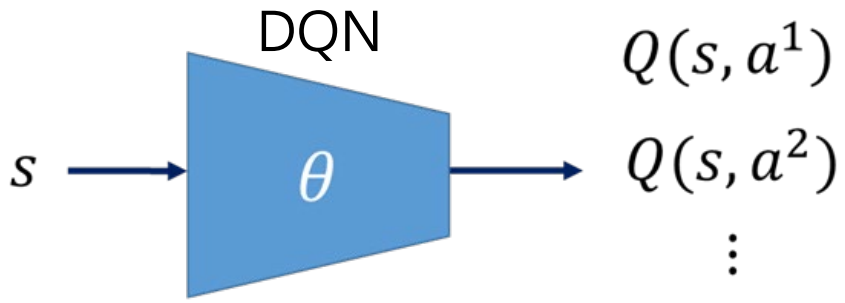
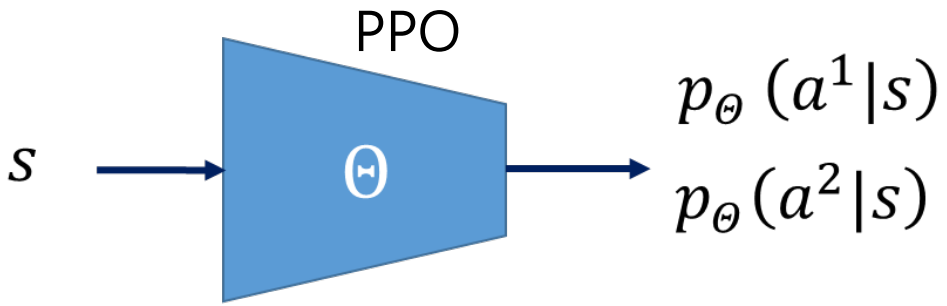
Recap: PPO, DQN and DDPG



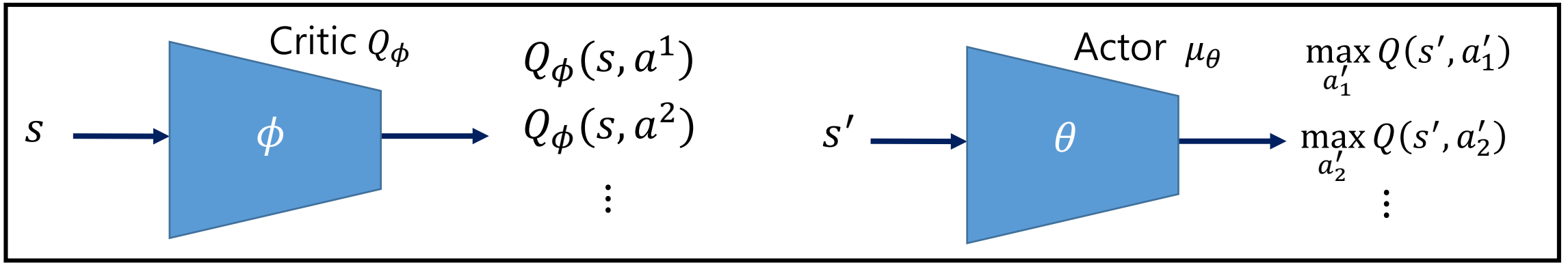
- Learn expected long-term reward $Q(s,a)$
- Use Bellman eq. to update $Q^*(s,a)$ based on $Q^*(s',a')$



- Learn policy $p_\Theta(a|s)$
- Use policy gradient (gradient of expected value of long-term rewards) to adjust $p_\Theta(a|s)$

DDPG

- Learn $Q(s,a)$ and a' that max $Q(s',a')$
- Use Bellman eq. to update $Q^*(s,a)$ and use $Q(s,a)$ to update policy network



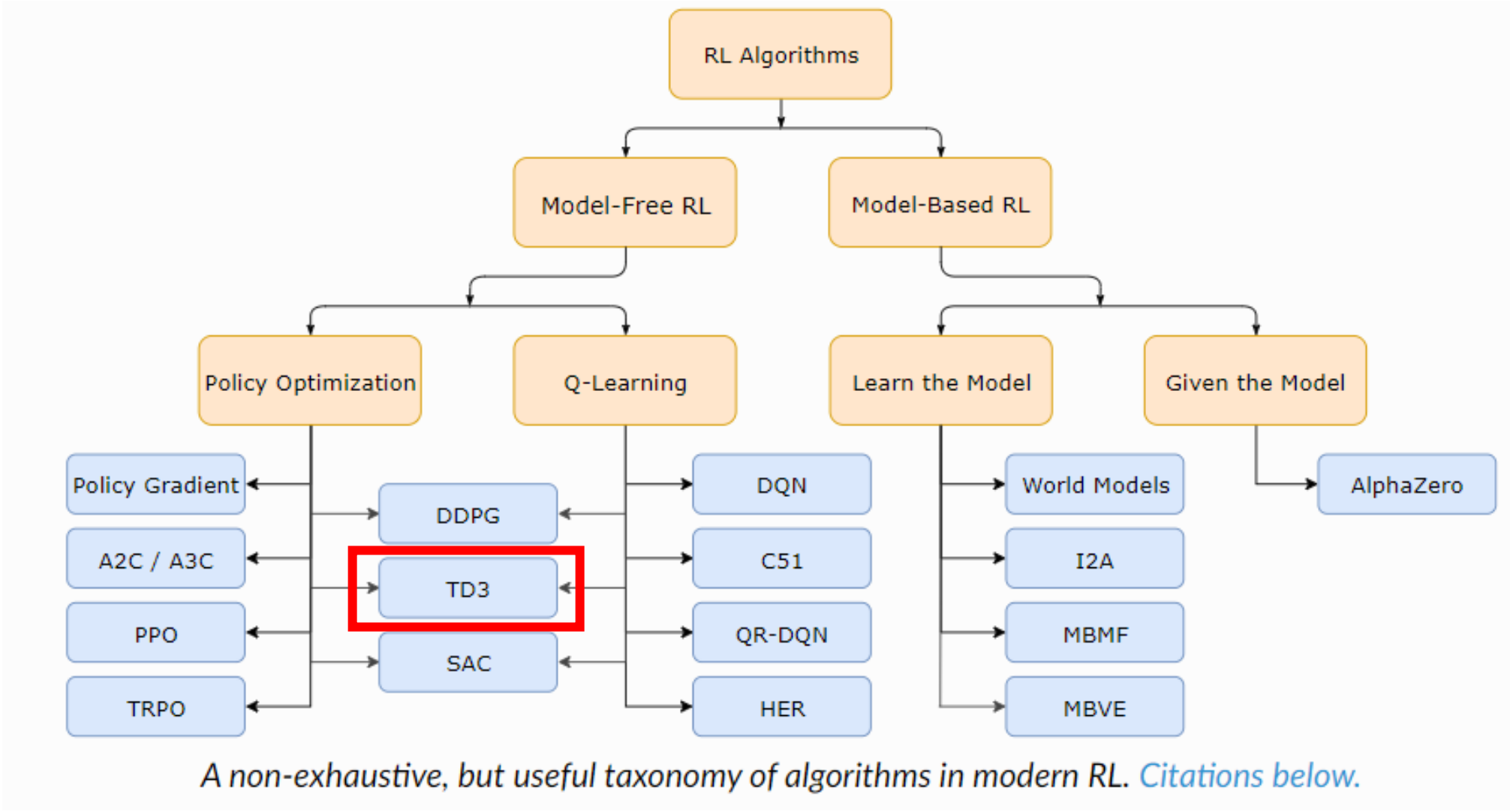
Introduction – Twin Delayed DDPG (TD3)

While DDPG can achieve great performance sometimes, it is frequently brittle with respect to hyper-parameters and other kinds of tuning. A common failure mode for DDPG is that the learned Q-function begins to dramatically overestimate Q-values, which then leads to the policy breaking, because it exploits the errors in the Q-function. Twin Delayed DDPG (TD3) is an algorithm that addresses this issue by introducing three critical tricks.

Reference: Twin Delayed DDPG — Spinning Up documentation (openai.com)

TD3

Popular Q-learning based RL algorithms include DQN, DDPG, TD3 and SAC.



A non-exhaustive, but useful taxonomy of algorithms in modern RL. Citations below.

Tricks to overcome DDPG problems

Trick one – Clipped Double-Q Learning

TD3 learns two Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

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.

Trick three – Target Policy Smoothing

TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

Target Policy Smoothing

Actions used to form the Q-learning target are based on the target policy network $\mu_{\theta_{target'}}$ but with clipped noise added on each dimension of the action. After adding the clipped noise, the target action is then clipped to lie in the valid action range (all valid actions, a, satisfy $a_{Low} \leq a \leq a_{High}$).

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

Target policy network in DDPG

$$\mu_{\theta_{target}}(s') \approx arg\max_{a'} Q(s', a')$$

Clipped double-Q learning

Both Q-functions use a single target, calculated using whichever of the two Q-functions gives a smaller target value.

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

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

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

Learning policy

The policy is learned just by maximizing Q_{ϕ_1}.

$$\max_{\theta} \mathop{\mathbb{E}}_{s \sim \mathcal{D}} \left[Q_{\phi_1}(s, \mu_\theta(s)) \right]$$

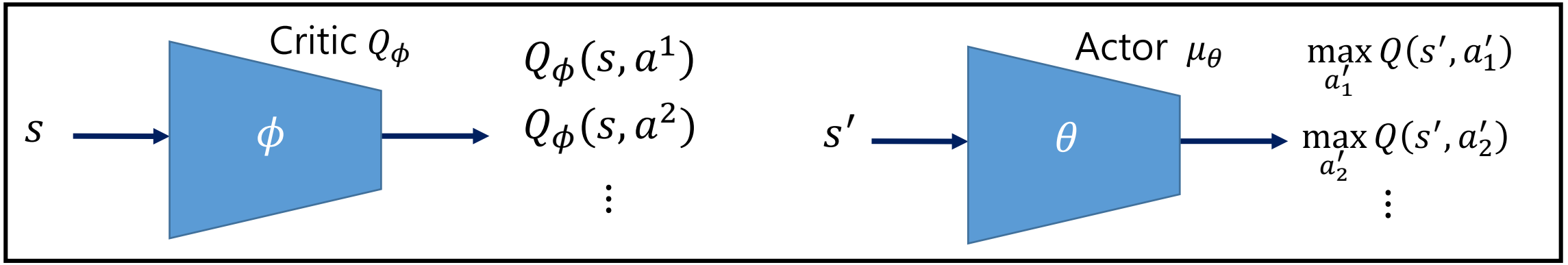
In TD3, the policy is updated less frequently than the Q-functions are. This helps damp the volatility that normally arises in DDPG because of how a policy update changes the target.

Exploration vs. Exploitation

TD3 trains a deterministic policy in an off-policy way. 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 TD3 policies explore better, we add noise to their actions at training time, typically uncorrelated mean-zero Gaussian noise. To facilitate getting higher-quality training data, you may reduce the scale of the noise over the course of training.

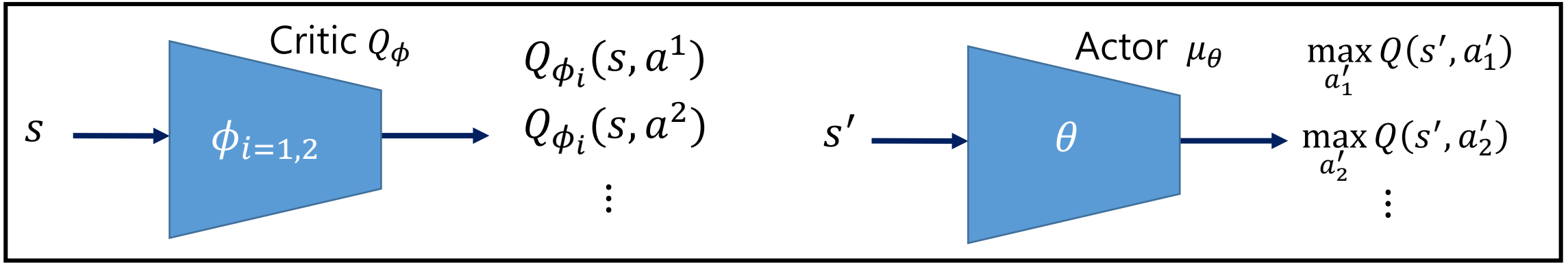
Recap: DDPG → TD3

DDPG



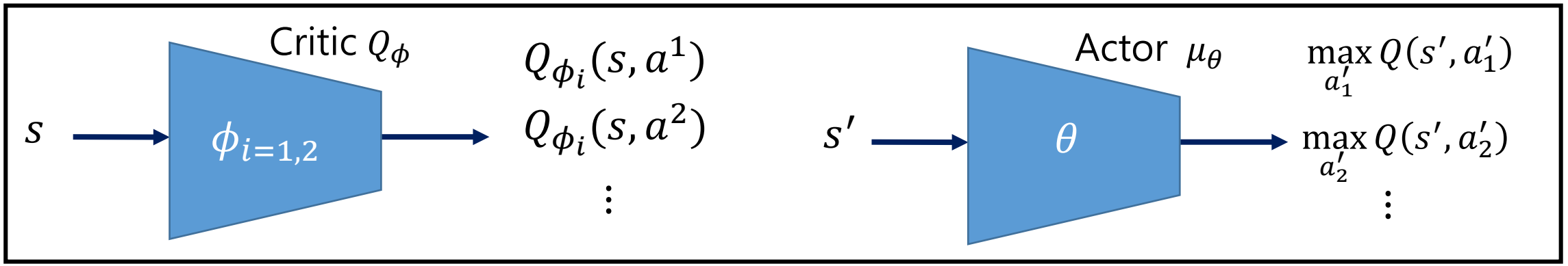
TD3

- Learn two Q functions $\phi_{i=1,2}$ and a' that max$\phi_{1,}(s', a')$
- Use Bellman eq. to update $Q^*(s, a)$ use $Q(s, a)$ to update policy network



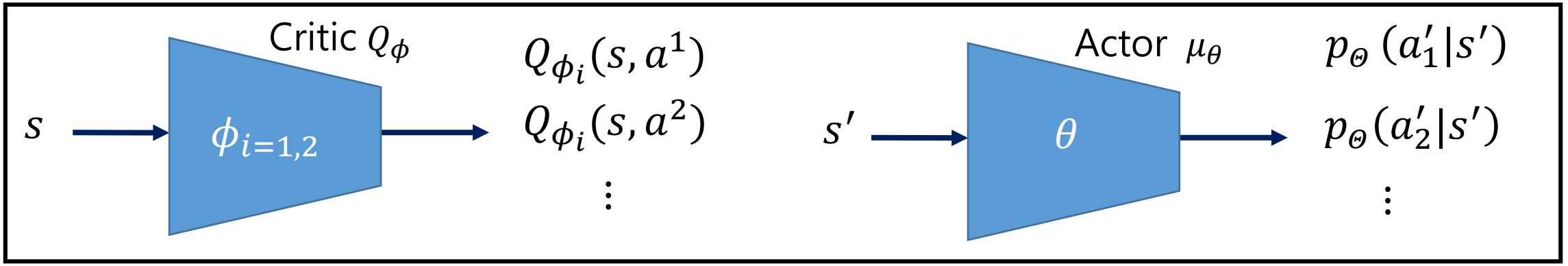
TD3 → SAC (Soft Actor Critic)

TD3



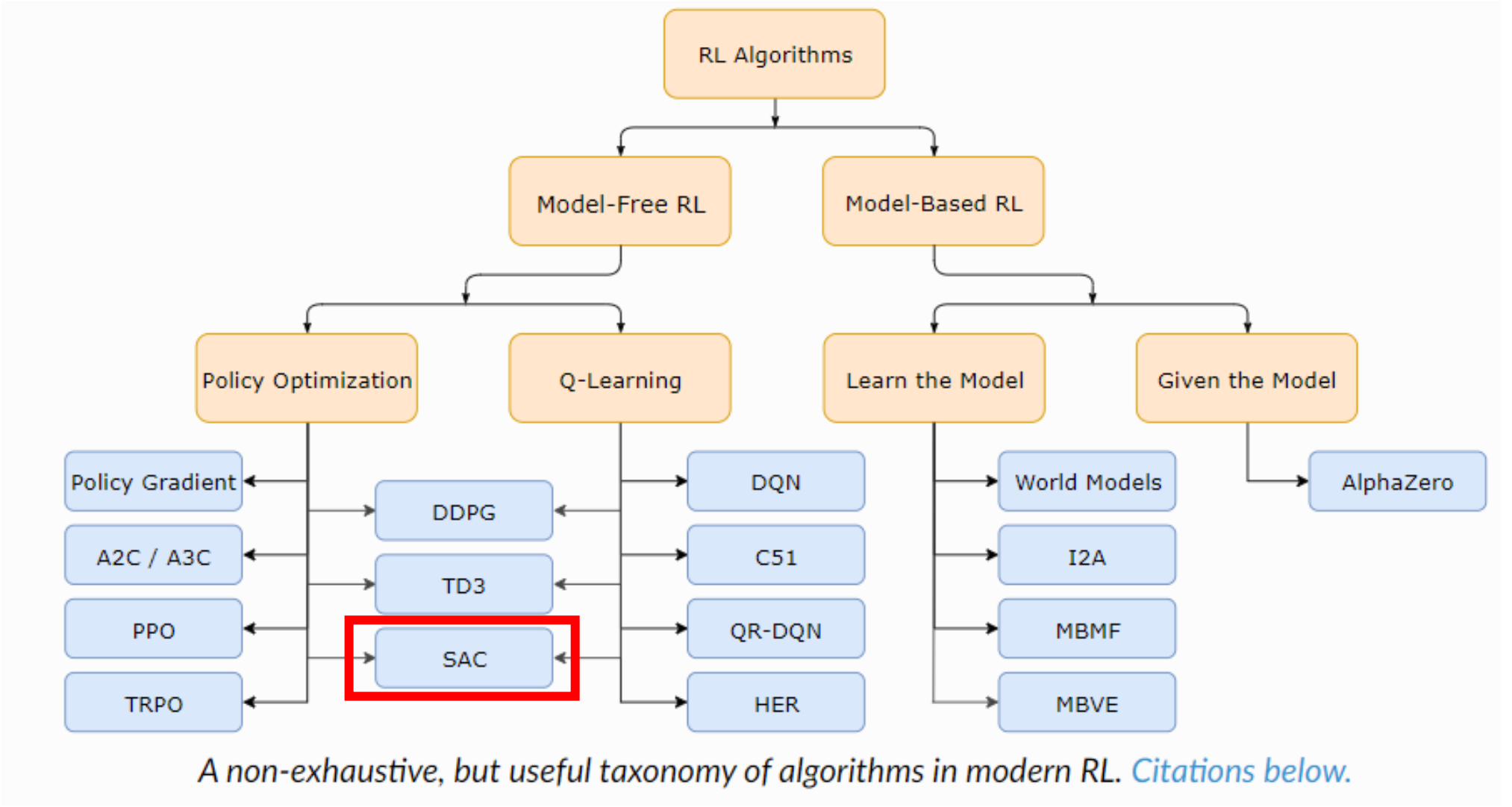
SAC

- Learn two Q functions $\phi_{i=1,2}$ and a stochastic policy
- Use Bellman eq. to update $Q^*(s, a)$ and use $Q(s, a)$ + entropy regularization to update policy network



SAC

Popular Q-learning based RL algorithms include DQN, DDPG, TD3 and SAC.



A non-exhaustive, but useful taxonomy of algorithms in modern RL. Citations below.

Policy, Q_1, Q_2 networks

```
self.Normal    = NormalPolicyNet(input_dim=input_dim, action_dim=action_dim)
self.Normal_optimizer = optim.Adam(self.Normal.parameters(), lr=1e-3)

self.Q1        = QNet(input_dim=input_dim, action_dim=action_dim)
self.Q1_targ   = QNet(input_dim=input_dim, action_dim=action_dim)
self.Q1_targ.load_state_dict(self.Q1.state_dict())
self.Q1_optimizer = optim.Adam(self.Q1.parameters(), lr=1e-3)

self.Q2        = QNet(input_dim=input_dim, action_dim=action_dim)
self.Q2_targ   = QNet(input_dim=input_dim, action_dim=action_dim)
self.Q2_targ.load_state_dict(self.Q2.state_dict())
self.Q2_optimizer = optim.Adam(self.Q2.parameters(), lr=1e-3)
```

μ_θ

Q_{ϕ_1}

$Q_{\phi_{1,target}}$

Q_{ϕ_2}

$Q_{\phi_{2,target}}$

code: https://github.com/zhihanyang2022/pytorch-sac

Introduction

Soft Actor Critic (SAC) is an algorithm that optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and DDPG-style approaches.

Reference: https://spinningup.openai.com/en/latest/algorithms/sac.html

Stochastic policy in SAC

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.

Stochastic policy in SAC $\tilde{a}' \backsim \pi_\theta(\cdot\,|s')$

Target policy smoothing in TD3 $a'(s') = clip\left(\mu_{\theta_{target}}(s') + clip(\epsilon, -c, c), a_{Low}, a_{High}\right)$

Target policy network in DDPG $\mu_{\theta_{targ}}(s') = arg\max_{a'} Q(s', a')$

Entropy regularization

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 RL

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.

$$H(P) = \underset{x \backsim P}{\mathrm{E}}[-logP(x)]$$

$$\pi^* = arg\max_{\pi} \underset{x \backsim \pi}{\mathrm{E}} \left[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot\,|s_t))) \,|s_0 = s \right]$$

Entropy-regularized RL

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

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

$$V^{\pi}(s) = \mathop{\mathrm{E}}_{\tau \backsim \pi}[Q^{\pi}(s,a)] + \alpha H(\pi(\cdot\,|s))$$

$$\mathrm{Q}^{\pi}(s,a) = \mathop{\mathrm{E}}_{\substack{s' \backsim P \\ a' \backsim \pi}}[R(s,a,s') + \gamma(\mathrm{Q}^{\pi}(s',a') + \alpha H(\pi(\cdot\,|s'))]$$

$$= \mathop{\mathrm{E}}_{s' \backsim P}[R(s,a,s') + \gamma V^{\pi}(s')]$$

Learning the Q functions

Unlike in TD3, the next-state actions used in the target come from the current policy instead of a target policy.

Q learning in SAC

$$\tilde{a}' \backsim \pi_\theta(\cdot\,|s')$$

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

$$L(\phi_{i=1,2},\mathcal{D}) = \underset{(s,a,r,s',a')\backsim\mathcal{D}}{\mathrm{E}}\left[\left(\phi_{i=1,2}(s,a) - y(r,s',d)\right)^2\right])$$

Q learning in TD3

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

$$L(\phi_{i=1,2},\mathcal{D}) = \underset{(s,a,r,s',a')\backsim\mathcal{D}}{\mathrm{E}}\left[\left(Q_{\phi_{i=1,2}}(s,a) - y(r,s',d)\right)^2\right])$$

Learning the Q functions

```
na, log_pi_na_given_ns = self.sample_action_and_compute_log_pi(b.ns, use_reparametrization_trick=False)
```

$$\tilde{a}' \backsim \pi_\theta(\cdot\,|s')$$

```
targets = b.r + self.gamma * (1 - b.d) * \
          (self.min_i_12(self.Q1_targ(b.ns, na), self.Q2_targ(b.ns, na)) - self.alpha * log_pi_na_given_ns)
```

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

```
Q1_predictions = self.Q1(b.s, b.a)
Q1_loss = torch.mean((Q1_predictions - targets) ** 2)
```

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

```
Q2_predictions = self.Q2(b.s, b.a)
Q2_loss = torch.mean((Q2_predictions - targets) ** 2)
```

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

code: https://github.com/zhihanyang2022/pytorch-sac

Reparameterization trick to sample next action

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. Following the authors of the SAC paper, we use a squashed Gaussian policy, which means that samples are obtained according to

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

```
def sample_action_and_compute_log_pi(self, state: torch.tensor, use_reparametrization_trick: bool):
        mu_given_s = self.Normal(state)
        u = mu_given_s.rsample() if use_reparametrization_trick else mu_given_s.sample()
        a = torch.tanh(u)
```

$\mu_\theta(s) + \sigma_\theta(s)$

$\tilde{a}_\theta = tanh\big(\mu_\theta(s) + \sigma_\theta(s)\big)$

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 $V^{\pi}(s)$.

$$\max_{\theta} V^{\pi_\theta}(s) = \mathop{\mathrm{E}}_{a\backsim\pi}[Q^{\pi}(s,a)] + \alpha H\big(\pi(\cdot|s)\big)$$
$$= \mathop{\mathrm{E}}_{a\backsim\pi}[Q^{\pi}(s,a) - \alpha log\pi(a|s)]$$

$$H(P) = \mathop{\mathrm{E}}_{x\backsim P}[-logP(x)]$$

Policy learning in TD3 $\max_{\theta} \mathop{\mathrm{E}}_{s\backsim\mathcal{D}}\left[Q_{\phi_1}(s,\mu_\theta(s))\right]$

Policy learning in DDPG $\max_{\theta} \mathop{\mathrm{E}}_{s\backsim\mathcal{D}}\left[Q_{\phi_1}(s,\mu_\theta(s))\right]$

Sample action and calculate log probability

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

$$\max_\theta V^{\pi_\theta}(s) = \mathop{\mathrm{E}}_{a \sim \pi}[Q^\pi(s, \tilde{a}_\theta) - \alpha log\pi(a|s)]$$

```
def sample_action_and_compute_log_pi(self, state: torch.tensor, use_reparametrization_trick: bool):

        mu_given_s = self.Normal(state)

        u = mu_given_s.rsample() if use_reparametrization_trick else mu_given_s.sample()

        a = torch.tanh(u)

        log_pi_a_given_s = mu_given_s.log_prob(u) - (2 * (np.log(2) - u - F.softplus(-2 * u))).sum(dim=1)

        return a, log_pi_a_given_s
```

$\mu_\theta(s) + \sigma_\theta(s)$

$\tilde{a}_\theta = tanh\big(\mu_\theta(s) + \sigma_\theta(s)\big)$

$log\pi(a|s)$

Learning the policy

```
a, log_pi_a_given_s = self.sample_action_and_compute_log_pi(b.s, use_reparametrization_trick=True)
```

$$\tilde{a}_\theta = tanh(\mu_\theta(s) + \sigma_\theta(s) \odot \xi)$$

```
policy_loss = - torch.mean(self.min_i_12(self.Q1(b.s, a), self.Q2(b.s, a)) - self.alpha * log_pi_a_given_s)
```

$$\max V^\pi(s) = \mathop{\mathrm{E}}_{a \backsim \pi}[Q^\pi(s, \tilde{a}_\theta) - \alpha log \pi(\tilde{a}_\theta | s)]$$

Learning the Policy

SAC policy has two key differences from the policies we use in the 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 the PPO policies. It also changes the distribution: before the $tanh$ the SAC policy is a factored Gaussian like the other algorithms' policies, but after the $tanh$ it is not.
2. **The way standard deviations are parameterized**. In PPO, we represent the log std devs with state-independent parameter vectors. In SAC, we represent the log std devs as outputs from the neural network, meaning that they depend on state in a complex way. SAC with state-independent log std devs, in our experience, did not work

Update the target nets

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

ρ is a hyperparameter between 0 and 1 (usually close to 1). (This hyperparameter is called polyak in our code).

```
with torch.no_grad():
    self.polyak_update(old_net=self.Q1_targ, new_net=self.Q1)
    self.polyak_update(old_net=self.Q2_targ, new_net=self.Q2)
```

Exploration vs. Exploitation

SAC trains a stochastic policy with entropy regularization, and explores in an on-policy way. The entropy regularization coefficient α explicitly controls the explore-exploit trade off, with higher α corresponding to more exploration, and lower α corresponding to more exploitation. The right coefficient (the one which leads to the stablest/highest-reward learning) may vary from environment to environment, and could require careful tuning.