a}) - \max_{\hat{a}} Q^{target}(s', \hat{a})\right). \end{aligned} \tag{6.11}$$

Considering the noise $Y_{s'}^{\hat{a}}$, some of the Q-values might be too small, while others might be too large. The max operator always picks the largest value for each state, which will make it sensitive to overestimate the correct Q-values for some actions. In this case, this noise will lead to $\mathbb{E}[Z_s] > 0$ and cause the overestimation problem.

The TD3 algorithm incorporates the idea of double Q-learning in DDPG; it establishes two Q-value networks to compute the value of the next state:

$$Q_{\theta_1'}\left(s', a'\right) = Q_{\theta_1'}\left(s', \pi_{\phi_1}(s')\right), \tag{6.12}$$

$$Q_{\theta_2'}\left(s', a'\right) = Q_{\theta_2'}\left(s', \pi_{\phi_1}(s')\right). \tag{6.13}$$

Use the minimum of the two values (clipped) to compute the Bellman equation:

$$Y_1 = r + \gamma \min_{i=1,2} Q_{\theta_i'}\left(s', \pi_{\phi_1}(s')\right). \tag{6.14}$$

With clipped double Q-learning, the value target will not introduce additional overestimation over using the standard Q-learning target. While this update rule may induce an underestimation bias, this is preferable to an overestimation bias. Because unlike overestimated actions, the value of underestimated actions will not be explicitly propagated through the policy update (Fujimoto et al. 2018).

For the second technique, we know that the target network is a good tool to achieve stability in deep reinforcement learning. As deep function approximators require multiple gradient updates to converge, target networks provide a stable objective in the learning procedure and allow better coverage of the training data. So, if target networks can be used to reduce the error over multiple updates, and policy updates on high-error states cause divergent behavior, then the policy network should be updated at a lower frequency than the value network, to first minimize error before introducing a policy update. In this way, the TD3 algorithm reduces the update frequency of the policy function. It only updates the policy and target networks after a fixed number of updates d to the critic. The less frequent policy updates can make the update of Q-value function has a smaller variance, and thus a higher quality policy can be obtained.

For the third technique, a concern with deterministic policies is that they can overfit to narrow peaks in the value estimate. In TD3 paper, the author enforces the notion that similar actions should have similar value, so fitting the value of a small area around the target action makes sense:

$$y = r + \mathbb{E}_{\epsilon}\left[Q_{\theta'}\left(s', \pi_{\phi'}(s') + \epsilon\right)\right]. \tag{6.15}$$

By adding a truncated normal distribution noise to each action as a regularization, the computation of Q-values can be smoothed to avoid overfitting.

This makes a modified target update:

$$y = r + \gamma Q_{\theta'}\left(s', \pi_{\phi'}(s') + \epsilon\right), \epsilon \sim clip(N(0, \sigma), -c, c). \tag{6.16}$$

The whole pseudocode shows in Algorithm 2.

Algorithm 2 TD3

```
1: **Hyperparameters**: soft update factor $\rho$, reward discount factor $\gamma$, clip factor $c$
2: **Input**: empty replay buffer $\mathcal{D}$, initial parameters $\theta_1, \theta_2$ of critic networks $Q_{\theta_1}, Q_{\theta_2}$, initial parameters $\phi$ of actor network $\pi_\phi$
3: Initialize target networks $\hat{\theta}_1 \leftarrow \theta_1, \hat{\theta}_2 \leftarrow \theta_2, \hat{\phi} \leftarrow \phi$
4: **for** $t = 1$ to $T$ do **do**
5:     Select action with exploration noise $A_t \sim \pi_\phi(S_t) + \epsilon, \epsilon \sim \mathcal{N}(0, \sigma)$
6:     Observe reward $R_t$ and new state $S_{t+1}$
7:     Store transition tuple $(S_t, A_t, R_t, D_t, S_{t+1})$ in $\mathcal{D}$
8:     Sample mini-batch of $N$ transitions $(S_t, A_t, R_t, D_t, S_{t+1})$ from $\mathcal{D}$
9:     $\tilde{a}_{t+1} \leftarrow \pi_{\phi'}(S_{t+1}) + \epsilon, \epsilon \sim clip(\mathcal{N}(0, \tilde{\sigma}, -c, c))$
10:    $y \leftarrow R_t + \gamma(1 - D_t) \min_{i=1,2} Q_{\theta_i'}(S_{t+1}, \tilde{a}_{t+1})$
11:    Update critics $\theta_i \leftarrow \arg\min_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(S_t, A_t))^2$
12:    **if** $t$ mod $d$ **then**
13:        Update $\phi$ by the deterministic policy gradient:
14:        $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(S_t, A_t)|_{A_t=\pi_\phi(S_t)} \nabla_\phi \pi_\phi(S_t)$
15:        Update target networks:
16:        $\hat{\theta}_i \leftarrow \rho\theta_i + (1 - \rho)\hat{\theta}_i$
17:        $\hat{\phi} \leftarrow \rho\phi + (1 - \rho)\hat{\phi}$
18:    **end if**
19: **end for**
```

6.4 Soft Actor-Critic (SAC)

The soft actor-critic (SAC) algorithm follows the idea of maximum entropy reinforcement learning, where instead of maximizing the discounted cumulative reward, the optimal policy aims to maximize its entropy regularized reward, thus encouraging the exploration of the policy.

$$\max_{\pi_\theta} \mathbb{E}\left[\sum_t \gamma^t \left(r(S_t, A_t) + \alpha\mathcal{H}(\pi_\theta(\cdot|S_t))\right)\right], \tag{6.17}$$

where α is the regularization coefficient. Maximum entropy reinforcement learning has been well explored in the literature (Ziebart et al. 2008; Levine and Koltun 2013; Fox et al. 2016; Nachum et al. 2017; Haarnoja et al. 2017). Here we only introduce the idea of soft policy iteration which serves as the fundamental of the SAC method.

6.4.1 Soft Policy Iteration

Soft policy iteration is a general algorithm for learning the optimal maximum entropy policies with provable guarantees. Similar to policy iteration, soft policy iteration also has two steps: soft policy evaluation and soft policy improvement.

Let

$$V^{\pi}(s) = \mathbb{E}\left[\sum_t \gamma^t \left(r(S_t, A_t) + \alpha\mathcal{H}(\pi(\cdot|S_t))\right)\right], \tag{6.18}$$

where $S_0 = s$. Further let

$$Q(s, a) = r(s, a) + \gamma\mathbb{E}\left[V(s')\right], \tag{6.19}$$

where $s' \sim \Pr(\cdot|s, a)$ is the next state. It is straightforward to verify that

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}\left[Q(s, a) - \alpha\log(a|s)\right]. \tag{6.20}$$

In the soft policy evaluation step, define the Bellman backup operator $\mathcal{T}$ by

$$\mathcal{T}^{\pi}Q(s, a) = r(s, a) + \gamma\mathbb{E}\left[V^{\pi}(s')\right]. \tag{6.21}$$

Similar to policy evaluation, one can prove that for any map $Q^0 : \mathcal{S}\times\mathcal{A}\to\mathbb{R}$, Q^k will converge to the soft Q-value of π, where $Q^k = \mathcal{T}^{\pi}Q^{k-1}$.

In the policy improvement step, we solve the entropy regularized reward maximization problem with the current Q values.

$$\pi(\cdot|s) = \arg\max_{\pi}\mathbb{E}_{a\sim\pi}\left[Q(s, a) + \alpha\mathcal{H}(\pi)\right]. \tag{6.22}$$

Solving the above optimization problem (Fox et al. 2016; Nachum et al. 2017), one can get that

$$\pi(\cdot|s) = \frac{\exp\left(\frac{1}{\alpha}Q(s, \cdot)\right)}{Z(s)}, \tag{6.23}$$

where $Z(s)$ is the normalizing factor, i.e. $Z(s) = \sum_a \exp\left(\frac{1}{\alpha}Q(s, a)\right)$. Given that the optimal π may not be representable in the policy model, we instead update the policy by

$$\pi(\cdot|s) = \arg\min_{\pi\in\Pi} D_{\mathrm{KL}}\left(\pi(\cdot|s)\|\frac{\exp\left(\frac{1}{\alpha}Q(s, \cdot)\right)}{Z(s)}\right). \tag{6.24}$$

Not surprising, one can also prove the policy monotonically improvement property for the above soft policy improvement step, even with the projection to Π using KL-divergence. The next theorem shows that soft policy iteration, similar to policy iteration, converges to the optimal solution.

Theorem 6.1 *Let $\pi_0 \in \Pi$ be any initialized policy. Assume that by performing soft policy iteration steps, π_0 converges to $\pi*$. Then $Q^{\pi*}(s, a) \geq Q^{\pi}(s, a)$ for any $(s, a) \in \mathcal{S} \times \mathcal{A}$ and any $\pi \in \Pi$.*

We omit all the proofs of this section of this book. Interested readers can refer to (Haarnoja et al. 2018b) for more details.

6.4.2 SAC

SAC extends soft policy iteration to the setting with function approximation which is more practical. Instead of estimating the true Q value of the policy π for policy improvement, SAC performs an alternative optimization on both the value function and the policy.

Consider a parametrized Q function $Q_{\phi}(s, a)$ and policy π_{θ}. Here we consider the continuous action setting, where the output of π_{θ} is a Gaussian mean and covariance. Similarly, the Q function can be learned by minimizing the soft Bellman residual,

$$J_Q(\phi) = \mathbb{E}\left[\left(Q(S_t, A_t) - r(S_t, A_t) - \gamma \mathbb{E}_{S_{t+1}}\left[V_{\tilde{\phi}}(S_{t+1})\right]\right)^2\right]. \tag{6.25}$$

where $V_{\tilde{\phi}}(s) = \mathbb{E}_{\pi_{\theta}}\left[Q_{\tilde{\phi}}(s, a) - \alpha \log \pi_{\theta}(a|s)\right]$, and $Q_{\tilde{\phi}}$ is a target Q network, whose parameter $\tilde{\phi}$ is obtained as an exponentially moving average of ϕ. Moreover, the policy π_{θ} can be learned by minimizing the expected KL-divergence.

$$J_{\pi}(\theta) = \mathbb{E}_{s \sim \mathcal{D}}\Big[\mathbb{E}_{a \sim \pi_{\theta}}\left[\alpha \log \pi_{\theta}(a|s) - Q_{\phi}(s, a)\right]\Big]. \tag{6.26}$$

In practice, SAC uses two Q-networks (as well as two target Q-networks) to mitigate the biased Q value problem, i.e. $Q_{\phi}(s, a) = \min\left(Q_{\phi_1}(s, a), Q_{\phi_2}(s, a)\right)$. Note that $J_{\pi}(\theta)$ has the expectation taken on π_{θ}. To optimize $J_{\pi}(\theta)$, one option is to use the idea of likelihood ratio gradient estimator (Williams 1992). However, in the continuous action setting, one can instead use the reparametrization trick for the policy network, which usually results in a lower variance estimator. To do that, we reparametrize π_{θ} as an action network taking both state s and a standard Gaussian noise ϵ as its input.

$$a = f_{\theta}(s, \epsilon). \tag{6.27}$$

Plugging into $J_{\pi}(\theta)$,

$$J_{\pi}(\theta) = \mathbb{E}_{s \sim \mathcal{D}, \epsilon \sim \mathcal{N}}\left[\alpha \log \pi_{\theta}(f_{\theta}(s, \epsilon)|s) - Q_{\phi}(s, f_{\theta}(s, \epsilon))\right], \tag{6.28}$$

where $\mathcal{N}$ is the standard Gaussian distribution, and π_θ is now defined implicitly in terms of f_θ.

Finally, SAC also provides a way in automatically update the regularization coefficient α, by minimizing the following loss:

$$J(\alpha) = \mathbb{E}_{a \sim \pi_\theta} \left[-\alpha \log \pi_\theta(a|s) - \alpha\kappa \right], \tag{6.29}$$

where κ is a hyperparameter interpreted as the target entropy. Such updating schemes for α are also called automating entropy adjustments. The intuition behind $J(\alpha)$ is the dual form of the original policy optimization problem with the constraint that the average entropy at each time step should be at least κ. For more rigorous statement around automating entropy adjustment, please refer to the SAC paper (Haarnoja et al. 2018b). We summarize the SAC algorithm in Algorithm 3.

Algorithm 3 Soft actor-critic (SAC)

```
Hyperparameters: target entropy $\kappa$, step sizes $\lambda_Q$, $\lambda_\pi$, $\lambda_\alpha$, exponentially moving average coefficient $\tau$
Input: initial policy parameters $\theta$, initial Q value function parameters $\phi_1$ and $\phi_2$
$\mathcal{D} = \emptyset$; $\tilde{\phi}_i = \phi_i$, for $i = 1, 2$
for $k = 0, 1, 2, \ldots$ do
    for $t = 0, 1, 2, \ldots$ do
        Sample $A_t$ from $\pi_\theta(\cdot|S_t)$, collect $(R_t, S_{t+1})$
        $\mathcal{D} = \mathcal{D} \cup \{S_t, A_t, R_t, S_{t+1}\}$
    end for
    Perform multiple step of gradients:
        $\phi_i = \phi_i - \lambda_Q \nabla J_Q(\phi_i)$ for $i = 1, 2$
        $\theta = \theta - \lambda_\pi \nabla_\theta J_\pi(\theta)$
        $\alpha = \alpha - \lambda_\alpha \nabla J(\alpha)$
        $\tilde{\phi}_i = (1 - \tau)\phi_i + \tau\tilde{\phi}_i$ for $i = 1, 2$
end for
Output $\theta$, $\phi_1$, $\phi_2$
```

6.5 Examples

This section will share examples of DDPG, TD3, and SAC. They are all actor-critic methods and use a Q-network as a critic. Examples are based on the OpenAI Gym Environment. Since these algorithms are based on continuous action space, “Pendulum-v0” environment is used.

6.5.1 Related Gym Environment

As mentioned above, Pendulum-v0 is a classical inverted pendulum environment with three-dimensional observation space and one-continuous action space. At each step, the environment returns a reward affected by the current rotation angle, speed, and acceleration. The goal of this task is to turn the pendulum upside down to gain the maximum score.

6.5.2 DDPG: Pendulum-v0

DDPG uses off-policy data and TD methods. The structure of DDPG class can be shown as follows:

```
class DDPG(object):
   def __init__(self, action_dim, state_dim, action_range):
      ...
   def ema_update(self):
      ...
   def get_action(self, s, greedy=False):
      ...
   def learn(self):
      ...
   def store_transition(self, s, a, r, s_):
      ...
   def save(self):
      ...
   def load(self):
      ...
```

There are four networks created in the initialization function. They are the actor, critic, actor target, and critic target. The parameters of the target network will be replaced with the corresponding network parameters.

```
class DDPG(object):
   def __init__(self, action_dim, state_dim, action_range):
      self.memory = np.zeros((MEMORY_CAPACITY, state_dim * 2 +
          action_dim + 1), dtype=np.float32)
      self.pointer = 0
      self.action_dim, self.state_dim, self.action_range =
          action_dim, state_dim, action_range
      self.var = VAR

      W_init = tf.random_normal_initializer(mean=0, stddev=0.3)
      b_init = tf.constant_initializer(0.1)
```

```
def get_actor(input_state_shape, name=''):
   input_layer = tl.layers.Input(input_state_shape,
       name='A_input')
   layer = tl.layers.Dense(n_units=64, act=tf.nn.relu,
       W_init=W_init, b_init=b_init,
       name='A_l1')(input_layer)
   layer = tl.layers.Dense(n_units=64, act=tf.nn.relu,
       W_init=W_init, b_init=b_init, name='A_l2')(layer)
   layer = tl.layers.Dense(n_units=action_dim,
       act=tf.nn.tanh, W_init=W_init, b_init=b_init,
       name='A_a')(layer)
   layer = tl.layers.Lambda(lambda x: action_range *
       x)(layer)
   return tl.models.Model(inputs=input_layer,
       outputs=layer, name='Actor' + name)

def get_critic(input_state_shape, input_action_shape,
    name=''):
   state_input = tl.layers.Input(input_state_shape,
       name='C_s_input')
   action_input = tl.layers.Input(input_action_shape,
       name='C_a_input')
   layer = tl.layers.Concat(1)([state_input, action_input])
   layer = tl.layers.Dense(n_units=64, act=tf.nn.relu,
       W_init=W_init, b_init=b_init, name='C_l1')(layer)
   layer = tl.layers.Dense(n_units=64, act=tf.nn.relu,
       W_init=W_init, b_init=b_init, name='C_l2')(layer)
   layer = tl.layers.Dense(n_units=1, W_init=W_init,
       b_init=b_init, name='C_out')(layer)
   return tl.models.Model(inputs=[state_input,
       action_input], outputs=layer, name='Critic' + name)

self.actor = get_actor([None, state_dim])
self.critic = get_critic([None, state_dim], [None,
    action_dim])
self.actor.train()
self.critic.train()

def copy_para(from_model, to_model):
   for i, j in zip(from_model.trainable_weights,
       to_model.trainable_weights):
      j.assign(i)

self.actor_target = get_actor([None, state_dim],
    name='_target')
copy_para(self.actor, self.actor_target)
self.actor_target.eval()

self.critic_target = get_critic([None, state_dim], [None,
    action_dim], name='_target')
copy_para(self.critic, self.critic_target)
self.critic_target.eval()
```

```
    self.ema = tf.train.ExponentialMovingAverage(decay=1 -
        TAU) # soft replacement

    self.actor_opt = tf.optimizers.Adam(LR_A)
    self.critic_opt = tf.optimizers.Adam(LR_C)
```

During the training process, the parameters of the target network will be updated by a moving average.

```
def ema_update(self):
    paras = self.actor.trainable_weights +
        self.critic.trainable_weights
    self.ema.apply(paras)
    for i, j in zip(self.actor_target.trainable_weights +
        self.critic_target.trainable_weights, paras):
        i.assign(self.ema.average(j))
```

As the policy network is a deterministic policy network, we need to add some randomness if we do not choose actions by greedy methods. We used a normal distribution here and the value of variance decreases in the process of the update iteration. The randomness can also be changed to other methods, such as an O–U noise.

```
def get_action(self, state, greedy=False):
    a = self.actor(np.array([s], dtype=np.float32))[0]
    if greedy:
        return a

    # add randomness to action selection for exploration
    return np.clip(np.random.normal(a, self.var),
                   -self.action_range,
                   self.action_range)
```

In the `learn()` function, we sample the off-policy data from the replay buffer and use the Bellman equation to learn the Q-function. After that, the policy can be learned by maximizing the Q-value. Finally, target networks will be updated with Polyak averaging (Polyak 1964) by using formula $\theta^{Q'} \leftarrow \rho\theta^{Q} + (1-\rho)\theta^{Q'}, \theta^{\pi'} \leftarrow \rho\theta^{\pi} + (1-\rho)\theta^{\pi'}$.

```
def learn(self):
    self.var *= .9995
    indices = np.random.choice(MEMORY_CAPACITY,
        size=BATCH_SIZE)
    bt = self.memory[indices, :]
    bs = bt[:, :self.s_dim]
    ba = bt[:, self.s_dim:self.s_dim + self.a_dim]
    br = bt[:, -self.s_dim - 1:-self.s_dim]
    bs_ = bt[:, -self.s_dim:]

    with tf.GradientTape() as tape:
        a_ = self.actor_target(bs_)
```

```
            q_ = self.critic_target([bs_, a_])
            y = br + GAMMA * q_
            q = self.critic([bs, ba])
            td_error = tf.losses.mean_squared_error(y, q)
        c_grads = tape.gradient(td_error,
            self.critic.trainable_weights)
        self.critic_opt.apply_gradients(zip(c_grads,
            self.critic.trainable_weights))

        with tf.GradientTape() as tape:
            a = self.actor(bs)
            q = self.critic([bs, a])
            a_loss = -tf.reduce_mean(q) # maximize the q
        a_grads = tape.gradient(a_loss,
            self.actor.trainable_weights)
        self.actor_opt.apply_gradients(zip(a_grads,
            self.actor.trainable_weights))
        self.ema_update()
```

The `store_transition()` function uses a replay buffer to store the transition of each step.

```
    def store_transition(self, s, a, r, s_):
        s = s.astype(np.float32)
        s_ = s_.astype(np.float32)
        transition = np.hstack((s, a, [r], s_))
        index = self.pointer % MEMORY_CAPACITY # replace the old
            memory with new memory
        self.memory[index, :] = transition
        self.pointer += 1
```

The main function is straightforward, the agent interacts with the environment at each step, stores data into the replay buffer, and uses sampled batch data from the replay buffer to update the networks.

```
    env = gym.make(ENV_ID).unwrapped

    # reproducible
    env.seed(RANDOM_SEED)
    np.random.seed(RANDOM_SEED)
    tf.random.set_seed(RANDOM_SEED)

    state_dim = env.observation_space.shape[0]
    action_dim = env.action_space.shape[0]
    action_range = env.action_space.high # scale action,
        [-action_range, action_range]

    agent = DDPG(action_dim, state_dim, action_range)
    t0 = time.time()

    if args.train: # train
        all_episode_reward = []
    for episode in range(TRAIN_EPISODES):
        state = env.reset()
        episode_reward = 0
        for step in range(MAX_STEPS):
            if RENDER:
                env.render()
            # Add exploration noise
            action = agent.get_action(state)
            state_, reward, done, info = env.step(action)
            agent.store_transition(state, action, reward, state_)
            if agent.pointer > MEMORY_CAPACITY:
                agent.learn()
            state = state_
            episode_reward += reward
            if done:
                break

        if episode == 0:
            all_episode_reward.append(episode_reward)
        else:
            all_episode_reward.append(all_episode_reward[-1] *
                 0.9 + episode_reward * 0.1)
        print(
            'Training | Episode: {}/{} | Episode Reward: {:.4f}
                 | Running Time: {:.4f}'.format(
                episode+1, TRAIN_EPISODES, episode_reward,
                time.time() - t0
            )
        )

    agent.save()
    plt.plot(all_episode_reward)
    if not os.path.exists('image'):
        os.makedirs('image')
    plt.savefig(os.path.join('image', 'ddpg.png'))
```

The agent can be tested after training.

```
if args.test:
    # test
    agent.load()
    for episode in range(TEST_EPISODES):
        state = env.reset()
        episode_reward = 0
        for step in range(MAX_STEPS):
            env.render()
            state, reward, done, info =
                 env.step(agent.get_action(state, greedy=True))
            episode_reward += reward
            if done:
                break
        print(
            'Testing | Episode: {}/{} | Episode Reward: {:.4f} |
                 Running Time: {:.4f}'.format(
                episode + 1, TEST_EPISODES, episode_reward,
                time.time() - t0))
```

6.5.3 *TD3: Pendulum-v0*

TD3 code uses these classes: `ReplayBuffer` class, `QNetwork` class, `PolicyNetwork` class, and `TD3` class.

`ReplayBuffer` class is used to build a replay buffer, so it should have the function of `push()` and `sample()`.

```
class ReplayBuffer:
   def __init__(self, capacity):
      ...
   def push(self, state, action, reward, next_state, done):
      ...
   def sample(self, batch_size):
      ...
   def __len__(self):
      ...
```

The `push()` function appends data to the buffer and move the pointer.

```
   def push(self, state, action, reward, next_state, done):
      if len(self.buffer) < self.capacity:
         self.buffer.append(None)
      self.buffer[self.position] = (state, action, reward,
          next_state, done)
      self.position = int((self.position + 1) % self.capacity) #
          as a ring buffer
```

The `sample()` function simply samples data from the buffer and returns.

```
   def sample(self, batch_size):
      batch = random.sample(self.buffer, batch_size)
      state, action, reward, next_state, done = map(np.stack,
          zip(*batch)) # stack for each element
      return state, action, reward, next_state, done
```

By refactoring the `__len__()` function, the buffer size will be returned when called by the `len()` function.

```
   def __len__(self):
      return len(self.buffer)
```

The `QNetwork` class is used to build the *Q*-network for critic. It is another coding way for building networks.

```
class QNetwork(Model):
   def __init__(self, num_inputs, num_actions, hidden_dim,
       init_w=3e-3):
      super(QNetwork, self).__init__()
      input_dim = num_inputs + num_actions
      w_init = tf.random_uniform_initializer(-init_w, init_w)
      self.linear1 = Dense(n_units=hidden_dim, act=tf.nn.relu,
          W_init=w_init, in_channels=input_dim, name='q1')
      self.linear2 = Dense(n_units=hidden_dim, act=tf.nn.relu,
          W_init=w_init, in_channels=hidden_dim, name='q2')
      self.linear3 = Dense(n_units=1, W_init=w_init,
          in_channels=hidden_dim, name='q3')

   def forward(self, input):
      x = self.linear1(input)
      x = self.linear2(x)
      x = self.linear3(x)
      return x
```

The `PolicyNetwork` class is used to build the policy network for the actor. While building the network, it also adds `evaluate()`, `get_action()`, `sample_action()` functions.

```
class PolicyNetwork(Model):
   def __init__(self, num_inputs, num_actions, hidden_dim,
       action_range=1., init_w=3e-3):
      ...
   def forward(self, state):
      ...
   def evaluate(self, state, eval_noise_scale):
      ...
   def get_action(self, state, explore_noise_scale,
       greedy=False):
      ...
   def sample_action(self):
      ...
```

The following part of the code shows how to build the network:

```
class PolicyNetwork(Model):
   def __init__(self, num_inputs, num_actions, hidden_dim,
       action_range=1., init_w=3e-3):
      super(PolicyNetwork, self).__init__()
      w_init = tf.random_uniform_initializer(-init_w, init_w)
      self.linear1 = Dense(n_units=hidden_dim, act=tf.nn.relu,
          W_init=w_init, in_channels=num_inputs, name='policy1')
      self.linear2 = Dense(n_units=hidden_dim, act=tf.nn.relu,
          W_init=w_init, in_channels=hidden_dim, name='policy2')
```

```
        self.linear3 = Dense(n_units=hidden_dim, act=tf.nn.relu,
            W_init=w_init, in_channels=hidden_dim, name='policy3')
        self.output_linear = Dense(n_units=num_actions,
            W_init=w_init,
            b_init=tf.random_uniform_initializer(-init_w, init_w),
            in_channels=hidden_dim, name='policy_output')
        self.action_range = action_range
        self.num_actions = num_actions

    def forward(self, state):
        x = self.linear1(state)
        x = self.linear2(x)
        x = self.linear3(x)
        output = tf.nn.tanh(self.output_linear(x)) # unit range
            output [-1, 1]
        return output
```

The `evaluate()` function generates actions with the states for calculating gradients. It uses the trick of target policy smoothing for generating noisy actions.

```
    def evaluate(self, state, eval_noise_scale):
        state = state.astype(np.float32)
        action = self.forward(state)
        action = self.action_range * action
        # add noise
        normal = Normal(0, 1)
        eval_noise_clip = 2 * eval_noise_scale
        noise = normal.sample(action.shape) * eval_noise_scale
        noise = tf.clip_by_value(noise, -eval_noise_clip,
            eval_noise_clip)
        action = action + noise
        return action
```

The `get_action()` function generates actions with the states to interact with the environment.

```
    def get_action(self, state, explore_noise_scale,
         greedy=False):
        action = self.forward([state])
        action = self.action_range * action.numpy()[0]
        if greedy:
            return action
        # add noise
        normal = Normal(0, 1)
        noise = normal.sample(action.shape) * explore_noise_scale
        action += noise
        return action.numpy()
```

The `sample_action()` function is used to generate random actions at the beginning of training.

```
   def sample_action(self, ):
      a = tf.random.uniform([self.num_actions], -1, 1)
      return self.action_range * a.numpy()
```

The `TD3` class is the core content of the TD3 algorithm.

```
class TD3():
   def __init__(self, state_dim, action_dim, replay_buffer,
       hidden_dim, action_range,
       policy_target_update_interval=1, q_lr=3e-4,
       policy_lr=3e-4): # create replay buffer and networks
      ...
   def target_ini(self, net, target_net): # hard-copy update for
       initializing target networks
      ...
   def target_soft_update(self, net, target_net, soft_tau):
       #soft update the target net with Polyak averaging
      ...
   def update(self, batch_size, eval_noise_scale,
       reward_scale=10., gamma=0.9, soft_tau=1e-2): # update all
       networks in TD3
      ...
   def save(self): # save trained weights
      ...
   def load(self): # load trained weights
      ...
```

The initialization function creates twin q-networks, policy-networks, and their target networks. Six networks are established in total.

```
class TD3():
   def __init__(self, state_dim, action_dim, replay_buffer,
       hidden_dim, action_range,
       policy_target_update_interval=1, q_lr=3e-4,
       policy_lr=3e-4):
      self.replay_buffer = replay_buffer

      # initialize all networks
      self.q_net1 = QNetwork(state_dim, action_dim, hidden_dim)
      self.q_net2 = QNetwork(state_dim, action_dim, hidden_dim)
      self.target_q_net1 = QNetwork(state_dim, action_dim,
          hidden_dim)
      self.target_q_net2 = QNetwork(state_dim, action_dim,
          hidden_dim)
      self.policy_net = PolicyNetwork(state_dim, action_dim,
          hidden_dim, action_range)
      self.target_policy_net = PolicyNetwork(state_dim,
          action_dim, hidden_dim, action_range)
      print('Q Network (1,2): ', self.q_net1)
```

```
        print('Policy Network: ', self.policy_net)

        # initialize weights of target networks
        self.target_q_net1 = self.target_ini(self.q_net1,
            self.target_q_net1)
        self.target_q_net2 = self.target_ini(self.q_net2,
            self.target_q_net2)
        self.target_policy_net = self.target_ini(self.policy_net,
            self.target_policy_net)

        # set train mode
        self.q_net1.train()
        self.q_net2.train()
        self.target_q_net1.train()
        self.target_q_net2.train()
        self.policy_net.train()
        self.target_policy_net.train()

        self.update_cnt = 0
        self.policy_target_update_interval =
            policy_target_update_interval

        self.q_optimizer1 = tf.optimizers.Adam(q_lr)
        self.q_optimizer2 = tf.optimizers.Adam(q_lr)
        self.policy_optimizer = tf.optimizers.Adam(policy_lr)
```

The `target_ini()` function and `target_soft_update()` function are used to update the parameters of target networks. The difference is that the former one is a hard copy replacement and the latter one updates the parameters by Polyak averaging.

```
    def target_ini(self, net, target_net):=
        for target_param, param in
            zip(target_net.trainable_weights,
            net.trainable_weights):
           target_param.assign(param)
        return target_net

    def target_soft_update(self, net, target_net, soft_tau):=
        for target_param, param in
            zip(target_net.trainable_weights,
            net.trainable_weights):
           target_param.assign(target_param * (1.0 - soft_tau) +
               param * soft_tau) # copy weight value into target
               parameters
        return target_net
```

Next is the most critical part: the `update()` function. This part fully reflects the three tricks of the TD3 algorithm.

At the beginning of the function, we first sample data from the replay buffer.

```
    def update(self, batch_size, eval_noise_scale,
        reward_scale=10., gamma=0.9, soft_tau=1e-2):
        ''' update all networks in TD3 '''
        self.update_cnt += 1

        # sample batch data
        state, action, reward, next_state, done =
            self.replay_buffer.sample(batch_size)
        reward = reward[:, np.newaxis] # expand dim
        done = done[:, np.newaxis]
```

Next, we use a target policy smoothing trick by adding noise to the target action. This makes it harder for the policy to exploit Q-value function errors by smoothing out Q values along changes in action.

```
        # Trick Three: Target Policy Smoothing. Add noise to the
            target action
        new_next_action = self.target_policy_net.evaluate(
           next_state, eval_noise_scale=eval_noise_scale
        ) # clipped normal noise

        # normalize with batch mean and std
        reward = reward_scale * (reward - np.mean(reward, axis=0))
            / np.std(reward, axis=0)
```

The next trick is clipped double-Q learning. It learns two Q-value functions and uses the smaller Q-value to form the targets in the Bellman error loss function. In this way, the overestimation of the Q-value can be mitigated.

```
        # Training Q Function
        target_q_input = tf.concat([next_state, new_next_action],
            1) # the dim 0 is number of samples

        # Trick One: Clipped Double-Q Learning. Use the smaller
            Q-value.
        target_q_min =
            tf.minimum(self.target_q_net1(target_q_input),
            self.target_q_net2(target_q_input))

        target_q_value = reward + (1 - done) * gamma *
            target_q_min # if done==1, only reward
        q_input = tf.concat([state, action], 1) # input of q_net

        with tf.GradientTape() as q1_tape:
           predicted_q_value1 = self.q_net1(q_input)
           q_value_loss1 =
               tf.reduce_mean(tf.square(predicted_q_value1 -
               target_q_value))
        q1_grad = q1_tape.gradient(q_value_loss1,
            self.q_net1.trainable_weights)
```

```
self.q_optimizer1.apply_gradients(zip(q1_grad,
    self.q_net1.trainable_weights))

with tf.GradientTape() as q2_tape:
   predicted_q_value2 = self.q_net2(q_input)
   q_value_loss2 =
       tf.reduce_mean(tf.square(predicted_q_value2 -
       target_q_value))
q2_grad = q2_tape.gradient(q_value_loss2,
    self.q_net2.trainable_weights)
self.q_optimizer2.apply_gradients(zip(q2_grad,
    self.q_net2.trainable_weights))
```

The last trick is the "delayed" policy updates trick. The policy and its target networks are updated less frequently than the Q-value function. The paper recommends one policy update for every two Q-value function updates.

```
# Training Policy Function
# Trick Two: ``Delayed''İ Policy Updates. Update the
    policy less frequently
if self.update_cnt % self.policy_target_update_interval ==
    0:
   with tf.GradientTape() as p_tape:
      new_action = self.policy_net.evaluate(
         state, eval_noise_scale=0.0
      ) # no noise, deterministic policy gradients
      new_q_input = tf.concat([state, new_action], 1)
      # ''' implementation 1 '''
      # predicted_new_q_value =
          tf.minimum(self.q_net1(new_q_input),self.q_net2
          (new_q_input))
      ''' implementation 2 '''
      predicted_new_q_value = self.q_net1(new_q_input)
      policy_loss = -tf.reduce_mean(predicted_new_q_value)
   p_grad = p_tape.gradient(policy_loss,
       self.policy_net.trainable_weights)
   self.policy_optimizer.apply_gradients(zip(p_grad,
       self.policy_net.trainable_weights))

   # Soft update the target nets
   self.target_q_net1 =
       self.target_soft_update(self.q_net1,
       self.target_q_net1, soft_tau)
   self.target_q_net2 =
       self.target_soft_update(self.q_net2,
       self.target_q_net2, soft_tau)
   self.target_policy_net =
       self.target_soft_update(self.policy_net,
       self.target_policy_net, soft_tau)
```

The following code is part of the training code. We first create the environment and the agent.

```
# initialization of env
env = gym.make(ENV_ID).unwrapped
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
action_range = env.action_space.high # scale action,
    [-action_range, action_range]

# reproducible
env.seed(RANDOM_SEED)
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
tf.random.set_seed(RANDOM_SEED)

# initialization of buffer
replay_buffer = ReplayBuffer(REPLAY_BUFFER_SIZE)
# initialization of trainer
agent = TD3(state_dim, action_dim, action_range, HIDDEN_DIM,
    replay_buffer,
        POLICY_TARGET_UPDATE_INTERVAL, Q_LR, POLICY_LR)
t0 = time.time()
```

Before starting each episode, it is necessary to do some initialization. In this code, the training time is limited by the total number of steps instead of the max episode iterations. And because the network is built in different ways, you need an extra call of the function before using it.

```
# training loop
if args.train:
   frame_idx = 0
   all_episode_reward = []
   # need an extra call here to make inside functions be able
       to use model.forward
   state = env.reset().astype(np.float32)
   agent.policy_net([state])
   agent.target_policy_net([state])
```

At the very beginning, a random sample was used by the agent to provide enough data for the update. After that the agent was left to interact with the environment and update at every step as usual.

```
   for episode in range(TRAIN_EPISODES):
      state = env.reset().astype(np.float32)
      episode_reward = 0
      for step in range(MAX_STEPS):
         if RENDER:
            env.render()
         if frame_idx > EXPLORE_STEPS:
            action = agent.policy_net.get_action(state,
                EXPLORE_NOISE_SCALE)
         else:
            action = agent.policy_net.sample_action()

         next_state, reward, done, _ = env.step(action)
         next_state = next_state.astype(np.float32)
         done = 1 if done is True else 0

         replay_buffer.push(state, action, reward,
             next_state, done)
         state = next_state
         episode_reward += reward
         frame_idx += 1

         if len(replay_buffer) > BATCH_SIZE:
            for i in range(UPDATE_ITR):
               agent.update(BATCH_SIZE, EVAL_NOISE_SCALE,
                   REWARD_SCALE)
         if done:
            break
```

Finally, we provide the necessary functions to visualize the training process and save the agent.

```
         if episode == 0:
            all_episode_reward.append(episode_reward)
         else:
            all_episode_reward.append(all_episode_reward[-1] *
                0.9 + episode_reward * 0.1)
         print(
            'Training | Episode: {}/{} | Episode Reward: {:.4f}
                | Running Time: {:.4f}'.format(
               episode+1, TRAIN_EPISODES, episode_reward,
               time.time() - t0
            )
         )
      agent.save()
      plt.plot(all_episode_reward)
      if not os.path.exists('image'):
         os.makedirs('image')
      plt.savefig(os.path.join('image', 'td3.png'))
```

6.5.4 SAC: Pendulum-v0

SAC is an entropy method. The target q value uses the minimum of the two Q network and log probability of policy $\pi(\tilde{a}|s)$. The example code used `ReplayBuffer` class, `SoftQNetwork` class, `PolicyNetwork` class, and `SAC` class.

The `ReplayBuffer` class and the `SoftQNetwork` class are almost the same as `ReplayBuffer` class and `QNetwork` class in TD3 code, so we can skip them and see the code after.

```
class ReplayBuffer: # a ring buffer for storing transitions and
    sampling for training
   def __init__(self, capacity):
       ...
   def push(self, state, action, reward, next_state, done):
       ...
   def sample(self, batch_size):
       ...
   def __len__(self):
       ...

class SoftQNetwork(Model): # the network for evaluate values of
    state-action pairs: Q(s,a)
   def __init__(self, num_inputs, num_actions, hidden_dim,
       init_w=3e-3):
       ...
   def forward(self, input):
       ...
```

The `PolicyNetwork` class is also similar. The difference is that SAC uses a stochastic policy network instead of a deterministic policy network, which causes some differences.

```
class PolicyNetwork(Model):
   def __init__(self, num_inputs, num_actions, hidden_dim,
       action_range=1., init_w=3e-3, log_std_min=-20,
       log_std_max=2):
       ...
   def forward(self, state):
       ...
   def evaluate(self, state, epsilon=1e-6):
       ...
   def get_action(self, state, greedy=False):
       ...
   def sample_action(self):
       ...
```

The stochastic network outputs mean values and log standard deviations to depict an action distribution. Therefore, the network has two outputs.

```
class PolicyNetwork(Model):
   def __init__(self, num_inputs, num_actions, hidden_dim,
       action_range=1., init_w=3e-3, log_std_min=-20,
       log_std_max=2):
       super(PolicyNetwork, self).__init__()
       self.log_std_min = log_std_min
       self.log_std_max = log_std_max
       w_init = tf.keras.initializers.glorot_normal(seed=None)
```

```
    self.linear1 = Dense(n_units=hidden_dim, act=tf.nn.relu,
        W_init=w_init, in_channels=num_inputs, name='policy1')
    self.linear2 = Dense(n_units=hidden_dim, act=tf.nn.relu,
        W_init=w_init, in_channels=hidden_dim, name='policy2')
    self.linear3 = Dense(n_units=hidden_dim, act=tf.nn.relu,
        W_init=w_init, in_channels=hidden_dim, name='policy3')
    self.mean_linear = Dense(n_units=num_actions,
        W_init=w_init,
        b_init=tf.random_uniform_initializer(-init_w, init_w),
        in_channels=hidden_dim, name='policy_mean')
    self.log_std_linear = Dense(n_units=num_actions,
        W_init=w_init,
        b_init=tf.random_uniform_initializer(-init_w, init_w),
        in_channels=hidden_dim, name='policy_logstd')
    self.action_range = action_range
    self.num_actions = num_actions
```

A clip method on log standard deviations is used in the `forward()` function to prevent the standard deviation value from becoming too large.

```
def forward(self, state):
    x = self.linear1(state)
    x = self.linear2(x)
    x = self.linear3(x)
    mean = self.mean_linear(x)
    log_std = self.log_std_linear(x)
    log_std = tf.clip_by_value(log_std, self.log_std_min,
        self.log_std_max)
    return mean, log_std
```

The `evaluate()` function uses a reparameterization trick to sample actions from the action distribution so that the gradient here can be propagated back. It also calculates the log probability of the sampled actions on the original action distribution.

```
def evaluate(self, state, epsilon=1e-6):
    state = state.astype(np.float32)
    mean, log_std = self.forward(state)
    std = tf.math.exp(log_std) # no clip in evaluation, clip
        affects gradients flow
    normal = Normal(0, 1)
    z = normal.sample(mean.shape)
    action_0 = tf.math.tanh(mean + std * z) # TanhNormal
        distribution as actions; reparameterization trick
    action = self.action_range * action_0
    # according to original paper, with an extra last term for
        normalizing different action range
    log_prob = Normal(mean, std).log_prob(mean + std * z) -
        tf.math.log(1. - action_0 ** 2 + epsilon) -
        np.log(self.action_range)
    # both dims of normal.log_prob and -log(1-a**2) are
        (N,dim_of_action);
    # the Normal.log_prob outputs the same dim of input
        features instead of 1 dim probability,
    # needs sum up across the dim of actions to get 1 dim
        probability; or else use Multivariate Normal.
    log_prob = tf.reduce_sum(log_prob, axis=1)[:, np.newaxis]
        # expand dim as reduce_sum causes 1 dim reduced
    return action, log_prob, z, mean, log_std
```

The `get_action()` function is a simplified version of the previous function. It only samples actions from the action distributions.

```
    def get_action(self, state, greedy=False):
        mean, log_std = self.forward([state])
        std = tf.math.exp(log_std)
        normal = Normal(0, 1)
        z = normal.sample(mean.shape)
        action = self.action_range * tf.math.tanh(
            mean + std * z
        ) # TanhNormal distribution as actions; reparameterization
            trick

        action = self.action_range * tf.math.tanh(mean) if greedy
            else action
        return action.numpy()[0]
```

The `sample_action()` function is much more simple. It is only used at the very beginning of training to sample data for the first update.

```
    def sample_action(self, ):
        a = tf.random.uniform([self.num_actions], -1, 1)
        return self.action_range * a.numpy()
```

The structure of the SAC we will talk about next is as follows:

```
class SAC():
    def __init__(self, state_dim, action_dim, replay_buffer,
         hidden_dim, action_range, soft_q_lr=3e-4, policy_lr=3e-4,
         alpha_lr=3e-4): # create networks and variables
        ...
    def target_ini(self, net, target_net): # hard-copy update for
         initializing target networks
        ...
    def target_soft_update(self, net, target_net, soft_tau): #
         soft update the target net with Polyak averaging
        ...
    def update(self, batch_size, reward_scale=10.,
         auto_entropy=True, target_entropy=-2, gamma=0.99,
         soft_tau=1e-2): # update all networks in SAC
        ...
    def save(self): # save trained weights
        ...
    def load(self): # load trained weights
        ...
```

There are 5 networks in SAC algorithm. They are two soft Q-networks and their target networks, and a stochastic policy network. An alpha variable is also needed as a trade-off coefficient for the entropy regularization.

```
class SAC():
   def __init__(self, state_dim, action_dim, replay_buffer,
       hidden_dim, action_range, soft_q_lr=3e-4, policy_lr=3e-4,
       alpha_lr=3e-4):
      self.replay_buffer = replay_buffer

      # initialize all networks
      self.soft_q_net1 = SoftQNetwork(state_dim, action_dim,
          hidden_dim)
      self.soft_q_net2 = SoftQNetwork(state_dim, action_dim,
          hidden_dim)
      self.target_soft_q_net1 = SoftQNetwork(state_dim,
          action_dim, hidden_dim)
      self.target_soft_q_net2 = SoftQNetwork(state_dim,
          action_dim, hidden_dim)
      self.policy_net = PolicyNetwork(state_dim, action_dim,
          hidden_dim, action_range)
      self.log_alpha = tf.Variable(0, dtype=np.float32,
          name='log_alpha')
      self.alpha = tf.math.exp(self.log_alpha)
      print('Soft Q Network (1,2): ', self.soft_q_net1)
      print('Policy Network: ', self.policy_net)
      # set mode
      self.soft_q_net1.train()
      self.soft_q_net2.train()
      self.target_soft_q_net1.eval()
      self.target_soft_q_net2.eval()
      self.policy_net.train()

      # initialize weights of target networks
      self.target_soft_q_net1 =
          self.target_ini(self.soft_q_net1,
          self.target_soft_q_net1)
      self.target_soft_q_net2 =
          self.target_ini(self.soft_q_net2,
          self.target_soft_q_net2)

      self.soft_q_optimizer1 = tf.optimizers.Adam(soft_q_lr)
      self.soft_q_optimizer2 = tf.optimizers.Adam(soft_q_lr)
      self.policy_optimizer = tf.optimizers.Adam(policy_lr)
      self.alpha_optimizer = tf.optimizers.Adam(alpha_lr)
```

Let us introduce the `update()` function next. The other functions are the same as the previous code, so we can skip them. As usual, at the beginning of the `update()` function, we sample data from the replay buffer first. A normalization on reward values can improve the training effect.

```
   def update(self, batch_size, reward_scale=10.,
       auto_entropy=True, target_entropy=-2, gamma=0.99,
       soft_tau=1e-2):
      state, action, reward, next_state, done =
          self.replay_buffer.sample(batch_size)
      reward = reward[:, np.newaxis] # expand dim
      done = done[:, np.newaxis]
      reward = reward_scale * (reward - np.mean(reward, axis=0))
          / (
            np.std(reward, axis=0) + 1e-6
      ) # normalize with batch mean and std; plus a small number
          to prevent numerical problem
```

After that, we will calculate the target Q-value based on the next state. SAC used the minimum of the two target networks which is the same as TD3. But they differ in that SAC adds entropy regularization when calculating target q value. The `log_prob` part here is the entropy which is a measure of randomness in the policy.

```
      # Training Q Function
      new_next_action, next_log_prob, _, _, _ =
          self.policy_net.evaluate(next_state)
      target_q_input = tf.concat([next_state, new_next_action],
          1) # the dim 0 is number of samples
      target_q_min = tf.minimum(
         self.target_soft_q_net1(target_q_input),
             self.target_soft_q_net2(target_q_input)
      ) - self.alpha * next_log_prob
      target_q_value = reward + (1 - done) * gamma *
          target_q_min # if done==1, only reward
```

After calculating the target Q-value, training the Q function is simple.

```
      q_input = tf.concat([state, action], 1) # the dim 0 is
          number of samples
      with tf.GradientTape() as q1_tape:
         predicted_q_value1 = self.soft_q_net1(q_input)
         q_value_loss1 =
             tf.reduce_mean(tf.losses.mean_squared_error
             (predicted_q_value1, target_q_value))
      q1_grad = q1_tape.gradient(q_value_loss1,
          self.soft_q_net1.trainable_weights)
      self.soft_q_optimizer1.apply_gradients(zip(q1_grad,
          self.soft_q_net1.trainable_weights))
      with tf.GradientTape() as q2_tape:
         predicted_q_value2 = self.soft_q_net2(q_input)
         q_value_loss2 =
             tf.reduce_mean(tf.losses.mean_squared_error
             (predicted_q_value2, target_q_value))
```

```
q2_grad = q2_tape.gradient(q_value_loss2,
    self.soft_q_net2.trainable_weights)
self.soft_q_optimizer2.apply_gradients(zip(q2_grad,
    self.soft_q_net2.trainable_weights))
```

The policy loss is the expected future return plus expected future entropy. By maximizing the loss function, the policy will be trained to maximize a trade-off between expected return and entropy.

```
# Training Policy Function
with tf.GradientTape() as p_tape:
   new_action, log_prob, z, mean, log_std =
       self.policy_net.evaluate(state)
   new_q_input = tf.concat([state, new_action], 1) # the
       dim 0 is number of samples
   ''' implementation 1 '''
   predicted_new_q_value =
       tf.minimum(self.soft_q_net1(new_q_input),
       self.soft_q_net2(new_q_input))
   # ''' implementation 2 '''
   # predicted_new_q_value = self.soft_q_net1(new_q_input)
   policy_loss = tf.reduce_mean(self.alpha * log_prob -
       predicted_new_q_value)
p_grad = p_tape.gradient(policy_loss,
    self.policy_net.trainable_weights)
self.policy_optimizer.apply_gradients(zip(p_grad,
    self.policy_net.trainable_weights))
```

Finally, we update the entropy trade-off coefficient alpha and target networks.

```
# Updating alpha w.r.t entropy
# alpha: trade-off between exploration (max entropy) and
    exploitation (max Q)
if auto_entropy is True:
   with tf.GradientTape() as alpha_tape:
      alpha_loss = -tf.reduce_mean((self.log_alpha *
          (log_prob + target_entropy)))
   alpha_grad = alpha_tape.gradient(alpha_loss,
       [self.log_alpha])
   self.alpha_optimizer.apply_gradients(zip(alpha_grad,
       [self.log_alpha]))
   self.alpha = tf.math.exp(self.log_alpha)
else: # fixed alpha
   self.alpha = 1.
   alpha_loss = 0

# Soft update the target value nets
self.target_soft_q_net1 =
    self.target_soft_update(self.soft_q_net1,
    self.target_soft_q_net1, soft_tau)
```

```
        self.target_soft_q_net2 =
            self.target_soft_update(self.soft_q_net2,
            self.target_soft_q_net2, soft_tau)
```

The main loop process of training is the same as TD3. First, build the environment and the agent.

```
    # initialization of env
    env = gym.make(ENV_ID).unwrapped
    state_dim = env.observation_space.shape[0]
    action_dim = env.action_space.shape[0]
    action_range = env.action_space.high # scale action,
        [-action_range, action_range]

    # reproducible
    env.seed(RANDOM_SEED)
    random.seed(RANDOM_SEED)
    np.random.seed(RANDOM_SEED)
    tf.random.set_seed(RANDOM_SEED)

    # initialization of buffer
    replay_buffer = ReplayBuffer(REPLAY_BUFFER_SIZE)
    # initialization of trainer
    agent = SAC(state_dim, action_dim, action_range, HIDDEN_DIM,
                   replay_buffer, SOFT_Q_LR, POLICY_LR, ALPHA_LR)
    t0 = time.time()
```

Then, use the agent to interact with the environment and store sampled data for updates. Before the first update, a random action sample is used.

```
    # training loop
    if args.train:
        frame_idx = 0
        all_episode_reward = []

        # need an extra call here to make inside functions be able
            to use model.forward
        state = env.reset().astype(np.float32)
        agent.policy_net([state])

        for episode in range(TRAIN_EPISODES):
            state = env.reset().astype(np.float32)
            episode_reward = 0
            for step in range(MAX_STEPS):
                if RENDER:
                    env.render()
                if frame_idx > EXPLORE_STEPS:
                    action = agent.policy_net.get_action(state)
                else:
                    action = agent.policy_net.sample_action()
                next_state, reward, done, _ = env.step(action)
                next_state = next_state.astype(np.float32)
                done = 1 if done is True else 0
                replay_buffer.push(state, action, reward,
                    next_state, done)
                state = next_state
                episode_reward += reward
                frame_idx += 1
```

When enough data are collected, we can start to update at each step.

```
            if len(replay_buffer) > BATCH_SIZE:
               for i in range(UPDATE_ITR):
                  agent.update(
                     BATCH_SIZE, reward_scale=REWARD_SCALE,
                         auto_entropy=AUTO_ENTROPY,
                     target_entropy=-1. * action_dim
                  )
            if done:
               break
```

Through the above steps, the agent can become more and more powerful through updates. The next step is to better represent the training process.

```
         if episode == 0:
            all_episode_reward.append(episode_reward)
         else:
            all_episode_reward.append(all_episode_reward[-1] *
                0.9 + episode_reward * 0.1)
         print(
            'Training | Episode: {}/{} | Episode Reward: {:.4f}
                | Running Time: {:.4f}'.format(
               episode+1, TRAIN_EPISODES, episode_reward,
               time.time() - t0
            )
         )
```

Finally, save the agent and plot the figure.

```
      agent.save()
      plt.plot(all_episode_reward)
      if not os.path.exists('image'):
         os.makedirs('image')
      plt.savefig(os.path.join('image', 'sac.png'))
```

References

Fox R, Pakman A, Tishby N (2016) Taming the noise in reinforcement learning via soft updates. In: Proceedings of the thirty-second conference on uncertainty in artificial intelligence. AUAI Press, Corvallis, pp 202–211

Fujimoto S, van Hoof H, Meger D (2018) Addressing function approximation error in actor-critic methods. arXiv:180209477

Haarnoja T, Tang H, Abbeel P, Levine S (2017) Reinforcement learning with deep energy-based policies. In: Proceedings of the 34th international conference on machine learning, vol 70, pp 1352–1361. JMLR.org

Haarnoja T, Zhou A, Abbeel P, Levine S (2018a) Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor. arXiv:180101290

Haarnoja T, Zhou A, Hartikainen K, Tucker G, Ha S, Tan J, Kumar V, Zhu H, Gupta A, Abbeel P, et al (2018b) Soft actor-critic algorithms and applications. arXiv:181205905

It K, McKean H (1965) Diffusion processes and their sample paths. Die Grundlehren der math Wissenschaften, vol 125. Springer, Berlin

Levine S, Koltun V (2013) Guided policy search. In: International conference on machine learning, pp 1–9

Lillicrap TP, Hunt JJ, Pritzel A, Heess N, Erez T, Tassa Y, Silver D, Wierstra D (2015) Continuous control with deep reinforcement learning. arXiv:150902971

Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, Graves A, Riedmiller M, Fidjeland AK, Ostrovski G, Petersen S, Beattie C, Sadik A, Antonoglou I, King H, Kumaran D, Wierstra D, Legg S, Hassabis D (2015) Human-level control through deep reinforcement learning. Nature 518(7540):529–533

Nachum O, Norouzi M, Xu K, Schuurmans D (2017) Bridging the gap between value and policy based reinforcement learning. In: Advances in neural information processing systems, pp 2775–2785

Polyak BT (1964) Some methods of speeding up the convergence of iteration methods. USSR Comput Math Math Phys 4(5):1–17

Silver D, Lever G, Heess N, Degris T, Wierstra D, Riedmiller M (2014) Deterministic policy gradient algorithms. In: Proceedings of the 31st international conference on machine learning

Sutton RS, Barto AG (2018) Reinforcement learning: an introduction. MIT Press, Cambridge

Uhlenbeck GE, Ornstein LS (1930) On the theory of the Brownian motion. Phys. Rev. 36(5):823

Williams RJ (1992) Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach Learn 8(3–4):229–256

Ziebart BD, Maas AL, Bagnell JA, Dey AK (2008) Maximum entropy inverse reinforcement learning. In: Proceedings of the AAAI conference on artificial intelligence, Chicago, vol 8, pp 1433–1438