RL methods for continuous action space

Alexey Skrynnik

AGENDA

01 Deterministic Policy Gradient

02 Twin Delayed DDPG

03 Soft Actor-Critic

01

Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG)

- DDPG is an algorithm that concurrently learns:
  - A Q-function
  - A policy
- Uses off-policy data and the Bellman equation to learn the Q-function.
- Uses the Q-function to learn the policy.
- Closely connected to Q-learning and motivated by the idea that if we know the optimal Q-function $Q^*(s, a)$, we can determine the optimal action:

$$a^*(s) = \arg\max_a Q^*(s, a).$$

Learning Q-Function and Policy

- DDPG interleaves:
  - Learning an approximator for $Q^*(s, a)$.
  - Learning an approximator for $a^*(s)$.
- Adapted specifically for **continuous action spaces**.
- The key challenge: Computing the max over actions in $\max_a Q^*(s, a)$.

Challenge of Maximization in Continuous Spaces

- In **discrete action spaces**:
  - Maximization is simple: compute $Q(s, a)$ for all possible actions.
- In **continuous action spaces**:
  - Cannot exhaustively evaluate all actions.
  - Solving $\max_a Q^*(s, a)$ directly is computationally expensive.
- Assumption: $Q^*(s, a)$ is **differentiable w.r.t. actions**.

Challenge of Maximization in Continuous Spaces

Because the action space is continuous, the function $Q^*(s, a)$ is presumed to be differentiable with respect to the action argument. This allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute $\max_a Q(s, a)$, we can approximate it with:

$$\max_a Q(s, a) \approx Q(s, \mu(s))$$

The Q-Learning Side of DDPG

- The Bellman equation for the optimal action-value function $Q^*(s, a)$:

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

- Here, $s' \sim P$ means that the next state s' is sampled from the environment's transition distribution $P(\cdot|s, a)$.
- This equation forms the basis for learning an approximator to $Q^*(s, a)$.

Learning an Approximate Q-Function

- We use a neural network $Q_\phi(s, a)$ with parameters ϕ to approximate $Q^*(s, a)$.
- We train it using a replay buffer $\mathcal{D}$ containing transitions (s, a, r, s', d).
- The Mean-Squared Bellman Error (MSBE) loss function is:

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

Understanding the Terminal State Condition

- The term $(1 - d)$ ensures that:
  - If $d = 1$ (terminal state), then $\max_{a'} Q_\phi(s', a')$ is ignored.
  - If $d = 0$ (non-terminal state), then we bootstrap using the next Q-value.
- This follows the Python convention where True $= 1$ and False $= 0$.
- Ensures that Q-function stops accumulating rewards after a terminal state.

Tricks Used in Q-Learning

- Most Q-learning algorithms, including DQN and DDPG, minimize the MSBE loss.
- Two key tricks used in all of them:
  1. Target networks: Use a separate, slowly updated network $Q_{\phi_{\text{targ}}}$ to stabilize training.
  2. Replay buffer: Store past experiences and sample them randomly to decorrelate updates.

Target Network Updates in DDPG

- In DQN-based algorithms, the target network is copied from the main network every fixed number of steps.
- In DDPG, the target network is updated once per main network update using Polyak averaging:

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

- Here, ρ is a hyperparameter between 0 and 1 (typically close to 1).
- This approach makes the target network update more smoothly over time.

Handling the Max Over Actions in DDPG

- In continuous action spaces, computing $\max_a Q(s, a)$ is challenging.
- Instead of direct optimization, DDPG uses a **target policy network** to approximate the maximizing action:

$$\tilde{a} = \mu_{\theta_{\text{targ}}}(s')$$

- Like the target Q-function, the target policy network is updated via **Polyak averaging**.
- This ensures that target actions change gradually, improving stability.

Q-Learning in DDPG

- The Q-function in DDPG is trained by minimizing the **Mean-Squared Bellman Error (MSBE)**:

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

- Here, $\mu_{\theta_{\text{targ}}}$ is the target policy network.
- This loss is minimized with **stochastic gradient descent**.

Policy Learning in DDPG

- The goal is to learn a **deterministic policy** $\mu_\theta(s)$ that maximizes $Q_\phi(s, a)$.
- Since $Q_\phi(s, a)$ is differentiable w.r.t. a, we can **directly optimize the policy**:

$$\max_\theta \mathop{\mathrm{E}}_{s \sim \mathcal{D}} [Q_\phi(s, \mu_\theta(s))]$$

- We use **gradient ascent** to update policy parameters θ.
- The Q-function parameters ϕ are treated as **constants** during policy optimization.

Exploration vs. Exploitation in DDPG

- DDPG trains a **deterministic policy** in an **off-policy** manner.
- A purely deterministic policy would not explore enough early on.
- To encourage exploration, we add **noise** to actions during training.
- The original DDPG paper suggested **Ornstein-Uhlenbeck (OU) noise**, but recent research shows that:
  - **Uncorrelated Gaussian noise** (mean-zero) works just as well.
  - Gaussian noise is simpler and preferred in practice.
- Optionally, noise scale can be **reduced over time** to improve exploitation.
- Our implementation **keeps noise fixed** throughout training.

DDPG Pseudocode

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

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

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

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

02

Twin Delayed DDPG

Limitations of DDPG

- While DDPG can achieve great performance, it is **often brittle** with respect to hyperparameters and tuning.
- A common failure mode is **Q-function overestimation**, leading to policy instability.
- The learned policy exploits errors in the Q-function, breaking performance.
- **Twin Delayed DDPG (TD3)** addresses these issues with three key tricks.

Trick One: Clipped Double-Q Learning

- TD3 learns **two Q-functions** instead of one (hence “twin”).
- It uses the **minimum of the two Q-values** to form targets in the Bellman error loss function.
- This reduces **overestimation bias** in Q-learning.

Trick Two: Delayed Policy Updates

- TD3 updates the **policy and target networks less frequently** than the Q-function.
- The paper recommends **one policy update for every two Q-function updates**.
- This reduces **variability in policy updates**, leading to more stable learning.

Trick Three: Target Policy Smoothing

- **Noise is added** to the target action to prevent overfitting to Q-function errors.
- This **smooths out Q-values** along small action changes, improving robustness.
- Helps avoid **exploiting incorrect sharp Q-function peaks**.

Key Equations: Target Policy Smoothing

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

- Target actions are **perturbed by clipped Gaussian noise**.
- The perturbed action is then **clipped within the valid action range**.
- Prevents **excessively sharp Q-function gradients** from harming training.

Clipped Double-Q Learning

→ Both Q-functions use a **single target**, determined by the **smaller Q-value**:

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{i,\text{targ}}}(s', a'(s')).$$

$$L(\phi_1, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_{\phi_1}(s, a) - y(r, s', d) \right)^2 \right]$$

$$L(\phi_2, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_{\phi_2}(s, a) - y(r, s', d) \right)^2 \right]$$

Policy Learning in TD3

- The policy is learned by **maximizing the Q-function**:

$$\max_{\theta} \underset{s \sim \mathcal{D}}{\mathrm{E}} \left[Q_{\phi_1}(s, \mu_\theta(s)) \right].$$

- **Same as in DDPG**, but with an important difference:
- In TD3, the **policy is updated less frequently** than the Q-functions.
- This **reduces training instability** caused by rapid target shifts.

Algorithm 3 Twin Delayed DDPG

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$

Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$

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 j in range(however many updates) **do**

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

      Compute target actions

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

if $j \mod \texttt{policy\_delay} = 0$ **then**

Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

Update target networks with

$$\begin{aligned} \phi_{\text{targ},i} &\leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2 \\ \theta_{\text{targ}} &\leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta \end{aligned}$$

end if

end for

end if

until convergence

03

Soft Actor-Critic

Soft Actor-Critic

The Soft Actor-Critic (SAC) algorithm optimizes a stochastic policy in an off-policy manner, bridging the gap between stochastic policy optimization and DDPG-style approaches. It is not a direct successor to TD3 (as it was published around the same time) but incorporates the clipped double-Q trick. Additionally, due to the inherent stochasticity of the policy in SAC, it also benefits from target policy smoothing.

Soft Actor-Critic

A central feature of SAC is entropy regularization.

The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. This has a close connection to the exploration-exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on. It can also prevent the policy from prematurely converging to a bad local optimum.

Entropy-Regularized Reinforcement Learning

Entropy is a quantity which, roughly speaking, says how random a random variable is. If a coin is weighted so that it almost always comes up heads, it has low entropy; if it's evenly weighted and has a half chance of either outcome, it has high entropy.

Let x be a random variable with probability mass or density function P. The entropy H of x is computed from its distribution P according to

$$H(P) = \mathbb{E}_{x \sim P}\left[-\log P(x)\right].$$

Entropy-Regularized Reinforcement Learning

In entropy-regularized reinforcement learning, the agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep. This changes the RL problem to:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H\big(\pi(\cdot|s_t)\big)\right)\right],$$

where $\alpha > 0$ is the trade-off coefficient. We assume an infinite-horizon discounted setting.

Entropy-Regularized Reinforcement Learning

The value function in this setting is modified to include the entropy bonuses from every timestep:

$$V^{\pi}(s) = \mathbb{E}_{\tau \sim \pi}\left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H\big(\pi(\cdot|s_t)\big)\right) \middle| s_0 = s\right].$$

Q^{π} is changed to include the entropy bonuses from every timestep **except the first**:

$$Q^{\pi}(s, a) = \mathbb{E}_{\tau \sim \pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) + \alpha \sum_{t=1}^{\infty} \gamma^t H\big(\pi(\cdot|s_t)\big) \middle| s_0 = s, a_0 = a\right].$$

Entropy-Regularized Q-Function

With these definitions, V^π and Q^π are connected by:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi}\left[Q^\pi(s,a)\right] + \alpha H\big(\pi(\cdot|s)\big).$$

The Bellman equation for Q^π is:

$$Q^\pi(s,a) = \mathbb{E}_{s'\sim P}\left[R(s,a,s') + \gamma \mathbb{E}_{a'\sim\pi}\left[Q^\pi(s',a') + \alpha H\big(\pi(\cdot|s')\big)\right]\right].$$

Which simplifies to:

$$Q^\pi(s,a) = \mathbb{E}_{s'\sim P}\left[R(s,a,s') + \gamma V^\pi(s')\right].$$

Soft Actor-Critic и TD3

SAC concurrently learns a policy π_θ and two Q-functions Q_{ϕ_1}, Q_{ϕ_2}. There are two variants of SAC that are currently standard: one that uses a fixed entropy regularization coefficient α, and another that enforces an entropy constraint by varying α over the course of training.

The Q-functions are learned in a similar way to TD3, but with a few key differences.

First, what's similar?

- Like in TD3, both Q-functions are learned with MSBE minimization, by regressing to a single shared target.
- Like in TD3, the shared target is computed using target Q-networks, and the target Q-networks are obtained by polyak averaging the Q-network parameters over the course of training.
- Like in TD3, the shared target makes use of the clipped double-Q trick.

Soft Actor-Critic and TD3

What is different?

- Unlike in TD3, the target also includes a term that comes from SAC's use of entropy regularization.
- Unlike in TD3, the next-state actions used in the target come from the current policy instead of a target policy.
- Unlike in TD3, there is no explicit target policy smoothing. TD3 trains a deterministic policy, and so it accomplishes smoothing by adding random noise to the next-state actions. SAC trains a stochastic policy, and so the noise from that stochasticity is sufficient to get a similar effect.

Approximating the Expectation

Before we give the final form of the Q-loss, let's take a moment to discuss how the contribution from entropy regularization comes in. We'll start by taking our recursive Bellman equation for the entropy-regularized Q^π from earlier and rewriting it slightly using the definition of entropy:

$$
\begin{aligned}
Q^\pi(s,a) &= \mathbb{E}_{s'\sim P, a'\sim\pi}\left[R(s,a,s') + \gamma\left(Q^\pi(s',a') + \alpha H\big(\pi(\cdot|s')\big)\right)\right] \\
&= \mathbb{E}_{s'\sim P, a'\sim\pi}\left[R(s,a,s') + \gamma\left(Q^\pi(s',a') - \alpha \log \pi(a'|s')\right)\right].
\end{aligned}
$$

Approximating the Expectation

The right-hand side (RHS) of the Bellman equation is an expectation over next states (which come from the replay buffer) and next actions (which come from the current policy, not the replay buffer).

Since it's an expectation, we can approximate it with samples:

$$Q^{\pi}(s,a) \approx r + \gamma \left(Q^{\pi}(s', \tilde{a}') - \alpha \log \pi(\tilde{a}'|s') \right), \quad \tilde{a}' \sim \pi(\cdot|s').$$

This sampled approximation is used in practice to estimate the Q-values efficiently.

Q-Loss in SAC

SAC sets up the Mean Squared Bellman Error (MSBE) loss for each Q-function using this kind of sample approximation for the target.

The only thing still undetermined is which Q-function gets used to compute the sample backup. Like TD3, SAC uses the clipped double-Q trick and takes the minimum Q-value between the two Q approximators.

Q-Loss in SAC

Putting it all together, the loss functions for the Q-networks in SAC are:

$$L(\phi_i, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathbb{E}} \left[\left(Q_{\phi_i}(s,a) - y(r,s',d) \right)^2 \right],$$

where the target is given by:

$$y(r,s',d) = r + \gamma(1-d)\left(\min_{j=1,2} Q_{\phi_{\text{targ},j}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s') \right),$$

where $\tilde{a}' \sim \pi_\theta(\cdot|s')$.

Learning the Policy

The policy should, in each state, act to maximize the expected future return plus expected future entropy. That is, it should maximize $V^{\pi}(s)$, which expands to:

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{a\sim\pi}\left[Q^{\pi}(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right) \\ &= \mathbb{E}_{a\sim\pi}\left[Q^{\pi}(s,a) - \alpha \log \pi(a|s)\right]. \end{aligned}$$

The policy optimization step in SAC is based on this objective.

Reparameterization Trick

The way we optimize the policy makes use of the **reparameterization trick**, in which a sample from $\pi_\theta(\cdot|s)$ is drawn by computing a deterministic function of state, policy parameters, and independent noise.

To illustrate: following the authors of the SAC paper, we use a squashed Gaussian policy, which means that samples are obtained according to:

$$\tilde{a}_\theta(s, \xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s) \odot \xi\right), \quad \xi \sim \mathcal{N}(0, I).$$

This formulation ensures smooth exploration while maintaining bounded action outputs.

Key Differences in SAC Policy

This policy has two key differences from the policies used in other policy optimization algorithms:

1. **The squashing function:** The $\tanh$ in the SAC policy ensures that actions are bounded to a finite range. This is absent in VPG, TRPO, and PPO policies. It also modifies the distribution: before the $\tanh$, the SAC policy is a factored Gaussian like the others, but after $\tanh$, it is not.
2. **State-dependent standard deviations:** In VPG, TRPO, and PPO, log standard deviations are represented by state-independent parameter vectors. In SAC, log standard deviations are outputs from the neural network, making them state-dependent.

Reparameterization Trick and Policy Loss

The reparameterization trick allows us to rewrite the expectation over actions into an expectation over noise:

$$\mathbb{E}_{a\sim\pi_\theta}\left[Q^{\pi_\theta}(s,a) - \alpha \log \pi_\theta(a|s)\right] = \mathbb{E}_{\xi\sim\mathcal{N}}\left[Q^{\pi_\theta}(s,\tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right].$$

Reparameterization Trick and Policy Loss

To get the policy loss, we substitute Q^{π_θ} with a function approximator. Unlike TD3, which uses only Q_{ϕ_1}, SAC uses the minimum of the two Q approximators:

$$\max_\theta \mathbb{E}_{s\sim\mathcal{D},\xi\sim\mathcal{N}}\left[\min_{j=1,2} Q_{\phi_j}(s,\tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right].$$

This is similar to DDPG and TD3, except for the min-double-Q trick, the stochasticity, and the entropy term.

SAC Pseudocode

Algorithm 5 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$ and 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 j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \qquad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \qquad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

12: Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

13: Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$$

where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.

14: Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

15: **end for**

16: **end if**

17: **until** convergence

Questions?