Deep RL Notes

Jaewan Park

Table of Contents

Basics

Markov Decision Process (MDP)

Symbol	Meaning
$\mathcal{S}$	state space
$\mathcal{A}$	action space
$\gamma \in (0, 1]$	discount factor
$p : \mathcal{S} \times \mathcal{A} \to \Delta(\mathbb{R} \times \mathcal{S})$	reward and state transition probability function
$r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$	reward function
$V^\pi : \mathcal{S} \to \mathbb{R}$	state-value function of policy π
$Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$	state action-value function of policy π

The reward function is a random variable with probability defined by:

$$\mathbb{P}\big(r(s, a) = r\big) = \sum_{s' \in \mathcal{S}} p(r, s' \mid s, a) \quad \left(\text{or } \int_{\mathcal{S}} p(r, s' \mid s, a) ds'\right)$$

We assume that the MDP is always **continual**, and that $\mathcal{S}$ contains an **absorbing state** <t> which satisfies

$$\int_0^1 p(r, s' \mid \texttt{<t>}, a) dr = \begin{cases} 1 & \text{if } s' = \texttt{<t>} \\ 0 & \text{otherwise} \end{cases}, \qquad \forall a \in \mathcal{A}.$$

Value Functions and the Optimal Policy

Value functions are defined by:

$$V^{\pi}(s) \coloneqq \mathbb{E}_{\substack{a_t \sim \pi(\cdot|s_t) \\ (r_t, s_{t+1}) \sim p(\cdot,\cdot|s_t, a_t)}} \left[\sum_{t=0}^{\infty} \gamma^t r_t \;\middle|\; s_0 = s \right],$$

$$Q^{\pi}(s, a) \coloneqq \mathbb{E}_{\substack{(r_t, s_{t+1}) \sim p(\cdot,\cdot|s_t, a_t) \\ a_t \sim \pi(\cdot|s_t)}} \left[\sum_{t=0}^{\infty} \gamma^t r_t \;\middle|\; s_0 = s, a_0 = a \right].$$

For convenience, the state-value function is also called the **V function**, and the state action-value function is also called the **Q function**.

We say a policy π^* is **optimal** if

$$V^{\pi^\star}(s) \geq V^{\pi}(s), \qquad \forall s \in \mathcal{S},\ \pi \in \Delta(\mathcal{A})^{\mathcal{S}}.$$

The overall objective of RL is to *find the optimal policy of a given MDP*.

Bellman Equations

Bellman equations for V, Q at π:

$$V(s) = \mathbb{E}_{\substack{a\sim\pi(\cdot|s) \\ (r,s')\sim p(\cdot,\cdot|s,a)}} \Big[r + \gamma V(s') \;\Big|\; s \Big]$$

$$Q(s,a) = \mathbb{E}_{\substack{(r,s')\sim p(\cdot,\cdot|s,a) \\ a'\sim\pi(\cdot|s')}} \Big[r + \gamma Q(s',a') \;\Big|\; s,a \Big]$$

Transition properties between V and Q at π:

$$V(s) = \mathbb{E}_{a\sim\pi(\cdot|s)} \Big[Q(s,a) \;\Big|\; s \Big]$$

$$Q(s,a) = \mathbb{E}_{(r,s')\sim p(\cdot,\cdot|s,a)} \Big[r + \gamma V(s') \;\Big|\; s,a \Big]$$

Theorem (Properties of value functions)

Let π be a policy, and assume $\gamma \in (0,1)$, $\mathcal{S}$ and $\mathcal{A}$ are finite, and rewards are bounded. Then, the following hold:

1. *V^π and Q^π satisfy the Bellman equations and transition properties at π.*
2. *Solutions to the Bellman equations at π exist, and if V and Q satisfy the Bellman equations at π, then $V = V^\pi$ and $Q = Q^\pi$.*

Bellman Optimality Equations

Bellman optimality equations for V, Q:

$$V(s) = \max_{a \in \mathcal{A}} \mathbb{E}_{(r,s') \sim p(\cdot,\cdot|s,a)} \Big[r + \gamma V(s') \;\Big|\; s, a \Big]$$

$$Q(s,a) = \mathbb{E}_{(r,s') \sim p(\cdot,\cdot|s,a)} \Big[r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') \;\Big|\; s, a \Big]$$

Theorem (Properties of optimal value functions)

Assume $\gamma \in (0,1)$, $\mathcal{S}$ and $\mathcal{A}$ are finite, and rewards are bounded. Then, the following hold:

1. *If $\pi^\star$ is an optimal policy, then $V^{\pi^\star}$ and $Q^{\pi^\star}$ satisfy the Bellman optimality equations.*
2. *Solutions to the Bellman optimality equations exist, and if $V^\star$ and $Q^\star$ satisfy the Bellman optimality equations,*

$$\pi^\star(s) := \operatorname*{argmax}_{a \in \mathcal{A}} \mathbb{E}_{(r,s') \sim p(\cdot,\cdot|s,a)} \Big[r + \gamma V^\star(s') \;\Big|\; s, a \Big] = \operatorname*{argmax}_{a \in \mathcal{A}} Q^\star(s,a)$$

is an optimal policy. In this case, $V^\star = V^{\pi^\star}$ and $Q^\star = Q^{\pi^\star}$.

Value-Based RL

Overview

Value-Based RL is a class of RL algorithms where we first learn a value function, then derive a policy by acting greedily or approximately greedily with respect to that value function.

There are two main settings:

1. **Known model**: We use exact *Dynamic Programming (DP)*, with two types of algorithms: *policy iteration (PI)* and *Value Iteration (VI)*.
2. **Unknown model**: We use a *sampling-based approximation* of DP, classified into *Monte Carlo (MC)* or *Temporal Difference (TD)* methods, according to the depth of sampling. Each class both contains *'PI-like'* and *'VI-like'* algorithms.

Known Model: Dynamic Programming

Assume that we know the transition model $p(s', a' \mid s, a)$, and $\mathcal{S}$ and $\mathcal{A}$ are small enough that iterating over them is computationally plausible.

Define the **Bellman operators** at π $\mathcal{B}^\pi_\mathcal{V} : \mathbb{R}^\mathcal{S} \to \mathbb{R}^\mathcal{S}$, $\mathcal{B}^\pi_\mathcal{Q} : \mathbb{R}^{\mathcal{S}\times\mathcal{A}} \to \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$ by:

$$\mathcal{B}^\pi_\mathcal{V}[V](s) = \mathbb{E}_{\substack{a\sim\pi(\cdot|s) \\ (r,s')\sim p(\cdot,\cdot|s,a)}}\left[r + \gamma V(s') \;\middle|\; s\right]$$

$$\mathcal{B}^\pi_\mathcal{Q}[Q](s,a) = \mathbb{E}_{\substack{(r,s')\sim p(\cdot,\cdot|s,a) \\ a'\sim\pi(\cdot|s')}}\left[r + \gamma Q(s', a') \;\middle|\; s, a\right]$$

Similarly, define the **Bellman optimality operators** $\mathcal{B}^\star_\mathcal{V} : \mathbb{R}^\mathcal{S} \to \mathbb{R}^\mathcal{S}$, $\mathcal{B}^\star_\mathcal{Q} : \mathbb{R}^{\mathcal{S}\times\mathcal{A}} \to \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$ by:

$$\mathcal{B}^\star_\mathcal{V}[V](s) = \max_{a\in\mathcal{A}} \mathbb{E}_{(r,s')\sim p(\cdot,\cdot|s,a)}\left[r + \gamma V(s') \;\middle|\; s, a\right]$$

$$\mathcal{B}^\star_\mathcal{Q}[Q](s,a) = \mathbb{E}_{(r,s')\sim p(\cdot,\cdot|s,a)}\left[r + \gamma \max_{a'\in\mathcal{A}} Q(s', a') \;\middle|\; s, a\right]$$

Then, we may run **Dynamic Programming (DP)** on the Bellman (optimality) equations, treating these operators as *fixed-point iterators*.

Policy Iteration

Policy Iteration (PI) iterates on the *Bellman equation*.

Algorithm 1 PI with V functions

while not converged **do**
  while not converged **do**
    $V \leftarrow \mathcal{B}^{\pi}_{\mathcal{V}}[V]$
  $\pi \leftarrow \text{Greedy}(V)$
return π

Algorithm 2 PI with Q functions

while not converged **do**
  while not converged **do**
    $Q \leftarrow \mathcal{B}^{\pi}_{\mathcal{Q}}[Q]$
  $\pi \leftarrow \text{Greedy}(Q)$
return π

The fixed-point iteration on the Bellman equation is called the **policy evaluation** step, and taking the greedy policy with respect to the evaluated value function is called the **policy improvement** step.

The following theorem shows under which conditions a unique fixed-point of a fixed-point iterator exists and the convergence to it is guaranteed.

Theorem (Banach Fixed Point Theorem)

Let $\mathcal{X}$ be a complete metric space, and let $\mathcal{T} : \mathcal{X} \to \mathcal{X}$ be a γ-contraction with $\gamma < 1$. Then, $\mathcal{T}$ has a unique fixed point $x^\star \in \mathcal{X}$. Furthurmore, for any $x \in \mathcal{X}$, we have $\mathcal{T}^k(x) \to x^\star$ as $k \to \infty$.

Policy Iteration (Cont'd)

Clearly V^π and Q^π are fixed-points to $\mathcal{B}^\pi_\mathcal{V}$ and $\mathcal{B}^\pi_\mathcal{Q}$. Then, the convergence of the policy evaluation step to these functions is guaranteed by:

Theorem (γ-Contractiveness of Bellman Operators)

$\mathcal{B}^\pi_\mathcal{V}$ and $\mathcal{B}^\pi_\mathcal{Q}$ are γ-contractions, i.e., for any $V_1, V_2 \in \mathbb{R}^\mathcal{S}$ and $Q_1, Q_2 \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$,

$$\|\mathcal{B}^\pi_\mathcal{V}[V_1] - \mathcal{B}^\pi_\mathcal{V}[V_2]\| \leq \gamma\|V_1 - V_2\|, \quad \|\mathcal{B}^\pi_\mathcal{Q}[Q_1] - \mathcal{B}^\pi_\mathcal{Q}[Q_2]\| \leq \gamma\|Q_1 - Q_2\|,$$

where $\|\cdot\|$ is measured by the supremum norm.

Finally, the convergence of PI to an optimal policy is guaranteed by:

Theorem (Policy Improvement Theorem)

Consider the PI iteration. Assume $\gamma \in (0, 1)$, $\mathcal{S}$ and $\mathcal{A}$ are finite, and rewards are bounded. Let π_{curr} the policy at the beginning of an iteration, and π_{next} the resulting policy at the end of that iteration. Then, for all iterations,

$$V^{\pi_{next}}(s) \geq V^{\pi_{curr}}(s), \quad \forall s \in \mathcal{S}.$$

Furthermore, after a finite number of iterations, V (or Q) becomes an optimal value function, and π becomes an optimal policy.

Value Iteration

Value Iteration (VI) iterates on the *Bellman optimality equation*.

Algorithm 3 VI with V functions

1: **while** not converged **do**
2: $\quad V \leftarrow \mathcal{B}^\star_{\mathcal{V}}[V]$
3: **return** Greedy(V)

Algorithm 4 VI with Q functions

1: **while** not converged **do**
2: $\quad Q \leftarrow \mathcal{B}^\star_{\mathcal{Q}}[Q]$
3: **return** Greedy(Q)

Clearly $V^\star$ and $Q^\star$ are fixed-points to $\mathcal{B}^\star_{\mathcal{V}}$ and $\mathcal{B}^\star_{\mathcal{Q}}$. Then, together with the Banach fixed point theorem, the convergence of VI's fixed-point iteration to these functions is guaranteed by:

Theorem (γ-Contractiveness of Bellman Optimality Operators)

$\mathcal{B}^\star_{\mathcal{V}}$ and $\mathcal{B}^\star_{\mathcal{Q}}$ are γ-contractions, i.e., for any $V_1, V_2 \in \mathbb{R}^{\mathcal{S}}$ and $Q_1, Q_2 \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$,

$$\|\mathcal{B}^\star_{\mathcal{V}}[V_1] - \mathcal{B}^\star_{\mathcal{V}}[V_2]\| \leq \gamma\|V_1 - V_2\|, \quad \|\mathcal{B}^\star_{\mathcal{Q}}[Q_1] - \mathcal{B}^\star_{\mathcal{Q}}[Q_2]\| \leq \gamma\|Q_1 - Q_2\|,$$

where $\|\cdot\|$ is measured by the supremum norm.

Since $\pi^\star = \text{Greedy}(V^\star) = \text{Greedy}(Q^\star)$, we can say that VI is guaranteed to find the optimal policy.

Unknown Model: Sample-Based Value Learning

When we do not know the model, we cannot directly compute the Bellman expectations over $(r, s') \sim p(\cdot, \cdot \mid s, a)$.

Instead, we collect samples and approximate expectations:

$$(s, a, r, s') \sim \text{experience.}$$

This gives sample-based analogues of PI and VI.

In general, we have two major families which differ in the *depth* of sampling:

- **Monte Carlo (MC)**: sample *full trajectories* and average the returns.
- **Temporal-Difference (TD)**: sample *1-step transitions* and bootstrap from current value estimates.

Learning V vs Q Functions

There is one subtlety to discuss in model-free value learning. In the previous slides, we illustrated both V and Q versions of PI and VI.

However, learning V may be slightly disturbing in sampling-based scenarios. Both PI and VI require a step where we compute the greedy policy with respect to some value function. For V, this is given by

$$\text{Greedy}(V)(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \mathbb{E}_{(r,s') \sim p(\cdot,\cdot|s,a)} \Big[r + \gamma V(s') \;\Big|\; s, a \Big],$$

while for Q, we have a simpler equation

$$\text{Greedy}(Q)(s) = \operatorname*{argmax}_{a \in \mathcal{A}} Q(s,a).$$

Therefore, if we use V, greedy action selection requires extra sampling for each action. This is not inherently worse, but we can clearly say that using Q is more convenient for model-free control.

For the remaining slides, we only consider cases where we learn Q.

Monte Carlo Methods

Monte Carlo (MC) methods collect *full trajectories*

$$(s_0, a_0, r_0, s_1, a_0, \ldots) \quad (\times N \text{ samples})$$

and compute the average return starting from each state–action pair:

$$\widehat{Q^\pi}(s, a) = \mathbb{E}_{t:s_t=s,a_t=a}\big[G_t\big].$$

Then, the optimal greedy policy $\text{Greedy}\Big(\widehat{Q^\pi}\Big)$ is returned.

Therefore, MC is *PI-like*. It is naturally hard to run a *VI-like* MC method, as we cannot approximate the *maximization over actions* in the Bellman optimality operator through sampling.

MC control can be on-policy, or off-policy with importance sampling.

Temporal-Difference Methods

Temporal-Difference (TD) methods use *1-step transitions* and *bootstrapping*.

Instead of waiting for the full rollout to end, TD samples multiple 1-step transitions [1] starting from various state-action pairs:

$$(s, a) \xrightarrow{p} (r, s').$$

We may aggregate the sampled transitions (s, a, r, s') into a single set $\mathcal{D}$.

Then, we use approximate the expectations in the PI or VI updates by averaging over sampled data.

TD sits between pure dynamic programming and Monte Carlo:

- like MC, it learns from samples;
- like DP, it uses Bellman-style bootstrapping.

[1] A more correct phrase may be *1-step transitions and (optionally) actions*, since in some cases we also sample the next action a' from the sampled next state s'.

SARSA

SARSA is a PI-like TD method. To do this, we additionally sample the next action a' from the sampled next state s':

$$(s, a) \xrightarrow{p} (r, s') \xrightarrow{\pi} a'.$$

Then, we may aggregate the sampled transitions (s, a, r, s', a') into a single set $\mathcal{D}^\pi$. This additional sampling step causes SARSA to be an **on-policy** method, i.e., it learns the value of the policy it actually follows.

After collecting transitions, SARSA approximates the Bellman operator by

$$\widehat{\mathcal{B}^\pi_Q[Q]}(s, a) = \mathbb{E}_{(r,s',a'):(s,a,r,s',a')\in\mathcal{D}^\pi}\Big[r + \gamma Q(s', a')\Big].$$

The other steps follow PI.

Q-Learning

Q-Learning is a VI-like TD method that approximates the Bellman optimality operator by

$$\widehat{\mathcal{B}^{\star}_{\mathcal{Q}}[Q]}(s,a) = \mathbb{E}_{(r,s'):(s,a,r,s')\in\mathcal{D}}\left[r + \gamma \max_{a'\in\mathcal{A}} Q(s',a')\right].$$

The other steps follow VI.

Unlike SARSA, Q-learning only approximates the expectation over $(r, s') \sim p(\cdot, \cdot \mid s, a)$, which does not depend on the policy. Therefore, Q-learning is naturally an **off-policy** method.

The fact that Q-learning can learn from off-policy samples gives flexibility in data collection. For example, one may use pre-collected data when online interaction is expensive, or use a separate exploration-oriented behavior policy to visit less-explored regions of the state-action space.

However, this flexibility also comes with caveats; the data must still provide sufficient coverage of relevant state-action pairs; otherwise, poor exploration and distribution shift can make the learned values unreliable.

Q-Learning with Function Approximation

We focus more on Q-learning. So far, we mostly considered the *tabular* setup:

- The state/action spaces are small.
- We can store one value estimate per state-action pair.
- We can update entries of a table directly.

However, when the space is large, this is no longer feasible.

Instead, we approximate the action-value function by a *parameterized function* Q_ϕ. The policy is then induced by the learned Q-function:

$$\pi_\phi(s) = \operatorname*{argmax}_{a \in \mathcal{A}} Q_\phi(s, a).$$

This is still value-based RL, but the value function is now represented by a learnable model.

Offline Q-Learning with FA: Fitted Q-Iteration

Suppose we are given an offline dataset $\mathcal{D}$ of one-step transitions. The underlying behavior policy used in action sampling doesn't matter.

Fitted Q-Iteration (FQI) replaces the inner update step $Q \leftarrow \widehat{\mathcal{B}^\star_Q[Q]}$ of ordinary Q-learning by a mini-optimization problem using the MSE loss:

$$\phi \leftarrow \operatorname*{argmin}_{\phi'} \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}} \left[\left(r + \gamma \max_{a' \in \mathcal{A}} Q_\phi(s', a') - Q_{\phi'}(s, a) \right)^2 \right]$$

This is basically supervised regression using the MSE loss. We may also use other types of losses such as the Huber loss. The optimization problem is typically solved by minibatch SGD over a fixed number of steps.

In actual implementation, ϕ and ϕ' should be treated separately. In particular, gradients should NOT be backpropagated through the regression target $r + \gamma \max_{a' \in \mathcal{A}} Q_\phi(s', a')$. Equivalently, the target should be computed under a *stop-gradient* operation.

Online Q-Learning with FA: Deep Q-Learning

When $\mathcal{D}$ is not fixed in advance but is instead collected through online interaction with the environment, the algorithm is often referred to as **Deep Q-Learning** or **Deep Q-Networks (DQN)** [1].

The updates are essentially the same as FQI, but the online collection of $\mathcal{D}$ gives us more flexibility in shaping the training dynamics.

However, DQNs also need some careful stabilization tools:

1. Target networks for moving targets
2. Replay buffers for data reuse and decorrelation
3. Double Q-learning for mitigating maximization overestimation

[1] The term *deep* refers to the use of deep neural networks for function approximation, which was central to the practical successes of function approximation-based online Q-learning.

Target Network Updates

In naive DQN, the bootstrapping targets are always computed from the current network. However, we may compute from a separate **target network** $Q_{\bar{\phi}}$ [1].

A natural choice is **periodic updates**, where the target network copied periodically from the current network:

$$\bar{\phi} \leftarrow \phi \qquad \text{every } C \text{ environment or gradient steps.}$$

This helps because the regression target stays approximately fixed for several updates.

A smoother alternative is **Polyak updates**:

$$\bar{\phi} \leftarrow \tau\bar{\phi} + (1-\tau)\phi,$$

where τ is close to 1, e.g. $\tau = 0.999$. This makes the target network slowly track the online network.

[1] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015

Replay Buffers

A **replay buffer** [1] stores transitions collected during previous interactions.

When updating Q_ϕ, instead of using only fresh data, we may also sample a few mini-batches from the buffer.

Replay buffers help because they:

- reuse data, improving sample efficiency;
- decorrelate updates by mixing transitions from different times;
- make online Q-learning resemble fitted Q-iteration;
- allow multiple gradient steps per environment step.

However, the buffer distribution may differ from the current policy distribution, so coverage and distribution shift matter.

[1] V. Mnih et al., "Playing Atari with Deep Reinforcement Learning," *NIPS 2013 Deep Learning Workshop*, 2013

Maximization Overestimation

Q-learning's target is computed from a noisy target network $Q_{\bar{\phi}}$. Suppose the network has zero-mean additive noise:

$$Q_{\bar{\phi}}(s,a) = Q^\star(s,a) + \epsilon_{s,a}, \qquad \mathbb{E}\Big[\epsilon_{s,a} \;\Big|\; s,a\Big] = 0.$$

The target takes the maximum among computed values. However, this max operator can introduce *positive bias*, even with unbiased noise:

$$\begin{aligned}\mathbb{E}\left[\max_{a\in\mathcal{A}} Q_{\bar{\phi}}(s,a) \;\middle|\; s\right] &\geq \max_{a\in\mathcal{A}} Q^\star(s,a) + \mathbb{E}\left[\max_{a\in\mathcal{A}} \epsilon_{s,a} \;\middle|\; s\right] \\ &\geq \max_{a\in\mathcal{A}} Q^\star(s,a) + \max_{a\in\mathcal{A}} \mathbb{E}\Big[\epsilon_{s,a} \;\Big|\; s,a\Big] = \max_{a\in\mathcal{A}} Q^\star(s,a).\end{aligned}$$

Intuitively, among noisy value estimates, the maximum tends to select actions whose values are overestimated.

Note that we may write

$$\max_{a\in\mathcal{A}} Q_{\bar{\phi}}(s,a) = Q_{\bar{\phi}}\Big(s, \operatorname*{argmax}_{a\in\mathcal{A}} Q_{\bar{\phi}}(s,a)\Big).$$

The same network is used in *choosing the best action* and *evaluating the action*. This is the key cause of the noise not canceling out despite unbiasedness.

Double DQN

Double Q-learning [1] reduces this correlation by decoupling the two objectives. First, write the approximated Bellman optimality operator in a decoupled way:

$$\widehat{\mathcal{B}^\star_\mathcal{Q}[Q_1, Q_2]}(s,a) = \mathbb{E}_{(r,s'):(s,a,r,s')\in\mathcal{D}}\left[r + \gamma Q_1\left(s', \operatorname*{argmax}_{a'\in\mathcal{A}} Q_2(s',a')\right)\right].$$

Then, actually maintain and update two Q-functions Q_A and Q_B:

$$Q_A \leftarrow \widehat{\mathcal{B}^\star_\mathcal{Q}[Q_B, Q_A]}, \quad Q_B \leftarrow \widehat{\mathcal{B}^\star_\mathcal{Q}[Q_A, Q_B]}.$$

The action is chosen according to one estimator, while its value is evaluated by the other. If their errors are sufficiently decorrelated, overestimation is reduced.

In DQN, we already have two networks: the current network Q_ϕ and the target network $Q_{\bar{\phi}}$. **Double DQN** [2] uses Q_ϕ to select the action and $Q_{\bar{\phi}}$ to evaluate it. Double DQN changes only the target computation, but it often helps a lot.

$$\phi \leftarrow \operatorname*{argmin}_{\phi'} \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma Q_{\bar{\phi}}\left(s', \operatorname*{argmax}_{a'\in\mathcal{A}} Q_\phi(s',a')\right) - Q_{\phi'}(s,a)\right)^2\right].$$

[1] H. van Hasselt, "Double Q-learning," *NIPS*, 2010

[2] H. van Hasselt et al., "Deep Reinforcement Learning with Double Q-learning," *AAAI*, 2016

Policy-Based RL

Policy Gradient Methods

In this section, we study **Policy-Based RL**. Policy-based RL models and optimizes the policy directly, unlike value-based RL where we always take the greedy policy w.r.t. an estimated value function.

We especially focus on **Policy Gradient (PG)** methods. PG aims to run *gradient descent* directly on the policy. This requires the policy to be a parameterized model π_θ. The optimization objective is then

$$\underset{\theta}{\text{maximize}} \quad \mathcal{J}(\theta) = \mathbb{E}_{s\sim p_0}\Big[V^{\pi_\theta}(s)\Big] = \frac{1}{1-\gamma}\mathbb{E}_{\substack{s\sim d_\gamma^{\pi_\theta,p_0}\\ a\sim\pi_\theta(\cdot|s)}}\Big[r(s,a)\Big],$$

where p_0 is the **initial state distribution** and $d_\gamma^{\pi_\theta,p_0}$ is the **discounted state occupancy distribution** of the Markov decision process:

$$d_\gamma^{\pi_\theta,p_0}(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t \mathbb{P}_{\substack{s_0\sim p_0\\ a_t\sim\pi_\theta(\cdot|s_t)\\ (r_t,s_{t+1})\sim p(\cdot,\cdot|s_t,a_t)}}\Big(s_t = s\Big).$$

Stationary State Distributions

Let d^{π_θ} denote the **stationary state distribution** of the MDP:

$$d^{\pi_\theta}(s) = \lim_{T \to \infty} \mathbb{P}_{\substack{s_0 \sim p_0 \\ a_t \sim \pi_\theta(\cdot|s_t) \\ (r_t, s_{t+1}) \sim p(\cdot,\cdot|s_t,a_t)}} \Big(s_T = s \Big).$$

The following two theorems characterize d^{π_θ} and its relationship to $d_\gamma^{\pi_\theta, p_0}$.

Theorem (Stationary State Distributions in MDPs)

In irreducible and positive recurrent MDPs, a unique stationary distribution exists. If the MDP is also aperiodic, the state distribution converges to the stationary distribution from any initial state distribution.

Theorem (Discounted State Occupancy Distributions at the Limit)

In irreducible and positive recurrent MDPs, the discounted state occupancy distribution converges to the stationary state distribution as $\gamma \to 1$.

Irreducibility and positive recurrence are plausible assumptions in many deep RL scenarios. A large γ is also common. Therefore, in the following slides, we assume the unique existence of d^{π_θ}, and will equivalently write $d_\gamma^{\pi_\theta, p_0} \approx d^{\pi_\theta}$.

REINFORCE

Computing the gradient of $\mathcal{J}(\theta)$ seems tricky, but the **Policy Gradient Theorem** gives us a strong, effective result that simplifies the computation.

Theorem (Policy Gradient Theorem)

In continuing (non-episodic) MDPs, we have

$$\nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{\substack{s \sim d^{\pi_\theta} \\ a \sim \pi_\theta(\cdot|s)}} \Big[\nabla_\theta \log \pi_\theta(a \mid s) Q^{\pi_\theta}(s, a) \Big].$$

Furthermore, we can also show that this is equal to

$$\nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{\substack{s \sim d^{\pi_\theta} \\ a \sim \pi_\theta(\cdot|s)}} \Big[\nabla_\theta \log \pi_\theta(a \mid s) \Big(\hat{Q}(s, a) - b(s) \Big) \Big],$$

if $\hat{Q}$ is an unbiased estimator of Q^{π_θ} and b is a function independent to a. b is often called a **baseline** function. Choices of $\hat{Q}$ and b may vary.

This entire algorithmic framework—taking first-order gradient updates using this transformation—is called **REINFORCE** [1].

[1] R. J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Machine Learning*, vol. 8, no. 3–4, pp. 229–256, May 1992

Natural Policy Gradient (NPG)

Aside from REINFORCE, we may consider constraining the update size, by putting a *forward KL-constraint* to the policy network update.

However, now the update is no longer a first-order gradient step; rather it becomes a mini-optimization problem:

$$\begin{aligned} \theta \quad \leftarrow \quad & \underset{\theta'}{\operatorname{argmax}} \quad && \mathcal{J}(\theta') \\ & \text{subject to} && \mathbb{E}_{s\sim d^{\pi_\theta}}\Big[D_{\mathrm{KL}}\big(\pi_\theta(\cdot \mid s) \,\big\|\, \pi_{\theta'}(\cdot \mid s)\big)\Big] \leq \delta \end{aligned}$$

This is hard to solve directly. We may use a *first-order approximation of the objective function* at θ:

$$\mathcal{J}(\theta') \approx \mathcal{J}(\theta) + \nabla_\theta \mathcal{J}(\theta)^\top (\theta' - \theta).$$

Also, we can use a *second-order approximation of the KL divergence* at θ:

$$D_{\mathrm{KL}}\big(\pi_\theta(\cdot \mid s) \,\big\|\, \pi_{\theta'}(\cdot \mid s)\big) \approx \frac{1}{2}(\theta' - \theta)^\top \mathbf{F}\big(\pi_\theta(\cdot \mid s)\big)(\theta' - \theta),$$

where $\mathbf{F}(\cdot)$ denotes the *Fisher information matrix* of the distribution.

Natural Policy Gradient (NPG) (Cont'd)

If we write

$$\mathbf{F}(\pi_\theta) = \mathbb{E}_{s\sim d^{\pi_\theta}}\Big[\mathbf{F}\big(\pi_\theta(\cdot \mid s)\big)\Big],$$

then combining the two approximations, the optimization problem simplifies to:

$$\begin{aligned} \theta \quad \leftarrow \quad & \underset{\theta'}{\operatorname{argmax}} \quad && \nabla_\theta \mathcal{J}(\theta)^\top(\theta' - \theta) \\ & \text{subject to} && \frac{1}{2}(\theta' - \theta)^\top \mathbf{F}(\pi_\theta)(\theta' - \theta) \le \delta \end{aligned}$$

We can solve this by taking a **Natural Gradient** [1] step:

$$\theta \leftarrow \theta + \alpha \mathbf{F}(\pi_\theta)^{-1} \nabla_\theta \mathcal{J}(\theta),$$

where α is the step size. This whole algorithm is called **Natural Policy Gradient (NPG)** [2].

Because explicitly computing and inverting $\mathbf{F}(\pi_\theta)$ is computationally intractable for large models, we commonly use the **Conjugate Gradient (CG)** method to approximate the Fisher-vector product $\mathbf{F}(\pi_\theta)^{-1}\nabla_\theta \mathcal{J}(\theta)$.

[1] S.-i. Amari, "Natural Gradient Works Efficiently in Learning," *Neural Computation*, vol. 10, no. 2, pp. 251–276, 1998

[2] S. Kakade, "A Natural Policy Gradient," *NIPS*, 2001

Advantage Estimates: Optimal

Getting back to REINFORCE, the term $A(s,a) := \hat{Q}(s,a) - b(s)$ is often called an **advantage** function. When choosing $\hat{Q}$ and b, our focus should be on *reducing gradient variance*.

For $\hat{Q}$, using Q^{π_θ} is a reasonable unbiased, minimum-variance choice.

For b, the lemma below suggests the choice of

$$\frac{\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[\left(\nabla_\theta \log \pi_\theta(a \mid s)\right)^2 Q^{\pi_\theta}(s,a) \mid s\right]}{\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[\left(\nabla_\theta \log \pi_\theta(a \mid s)\right)^2 \mid s\right]} \approx \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[Q^{\pi_\theta}(s,a) \mid s\right]$$

$$= V^{\pi_\theta}(s).$$

Lemma (Minimum-Variance Conditional Estimator)

Let $s \in \mathcal{S}$ and $a \in \mathcal{A}$ be random variable, and $w : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, $b : \mathcal{S} \to \mathbb{R}$ be functions. Then,

$$\operatorname*{argmin}_b \mathbb{E}_{a,s}\left[\left(w(s,a)(Q(s,a) - b(s))\right)^2\right] = \frac{\mathbb{E}_{a|s}\left[\left(w(s,a)\right)^2 Q(s,a) \mid s\right]}{\mathbb{E}_{a|s}\left[\left(w(s,a)\right)^2 \mid s\right]}$$

Actor-Critic Methods

However, Q^{π_θ} and V^{π_θ} are not strictly actionable. This brings us back to *value estimation* as in value-based RL (especially policy evaluation of PI).

Just like PI, we may train alongside a value network V_ψ that approximates V^{π_θ}, and train the policy using advantages computed from it. This approach is called the **Actor-Critic** method, where π_θ is the **actor** and V_ψ is the **critic**.

Let the resulting advantage function A_ψ. Then, the algorithm runs as:

Algorithm 5 On-Policy Actor-Critic

while not converged **do**

  sample a set of state-action pairs: ▷ use a reasonable proxy of d^{π_θ}

$$\mathcal{D}^{\pi_\theta} = \left\{(s,a) : s \sim d^{\pi_\theta}, a \sim \pi_\theta(\cdot \mid s)\right\}$$

  update the value network with loss: ▷ with any optimization algo

$$\mathcal{L}(\psi) = \mathbb{E}_{(s,a)\sim\mathcal{D}^{\pi_\theta}}\left[\left(\left[\!\left[A_\psi(s,a) + V_\psi(s)\right]\!\right] - V_\psi(s)\right)^2\right]$$

  update the policy network by: ▷ with first-order optimizer, learning rate η

$$\theta \leftarrow \theta + \eta\mathbb{E}_{(s,a)\sim\mathcal{D}^{\pi_\theta}}\left[\nabla_\theta \log \pi_\theta(a \mid s) A_\psi(s,a)\right]$$

return π_θ

Advantage Estimates: Practical

The following are some practical estimators of $Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)$.

- **Baselined MC return**: low bias, high variance

$$A_\psi^{\mathsf{MC}}(s_t, a_t) = \sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} - V_\psi(s_t)$$

- **TD residual**: high bias, low variance

$$A_\psi^{\mathsf{TD}}(s_t, a_t) = t_t + \gamma V_\psi(s_{t+1}) - V_\psi(s_t)$$

- **n-step TD residual**: note that $A_\psi^{\mathsf{TD}} = A_\psi^{\mathsf{TD}(1)}$ and $A_\psi^{\mathsf{MC}} = A_\phi^{\mathsf{TD}(\infty)}$

$$A_\psi^{\mathsf{TD}(n)}(s_t, a_t) = \sum_{t'=t}^{t+n-1} \gamma^{t'-t} r_{t'} + \gamma^n V_\psi(s_{t+n}) - V_\psi(s_t)$$

- **GAE** [1]: note that $A_\psi^{\mathsf{TD}} = A_\psi^{\mathsf{GAE}(0)}$ and $A_\psi^{\mathsf{MC}} = A_\psi^{\mathsf{GAE}(1)}$

$$A_\psi^{\mathsf{GAE}(\lambda)}(s_t, a_t) = \sum_{n=1}^{\infty} \lambda^{n-1} A_\psi^{\mathsf{TD}(n)}(s_t, a_t) = \sum_{t'=t}^{\infty} (\gamma\lambda)^{t'-t} A_\psi^{\mathsf{TD}}(s_{t'}, a_{t'})$$

[1] J. Schulman et al., "High-Dimensional Continuous Control Using Generalized Advantage Estimation," *ICLR*, 2016

Off-Policy Policy Gradients

To increase sample efficiency, we may use *off-policy* samples, by incorporating *importance sampling ratios*.

If we want to sample from a separate behavior policy b, we may use the following transformation:

$$\begin{aligned}
\nabla_\theta \mathcal{J}(\theta) &= \mathbb{E}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta(\cdot|s)}} \Big[\nabla_\theta \log \pi_\theta(a \mid s) A(s,a)\Big] \\
&= \mathbb{E}_{\substack{s\sim d^{b} \\ a\sim b(\cdot|s)}} \left[\frac{\pi_\theta(a \mid s)}{b(a \mid s)} \nabla_\theta \log \pi_\theta(a \mid s) A(s,a)\right] \\
&= \nabla_\theta \underbrace{\mathbb{E}_{\substack{s\sim d^{b} \\ a\sim b(\cdot|s)}} \left[\frac{\pi_\theta(a \mid s)}{b(a \mid s)} A(s,a)\right]}_{=:\mathcal{J}_{\text{off}}(\theta;b)}.
\end{aligned}$$

$\mathcal{J}_{\text{off}}(\theta; b)$ is often called the **surrogate objective**. When using first-order optimizers, we may treat $-\mathcal{J}_{\text{off}}(\theta; b)$ as a loss and backpropagate through it.

However, importance sampling estimators may become inaccurate when the sampling distribution is too far from the nominal distribution. We must explicitly constrain b to be close to π_θ.

Trust Region Methods: TRPO

Trust Region Policy Optimization (TRPO) [1] tackles the off-policy coverage problem in the setup where b is a stale old policy π_{θ_0}.

To do so, TRPO puts a *forward KL-constraint* to the policy network update. The update step is no longer a first-order gradient update; rather it becomes a mini-optimization problem:

$$\begin{aligned} \theta \quad \leftarrow \quad & \underset{\theta'}{\operatorname{argmax}} \quad && \mathcal{J}_{\text{off}}(\theta'; \pi_{\theta_0}) \\ & \text{subject to} && \mathbb{E}_{s \sim d^{\pi_{\theta_0}}} \Big[D_{\text{KL}} \big(\pi_{\theta_0}(\cdot \mid s) \,\big\|\, \pi_{\theta'}(\cdot \mid s) \big) \Big] \leq \delta \end{aligned}$$

Solving this problem resmbles NPG. Yet the first-order approximation of $\mathcal{J}_{\text{off}}(\theta'; \pi_{\theta'})$ at θ_0 requires an additional step:

$$\begin{aligned} \mathcal{J}_{\text{off}}(\theta'; \pi_{\theta'}) &\approx \mathcal{J}_{\text{off}}(\theta_0; \pi_{\theta'}) + \nabla_\theta \mathcal{J}_{\text{off}}(\theta_0; \pi_{\theta'})^\top (\theta' - \theta_0) \\ &= \mathcal{J}_{\text{off}}(\theta_0; \pi_{\theta'}) + \nabla_\theta \mathcal{J}(\theta_0)^\top (\theta' - \theta_0). \end{aligned}$$

[1] J. Schulman et al., "Trust Region Policy Optimization," *ICML*, 2015

Trust Region Methods: TRPO (Cont'd)

The gradient equivalence between the original objective and the surrogate objective transforms the problem into the exact same form as NPG:

$$\begin{aligned} \theta \quad \leftarrow \quad & \underset{\theta'}{\text{argmax}} \quad && \nabla_\theta \mathcal{J}(\theta_0)^\top (\theta' - \theta_0) \\ & \text{subject to} && \frac{1}{2}(\theta' - \theta_0)^\top \mathbf{F}(\pi_{\theta_0})(\theta' - \theta_0) \leq \delta \end{aligned}$$

Therefore, as in NPG, we can take the natural gradient step:

$$\theta \leftarrow \theta_0 + \alpha \mathbf{F}(\pi_{\theta_0})^{-1} \nabla \mathcal{J}(\theta_0).$$

Furthermore, TRPO suggests a *constraint-maximizing* choice of α:

$$\alpha = \sqrt{\frac{2\delta}{\nabla \mathcal{J}(\theta_0)^\top \mathbf{F}(\pi_{\theta_0}) \nabla \mathcal{J}(\theta_0)}},$$

which is the maximum safe step size that will hit the boundary of the trust region.

Trust Region Methods: PPO

Proximal Policy Optimization (PPO) [1] simplifies TRPO by replacing the KL-constraint with some proximal constraints. These simplifications again allow us to use first-order optimizers.

There are two versions of PPO, which are also often used together.

1. **PPO-Clip**: add *clipping* to the surrogate objective.

$$\mathbb{E}_{\substack{s\sim d^{\pi_{\theta_0}} \\ a\sim\pi_{\theta_0}(\cdot|s)}}\left[\min\left(\frac{\pi_\theta(a\mid s)}{\pi_{\theta_0}(a\mid s)}A(s,a), \operatorname{clip}\left(\frac{\pi_\theta(a\mid s)}{\pi_{\theta_0}(a\mid s)}, 1-\epsilon, 1+\epsilon\right)A(s,a)\right)\right].$$

2. **PPO-KL**: add a *forward KL-penalty* to the surrogate objective.

$$\mathbb{E}_{\substack{s\sim d^{\pi_{\theta_0}} \\ a\sim\pi_{\theta_0}(\cdot|s)}}\left[\frac{\pi_\theta(a\mid s)}{\pi_{\theta_0}(a\mid s)}A(s,a) - \beta D_{\mathrm{KL}}\big(\pi_{\theta_0}(\cdot\mid s)\,\big\|\,\pi_\theta(\cdot\mid s)\big)\right].$$

[1] J. Schulman et al., "Proximal Policy Optimization Algorithms," *arXiv*, 2017

Continuous Actions: DPG

Sampling actions become tricky when actions are continuous, i.e., $\mathcal{A} \subseteq \mathbb{R}^d$. In this case, a better approach may be to train a *deterministic policy* that directly outputs the maximizing action.

Let $\mu_\theta : \mathcal{S} \to \mathcal{A}$ a parameterized deterministic policy. Then, we can derive the deterministic version of the policy gradient theorem:

Theorem (Deterministic Policy Gradient Theorem)

In continuing (non-episodic) MDPs, we have

$$\nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{s \sim d^{\mu_\theta}} \left[\nabla_\theta \mu_\theta(s)^\top \nabla_a Q^{\mu_\theta}\left(s, \mu_\theta(s)\right) \right].$$

Similar to normal policy gradients, we can replace Q^{μ_θ} by any unbiased estimator of it. However, there are no *baselines* in deterministic settings, since subtracting an action-independent baseline function $b(s)$ will vanish by ∇_a.

Continuous Actions: DPG (Cont'd)

Deterministic Policy Gradient (DPG) [1] uses this theorem directly to run a deterministic version of actor-critic.

Since are no baselines in DPG, it is more convenient to maintain and train a Q network rather than a V network. This also allows us to use *off-policy* samples. The fully on-policy version of the algorithm runs as:

Algorithm 6 On-Policy DPG

1: **while** not converged **do**

2: sample a set of 1-step transitions: ▷ use a reasonable proxy of d^{μ_θ}

$$\mathcal{D}^{\mu_\theta} = \left\{(s, a, r, s') : s \sim d^{\mu_\theta}, a = \mu_\theta(s), (r, s') \sim p(\cdot, \cdot \mid s, a)\right\}$$

3: update the value network with loss: ▷ with any optimization algo

$$\mathcal{L}(\phi) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}^{\mu_\theta}} \left[\left(\left[\!\left[r + \gamma Q_\phi\big(s', \mu_\theta(s')\big)\right]\!\right] - Q_\phi(s, a)\right)^2\right]$$

4: update the policy network by: ▷ with first-order optimizer, learning rate η

$$\theta \leftarrow \theta + \eta \mathbb{E}_{(s,\cdot,\cdot,\cdot) \sim \mathcal{D}^{\mu_\theta}} \left[\nabla_\theta \mu_\theta(s)^\top \nabla_a Q_\phi\big(s, \mu_\theta(s)\big)\right]$$

5: **return** μ_θ

[1] D. Silver et al., "Deterministic Policy Gradient Algorithms," *ICML*, 2014

Enhancements in Critic Learning from DQNs: DDPG

Deep Deterministic Policy Gradient (DDPG) [1] is an enhancement over DPG that focuses on stabilization. Since DPG uses a Q-network as the critic, we can borrow standard tricks that were discussed with DQNs.

DDPG specifically uses:

1. **Target Networks**: Maintain target networks for both the actor and the critic, and apply Polyak updates. These are only used for critic learning.
2. **Off-Policy Samples**: To encourage exploration, use a noisy behavior policy

$$b(s) = \mu_\theta(s) + \epsilon, \qquad \epsilon \in \mathbb{R}^d.$$

   and construct an off-policy dataset $\mathcal{D}$. Original DDPG uses *time correlated Ornstein–Uhlenbeck noise*, but a simple *zero-mean Gaussian noise* is also known to work well. Also note that IS correction is not needed here, since actions are deterministic and do not require sampling.
3. **Replay Buffers**: Save past samples in $\mathcal{D}$ and reuse them (this was actually used in DPG too).

[1] T. P. Lillicrap et al., "Continuous control with deep reinforcement learning," *ICLR*, 2016

Enhancements in Critic Learning from DQNs: TD3

Twin Delayed Deep Deterministic (TD3) [1] adds more tricks on DDPG.

1. **Clipped Double Q-Learning**: Like double Q-learning, we maintain and train two critics to prevent overestimation. However, rather than always evaluating the target from the opposite critic, TD3 takes the *minimum* of the two for more pessimism. We update the critics with losses:

$$\mathcal{L}(\phi_1) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \min_{i\in\{1,2\}} Q_{\bar{\phi}_i}\left(s', \mu_{\bar{\theta}}(s')\right) - Q_{\phi_1}(s,a)\right)^2\right],$$

$$\mathcal{L}(\phi_2) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \min_{i\in\{1,2\}} Q_{\bar{\phi}_i}\left(s', \mu_{\bar{\theta}}(s')\right) - Q_{\phi_2}(s,a)\right)^2\right].$$

The policy is always updated using the first critic:

$$\theta \leftarrow \theta + \eta\mathbb{E}_{(s,\cdot,\cdot,\cdot)\sim\mathcal{D}}\left[\nabla_\theta \mu_\theta(s)^\top \nabla_a Q_{\phi_1}\left(s, \mu_\theta(s)\right)\right].$$

[1] S. Fujimoto et al., "Addressing Function Approximation Error in Actor-Critic Methods," *ICML*, 2018

Enhancements in Critic Learning from DQNs: TD3 (Cont'd)

2. **Delayed Policy Network Updates**: To slow down policy training and make the critic fully adapt to the policy, TD3 updates the policy network periodically.

3. **Delayed Target Network Updates**: The target networks, for both actor and critic, are also updated periodically, additional to the Polyak updates.

4. **Target Policy Smoothing**: If the critic accidentally assigns a very high value to a narrow region of action space, the actor may learn to choose exactly that action, even though the value estimate is just an artifact.

   To solve this, TD3 adds a small amount of clipped noise to the selected action in target computation. Replace the $\mu_{\bar{\theta}}(s')$ term in the critic learning target by

   $$\mu_{\bar{\theta}}(s') + \mathrm{clip}(\nu, -c, +c), \qquad \nu \sim \mathcal{N}(0, \sigma^2).$$

   This makes the learned critic smoother and discourages the actor from exploiting brittle overestimation errors.

Stochastic Continuous Control: SAC

Soft Actor-Critic (SAC) [1] [2] is another off-policy actor-critic method for continuous control. However, unlike DPG-like methods, SAC uses a *stochastic policy* as ordinary policy gradients.

The key idea is **Maximum-Entropy RL**, which optimizes both expected return and entropy:

$$\begin{aligned}\underset{\pi\in\Delta(\mathcal{A})^{\mathcal{S}}}{\text{maximize}} \quad \mathcal{J}_{\mathsf{MaxEnt}}(\pi) &= \mathbb{E}_{\substack{s_0\sim p_0\\ a_t\sim\pi(\cdot|s_t)\\ (r_t,s_{t+1})\sim p(\cdot,\cdot|s_t,a_t)}}\left[\sum_{t=0}^{\infty}\gamma^t\Big(r_t+\alpha\mathcal{H}\big(\pi(\cdot\mid s_t)\big)\Big)\right]\\ &= \mathbb{E}_{\substack{s_0\sim p_0\\ a_t\sim\pi(\cdot|s_t)\\ (r_t,s_{t+1})\sim p(\cdot,\cdot|s_t,a_t)}}\left[\sum_{t=0}^{\infty}\gamma^t\big(r_t-\alpha\log\pi(a_t\mid s_t)\big)\right].\end{aligned}$$

Hence, SAC prefers policies that achieve high task reward while remaining stochastic. This way, the policy is encouraged to explore more broadly and learn several equally good strategies.

---

[1] T. Haarnoja et al., "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," *ICML*, 2018

[2] T. Haarnoja et al., "Soft Actor-Critic Algorithms and Applications," *arXiv*, 2019

Stochastic Continuous Control: SAC (Cont'd)

To align with the maximum entropy RL objective, SAC defines **soft value functions** for a policy π:

$$\widetilde{V}^{\pi}(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}\Big[\widetilde{Q}^{\pi}(s,a) - \alpha\log\pi(a \mid s) \;\Big|\; s\Big],$$
$$\widetilde{Q}^{\pi}(s,a) = \mathbb{E}_{(r,s')\sim p(\cdot,\cdot|s,a)}\Big[r + \gamma\widetilde{V}^{\pi}(s') \;\Big|\; s,a\Big].$$

We can rearrange these into **soft Bellman equations** at π:

$$\widetilde{V}^{\pi}(s) = \mathbb{E}_{\substack{a\sim\pi(\cdot|s)\\(r,s')\sim p(\cdot,\cdot|s,a)}}\Big[r + \gamma\widetilde{V}^{\pi}(s') - \alpha\log\pi(a \mid s) \;\Big|\; s\Big],$$
$$\widetilde{Q}^{\pi}(s,a) = \mathbb{E}_{\substack{(r,s')\sim p(\cdot,\cdot|s,a)\\a'\sim\pi(\cdot|s')}}\Big[r + \gamma\Big(\widetilde{Q}^{\pi}(s',a') - \alpha\log\pi(a' \mid s')\Big) \;\Big|\; s,a\Big].$$

Similarly, we can also define the **soft Bellman operators** at π:

$$\widetilde{\mathcal{B}}^{\pi}_{\mathcal{V}}[V](s) = \mathbb{E}_{\substack{a\sim\pi(\cdot|s)\\(r,s')\sim p(\cdot,\cdot|s,a)}}\Big[r + \gamma V(s') - \alpha\log\pi(a \mid s) \;\Big|\; s\Big],$$
$$\widetilde{\mathcal{B}}^{\pi}_{\mathcal{Q}}[Q](s,a) = \mathbb{E}_{\substack{(r,s')\sim p(\cdot,\cdot|s,a)\\a'\sim\pi(\cdot|s')}}\Big[r + \gamma\big(Q(s',a') - \alpha\log\pi(a' \mid s')\big) \;\Big|\; s,a\Big].$$

Stochastic Continuous Control: SAC (Cont'd)

This is a very natural definition, since we can observe that maximum entropy RL objective is equivalent to maximizing with respect to the soft value functions, due to a telescoping sum:

$$\begin{aligned}
\mathcal{J}_{\mathsf{MaxEnt}}(\pi) &= \mathbb{E}_{\substack{s_0 \sim p_0 \\ a_t \sim \pi(\cdot|s_t) \\ (r_t, s_{t+1}) \sim p(\cdot,\cdot|s_t,a_t)}} \left[\sum_{t=0}^{\infty} \gamma^t \big(r_t - \alpha \log \pi(a_t \mid s_t) \big) \right] \\
&= \mathbb{E}_{\substack{s_0 \sim p_0 \\ a_t \sim \pi(\cdot|s_t) \\ (r_t, s_{t+1}) \sim p(\cdot,\cdot|s_t,a_t)}} \left[\sum_{t=0}^{\infty} \gamma^t \Big(\widetilde{V}^{\pi}(s_t) - \gamma \widetilde{V}^{\pi}(s_{t+1}) \Big) \right] \\
&= \mathbb{E}_{s \sim p_0} \Big[\widetilde{V}^{\pi}(s) \Big] = \mathbb{E}_{\substack{s \sim p_0 \\ a \sim \pi(\cdot|s)}} \Big[\widetilde{Q}^{\pi}(s,a) - \alpha \log \pi(a \mid s) \Big].
\end{aligned}$$

The minimizer of this objective is called the **soft optimal policy**. Under this characterization, we may also similarly define the notion of a **greedy policy** with respect to soft value functions. A natural definition is:

$$\widetilde{\mathsf{Greedy}}(V)(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \mathbb{E}_{(r,s') \sim p(\cdot,\cdot|s,a)} \Big[r + \gamma V(s') - \alpha \log \pi(a \mid s) \;\Big|\; s, a \Big]$$

$$\widetilde{\mathsf{Greedy}}(Q)(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \Big(Q(s,a) - \alpha \log \pi(a \mid s) \Big).$$

Stochastic Continuous Control: SAC (Cont'd)

Finally, using $\widetilde{\mathcal{B}}_{\mathcal{V}}$, $\widetilde{\mathcal{B}}_{\mathcal{Q}}$, and $\widetilde{\text{Greedy}}$, we may similarly run **soft policy iteration (SPI)** to find a soft optimal policy.

The convergence of SPI to an optimal policy is guaranteed by:

Theorem (γ-Contractiveness of Soft Bellman Operators)

$\widetilde{\mathcal{B}}^{\pi}_{\mathcal{V}}$ and $\widetilde{\mathcal{B}}^{\pi}_{\mathcal{Q}}$ are γ-contractions., i.e., for any $V_1, V_2 \in \mathbb{R}^{\mathcal{S}}$ and $Q_1, Q_2 \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$,

$$\left\|\widetilde{\mathcal{B}}^{\pi}_{\mathcal{V}}[V_1] - \widetilde{\mathcal{B}}^{\pi}_{\mathcal{V}}[V_2]\right\| \leq \gamma\|V_1 - V_2\|, \quad \left\|\widetilde{\mathcal{B}}^{\pi}_{\mathcal{Q}}[Q_1] - \widetilde{\mathcal{B}}^{\pi}_{\mathcal{Q}}[Q_2]\right\| \leq \gamma\|Q_1 - Q_2\|,$$

where $\|\cdot\|$ is measured by the supremum norm.

Theorem (Soft Policy Improvement Theorem)

Assume $\gamma \in (0,1)$, $\mathcal{A}$ has finite Lebesgue measure, and rewards are bounded. From a given π_{curr}, compute $\pi_{next} = \widetilde{\text{Greedy}}(V^{\pi_{curr}})$ or $\pi_{next} = \widetilde{\text{Greedy}}(Q^{\pi_{curr}})$. Then, either case,

$$\widetilde{V}^{\pi_{next}}(s) \geq \widetilde{V}^{\pi_{curr}}(s), \quad \forall s \in \mathcal{S}.$$

Furthermore, after a finite number of iterations, V (or Q) becomes an soft optimal value function, and π becomes an soft optimal policy.

Stochastic Continuous Control: SAC (Cont'd)

As an instance of SPI, SAC *approximately* performs actor-critic. While the policy update step may use first-order methods, SAC doesn't strictly rely on the *policy gradient theorem*, which makes it different from normal actor-critic.

Instead, direct greedy maximization already gives gradients with respect to the policy. For simple maximization, SAC uses Q functions, which also allows us to use *off-policy* samples. The fully on-policy version of the algorithm runs as:

Algorithm 7 On-Policy SAC (Simplified)

1: **while** not converged **do**

2: sample a set of 1-step transitions (and actions): ▷ use a reasonable proxy of p_0

$$\mathcal{D}^{\pi_\theta} = \left\{(s,a,r,s',a') : s \sim d^{p_0}, a \sim \pi_\theta(\cdot \mid s), (r,s') \sim p(\cdot,\cdot \mid s,a), a' \sim \pi_\theta(\cdot \mid s')\right\}$$

3: update the value network with loss: ▷ with any optimization algo

$$\mathcal{L}(\phi) = \mathbb{E}_{(s,a,r,s',a') \sim \mathcal{D}^{\pi_\theta}} \left[\left(\left[\!\left[r + \gamma \left(Q_\phi(s',a') - \alpha \log \pi_\theta(a' \mid s') \right) \right]\!\right] - Q_\phi(s,a) \right)^2 \right]$$

4: update the policy network with loss: ▷ with any optimization algo

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,\cdot,\cdot,\cdot) \sim \mathcal{D}^{\pi_\theta}} \left[\alpha \log \pi_\theta(a \mid s) - Q_\phi(s,a) \right]$$

5: **return** π_θ

Stochastic Continuous Control: SAC (Cont'd)

In practice, SAC uses off-policy transition samples collected and stored in a replay buffer $\mathcal{D}$.

Also, like DDQN or TD3, SAC uses two critics Q_{ϕ_1}, Q_{ϕ_2} and target critics $Q_{\bar{\phi}_1}, Q_{\bar{\phi}_2}$. For a transition $(s, a, r, s') \sim \mathcal{D}$, define the target

$$\widehat{Q}(s,a) = r + \gamma \mathbb{E}_{a' \sim \pi_\theta(\cdot|s')} \Big[\min_{i \in \{1,2\}} Q_{\bar{\phi}_i}(s', a') - \alpha \log \pi_\theta(a' \mid s') \Big].$$

Then, the critic losses are

$$\mathcal{L}(\phi_i) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \Big[\Big(\big[\!\big[\widehat{Q}(s,a) \big]\!\big] - Q_{\phi_i}(s,a) \Big)^2 \Big], \qquad i = 1, 2,$$

and the actor loss is

$$\mathcal{L}(\theta) = \mathbb{E}_{\substack{(s,\cdot,\cdot,\cdot) \sim \mathcal{D} \\ a \sim \pi_\theta(\cdot|s)}} \Big[\alpha \log \pi_\theta(a \mid s) - \min_{i \in \{1,2\}} Q_{\phi_i}(s,a) \Big].$$

Also, the temperature α is commonly tuned automatically using the following loss with a target entropy $\bar{\mathcal{H}}$:

$$\mathcal{L}(\alpha) = \mathbb{E}_{\substack{(s,\cdot,\cdot,\cdot) \sim \mathcal{D} \\ a \sim \pi_\theta(\cdot|s)}} \Big[-\alpha \Big(\log \pi_\theta(a \mid s) + \bar{\mathcal{H}} \Big) \Big].$$

Stochastic Continuous Control: SAC (Cont'd)

Combining all details, the overall SAC algorithm runs as follows:

Algorithm 8 SAC

while not converged **do**
  sample one step transitions and add then to $\mathcal{D}$
  while not converged **do**
    sample transitions from a replay buffer $\mathcal{D}$
    update critics using the soft Bellman target (minimize $\mathcal{L}(\phi_1)$, $\mathcal{L}(\phi_2)$)
    update the actor using the entropy-regularized objective (minimize $\mathcal{L}(\theta)$)
    (optionally) update α using the target-entropy loss (minimize $\mathcal{L}(\alpha)$)
    update target critics by Polyak averaging
return π_θ

In fact, although SAC was originally designed for *continuous actions*, we can also use the algorithm for *discrete actions* with no modification in design. The soft policy improvement theorem hold equivalently under the assumptions of the original policy improvement theorem for discrete actions.

In *offline RL*, SAC is often used as a base actor-critic method, but vanilla SAC can still choose OOD actions. Methods like CQL add pessimism to make it safer.

Model-Based RL

Overview

Model-Based RL is a class of RL algorithms where we first learn, or are given, an approximation to the environment model:

$$\widehat{p}_\eta(r, s' \mid s, a) \approx p(r, s' \mid s, a).$$

Once we have a model, we can query $\widehat{p}_\eta$ instead of the real environment. This gives us three common ways to use the model:

1. **Planning**: choose actions by optimizing predicted future rewards under $\widehat{p}_\eta$.
2. **Synthetic data generation**: generate imagined transitions and train a model-free learner on them.
3. **Value expansion**: use short model rollouts to construct better value-learning targets.

The main benefit is *sample efficiency*. The main risk is *model bias*: if $\widehat{p}_\eta$ is wrong, the policy or value function may learn from fake experience.

Learning a Dynamics Model

Suppose we have collected a replay buffer of real transitions

$$\mathcal{D}_{\text{env}} = \big\{(s, a, r, s') : (r, s') \sim p(\cdot, \cdot \mid s, a)\big\}.$$

The underlying policy may be arbitrary, or mixed.

A stochastic dynamics model can be trained by maximum likelihood:

$$\underset{\eta}{\text{maximize}} \;\; \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}_{\text{env}}} \Big[\log \widehat{p}_\eta(r, s' \mid s, a)\Big].$$

In continuous-control tasks, a common deterministic simplification is to predict the reward and the state difference:

$$\widehat{r}_\eta(s, a) \approx r, \qquad \widehat{f}_\eta(s, a) \approx s' - s,$$

with MSE loss:

$$\underset{\eta}{\text{minimize}} \;\; \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}_{\text{env}}} \Big[\big(\widehat{r}_\eta(s, a) - r\big)^2 + \Big\|\widehat{f}_\eta(s, a) - (s' - s)\Big\|^2\Big].$$

Then an imagined transition is generated by

$$\widehat{s}' = s + \widehat{f}_\eta(s, a), \qquad \widehat{r} = \widehat{r}_\eta(s, a).$$

Dyna

Dyna [1] is the classical template for combining *learning*, *planning*, and *acting*. At each real environment step, Dyna does two things

1. update the value function or policy using the real transition;
2. update the learned model, then sample imagined transitions from the model and perform extra value updates.

Algorithm 9 Dyna-Style Learning

while not converged **do**
  act in the real environment and observe (s, a, r, s')
  update the value function using (s, a, r, s')
  update the model $\widehat{p}_\eta$ using (s, a, r, s')
  for $m = 1, \ldots, M$ **do**
    sample a previously seen state-action pair $(\tilde{s}, \tilde{a})$
    sample $(\tilde{r}, \tilde{s}') \sim \widehat{p}_\eta(\cdot, \cdot \mid \tilde{s}, \tilde{a})$
    update the value function using $(\tilde{s}, \tilde{a}, \tilde{r}, \tilde{s}')$

Dyna gives the core idea, but in deep RL we should use the model in a more *restricted* way. If we use a learned neural dynamics model for long rollouts, the model may drift into states that are far from the real replay buffer.

[1] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM SIGART Bulletin*, vol. 2, no. 4, 1991

Model-Based Acceleration (MBA)

Model-Based Acceleration (MBA) [1] combines Q-learning with a learned model in continuous action settings.

The learned model is mainly used to generate additional transitions:

$$(s, a, \cdot, \cdot) \sim \mathcal{D}_{\text{env}}, \qquad (\widehat{r}, \widehat{s}') \sim \widehat{p}_\eta(\cdot, \cdot \mid s, a),$$

which are inserted into a model buffer $\mathcal{D}_\eta$. After generating model-based transitions, MBA trains the Q-function on both real and imagined data:

$$\mathcal{D}_{\text{train}} = \mathcal{D}_{\text{env}} \cup \mathcal{D}_\eta.$$

Since action maximization is hard in continuous action spaces, the paper uses **Normalized Advantage Functions (NAF)**, where the Q function is the sum of two parameterized networks:

$$Q_{\phi,\psi}(s,a) = V_\psi(s) \underbrace{-\frac{1}{2}\big(a - \mu_\phi(s)\big)^\top \Big(L_\phi(s) L_\phi(s)^\top\Big)\big(a - \mu_\phi(s)\big)}_{=: A_\phi(s,a)},$$

[1] S. Gu et al., "Continuous Deep Q-Learning with Model-based Acceleration," *ICML*, 2016

Model-Based Acceleration (MBA) (Cont'd)

The critic update is the usual fitted Q-learning objective:

$$\mathcal{L}(\phi, \psi) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}_{\text{train}}} \left[\left(\left[\!\left[r + \gamma \max_{a' \in \mathcal{A}} Q_{\phi,\psi}(s', a') \right]\!\right] - Q_{\phi,\psi}(s, a) \right)^2 \right].$$

With NAF, the maximization inside the target is available in closed-form:

$$\max_{a' \in \mathcal{A}} Q_{\phi,\psi}(s', a') = V_{\psi}(s'),$$

since $L_\phi(s) L_\phi(s)^\top$ is positive semidefinite.

The high-level idea is very Dyna-like: *use the real environment to learn a model, then use the model to create extra updates*. The caveat is that imagined transitions are only trustworthy near well-covered regions of $\mathcal{D}_{\text{env}}$.

Model-Based Value Expansion (MVE)

Model-Based Value Expansion (MVE) [1] combines actor-critic methods with a learned model in continuous action settings. MVE uses $\widehat{p}_\eta$ not for extra replay data, but as a way to construct a multi-step target for critic learning.

Since the model is accurate only near well-covered regions of $\mathcal{D}_{\text{env}}$, we first start from a real state-action pair $(s, a, \cdot, \cdot) \in \mathcal{D}_{\text{env}}$. Then we rollout the model for a *short* number of steps:

$$(\widehat{r}_t, \widehat{s}_{t+1}) \sim \widehat{p}_\eta(\cdot, \cdot \mid \widehat{s}_t, \widehat{a}_t),\ \widehat{a}_{t+1} \sim \pi_\theta(\cdot \mid \widehat{s}_{t+1}), \quad (t = 0, 1, \ldots, n-1),$$

where $\widehat{s}_0 = s$ and $\widehat{a}_0 = a$.

Then, we use the n-step bootstrapped return:

$$\sum_{t=0}^{n-1} \gamma^t \widehat{r}_t + \gamma^n Q_\phi(\widehat{s}_n, \widehat{a}_n)$$

as the critic learning target. The overall algorithm then follows standard model-free continuous action actor-critic methods, e.g., DDPG, TD3, SAC etc.

[1] V. Feinberg et al., "Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning," *ICML*, 2018

Model-Based Policy Optimization (MBPO)

Model-Based Policy Optimization (MBPO) [1] is also a method that combines actor-critic methods with a learned model in continuous action settings.

However, MBPO doesn't use $\widehat{p}_\eta$ for critic learning targets, but instead uses it to generate additional transitions, similar to MBA.

Other than that, the overall algorithm also follows standard model-free continuous action actor-critic methods. The original paper used SAC.

Remark

Overall, MBA / MVE / MBPO are all Dyna-like methods targeted for continuous action settings, with differences in *where the model is used*.

Method	Model usage	Underlying Algorithm
MBA	replay-buffer augmentation	Q-learning
MVE	value-target construction	actor-critic (DPG-like)
MBPO	replay-buffer augmentation	actor-critic (DPG-like)

[1] M. Janner et al., "When to Trust Your Model: Model-Based Policy Optimization," *NeurIPS*, 2019

Why Short Rollouts?

MVE illustrates a central tradeoff in model-based RL.

A longer model rollout can reduce dependence on bootstrapping, but it also increases model bias. If the learned model has local error

$$\epsilon_m(s,a) \approx d\big(p(\cdot,\cdot \mid s,a), \widehat{p}_\eta(\cdot,\cdot \mid s,a)\big),$$

then after rolling out under $\widehat{p}_\eta$, the later states follow the model-induced distribution, not the real distribution.

This creates two coupled problems:

1. **Compounding error**: small one-step errors accumulate over time.
2. **Distribution shift**: the model is queried on states/actions it was not trained on.

Therefore, many successful deep MBRL algorithms use *short* model rollouts.

Probabilistic Ensembles with Trajectory Sampling (PETS)

Probabilistic Ensembles with Trajectory Sampling (PETS) [1] focuses on learning an uncertainty-aware dynamics model.

Instead of a single deterministic model, PETS trains an ensemble of probabilistic models:

$$\widehat{p}_{\eta_j}(\cdot, \cdot \mid s, a) = \mathcal{N}\big(\mu_{\eta_j}(s, a), \Sigma_{\eta_j}(s, a)\big), \qquad j = 1, \ldots, B.$$

This captures two kinds of uncertainty:

1. **Aleatoric uncertainty**: randomness in the environment, represented by each model's predictive covariance $\Sigma_{\eta_j}(s, a)$.
2. **Epistemic uncertainty**: uncertainty due to limited data, represented by disagreement across ensemble members.

The model is used by propagating particles through the ensemble during planning. This allows the planner to estimate expected returns while accounting for model uncertainty.

---

[1] K. Chua et al., "Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models," *NeurIPS*, 2018

Probabilistic Ensembles with TS (PETS) (Cont'd)

PETS does not primarily train a global policy. Instead, at any real state $s \in \mathcal{S}$, it solves a finite-horizon planning problem under the learned ensemble model:

$$\underset{(a_0,\ldots,a_{n-1})\in\mathcal{A}^H}{\text{maximize}} \quad \mathbb{E}_{(\widehat{r}_t,\widehat{s}_{t+1})\sim\widehat{p}_\eta(\cdot,\cdot|\widehat{s}_t,a_t)}\left[\sum_{t=0}^{n-1}\gamma^t\widehat{r}_t \;\middle|\; \widehat{s}_0 = s\right].$$

In practice, this is usually approximated by the **Cross-Entropy Method (CEM)**:

Algorithm 10 CEM

1: initialize a distribution $f \in \Delta(\mathcal{A}^n)$ over length-n action sequences
2: **while** not converged **do**
3: sample N candidate action sequences from f
4: evaluate the return on each sequence, using rollouts from $\widehat{p}_\eta$ starting at s
5: refit f to the top-M sequences with highest return
6: **return** $(a_0^\star, \ldots, a_{n-1}^\star) \sim f$

Then, we can choose the first action $a_0^\star$ as the *pseudo-optimal* action at s. We may also periodically repeat *acting in the real world with CEM* and *fitting* $\widehat{p}_\eta$. This is called **model predictive control (MPC)**.

Offline RL

Overview

Suppose an offline dataset of 1-step transitions using a behavior policy b:

$$\mathcal{D}^b = \left\{(s, a, r, s') : s \sim d^b, a \sim b(\cdot \mid a), (r, s') \sim p(\cdot, \cdot \mid s, a)\right\}$$

is given, and no further interaction with the environment is allowed. That is, we can't sample the transition $(r, s') \sim p(\cdot, \cdot \mid s, a)$ during our training process.

Still, we want to maximize the expected return given a state, so we should maximize some utility function that aligns with this value. We may imagine of three straightforward approaches:

1. **Learn the V function of the behavior policy and maximize it**: This gives a policy that is strictly better than the behavior policy, due to the policy improvement theorem. This is exactly *one step* of *V function actor critic*.
2. **Iteratively learn the Q function of the current policy and maximize it**: This is exactly *Q function actor-critic*.
3. **Learn the optimal Q function and maximize it**: This is exactly *(fitted) Q-learning*.

Learning V vs Q Functions in Offline Settings

Due to the limits of offline data, the choice between V and Q is more nuanced in offline RL.

1. In PI, learning the Q function of b may be intractable, since the iteration proceeds by:

$$\underset{\phi'}{\text{minimize}} \quad \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}^b}\left[\left(\mathbb{E}_{a'\sim b(\cdot|s')}\left[r+\gamma Q_\phi(s',a')\right]-Q_{\phi'}(s,a)\right)^2\right],$$

   so we need an additional sample a' sampled from the behavior policy. It may not be guaranteed that $\mathcal{D}^b$ contains an action sampled from s'.

2. Also in PI, learning the V function of the current policy π may be intractable, since the iteration proceeds by:

$$\underset{\psi'}{\text{minimize}} \quad \mathbb{E}_{(s,\cdot,\cdot,\cdot)\sim\mathcal{D}^b}\left[\left(\mathbb{E}_{\substack{a\sim\pi(\cdot|s)\\(r,s')\sim p(\cdot,\cdot|s,a)}}\left[r+\gamma V_\psi(s')\right]-V_{\psi'}(s)\right)^2\right],$$

   so we need an additional sample a sampled from the current policy and the next transition. $\mathcal{D}^b$ should cover state-action pairs following the current policy, which is not guaranteed.

Learning V vs Q Functions in Offline Settings (Cont'd)

3. In VI, learning the optimal V function may be intractable, since the iteration proceeds by:

$$\underset{\psi'}{\text{minimize}}\ \mathbb{E}_{(s,\cdot,\cdot,\cdot)\in\mathcal{D}^b}\left[\left(\max_{a\in\mathcal{A}}\mathbb{E}_{(r,s')\sim p(\cdot,\cdot|s,a)}\left[r+\gamma V_\psi(s')\right]-V_{\psi'}(s)\right)^2\right],$$

   so we need transitions from all actions in $\mathcal{A}$. This is not guaranteed in $\mathcal{D}^b$ and is practically impossible.

Therefore, our remaining three choices are (we can call each in various names):

1. one-step V function PI = one-step V function actor-critic
2. Q function PI = Q function actor-critic = (fitted) SARSA
3. Q-function VI = (fitted) Q-learning

Distributional Shift

However, offline methods are often vulnerable to **distributional shift**.

In both PI or VI, the learned value model will only be accurate near seen inputs, i.e.,

$$(s, a) \in \mathcal{S} \times \mathcal{A} \quad \text{s.t.} \quad (s, a, \cdot, \cdot) \in \mathcal{D}^b \ \Big(\iff a \in \operatorname{supp}\big(b(\cdot \mid s)\big)\Big).$$

Therefore, the model will be unreliable for inputs far from the behavior policy's coverage, and policy learning (maximizing with respect to this learned value model) may be inaccurate.

This motivates us to constrain our policy to not move too far from the behavior policy during policy learning.

BC regularization

A minimalist approach to this end is **BC regularization**, where we add a small BC regularization term to the standard policy update objective of any method.

A representative example is **TD3+BC** [1], which adds a BC MSE term to the policy learning objective of TD3 (which is an instance of Q function actor-critic). The updated objective becomes:

$$\mathcal{J}(\theta) = \mathbb{E}_{(s,a,\cdot,\cdot)\sim\mathcal{D}}\Big[\lambda Q_\phi\big(s, \mu_\theta(s)\big) - \big(\mu_\theta(s) - a\big)^2\Big].$$

Using the deterministic policy gradient theorem, the gradient becomes:

$$\nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{(s,a,\cdot,\cdot)\sim\mathcal{D}}\Big[\lambda \nabla_\theta \mu_\theta(s)^\top \Big(\nabla_a Q_\phi\big(s, \mu_\theta(s)\big) - 2\big(\mu_\theta(s) - a\big)\Big)\Big].$$

This has an effect in making the policy stay close to the behavior policy, which directly addresses the distributional shift problem. We may similarly implement **SAC+BC**, **DDQN+BC**, etc.

[1] S. Fujimoto and S. S. Gu, "A Minimalist Approach to Offline Reinforcement Learning," *NeurIPS*, 2021

KL-Constrained Policy Improvement

A more sophisticated way to restrict policy updates is to add a **KL-constraint** to the policy update objective.

Suppose we have completely learned the target value function—V^b or Q^π or $Q^\star$—and we maximize an advantage function $\hat{A}$ induced by the learned value function.

Using reverse KL gives:

$$\begin{aligned} \underset{\pi(\cdot|s)}{\text{maximize}} \quad & \mathbb{E}_{a\sim\pi(\cdot|s)}\left[\hat{A}(s,a)\right] \\ \text{subject to} \quad & D_{\mathrm{KL}}\big(\pi(\cdot \mid s) \,\big\|\, b(\cdot \mid s)\big) \leq \delta \end{aligned}$$

Using forward KL gives:

$$\begin{aligned} \underset{\pi(\cdot|s)}{\text{maximize}} \quad & \mathbb{E}_{a\sim\pi(\cdot|s)}\left[\hat{A}(s,a)\right] \\ \text{subject to} \quad & D_{\mathrm{KL}}\big(b(\cdot \mid s) \,\big\|\, \pi(\cdot \mid s)\big) \leq \delta \end{aligned}$$

Explicit Policy Constraints: Lagrangian

One way to solve the constrained optimization problem is to modify the objective with a **Lagrange multiplier**.

Using reverse KL gives:

$$\begin{aligned}\underset{\pi(\cdot|s)}{\text{maximize}} \quad \mathcal{J}_{\text{RevKL}}(\pi; s) &= \mathbb{E}_{a\sim\pi(\cdot|s)}\Big[\hat{A}(s,a)\Big] - \beta D_{\text{KL}}\big(\pi(\cdot \mid s) \,\big\|\, b(\cdot \mid s)\big) \\ &= \mathbb{E}_{a\sim\pi(\cdot|s)}\left[\hat{A}(s,a) - \beta\frac{\log\pi(a \mid s)}{\log b(a \mid s)}\right]\end{aligned}$$

This becomes maximum entropy RL.

Using forward KL gives:

$$\begin{aligned}\underset{\pi(\cdot|s)}{\text{maximize}} \quad \mathcal{J}_{\text{FwdKL}}(\pi; s) &= \mathbb{E}_{a\sim\pi(\cdot|s)}\Big[\hat{A}(s,a)\Big] - \beta D_{\text{KL}}\big(b(\cdot \mid s) \,\big\|\, \pi(\cdot \mid s)\big) \\ &= \mathbb{E}_{a\sim\pi(\cdot|s)}\Big[\hat{A}(s,a)\Big] - \beta\mathbb{E}_{a\sim b(\cdot|s)}\left[\frac{\log b(a \mid s)}{\log\pi(a \mid s)}\right] \\ &\Leftrightarrow \mathbb{E}_{a\sim\pi(\cdot|s)}\Big[\hat{A}(s,a)\Big] + \beta\mathbb{E}_{a\sim b(\cdot|s)}\Big[\log\pi(a \mid s)\Big]\end{aligned}$$

The second term becomes a behavior cloning maximum likelihood loss.

Implicit Policy Constraints

Consider the reverse KL-constrained policy improvement problem with Lagrangian regularization.

We know from maximum entropy RL that the problem has the following closed-form solution:

$$\pi^\star(a \mid s) \propto b(a \mid s) \exp\left(\frac{1}{\beta}\hat{A}(s,a)\right).$$

This is not directly actionable since we cannot sample from $\pi^\star$, but we can attempt finding a policy that is *close* with π^*. Using forward KL as measure of closeness, this results in a *weighted maximum likelihood* form:

$$\begin{aligned}\widehat{\pi^\star}(\cdot \mid s) &= \operatorname*{argmin}_{\pi(\cdot|s)} D_{\mathrm{KL}}\left(\pi^\star(\cdot \mid s) \,\middle\|\, \pi(\cdot \mid s)\right) \\ &= \operatorname*{argmax}_{\pi(\cdot|s)} \mathbb{E}_{a\sim b(\cdot|s)}\left[\log \pi(a \mid s) \exp\left(\frac{1}{\beta}\hat{A}(s,a)\right)\right].\end{aligned}$$

In practical usage, we should implement this using function approximation.

Implicit Policy Constraints: AWR

With one-step V function actor-critic, we can run the following algorithm:

Algorithm 11 Fully Offline AWR

1: update the value network with loss: ▷ with any optimization algo

$$\mathcal{L}(\psi) = \mathbb{E}_{(s,a,\cdot,\cdot)\sim\mathcal{D}^b}\left[\left(\left[\!\left[A^{\mathsf{MC}}_{\psi}(s,a) + V_\psi(s)\right]\!\right] - V_\psi(s)\right)^2\right]$$

2: update the policy network with loss: ▷ with any optimization algo

$$\mathcal{L}(\theta) = -\mathbb{E}_{(s,a,\cdot,\cdot)\sim\mathcal{D}^b}\left[\log \pi_\theta(a \mid s) \exp\left(\frac{1}{\beta} A^{\mathsf{MC}}_{\psi}(s,a)\right)\right]$$

3: **return** π_θ

Since $V_\psi(s)$ vanishes in the value learning target, stopgrad is actually ignorable the inner-quadratic term collapses to $A^{\mathsf{MC}}_{\psi}(s,a)$. Also, we are assuming $\mathcal{D}^b$ contains full trajectories following b, so that we can compute the MC returns.

Advantage-Weighted Regression (AWR) [1] runs this process iteratively by incorporating online data. This becomes an actor-critic method where the first iteration is a *pretraining* step where V_ψ is initialized towards V^b, and the later iterations train V_ψ towards V^{π_θ}.

[1] X. B. Peng et al., "Advantage-Weighted Regression: Simple and Scalable Off-Policy Reinforcement Learning," *arXiv*, 2019

Implicit Policy Constraints: AWAC

With Q function actor-critic, we can run the following algorithm:

Algorithm 12 Fully Offline AWAC

1: **while** not converged **do**

2: update the value network with loss: ▷ with any optimization algo

$$\mathcal{L}(\phi) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}^b}\left[\left(\left[\!\left[\mathbb{E}_{a'\sim\pi(\cdot|s')}\left[r+\gamma Q_\phi(s',a')\right]\right]\!\right] - Q_\phi(s,a)\right)^2\right]$$

3: update the policy network with loss: ▷ with any optimization algo

$$\mathcal{L}(\theta) = -\mathbb{E}_{(s,a,r,s')\sim\mathcal{D}^b}\left[\log\pi_\theta(a|s)\exp\left(\frac{1}{\beta}\left(\mathbb{E}_{a'\sim\pi(\cdot|s')}\left[r+\gamma Q_\phi(s',a')\right] - Q_\phi(s,a)\right)\right)\right]$$

4: **return** π_θ

Since we can't collect full trajectories following π, we use the *1-step TD residual* as the advantage estimate.

Advantage Weighted Actor-Critic (AWAC) [1] incorporates online data into this algorithm (like AWR). Training data will graudally shift towards π's coverage.

[1] A. Nair et al., "AWAC: Accelerating Online Reinforcement Learning with Offline Datasets," *arXiv*, 2021

Implicit Q-Learning (IQL)

However, for Q-learning, constrained policy updates are not enough. A problem comes from value learning, where the loss for is:

$$\mathcal{L}(\phi) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}^b}\left[\left(\left[\!\left[r + \gamma \max_{a'\in\mathcal{A}} Q_\phi(s', a')\right]\!\right] - Q_\phi(s, a)\right)^2\right].$$

This introduces an additional source of distributional shift: *the value learning target contains evaluation from the current network*. This may be inaccurate if the input (s', a') is far from $\mathcal{D}^b$'s coverage.

Implicit Q-Learning (IQL) [1] solves this problem by augmenting one-step V function actor-critic to *mimic* Q-learning. If we can somehow bias value learning towards *optimal* value learning, a one step process is enough since it becomes VI.

Since we have seen that learning V or Q in VI is both problematic, we start by *learning both functions* using each other as targets.

[1] I. Kostrikov et al., "Offline Reinforcement Learning with Implicit Q-Learning," *ICLR*, 2022

Implicit Q-Learning (IQL) (Cont'd)

If we want to train V_ψ to approximate $V^\star$ and Q_ϕ to approximate $Q^\star$, we can naively imagine losses:

$$\mathcal{L}(\psi) = \mathbb{E}_{(s,a,\cdot,\cdot)\sim\mathcal{D}^b}\left[\Big(Q_\phi(s,a) - V_\psi(s)\Big)^2\right],$$

$$\mathcal{L}(\phi) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}^b}\left[\Big(r + \gamma V_\psi(s') - Q_\phi(s,a)\Big)^2\right].$$

$V^\star$ and $Q^\star$ are truly minimizers of these losses, but this is not enough since any V^π and Q^π induced from the same policy π are also minimizers of these losses. Therefore, we need some more structure to push training towards the optimal value functions.

IQL solves this by replacing the MSE loss in $\mathcal{L}(\psi)$ with an **expectile loss**:

$$\ell_2^\tau(x) = \begin{cases} \tau x^2 & \text{if } x > 0 \\ (1-\tau)x^2 & \text{otherwise} \end{cases}, \qquad \tau \in (0,1).$$

Then, $\mathcal{L}(\psi)$ becomes:

$$\mathcal{L}(\psi) = \mathbb{E}_{(s,a,\cdot,\cdot)\sim\mathcal{D}^b}\left[\ell_2^\tau\Big(Q_\phi(s,a) - V_\psi(s)\Big)\right].$$

Implicit Q-Learning (IQL) (Cont'd)

With large $\tau(\to 1)$, V_ψ is almost not penalized for negative deviations. Therefore, the system shifts toward an equilibrium where V_ψ estimates the maximum of Q_ϕ over the dataset support:

$$V_\psi(s) \to \max_{a:(s,a,\cdot,\cdot)\in\mathcal{D}^b} Q_\phi(s,a) \qquad \text{as} \qquad \tau \to 1.$$

Then, $\mathcal{L}(\phi)$ essentially becomes the Q-learning loss:

$$\mathcal{L}(\phi) \to \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}^b}\left[\left(\left[\!\left[r+\gamma \max_{a':(s',a',\cdot,\cdot)\in\mathcal{D}^b} Q_\phi(s',a')\right]\!\right] - Q_\phi(s,a)\right)^2\right].$$

The policy update step immediately follows AWR, but now that we have both Q and V functions, we can directly construct the advantage as:

$$A_{\phi,\psi}(s,a) \coloneqq Q_\phi(s,a) - V_\psi(s).$$

The loss for policy training becomes:

$$\mathcal{L}(\theta) = -\mathbb{E}_{(s,a,\cdot,\cdot)\sim\mathcal{D}^b}\left[\log \pi_\theta(a \mid s) \exp\left(\frac{1}{\beta} A_{\phi,\psi}(s,a)\right)\right].$$

Conservative Q-Learning (CQL)

Another approach is to regularize only *critic updates*, not policy updates, by implementing the principle of *pessimism*.

The target OOD problem in Q function actor-critic or Q-learning may have errors in all directions, but *overestimating* the value is actually more of a concern. This is because our next step, the policy learning step takes the *maximum* over the predicted values.

Conservative Q-Learning (CQL) [1] solves this problem by adding a conservative regularization term:

$$\mathcal{L}_{\text{CQL}}(\phi) := \mathbb{E}_{\substack{(s,\cdot,\cdot,\cdot)\sim\mathcal{D}^b \\ a\sim\mu(\cdot|s)}}\Big[Q_\phi(s,a)\Big] - \mathbb{E}_{(s,a,\cdot,\cdot)\sim\mathcal{D}^b}\Big[Q_\phi(s,a)\Big],$$

to the original value learning loss. μ here is some policy that effectively covers the entire action space. This way, the first term encourages the network to *push down* values of all actions, including OOD actions. The second term, on the other hand, encourages the network to *push up* values of ID actions.

[1] A. Kumar et al., "Conservative Q-Learning for Offline Reinforcement Learning," *NeurIPS*, 2020