Math Foundation RL

Basic Concepts

Discount Rate γ :

- γ is close to 0 $\rightarrow$ the *discounted return* is dominated by the rewards from the near future
- γ is close to 1 $\rightarrow$ dominated by the rewards from the far future

An *episode* is usually assumed to be *finite trajectory* (i.e. the agent stops at some *terminal states*). Task with episodes are called *episodic tasks*.

A positive reward represents encouragement to take such actions, a negative reward represents punishment to take such actions.

Converting episodic tasks to continuing tasks:

- Option 1: Treat the target state as a special *absorbing state*. Once the agent reaches an absorbing state, it will never leave and the consequent rewards $r = 0$.
- Option 2: Treat the target state as a normal state with a policy. The agent can still leave the target state and gain $r = +1$ when entering the target afterwards.

MDP: Markov Decision Process

Sets:

- State: the set of states S.
- Action: the set of actions $A(s)$ is associated for state $s \in S$.
- Reward: the set of rewards $R(s, a)$.

Probability distribution (or called *system model*):

- State transition probability: at state s, taking action a, the probability to transit to state s' is $p(s'|s, a)$.
- Reward probability: at state s, taking action a, the probability to get reward r is $p(r|s, a)$.

Policy: at state s, the probability to choose action a is $\pi(a|s)$.

Markov property: memoryless property

Bellman Equation

State Value

> *The value of one state relies on the values of other states ——Bootstrapping*

$$
\begin{aligned}
\textbf{trajectory}: \ & S_t \xrightarrow{A_t} R_{t+1}, S_{t+1} \xrightarrow{A_{t+1}} R_{t+2}, S_{t+2} \xrightarrow{A_{t+2}} R_{t+3}, S_{t+3}, \xrightarrow{A_{t+3}} \cdots \\
\textbf{discounted return}: \ & G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = R_{t+1} + \gamma G_{t+1} \\
\textbf{state value}: \ & v_\pi(s) = \mathbb{E}[G_t | S_t = s] \\
& \quad = \mathbb{E}[R_{t+1} + \gamma G_{t+1} | S_t = s] \\
& \quad = \mathbb{E}[R_{t+1} | S_t = s] + \gamma \mathbb{E}[G_{t+1} | S_t = s]
\end{aligned}
$$

First term of RHS (i.e. expectation of *immediate rewards*):

$$
\begin{aligned}
\mathbb{E}[R_{t+1} | S_t = s] &= \sum_a \pi(a|s) \mathbb{E}[R_{t+1} | S_t = a, A_t = a] \\
&= \sum_a \pi(a|s) \sum_r p(r|s,a) r \\
&\overset{\Delta}{=} r_\pi(s)
\end{aligned} \tag{1}
$$

Second term of RHS (i.e. expectation of *future rewards*):

$$
\begin{aligned}
\mathbb{E}[G_{t+1} | S_t = s] &= \sum_{s'} \mathbb{E}[G_{t+1} | S_{t+1} = s'] p(s'|s) \\
&= \sum_{s'} v_\pi(s') p(s'|s) \\
&= \sum_{s'} v_\pi(s') \sum_a p(s'|s,a) \pi(a|s) \overset{\Delta}{=} \sum_{s'} v_\pi(s') p_\pi(s'|s) \\
&= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) v_\pi(s')
\end{aligned} \tag{2}
$$

Bellman Equation

Therefore the **Bellman Equation** can conclude from (1) and (2):

$$
\begin{aligned}
v_\pi(s) &= \sum_a \pi(a|s) \left[\sum_r p(r|s,a) r + \gamma \sum_{s'} v_\pi(s') p(s'|s,a) \right] \\
&= r_\pi(s) + \gamma \sum_{s'} v_\pi(s') p_\pi(s'|s) \\
&= r_\pi(s_i) + \gamma \sum_{s_j} p_\pi(s_j|s_i) v_\pi(s_j);
\end{aligned} \tag{3}
$$

Suppose the states could be indexed as s_i $(i = 1, \ldots, n)$.

Matrix-vector form of **Bellman Equation**:

$$
v_\pi = r_\pi + \gamma P_\pi v_\pi
$$

where

$$
v_\pi = [v_\pi(s_1), \ldots, v_\pi(s_n)]^T \in \mathbb{R}^n
$$

$$
r_\pi = [r_\pi(s_1), \ldots, r_\pi(s_n)]^T \in \mathbb{R}^n
$$

$P_\pi \in \mathbb{R}^{n \times n}$, where $[P_\pi]_{ij} = p_\pi(s_j \mid s_i)$ is the *state transition matrix*

example:

$$\underbrace{\begin{bmatrix} v_\pi(s_1) \\ v_\pi(s_2) \\ v_\pi(s_3) \\ v_\pi(s_4) \end{bmatrix}}_{v_\pi} = \underbrace{\begin{bmatrix} r_\pi(s_1) \\ r_\pi(s_2) \\ r_\pi(s_3) \\ r_\pi(s_4) \end{bmatrix}}_{r_\pi} + \gamma \underbrace{\begin{bmatrix} p_\pi(s_1|s_1) & p_\pi(s_2|s_1) & p_\pi(s_3|s_1) & p_\pi(s_4|s_1) \\ p_\pi(s_1|s_2) & p_\pi(s_2|s_2) & p_\pi(s_3|s_2) & p_\pi(s_4|s_2) \\ p_\pi(s_1|s_3) & p_\pi(s_2|s_3) & p_\pi(s_3|s_3) & p_\pi(s_4|s_3) \\ p_\pi(s_1|s_4) & p_\pi(s_2|s_4) & p_\pi(s_3|s_4) & p_\pi(s_4|s_4) \end{bmatrix}}_{P_\pi} \underbrace{\begin{bmatrix} v_\pi(s_1) \\ v_\pi(s_2) \\ v_\pi(s_3) \\ v_\pi(s_4) \end{bmatrix}}_{v_\pi}$$

Solve the value via iterative solution:

$$v_{k+1} = r_\pi + \gamma P_\pi v_k v_k \to v_\pi, \quad k \to \infty$$

Action Value

$$q_\pi(s,a) = \mathbb{E}\left[G_t | S_t = s, A_t = a\right]$$

Conclude from (3) that:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \sum_a \mathbb{E}\left[G_t|S_t = s, A_t = a\right]\pi(a|s) \\ &= \sum_a \pi(a|s) \underbrace{\left[\sum_r p(r|s,a)r + \gamma \sum_{s'} v_\pi(s')p(s'|s,a)\right]}_{q_\pi(s,a)} \end{aligned}$$

Hence,

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a) \tag{4}$$

$$q_\pi(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} v_\pi(s')p(s'|s,a) \tag{5}$$

(4) and (5) are the two sides of the same coin:

- (4) shows how to obtain *state values* from *action values*.
- (5) shows how to obtain *action values* from *state values* .

BOE: Bellman Optimality Equation

Intro

Evaluating whether a policy is good: if

$$v_{\pi 1}(s) \geqslant v_{\pi 2}(s) \quad \text{for all } s \in \mathcal{S}$$

then π_1 is better than π_2.

Therefore a policy π^* is optimal if $v_{\pi^*}(s) \geqslant v_\pi(s)$ for all s and for any other policy π.

Bellman Optimality Equation (elementwise form):

$$
\begin{aligned}
v(s) &= \max_{\pi} \sum_{a} \pi(a|s) \left[\sum_{r} p(r|s,a) r + \gamma \sum_{s'} v(s') p(s'|s,a) \right], \quad s \in \mathcal{S} \\
&= \max_{\pi} \sum_{a} \pi(a|s) q(s,a), \quad s \in \mathcal{S}
\end{aligned}
$$

This equation shows how the state values (LHS) achieves its maximum (RHS) which corresponds to the optimal policy.

Remarks:

- $p(r|s,a)$, $p(s'|s,a)$, r, γ are known $\rightarrow$ $q(s,a)$ depends on $v(s')$.
- $v(s)$, $v(s')$ are unknown and to be calculated.
- The goal is to solve the equation to find the optimal policy π^*.

Bellman Optimality Equation (matrix-vector form):

$$
v = \max_{\pi}(r_\pi + \gamma P_\pi v)
$$

How to Solve

- Maximize the RHS expression elementwisely：

  $$
  \max_{\pi} \sum_{a} \pi(a|s) q(s,a) \leqslant \max_{a \in \mathcal{A}(s)} q(s,a) = q(s,a^*)
  $$

  where the optimality is achieved when

  $$
  \pi(a|s) = \begin{cases} 1 & a = a^* \\ 0 & a \neq a^* \end{cases} \quad where\ a^* = \arg\max_{a} q(s,a)
  $$

  This is because the RHS expression is actually the weighted sum of all possible $q(s,a)$ given a specific s. Since the weights (i.e. the possibility $\pi(a|s)$ for all a) adds up to 1, we just need to put all the weights on the largest $q(s,a)$ (i.e. assign the probability of 1 to $q(s,a^*)$, and 0 to the rest $q(s,a)$) to achieve the maximum.
- As for the matrix-vector form, rewrite BOE as: $v = \max_{\pi}(r_\pi + \gamma P_\pi v) \overset{\Delta}{=} f(v)$
- According to *Contraction Mapping Theorem*，BOE is a *contraction mapping* with a unique *fixed point* v^*.

  According to *value iteration algorithm*, the solution of BOE could be solved iteratively by

  $$
  v_{k+1} = f(v_k) = \max_{\pi}(r_\pi + \gamma P_\pi v_k)
  $$

  This sequence v_k converges to v^* exponentially fast given any initial guess v_0 . The convergence rate is determined by γ.
- Suppose

  $$
  \pi^* = \arg\max_{\pi} (r_\pi + \gamma P_\pi v^*)
  $$

  Then

$$v^* = r_{\pi^*} + \gamma P_{\pi^*} v^*$$

Therefore, π^* is a policy and $v^* = v_{\pi^*}$ is the corresponding state value.

- It can be proved mathematically that for the *fixed point* v^*,

$$v^* \geqslant v_\pi, \quad \forall \pi$$

therefore π^* is the optimal policy. It is a *deterministic greedy policy* as below:

$$\pi(a|s) = \begin{cases} 1 & a = a^* \\ 0 & a \neq a^* \end{cases}$$

where $a^* = \arg\max_a q^*(s,a)$ and $q^*(s,a)$ corresponds to $v^*(a)$.

> *The optimal policies are invariant to the linear transformation of the reward signals.*

Truncated Policy Iteration Algorithm

Pseudocode:

Initialization: The probability model $p(r|s,a)$ and $p(s'|s,a)$ for all (s,a) are known.

Aim: Search for the optimal state value and an optimal policy.

While v_k has not converged, **for** the kth iteration $(k = 0, 1, 2, \dots)$, **do**

Policy evaluation:

Initialization: select the initial guess as $v_k^{(0)} = v_{k-1}$. The maximum iteration is set to be $j_{truncate}$.

While $j < j_{truncate}$, **do**

For every state $s \in \mathcal{S}$, **do**

$$v_k^{(j+1)}(s) = \sum_a \pi_k(a|s) \left[\sum_r p(r|s,a) r + \gamma \sum_{s'} p(s'|s,a) v_k^{(j)}(s') \right]$$

Set $v_k = v_k^{(j_{truncate})}$ *# Note that when $j \to \infty$, $v_k^{(j)}$ converges to v_{π_k} of current π_k*

 # and it can be proved mathematically that for any iteration j, $v_k^{j-1} < v_k^j < v_k^{j_{truncate}}$.

 # S.t. v_{π_k} is taken as the evaluation of π_k, and we improve π_k based on v_{π_k}.

 # For computational efficiency, only iterate over $j < j_{truncate}$

 # because $v_k^{(j_{truncate})}$ is close enough to v_{π_k}.

Policy improvement:

For every state $s \in \mathcal{S}$, **do**

For every action $a \in \mathcal{A}(s)$, **do**

$q_k(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v_k(s')$

$a_k^*(s) = \arg\max_a q_k(s,a)$

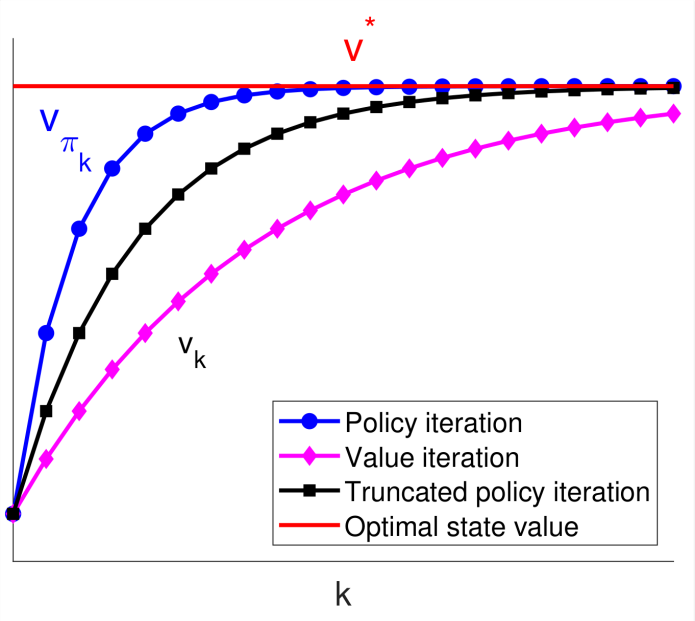
$\pi_{k+1}(a|s) = 1$ if $a = a_k^*$, and $\pi_{k+1}(a|s) = 0$ otherwise

 # Lemma: If $\pi_{k+1} = \arg\max_\pi (r_\pi + \gamma P_\pi v_{\pi_k})$ then $v_{\pi_k} < v_{\pi_{k+1}}$ for any k.

 # Theorem: The state value generated by the iteration converges to the optimal state value v^*,

 # as a result, the policy converges to an optimal policy.

The case of $j_{truncate} = 1$ is *Value Iteration Algorithm*, and the case of $j_{truncate} = \infty$ is *Policy Iteration Algorithm*.



MC: Monte Carlo Learning

> *model-free RL: When model is unavailable, we can use data.*

MC Basic algorithm

> *Many model-based and model-free RL algorithms fall into this framework.*

Pseudocode:

Initialization: Initial guess π_0.

Aim: Search for an optimal policy.

For the kth iteration ($k = 0, 1, 2, \dots$), **do**

For every state $s \in \mathcal{S}$, **do**

Collect sufficiently many episodes starting from (s, a) following π_k

Policy evaluation:

$q_{\pi_k}(s, a) \approx q_k(s, a) =$ average return of all the episodes starting from (s, a)

Policy improvement:

$a_k^*(s) = \arg\max_a q_k(s, a)$

$\pi_{k+1}(a|s) = 1$ if $a = a_k^*$, and $\pi_{k+1}(a|s) = 0$ otherwise

---

Since policy iteration is convergent, the convergence of MC Basic is also guaranteed to be convergent given sufficient episodes.

Episode length:

- When the episode length is short, only the states that are close to the target have nonzero state values.
- As the episode length increases, the states that are closer to the target have nonzero values earlier than those farther away.
- The episode length should be sufficiently long, but does not have to be infinitely long.

Visit: every time a state-action pair appears in the episode, it is called a visit of that state-action pair.

For an episode such as:

$$s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_4} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \dots$$

Methods to approximate $q_{\pi_k}(s, a)$ using the data:

- **Initial-visit method:** Just calculate the return and approximate $q_\pi(s_1, a_2)$.
- **Every-visit method:**

$$\begin{aligned}
s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_4} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \dots \quad & [\text{original episode}] \\
s_2 \xrightarrow{a_4} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \dots \quad & [\text{episode starting from } (s_2, a_4)] \\
s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \dots \quad & [\text{episode starting from } (s_1, a_2)] \\
s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \dots \quad & [\text{episode starting from } (s_2, a_3)] \\
s_5 \xrightarrow{a_1} \dots \quad & [\text{episode starting from } (s_5, a_1)]
\end{aligned}$$

Can estimate $q_\pi(s_1,a_2), q_\pi(s_2,a_4), q_\pi(s_2,a_3), q_\pi(s_5,a_1), \dots$

When to update the policy:

- Low efficiency: In the policy evaluation step, to collect all the episodes starting from a state-action pair and then use the average return to approximate the action value.
- High efficiency: Uses the return of a single episode to approximate the action value. In this way, we can improve the policy episode-by-episode.
- Will the second method cause problems?
  - In fact, we have applied its idea in the truncated policy iteration algorithm (i.e. approximating v_{π_k} via limited iterations).
  - Even if using the single episode we sampled as $q(s,a)$ is imprecise, as long as the relative values of $q(s,a)$ for different a are roughly correct, the policy still selects the action with higher return in $\arg\max_a q_k(s,a)$, thus improving on the right track.
  - When the policy gets better, **it generates episodes that are closer to the optimal trajectory, and consequently less random (i.e. the variance of $q_k(s,a)$ gets smaller and thus $q_k(s,a)$ converges to $q_{\pi_k}(s,a)$ continuously)**.

MC Exploring Starts

Pseudocode:

Initialization: Initial policy $\pi_0(a|s)$ and initial value $q(s,a)$ for all (s,a). $Returns(s,a) = [\,]$ for all (s,a).

Aim: Search for an optimal policy.

For each episode, **do**

Episode generation: Select a starting state-action pair (s_0,a_0) and ensure that all pairs can be possibly selected as starting point (this is the exploring-starts condition). Following the current policy, generate an episode of length T: $s_0, a_0, r_1, \dots, s_{T-1}, a_{T-1}, r_T$.

Initialization for each episode: $g \leftarrow 0$

For each step of the episode, $t = T-1, T-2, \dots, 0$, **do** *# Compute reversely from the end of the episode.*

$g \leftarrow \gamma g + r_{t+1}$ *# S.t. only need one step of calculation for each update of g.*

$Returns(s_t,a_t) \leftarrow Returns(s_t,a_t) \cup \{g\}$

Policy evaluation:

$q(s_t,a_t) \leftarrow$ average$(Returns(s_t,a_t))$

Policy improvement:

$\pi(a|s_t) = 1$ if $a = \arg\max_a q(s_t, a)$, and $\pi(a|s_t) = 0$ otherwise

---

> *What is exploring starts? Exploring starts means we need to generate sufficiently many episodes $\underbrace{starting}_{starts}$ from $\underbrace{every}_{exploring}$ state-action pair.*

In theory, only if every action value for every state is well explored, can we select the optimal actions correctly. Otherwise, if an action is not explored, this action may happen to be the optimal one and hence be missed.

MC ε-Greedy

> *What is a soft policy? A policy is soft if the probability to take any action is positive. With a soft policy, a few episodes that are sufficiently long can visit every state-action pair.*
>
> *Then, we do not need to have a large number of episodes starting from every state-action pair. Hence, the requirement of exploring starts can be removed.*
>
> - *Deterministic policy: e.g. greedy policy*
> - *Stochastic policy: e.g. soft policy*

Pseudocode:

Initialization: Initial policy $\pi_0(a|s)$ and initial value $q(s,a)$ for all (s,a). $Returns(s,a) = [\,]$ for all (s,a). $\varepsilon \in (0,1]$.

Aim: Search for the optimal state value and an optimal policy.

For each episode, **do**

Episode generation: Select a starting state-action pair (s_0, a_0) (the exploring-starts condition is not required). Following the current policy, generate an episode of length T: $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$.

Initialization for each episode: $g \leftarrow 0$

For each step of the episode, $t = T-1, T-2, \ldots, 0$, **do**

$g \leftarrow \gamma g + r_{t+1}$

$Returns(s_t, a_t) \leftarrow Returns(s_t, a_t) \cup \{g\}$

Policy evaluation:

$q(s_t, a_t) \leftarrow \text{average}(Returns(s_t, a_t))$

Policy improvement:

Let $a^* = \arg\max_a q(s_t, a)$ and

$$\pi(a|s_t) = \begin{cases} 1 - \frac{|\mathcal{A}(s_t)|-1}{|\mathcal{A}(s_t)|}\epsilon, & \text{for the greedy action} \\ \frac{1}{|\mathcal{A}(s_t)|}\epsilon, & \text{fot the other } |\mathcal{A}(s)| - 1 \text{ actions} \end{cases}$$

ε controls the balance between **exploration** and **exploitation**.

The advantage of ε-greedy policies is that they have strong exploration ability when ε is large.

The disadvantage is that ε-greedy polices are not optimal in general.

- ε cannot be too large. We can also use a decaying ε.

SA: Stochastic Approximation

SA refers to a broad class of stochastic iterative algorithms solving root finding or optimization problems.

Compared to many other root-finding algorithms such as gradient-based methods, SA is powerful in the sense that it **does not require to know the expression of the objective function nor its derivative** (model-free).

Robbins-Monro algorithm

Problem statement: Suppose we would like to find the root of the equation

$$g(w) = 0$$

The **Robbins-Monro (RM) algorithm** that can solve this problem is as follows:

$$w_{k+1} = w_k - a_k \tilde{g}(w_k, \eta_k), \quad k = 1, 2, 3, \ldots$$

where

- w_k is the kth estimate of the root
- $\tilde{g}(w_k, \eta_k) = g(w_k) + \eta_k$ is the kth noisy observation
- a_k is a positive coefficient.

The function $g(w)$ is viewed as a black box, for which only the input sequence $\{w_k\}$ and output sequence (noisy) $\{\tilde{g}(w_k, \eta_k)\}$ are available. So this algorithm relies on data instead of model.

> ***Robbins-Monro Theorem***
>
> *In the Robbins-Monro algorithm, if*
>
> 1. $0 < c_1 \leq \nabla_w g(w) \leq c_2$ *for all* w;
> 2. $\sum_{k=1}^{\infty} a_k = \infty$ *and* $\sum_{k=1}^{\infty} a_k^2 < \infty$;
> 3. $\mathbb{E}\left[\eta_k | \mathcal{H}_k\right] = 0$ *and* $\mathbb{E}\left[\eta_k^2 \mid \mathcal{H}_k\right] < \infty$;
>
> *where*
>
> $$\mathcal{H}_k = \{w_k, w_{k-1}, \ldots\},$$
>
> *then* w_k *converges w.p.1 to the root* w^* *satisfying* $g(w^*) = 0$.

Explanations of the three conditions:

- Condition1：
  - g should be monotonically increasing, which ensures that the root of $g(w) = 0$ exists and is unique.
  - This condition requires that $g(w)$ is convex.
- Condition 2:
  - $\sum_{k=1}^{\infty} a_k^2 < \infty$ ensures that a_k converges to zero as $k \to \infty$, so w_k converges to w^* as well.

    Also, if $w_k \to w^*, g(w_k) \to 0$ and $\tilde{g}(w_k, \eta_k)$ is dominant by η_k. This randomness should be limited.
  - $\sum_{k=1}^{\infty} a_k = \infty$ ensures that a_k do not converge to zero too fast. Otherwise a_k might converge to zero too early when there is still quite a distance between w_k and w^*, and in this case w_k is not able to get closer to w^*.
- Condition 3: The noise should be unbiased and its variance should be limited.

BGD, SGD, MBGD

Problem setup: Suppose we aim to solve the following optimization problem:

$$\min_w J(w) = \mathbb{E}[f(w, X)]$$

- w is the parameter to be optimized.
- X is a random variable. The expectation is with respect to X.
- w and X can be either scalars or vectors. The function $f(\cdot)$ is a scalar.

The *Batch Gradient Descent*, *Stochastic Gradient Descent*, *Mini-batch Gradient Descent* algorithms solving this problem are, respectively,

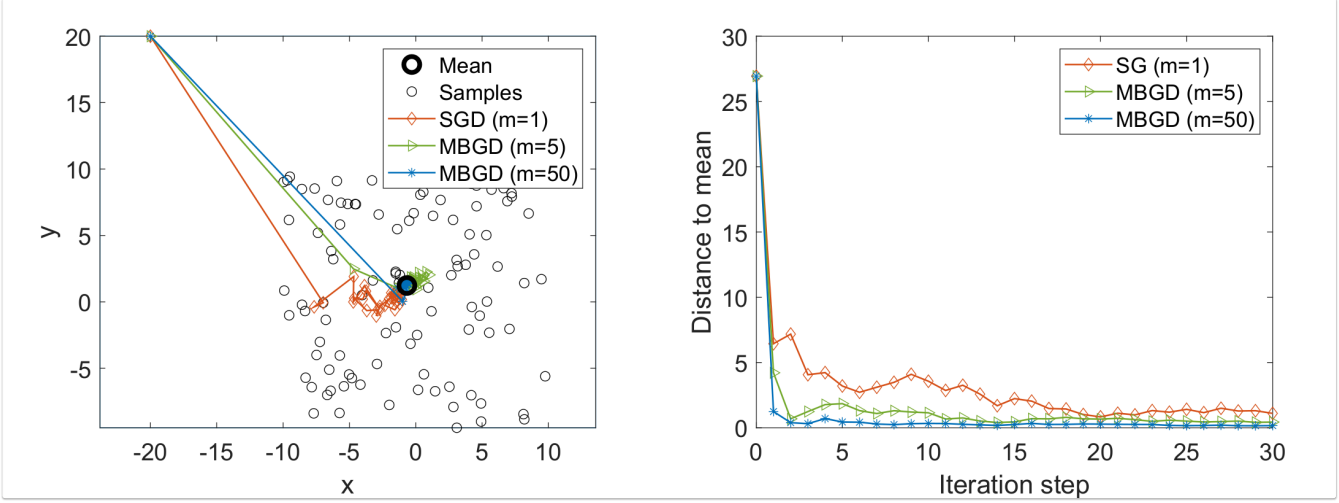
$$w_{k+1} = w_k - \alpha_k \underbrace{\frac{1}{n}\sum_{i=1}^{n} \nabla_w f(w_k, x_i)}_{\approx \mathbb{E}[\nabla_w f(w_k, X)]}, \qquad (BGD)$$

All the samples are used in every iteration, so the approximation to the true gradient $\mathbb{E}\left[\nabla_w f\left(w_k, X\right)\right]$ is close.

$$w_{k+1} = w_k - \alpha_k \underbrace{\frac{1}{m}\sum_{j\in\mathcal{I}_k} \nabla_w f(w_k, x_j)}_{\approx \mathbb{E}[\nabla_w f(w_k, X)]}, \qquad (MBGD)$$

$\mathcal{I}_k$ is a subset of $\{1, \ldots, n\}$ with the size $|\mathcal{I}_k| = m$. The set $\mathcal{I}_k$ is obtained by m times idd samplings.

$$w_{k+1} = w_k - \alpha_k \nabla_w f(w_k, x_k). \qquad (SGD)$$



TD: Temporal Difference Learning

Problem statement:

- Given policy π, the aim is to estimate the state values $\{v_\pi(s)\}_{s\in S}$ **under** π.

The TD learning algorithm is

$$\underbrace{v_{t+1}(s_t)}_{new\ estimate} = \underbrace{v_t(s_t)}_{current\ estimate} - \alpha_t(s_t)\underbrace{[v_t(s_t) - \overbrace{[r_{t+1} + \gamma v_t(s_{t+1})]}^{TD\ target\ \bar{v}_t}}_{TD\ error\ \delta_t} \qquad (1)$$

$$v_{t+1}(s) = v_t(s), \quad \forall s \neq s_t, \qquad (2)$$

where $t = 0, 1, 2, \ldots$

- Here, $v_t(s_t)$ is the estimated state value of $v_\pi(s_t)$; $\alpha_t(s_t)$ is the learning rate of s_t at time t.
- At time t, only the value of the visited state s_t is updated whereas the values of the unvisited states $s \neq s_t$ remain unchanged.
- The update in (2) will be omitted when the context is clear.

Observation: The new estimate $v_{t+1}(s_t)$ is a combination of the current estimate $v_t(s_t)$ and the TD error.

- Interpretation of *TD target* $\bar{v}_t$: $v_t(s_t)$ is the estimate for state value at s_t before taking the action a_t. At s_{t+1} (i.e. after taking a_t), the agent gets the true reward r_{t+1} along with the estimate $v_t(s_{t+1})$ for the new state s_{t+1}. Then $r_{t+1}+\gamma v_t(s_{t+1})$ is considered more precise in estimating the true state value $v_\pi(s_t)$ (**as it contains the real feedback from a step forward**), and thus becomes the *target* for updating $v_t(s_t)$.
- Concluded from above, TD error can be interpreted as **innovation**, which means new information obtained from the experience.
- At every time step, the current estimate $v_t(s_t)$ is updated by subtracting the error δ_t to TD target $\bar{v}_t$, therefore $v_t(s_t)$ is driven towards $\bar{v}_t$.
- If $v_t = v_\pi$, then δ_t should be zero (in the expectation sense), i.e.

  $$\mathbb{E}\left[\delta_{\pi,t}|S_t=s_t\right]=v_\pi(s_t)-\mathbb{E}\left[R_{t+1}+\gamma v_\pi(S_{t+1})|S_t=s_t\right]=0$$

  Hence, if δ_t is not zero, then v_t is not equal to v_π .

Other properties: The TD algorithm in (1) **only estimates the state value** of a given policy. It does not estimate the action values nor search for optimal policies.

Sarsa

Sarsa is the abbreviation of state-action-reward-state-action.

The aim is to estimate the action values of a given policy π.

Suppose we have some experience $\{(s_t,a_t,r_{t+1},s_{t+1},a_{t+1})\}_t$.

We can use the following Sarsa algorithm to estimate the action values:

$$q_{t+1}\left(s_t,a_t\right)=q_t\left(s_t,a_t\right)-\alpha_t\left(s_t,a_t\right)\left[q_t\left(s_t,a_t\right)-\left[r_{t+1}+\gamma q_t\left(s_{t+1},a_{t+1}\right)\right]\right.$$

$$q_{t+1}(s,a)=q_t(s,a),\quad \forall(s,a)\neq(s_t,a_t),$$

where $t=0,1,2,\ldots$

- $q_t\left(s_t,a_t\right)$ is an estimate of $q_\pi\left(s_t,a_t\right)$;
- $\alpha_t\left(s_t,a_t\right)$ is the learning rate depending on s_t,a_t.

> *Note that the relationship between Sarsa and the previous TD learning algorithm is that We can obtain Sarsa by replacing the state value estimate $v(s)$ in the TD algorithm with the action value estimate $q(s,a)$. As a result,* ***Sarsa is an action-value version of the TD algorithm****.*

The expression of Sarsa suggests that it is a stochastic approximation algorithm solving the following equation:

$$q_\pi(s,a)=\mathbb{E}\left[R+\gamma q_\pi\left(S',A'\right)\mid s,a\right],\quad \forall s,a.$$

This is another expression of the Bellman equation expressed in terms of action values.

Since the ultimate goal of RL is to find optimal policies, we combine Sarsa with a policy improvement step. The combined algorithm is also called Sarsa.

Pseudocode:

For each episode, **do**

If the current s_t is not the target state, do

Collect the experience $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$: In particular, take action a_t following $\pi_t(s_t)$, generate r_{t+1}, s_{t+1}, and then take action a_{t+1} following $\pi_t(s_{t+1})$.

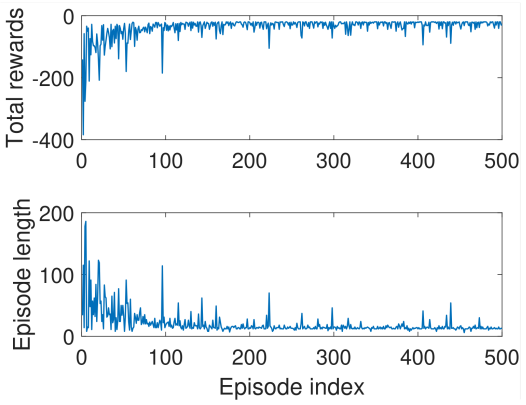
Update q-value:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)\Big[q_t(s_t, a_t) - [r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1})]\Big]$$

Update policy:

$$\pi_{t+1}(a|s_t) = 1 - \tfrac{\epsilon}{|\mathcal{A}|}(|\mathcal{A}| - 1) \text{ if } a = \arg\max_a q_{t+1}(s_t, a)$$

$$\pi_{t+1}(a|s_t) = \tfrac{\epsilon}{|\mathcal{A}|} \text{ otherwise}$$



A variant of Sarsa is the Expected Sarsa algorithm:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)\left[q_t(s_t, a_t) - (r_{t+1} + \gamma \mathbb{E}\left[q_t(s_{t+1}, A)\right])\right],$$

$$q_{t+1}(s, a) = q_t(s, a), \quad \forall (s, a) \neq (s_t, a_t),$$

where

$$\mathbb{E}\left[q_t\left(s_{t+1}, A\right)\right]=\sum_a \pi_t\left(a \mid s_{t+1}\right) q_t\left(s_{t+1}, a\right) \doteq v_t\left(s_{t+1}\right)$$

Compared to Sarsa:

- The *TD target* is changed into expectation form.
- Need more computation. But it is beneficial in the sense that it reduces the estimation variances because it reduces random variables in Sarsa from $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ to $(s_t, a_t, r_{t+1}, s_{t+1})$.

n-step Sarsa

n-step Sarsa: can unify Sarsa and Monte Carlo learning.

The definition of action value is

$$q_\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a].$$

The discounted return G_t can be written in different forms as

$$\begin{aligned}
\text{Sarsa} \quad &\longleftarrow \quad G_t^{(1)} = R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}),\\
&\qquad\quad G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, A_{t+2}),\\
&\qquad\quad \vdots\\
n\text{-step Sarsa} \quad &\longleftarrow \quad G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n q_\pi(S_{t+n}, A_{t+n}),\\
&\qquad\quad \vdots\\
\text{MC} \quad &\longleftarrow \quad G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots
\end{aligned}$$

It should be noted that $G_t = G_t^{(1)} = G_t^{(2)} = G_t^{(n)} = G_t^{(\infty)}$, where the superscripts merely indicate the different decomposition structures of G_t.

- Sarsa aims to solve

$$q_\pi(s,a) = \mathbb{E}[G_t^{(1)}|s,a] = \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|s,a].$$

- MC learning aims to solve

$$q_\pi(s,a) = \mathbb{E}[G_t^{(\infty)}|s,a] = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots|s,a].$$

- An intermediate algorithm called n-step Sarsa aims to solve

$$q_\pi(s,a) = \mathbb{E}[G_t^{(n)}|s,a] = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n q_\pi(S_{t+n}, A_{t+n})|s,a].$$

- The algorithm of n-step Sarsa is

$$q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - [r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n q_t(s_{t+n}, a_{t+n})]\Big].$$

n-step Sarsa is more general because it becomes the (one-step) Sarsa algorithm when $n = 1$ and the MC learning algorithm when $n = \infty$.

Q-learning

Introduce on-policy learning and off-policy learning:

There exist two policies in a TD learning task:

- The behavior policy is used to generate experience samples.
- The target policy is constantly updated toward an optimal policy.

On-policy vs off-policy:

- When the behavior policy is the same as the target policy, such kind of learning is called on-policy.
- When they are different, the learning is called off-policy.

Evaluating action value by Q-learning:

$$q_{t+1}\left(s_t, a_t\right)=q_t\left(s_t, a_t\right)-\alpha_t\left(s_t, a_t\right)\left[q_t\left(s_t, a_t\right)-\left[r_{t+1}+\gamma \max_{a \in \mathcal{A}} q_t\left(s_{t+1}, a\right)\right]\right],$$

$$q_{t+1}(s, a)=q_t(s, a), \quad \forall(s, a) \neq\left(s_t, a_t\right),$$

Pseudocode: Policy searching by Q-learning (on-policy version)

For each episode, **do**

If the current st is not the target state, **do**

Collect the experience$(s_t, a_t, r_{t+1}, s_{t+1})$: In particular, take action a_t following $\pi_t(s_t)$, generate r_{t+1}, s_{t+1}.

Update q-value:

$$q_{t+1}\left(s_t, a_t\right)=q_t\left(s_t, a_t\right)-\alpha_t\left(s_t, a_t\right)\left[q_t\left(s_t, a_t\right)-\left[r_{t+1}+\gamma \max_{a \in \mathcal{A}} q_t\left(s_{t+1}, a\right)\right]\right],$$

Update policy:

$$\pi_{t+1}(a|s_t)=1-\frac{\epsilon}{|\mathcal{A}|}(|\mathcal{A}|-1) \text{ if } a=\arg\max_a q_{t+1}(s_t, a)$$

$$\pi_{t+1}(a|s_t)=\frac{\epsilon}{|\mathcal{A}|} \text{ otherwise}$$

Pseudocode: Policy searching by Q-learning (off-policy version)

For each episode $\{s_0, a_0, r_1, s_1, a_1, r_2, \ldots\}$ generated by π_b, **do**

For each step $t = 0, 1, 2, \ldots$ of the episode, **do**

Collect the experience$(s_t, a_t, r_{t+1}, s_{t+1})$: In particular, take action a_t following $\pi_t(s_t)$, generate r_{t+1}, s_{t+1}.

Update q-value:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)\left[q_t(s_t, a_t) - [r_{t+1} + \gamma \max_{a \in \mathcal{A}} q_t(s_{t+1}, a)]\right],$$

Update policy:

$$\pi_{T,t+1}(a|s_t) = 1 \text{ if } a = \arg\max_a q_{t+1}(s_t, a)$$

$$\pi_{T,t+1}(a|s_t) = 0 \text{ otherwise}$$

---

where

- π_b is the behavior policy for generating a_t in s_t. It can be any policy.
- π_T is the target policy.

Q-learning aims to solve the Bellman optimality equation:

$$q(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_a q(S_{t+1}, a) \mid S_t = s, A_t = a\right], \quad \forall s, a.$$

Value Function Approximation

Algorithm for state value estimation

Policy evaluation problem:

Let $v_\pi(s)$ and $\hat{v}(s, w)$ be the true state value and a function for approximation. Our goal is to find an optimal parameter w so that $\hat{v}(s, w)$ can best approximate $v_\pi(s)$ for every s. To find the optimal w, we need two steps:

- The first step is to define an objective function.
- The second step is to derive algorithms optimizing the objective function.

The objective function is

$$J(w) = \mathbb{E}\left[\left(v_\pi(S) - \hat{v}(S, w)\right)^2\right]$$

Our goal is to find the best w that can minimize $J(w)$.

The expectation is with respect to the random variable $S \in \mathcal{S}$. What is the probability distribution of S?

An intuitive way is to use a **uniform distribution** by treating all the states to be equally important by setting the probability of each state as $1/|\mathcal{S}|$.

- In this case, the objective function becomes

$$J(w) = \mathbb{E}\left[\left(v_\pi(S) - \hat{v}(S, w)\right)^2\right] = \frac{1}{|\mathcal{S}|}\sum_{s\in\mathcal{S}}\left(v_\pi(s) - \hat{v}(s, w)\right)^2$$

- The states may not be equally important. For example, some states may be rarely visited by a policy. Hence, this way does not consider the real dynamics of the Markov process under the given policy.

Hence introducing **stationary distribution**. In short, it describes the long-run behavior of a Markov process.

- Let $\{d_\pi(s)\}_{s\in\mathcal{S}}$ denote the stationary distribution of the Markov process under policy π. By definition, $d_\pi(s) \geq 0$ and $\sum_{s\in\mathcal{S}} d_\pi(s) = 1$.
- The objective function can be rewritten as a weighted squared error.

$$J(w) = \mathbb{E}\left[\left(v_\pi(S) - \hat{v}(S, w)\right)^2\right] = \sum_{s\in\mathcal{S}} d_\pi(s)\left(v_\pi(s) - \hat{v}(s, w)\right)^2$$

- Since more frequently visited states have higher values of $d_\pi(s)$, their weights in the objective function are also higher than those rarely visited states.

> *Stationary distribution is also called steady-state distribution, or limiting distribution.*
>
> *Let $n_\pi(s)$ denote the number of times that s has been visited in a very long episode generated by π. Then, $d_\pi(s)$ can be approximated by*
>
> $$d_\pi(s) \approx \frac{n_\pi(s)}{\sum_{s'\in\mathcal{S}} n_\pi(s')}$$

Optimization algorithms

While we have the objective function, the next step is to optimize it. To minimize the objective function $J(w)$, we can use the gradient-descent algorithm:

$$w_{k+1} = w_k - \alpha_k \nabla_w J(w_k)$$

The true gradient is

$$\begin{aligned}\nabla_w J(w) &= \nabla_w \mathbb{E}\left[\left(v_\pi(S) - \hat{v}(S,w)\right)^2\right] \\ &= \mathbb{E}\left[\nabla_w \left(v_\pi(S) - \hat{v}(S,w)\right)^2\right] \\ &= 2\mathbb{E}\left[\left(v_\pi(S) - \hat{v}(S,w)\right)\left(-\nabla_w \hat{v}(S,w)\right)\right] \\ &= -2\mathbb{E}\left[\left(v_\pi(S) - \hat{v}(S,w)\right)\nabla_w \hat{v}(S,w)\right]\end{aligned}$$

We can use the stochastic gradient to replace the true gradient:

$$w_{t+1} = w_t + \alpha_t \left(v_\pi\left(s_t\right) - \hat{v}\left(s_t, w_t\right)\right)\nabla_w \hat{v}\left(s_t, w_t\right)$$

where s_t is a sample of S. Here, $2\alpha k$ is merged to αk.

- This algorithm is not implementable because it requires the true state value v_π, which is the unknown to be estimated.

  Therefore, we can replace $v_\pi(s_t)$ with an approximation so that the algorithm is implementable.

In particular,

- First, Monte Carlo learning with function approximation:

  Let g_t be the discounted return starting from s_t in the episode. Then, g_t can be used to approximate $v_\pi(s_t)$. The algorithm becomes

  $$w_{t+1} = w_t + \alpha_t \left(g_t - \hat{v}\left(s_t, w_t\right)\right)\nabla_w \hat{v}\left(s_t, w_t\right)$$

- Second, TD learning with function approximation:

  By the spirit of TD learning, $r_{t+1} + \gamma \hat{v}\left(s_{t+1}, w_t\right)$ can be viewed as an approximation of $v_\pi(s_t)$. Then, the algorithm becomes

  $$w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma \hat{v}\left(s_{t+1}, w_t\right) - \hat{v}\left(s_t, w_t\right)\right]\nabla_w \hat{v}\left(s_t, w_t\right)$$

Pseudocode: TD learning with function approximation

Initialization: A function $\hat{v}(s,w)$ that is a differentiable in w. Initial parameter w_0.

Aim: Approximate the true state values of a given policy π.

For each episode generated following the policy π, **do**

  For each step (s_t, r_{t+1}, s_{t+1}), **do**

    In the general case,

$$w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma \hat{v}\left(s_{t+1}, w_t\right) - \hat{v}\left(s_t, w_t\right)\right] \nabla_w \hat{v}\left(s_t, w_t\right)$$

In the linear case,

$$w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma \phi^T\left(s_{t+1}\right) w_t - \phi^T\left(s_t\right) w_t\right] \phi\left(s_t\right)$$

Selection of function approximators:

An important question that has not been answered: How to select the function $\hat{v}(s, w)$?

The first approach, which was widely used before, is to use a linear function.

$$\hat{v}(s, w) = \phi^T(s) w$$

- Here, $\phi(s)$ is the feature vector, which can be a polynomial basis, Fourier basis, ...
- Disadvantage: Difficult to select appropriate feature vectors.
- Advantage: The theoretical properties of the TD algorithm in the linear case can be much better understood than in the nonlinear case.

Linear function approximation is still powerful in the sense that the tabular representation is merely a special case of linear function approximation:

- First, consider the special feature vector for state s:

  $$\phi(s) = e_s \in \mathbb{R}^{|\mathcal{S}|}$$

  where e_s is a vector with the sth entry as 1 and the others as 0.
- In this case,

  $$\hat{v}(s, w) = e_s^T w = w(s)$$

  where $w(s)$ is the sth entry of w.
- Recall that the TD-Linear algorithm is

  $$w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma \phi^T\left(s_{t+1}\right) w_t - \phi^T\left(s_t\right) w_t\right] \phi\left(s_t\right)$$

  When $= \phi(s_t) = e_s$, the above algorithm becomes

  $$w_{t+1} = w_t + \alpha_t \left(r_{t+1} + \gamma w_t\left(s_{t+1}\right) - w_t\left(s_t\right)\right) e_s$$

  This is a vector equation that merely updates the s_tth entry of w_t.
- Multiplying $e_{s_t}^T$ on both sides of the equation gives

  $$w_{t+1}\left(s_t\right) = w_t\left(s_t\right) + \alpha_t \left(r_{t+1} + \gamma w_t\left(s_{t+1}\right) - w_t\left(s_t\right)\right)$$

  which is exactly the tabular TD algorithm.

The second approach, which is widely used nowadays, is to use a neural network as a nonlinear function approximator.

- The input of the NN is the state, the output is $\hat{v}(s,w)$, and the network parameter is w.

Sarsa with function approximation

So far, we merely considered the problem of state value estimation. That is we hope $\hat{v} \approx v_\pi$. To search for optimal policies, we need to estimate action values.

The Sarsa algorithm with value function approximation is

$$w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, w_t) - \hat{q}(s_t, a_t, w_t)\right] \nabla_w \hat{q}(s_t, a_t, w_t)$$

This is the same as the algorithm introduced previously except that $\hat{v}$ is replaced by $\hat{q}$.

To search for optimal policies, we can combine policy evaluation and policy improvement.

Pseudocode: Sarsa with function approximation

Aim: Search a policy that can lead the agent to the target from an initial state-action pair (s_0, a_0).

For each episode, **do**

If the current s_t is not the target state, **do**

Take action a_t following $\pi_t(s_t)$, generate r_{t+1}, s_{t+1}, and then take action a_{t+1} following $\pi_t(s_{t+1})$

Value update (parameter update):

$$w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, w_t) - \hat{q}(s_t, a_t, w_t)\right] \nabla_w \hat{q}(s_t, a_t, w_t)$$

Policy update:

$$\pi_{t+1}(a|s_t) = 1 - \frac{\varepsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1) \text{ if } a = \arg\max_{a \in \mathcal{A}(s_t)} \hat{q}(s_t, a, w_{t+1})$$

$$\pi_{t+1}(a|s_t) = \frac{\varepsilon}{|\mathcal{A}(s)|} \text{ otherwise}$$

Deep Q-learning

Deep Q-learning aims to minimize the objective function/loss function:

$$J(w)=\mathbb{E}\left[\left(R+\gamma\max_{a\in\mathcal{A}(S')}\hat{q}\left(S',a,w\right)-\hat{q}(S,A,w)\right)^2\right]$$

where (S,A,R,S') are random variables.

- This is actually the Bellman optimality error. That is because

$$q(s,a)=\mathbb{E}\left[R_{t+1}+\gamma\max_{a\in\mathcal{A}(S_{t+1})}q\left(S_{t+1},a\right)\mid S_t=s,A_t=a\right],\quad\forall s,a$$

  The value of $R+\gamma\max_{a\in\mathcal{A}(S')}\hat{q}\left(S',a,w\right)-\hat{q}(S,A,w)$ should be zero in the expectation sense.

We use gradient decsent to minimize the objective function, but it is tricky because in $J(w)$, the parameter w not only appears in $\hat{q}(S,A,w)$ but also in

$$R+\gamma\max_{a\in\mathcal{A}(S')}\hat{q}\left(S',a,w\right)\triangleq y$$

For the sake of simplicity, we can assume that w in y is fixed (at least for a while) when we calculate the gradient. To do that, we can introduce two networks.

- One is a main network representing $\hat{q}(s,a,w)$
- The other is a target network $\hat{q}(s,a,w_T)$

The objective function in this case degenerates to

$$J=\mathbb{E}\left[\left(R+\gamma\max_{a\in\mathcal{A}(S')}\hat{q}\left(S',a,w_T\right)-\hat{q}(S,A,w)\right)^2\right]$$

where w_T is the target network parameter.

When w_T is fixed, the gradient of J can be easily obtained as

$$\nabla_w J=\mathbb{E}\left[\left(R+\gamma\max_{a\in\mathcal{A}(S')}\hat{q}\left(S',a,w_T\right)-\hat{q}(S,A,w)\right)\nabla_w\hat{q}(S,A,w)\right]$$

The basic idea of deep Q-learning is to use the gradient-descent algorithm to minimize the objective function. Such an optimization process evolves two important techniques that deserve special attention.

Main network & Target network

Let w and w_T denote the parameters of the main and target networks, respectively. They are set to be the same initially.

In every iteration, we draw a mini-batch of samples $\{(s,a,r,s')\}$ from the replay buffer (will be explained in Experience replay). The inputs of the networks include state s and action a.

The target output is

$$y_T \triangleq r + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}\left(S', a, w_T\right)$$

Then, we directly minimize the TD error or called loss function $(y_T - \hat{q}(s, a, w))^2$ over the mini-batch $\{(s, a, y_T)\}$.

Experience replay

After we have collected some experience samples, we **do NOT use these samples in the order they were collected**. Instead, we store them in a set, called replay buffer $\mathcal{B} \triangleq \{(s, a, r, s')\}$.

Every time we train the neural network, we can draw a mini-batch of random samples from the replay buffer, so that **we can avoid the batch being composed of correlated data in the same episode and instead using i.i.d. data** for each batch during training, which is beneficial for the networks to learn properly.

The draw of samples is called **experience replay**, and it should follow a uniform distribution (in order to sample i.i.d. data mentioned above).

> *To be specific, let's review the objective function:*
>
> $$J = \mathbb{E}\left[\left(R + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}\left(S', a, w_T\right) - \hat{q}(S, A, w)\right)^2\right]$$
>
> - $(S, A) \sim d$: *(S, A) is an index and treated as a single random variable*
> - $R \sim p(R|S, A), \quad S' \sim (S'|S, A)$: *$R$ and S are determined by the system model.*
> - *The distribution of the state-action pair (S, A) is assumed **to be uniform**.*
>
> *However, the samples are not uniformly collected because they are generated consequently by certain policies. **To break the correlation between consequent samples**, we can use the experience replay technique by **uniformly drawing samples from the replay buffer**.*

Pseudocode: Deep Q-learning (off-policy version)

Aim: Learn an optimal target network to approximate the optimal action values from the experience samples generated by a behavior policy π_b.

Store the experience samples generated by π_b in a replay buffer $\mathcal{B} = \{(s, a, r, s')\}$

For each iteration, **do**

Uniformly draw a mini-batch of samples from $\mathcal{B}$

For each sample (s, a, r, s'), calculate the target value as $y_T \triangleq r + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}(S', a, w_T)$, where w_T is the parameter of the target network

Update the main network to minimize $(y_T - \hat{q}(s, a, w))^2$ using the mini-batch $\{(s, a, y_T)\}$

Set $w_T = w$ every C iterations

Policy Gradient Methods

Previously, policies have mostly been represented by tables, where each entry of the table is indexed by a state and an action, and we can directly access or change a value in the table.

Now, policies can be represented by parameterized functions (e.g. a neural network):

$$\pi(a|s, \theta)$$

where $\theta \in \mathbb{R}^m$ is a parameter vector.

Differences between tabular and function representations:

- First, how to define optimal policies?

  When represented as a table, a policy π is optimal if it can maximize every state value.

  When represented by a function, a policy π is optimal if it can maximize certain scalar metrics (or objective functions) $J(\theta)$.
- Second, how to access the probability of an action?

  In the tabular case, the probability of taking a at s can be directly accessed by looking up the tabular policy.

  In the case of function representation, we need to calculate the value of $\pi(a|s, \theta)$ given the function structure and the parameter.
- Third, how to update policies?

  When represented by a table, a policy π can be updated by directly changing the entries in the table.

  When represented by a parameterized function, a policy π cannot be updated in this way anymore. Instead, it can only be updated by optimizing the parameter θ:

  $$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t)$$

Metrics to define optimal policies

There are two metrics. The first metric is the average state value or simply called average value, and the second metric is average one-step reward or simply average reward.

Average value

The metric is defined as:

$$\bar{v}_\pi = \sum_{s\in\mathcal{S}} d(s)v_\pi(s)$$

where

- $\bar{v}_\pi$ is a weighted average of the state values
- $d(s) \geq 0$ is the weight for state s. Since $\sum_{s\in\mathcal{S}} d(s) = 1$, we can interpret $d(s)$ as a probability distribution. Then, the metric can be written as

  $$\bar{v}_\pi = \mathbb{E}[v_\pi(S)]$$

  where $S \sim d$.

How to select the distribution d? There are two cases.

The first case is that d is independent of the policy π.

- In this case, we specifically denote d as d_0 and $\bar{v}_\pi$ as $\bar{v}_\pi^0$.
- How to select d_0? One trivial way is to treat all the states equally important and hence select $d_0(s) = 1/|\mathcal{S}|$.

  Another important case is that we are only interested in a specific state s_0. For example, the episodes in some tasks always start from the same state s_0. Then, we only care about the long-term return starting from s_0. In this case,

  $$d_0(s_0) = 1,\ d_0(s \neq s_0) = 0$$

  The first case is that d depends on the policy π.
- A common way to select d as $d_\pi(s)$, which is the stationary distribution under π.
- The interpretation of selecting d_π is as follows:

  If one state is frequently visited in the long run, it is more important and deserves more weight.

  If a state is hardly visited, then we give it less weight.

Average reward

In particular, the metric is

$$\bar{r}_\pi \triangleq \sum_{s\in\mathcal{S}} d_\pi(s)r_\pi(s) = \mathbb{E}\left[r_\pi(S)\right]$$

where $S \sim d_\pi$. Here,

$$r_\pi(s) \triangleq \sum_{a\in\mathcal{A}} \pi(a|s)r(s,a)$$

is the average of the one-step immediate reward that can be obtained starting from state s, and

$$r(s,a) = \mathbb{E}[R|s,a] = \sum_r rp(r|s,a)$$

- The weight d_π is the stationary distribution.
- As its name suggests, $\bar{r}_\pi$ is simply a weighted average of the one-step immediate rewards.

An important property is that

$$\begin{aligned}\lim_{n\to\infty}\frac{1}{n}\mathbb{E}\left[\sum_{k=1}^{n} R_{t+k}|S_t = s_0\right] &= \lim_{n\to\infty}\frac{1}{n}\mathbb{E}\left[\sum_{k=1}^{n} R_{t+k}\right] \\ &= \sum_s d_\pi(s) r_\pi(s) \\ &= \bar{r}_\pi\end{aligned}$$

Note that the LHS is exactly the average single-step reward $(R_{t+1}, R_{t+2}, \ldots)$ along a trajectory generated by the agent following a given policy.

> *Remarks about the metrics:*
>
> - *All these metrics are functions of π. Since π is parameterized by θ, these metrics are functions of θ. Therefore, we can search for the optimal values of θ to maximize these metrics. This is the basic idea of policy gradient methods.*
> - *Intuitively, $\bar{r}_\pi$ is more short-sighted because **it merely considers the immediate rewards, whereas $\bar{v}_\pi$ considers the total reward overall steps**.*
> - *However, the two metrics are equivalent to each other. In the discounted case where $\gamma < 1$, it holds that*
>
> $$\bar{r}_\pi = (1-\gamma)\bar{v}_\pi$$

Gradients of the metrics

Summary of the results about the gradients:

$$\nabla_\theta J(\theta) = \sum_{s\in\mathcal{S}} \eta(s) \sum_{a\in\mathcal{A}} \nabla_\theta \pi(a|s,\theta) q_\pi(s,a)$$

where

- $J(\theta)$ can be $\bar{v}_\pi$, $\bar{r}_\pi$, or $\bar{v}^0_\pi$
- "$=$" may denote "strict equality", "approximation", or "proportional to"
- η is a distribution or weight of the states

Some specific results:

$$\nabla_\theta \bar{r}_\pi \simeq \sum_s d_\pi(s) \sum_a \nabla_\theta \pi(a|s,\theta) q_\pi(s,a)$$
$$\nabla_\theta \bar{v}_\pi = \frac{1}{1-\gamma} \nabla_\theta \bar{r}_\pi$$
$$\nabla_\theta \bar{v}_\pi^0 = \sum_{s\in\mathcal{S}} \rho_\pi(s) \sum_{a\in\mathcal{A}} \nabla_\theta \pi(a|s,\theta) q_\pi(s,a)$$

A compact and useful form of the gradient:

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_{s\in\mathcal{S}} \eta(s) \sum_{a\in\mathcal{A}} \nabla_\theta \pi(a|s,\theta) q_\pi(s,a) \\ &= \mathbb{E}\left[\nabla_\theta \ln \pi(A|S,\theta) q_\pi(S,A)\right]\end{aligned}$$

where $S \sim \eta$ and $A \sim \pi(A|S,\theta)$

Advantage of this form: we can use samples to approximate the gradient.

> *Remarks:*
>
> - *Because we need to calculate $\ln \pi(a|s,\theta)$, we must ensure that for all s, a, θ*
>
> $$\pi(a|s) > 0$$
>
> *This can be archived by applying softmax to normalize the entries in a vector.*
> - *Such a form based on the softmax function can be realized by a neural network whose input is s and parameter is θ. The network has $|\mathcal{A}|$ outputs, each of which corresponds to $\pi(a|s,\theta)$ for an action a. The activation function of the output layer should be softmax.*
> - *Since $\pi(a|s,\theta) > 0$ for all a, the parameterized policy is stochastic and hence exploratory. But there also exist **deterministic policy gradient (DPG)** methods.*

Gradient-ascent algorithm (REINFORCE)

Now, we are ready to present the first policy gradient algorithm to find optimal policies. The gradient-ascent algorithm maximizing $J(\theta)$ is

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha \nabla_\theta J(\theta) \\ &= \theta_t + \alpha \mathbb{E}\left[\nabla_\theta \ln \pi\left(A|S,\theta_t\right) q_\pi(S,A)\right]\end{aligned}$$

The true gradient can be replaced by a stochastic one:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \ln \pi\left(a_t|s_t,\theta_t\right) q_\pi\left(s_t,a_t\right)$$

Furthermore, since q_π is unknown, it can be approximated:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \ln \pi\left(a_t|s_t,\theta_t\right) q_t\left(s_t,a_t\right)$$

There are different methods to approximate $q_\pi(s_t,a_t)$, e.g. Monte-Carlo based method, REINFORCE and TD.

Remark 1: How to do sampling?

$$\mathbb{E}_{S\sim d, A\sim\pi}\left[\nabla_\theta \ln \pi\left(A|S,\theta_t\right) q_\pi(S,A)\right] \longrightarrow \nabla_\theta \ln \pi\left(a|s,\theta_t\right) q_\pi(s,a)$$

- How to sample S? $S \sim d$, where the distribution d is a long-run behavior under π.
- How to sample A? $A \sim \pi(A|S,\theta)$. Hence, a_t should be sampled following $\pi(\theta_t)$ at s_t.
- Therefore, the policy gradient method is on-policy.

Remark 2: How to interpret this algorithm?

Since

$$\nabla_\theta \ln \pi\left(a_t|s_t,\theta_t\right) = \frac{\nabla_\theta \pi\left(a_t|s_t,\theta_t\right)}{\pi\left(a_t|s_t,\theta_t\right)}$$

the algorithm can be rewritten as

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha \nabla_\theta \ln \pi\left(a_t \mid s_t,\theta_t\right) q_t\left(s_t,a_t\right) \\ &= \theta_t + \alpha \underbrace{\left(\frac{q_t\left(s_t,a_t\right)}{\pi\left(a_t \mid s_t,\theta_t\right)}\right)}_{\beta_t} \nabla_\theta \pi\left(a_t \mid s_t,\theta_t\right)\end{aligned}$$

Therefore, we have the important expression of the algorithm:

$$\theta_{t+1} = \theta_t + \alpha\beta_t \nabla_\theta \pi\left(a_t|s_t,\theta_t\right)$$

It is a gradient-ascent algorithm for maximizing $\pi(a_t|s_t,\theta)$:

Intuition: When $\alpha\beta_t$ is sufficiently small

- If $\beta_t > 0$, the probability of choosing (s_t,a_t) is enhanced (LR>0 for gradient ascent):

  $$\pi\left(a_t|s_t,\theta_{t+1}\right) > \pi\left(a_t|s_t,\theta_t\right)$$

  The greater β_t is, the stronger the enhancement is.
- If $\beta_t > 0$, then $\pi\left(a_t|s_t,\theta_{t+1}\right) < \pi\left(a_t|s_t,\theta_t\right)$

The coefficient β_t can well balance exploration and exploitation:

- First, β_t is proportional to $q_t(s_t,a_t)$.
  If $q_t(s_t,a_t)$ is great, then β_t is great. Therefore, the algorithm intends to enhance actions with greater values.
- Second, β_t is inversely proportional to $\pi(a_t|s_t,\theta_t)$.
  If $\pi(a_t|s_t,\theta_t)$ is small, then β_t is large.
  Therefore, the algorithm intends to explore actions that have low probabilities.

Recall that

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \ln \pi\left(a_t|s_t,\theta_t\right) q_\pi\left(s_t,a_t\right)$$

is replaced by

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \ln \pi (a_t | s_t, \theta_t) q_t (s_t, a_t)$$

where $q_t(s_t, a_t)$ is an approximation of $q_\pi(s_t, a_t)$.

If $q_\pi(s_t, a_t)$ is approximated by Monte Carlo estimation, the algorithm has a specifics name, REINFORCE.

Pseudocode: Policy Gradient by Monte Carlo (REINFORCE)

Initialization: A parameterized function $\pi(a|s, \theta)$, $\gamma \in (0, 1)$, and $\alpha > 0$.

Aim: Search for an optimal policy maximizing $J(\theta)$.

For the kth iteration, **do**

Select s_0 and generate an episode following $\pi(\theta_k)$. Suppose the episode is $\{s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T\}$.

For $t = 0, 1, \ldots, T-1$, **do**

Value update: $q_t (s_t, a_t) = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$

Policy update: $\theta_{t+1} = \theta_t + \alpha \nabla_\theta \ln \pi (a_t \mid s_t, \theta_t) q_t (s_t, a_t)$

$\theta_k = \theta_T$

Actor-Critic Methods

The simplest actor-critic (QAC)

Revisit the stochastic gradient-ascent algorithm maximizing $J(\theta)$:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \ln \pi (a_t | s_t, \theta_t) q_t (s_t, a_t)$$

We can see "actor" and "critic" from this algorithm:

- This algorithm corresponds to actor.
- The algorithm estimating $q_t(s, a)$ corresponds to critic.

How to get $q_t(s_t, a_t)$?

So far, we have studied two ways to estimate action values:

- Monte Carlo learning: If MC is used, the corresponding algorithm is called REINFORCE or Monte Carlo policy gradient.
- Temporal-difference learning: If TD is used, such kind of algorithms are usually called actor-critic.

Pseudocode: The simplest actor-critic algorithm (QAC)

Aim: Search for an optimal policy by maximizing $J(\theta)$.

At time step t in each episode, **do**

Generate a_t following $\pi(a|s_t,\theta_t)$, observe r_{t+1}, s_{t+1}, and then generate a_{t+1} following $\pi(a|s_{t+1},\theta_t)$.

Critic (value update):

$$w_{t+1} = w_t + \alpha_w \left[r_{t+1} + \gamma q\left(s_{t+1}, a_{t+1}, w_t\right) - q\left(s_t, a_t, w_t\right)\right] \nabla_w q\left(s_t, a_t, w_t\right)$$

Actor (policy update):

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \ln \pi\left(a_t|s_t,\theta_t\right) q\left(s_t, a_t, w_{t+1}\right)$$

> *Remarks:*
>
> - *The critic corresponds to "SARSA+value function approximation".*
> - *The actor corresponds to the policy update algorithm.*
> - *The algorithm is **on-policy**, and since the policy is stochastic, no need to use techniques like ε-greedy.*
> - *This particular actor-citric algorithm is sometimes referred to as **Q Actor-Critic (QAC)**.*

Advantage actor-critic (A2C)

The core idea is to introduce a baseline to reduce variance.

Property: the policy gradient is invariant to an additional baseline:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{S\sim\eta, A\sim\pi}\left[\nabla_\theta \ln \pi\left(A \mid S, \theta_t\right) q_\pi(S, A)\right] \\ &= \mathbb{E}_{S\sim\eta, A\sim\pi}\left[\nabla_\theta \ln \pi\left(A \mid S, \theta_t\right)\left(q_\pi(S, A) - b(S)\right)\right]\end{aligned}$$

Here, the additional baseline $b(S)$ is a scalar function of S. The property can be prooved mathematically.

Why is the baseline useful?

The gradient is $\nabla_\theta J(\theta) = \mathbb{E}[X]$ where

$$X(S, A) \triangleq \nabla_\theta \ln \pi\left(A|S,\theta_t\right)\left[q_\pi(S, A) - b(S)\right]$$

We have

- $\mathbb{E}[X]$ is invariant to $b(S)$.

- $\mathrm{var}(X)$ is NOT invariant to $b(S)$, because $\mathrm{tr}[\mathrm{var}(X)] = \mathbb{E}[X^T X] - \bar{x}^T \bar{x}$ and

$$\begin{aligned}\mathbb{E}[X^T X] &= \mathbb{E}\left[(\nabla_\theta \ln \pi)^T (\nabla_\theta \ln \pi)(q_\pi(S,A) - b(S))^2\right] \\ &= \mathbb{E}\left[\|\nabla_\theta \ln \pi\|^2 (q_\pi(S,A) - b(S))^2\right]\end{aligned}$$

  Imagine b is huge (e.g., 1 millon)

Our goal: Select an optimal baseline b to minimize $\mathrm{var}(X)$

Benefit: when we use a random sample to approximate $\mathbb{E}[X]$, the estimation variance would also be small.

In the algorithms of REINFORCE and QAC,there is no baseline. Or, we can say $b = 0$, which is not guaranteed to be a good baseline.

The optimal baseline that can minimize $\mathrm{var}(X)$ is, for any $s \in S$,

$$b^*(s) = \frac{\mathbb{E}_{A\sim\pi}[\|\nabla_\theta \ln \pi(A|s,\theta_t)\|^2 q_\pi(s,A)]}{\mathbb{E}_{A\sim\pi}[\|\nabla_\theta \ln \pi(A|s,\theta_t)\|^2]}$$

Although this baseline is optimal, it is complex. We can remove the weight $\|\nabla_\theta \ln \pi(A|s,\theta_t)\|^2$ and select the suboptimal baseline:

$$b(s) = \mathbb{E}_{A\sim\pi}[q_\pi(s,A)] = v_\pi(s)$$

which is the state value of s!

Therefore, when $b(s) = v_\pi(s)$, the gradient-ascent algorithm is

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha \mathbb{E}\left[\nabla_\theta \ln \pi(A|S,\theta_t)\left[q_\pi(S,A) - b(S)\right]\right] \\ &= \theta_t + \alpha \mathbb{E}\left[\nabla_\theta \ln \pi(A|S,\theta_t)\delta_\pi(S,A)\right]\end{aligned}$$

where

$$\delta_\pi(S,A) \triangleq q_\pi(S,A) - v_\pi(s)$$

is called the advantage function.

The stochastic version of this algorithm is

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha \nabla_\theta \ln \pi(a_t \mid s_t,\theta_t)\left[q_t(s_t,a_t) - v_t(s_t)\right] \\ &= \theta_t + \alpha \nabla_\theta \ln \pi(a_t \mid s_t,\theta_t)\delta_t(s_t,a_t)\end{aligned}$$

Moreover, the algorithm can be re-expressed as

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha \nabla_\theta \ln \pi(a_t|s_t,\theta_t)\delta_t(s_t,a_t) \\ &= \theta_t + \alpha \frac{\nabla_\theta \pi(a_t|s_t,\theta_t)}{\pi(a_t|s_t,\theta_t)}\delta_t(s_t,a_t) \\ &= \theta_t + \alpha \underbrace{\left(\frac{\delta_t(s_t,a_t)}{\pi(a_t|s_t,\theta_t)}\right)}_{\text{step size}} \nabla_\theta \pi(a_t|s_t,\theta_t)\end{aligned}$$

The step size is proportional to the relative value δ_t rather than the absolute value q_t, which is more reasonable. It can still well balance exploration and exploitation.

Furthermore, the advantage function is approximated by the TD error:

$$\delta_t = q_t(s_t, a_t) - v_t(s_t) \to r_{t+1} + \gamma v_t(s_{t+1}) - v_t(s_t)$$

- Benefit: only need one network to approximate $v_\pi(s)$ rather than two networks for $q_\pi(s,a)$ and $v_\pi(s)$.

Pseudocode: Advantage actor-critic (A2C) or TD actor-critic

Aim: Search for an optimal policy maximizing $J(\theta)$.

At time step t in each episode, **do**

Generate a_t following $\pi(a|s_t, \theta_t)$ and then observe r_{t+1}, s_{t+1}.

TD error (advantage function):

$$\delta_t = r_{t+1} + \gamma v_t(s_{t+1}, w_t) - v_t(s_t, w_t)$$

Critic (value update):

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w v(s_t, w_t)$$

Actor (policy update):

$$\theta_{t+1} = \theta_t + \alpha_\theta \delta_t \nabla_\theta \ln \pi(a_t \mid s_t, \theta_t)$$

Off-policy actor-critic

Policy gradient is on-policy, because the gradient is $\nabla_\theta J(\theta) = \mathbb{E}_{S\sim\eta, A\sim\pi}[*]$. However, by importance sampling, we can convert it to off-policy.

Illustrative examples

Consider a random variable $X \in \mathcal{X} = \{+1, -1\}$, the probability distribution of X is p_0:

$$p_0(X = +1) = 0.5, \quad p_0(X = -1) = 0.5$$

then the expectation of X is 0.

Question: how to estimate $\mathbb{E}[X]$ by using some samples $\{x_i\}$?

- Case 1: The samples $\{x_i\}$ are generated according to p_0:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \to \mathbb{E}[X]$$

- Case 2: The samples $\{x_i\}$ are generated according to another distribution p_1:

  $$p_0(X=+1)=0.8, \quad p_0(X=-1)=0.2$$

  If we use the average of the samples, then without suprising

  $$\bar{x}=\frac{1}{n}\sum_{i=1}^{n}x_i \to \mathbb{E}_{X\sim p_1}[X]=0.6\neq \mathbb{E}_{X\sim p_0}[X]$$

  Can we use $\{x_i\}\sim p_1$ to estimate $\mathbb{E}_{X\sim p_0}[X]$? We aim to do that because we may have to estimate $\mathbb{E}_{X\sim\pi}[*]$ where π is the *target policy* (p_0) based on the samples of a *behavior policy* β (p_1).

We can achieve that by using the importance sampling technique.

Importance sampling

Note that

$$\mathbb{E}_{X\sim p_0}[X]=\sum_x p_0(x)x=\sum_x p_1(x)\underbrace{\frac{p_0(x)}{p_1(x)}}_{f(x)}x=\mathbb{E}_{X\sim p_1}[f(X)]$$

Thus, we can estimate $\mathbb{E}_{X\sim p_1}[f(X)]$ in order to estimate $\mathbb{E}_{X\sim p_0}[X]$.

How to estimate $\mathbb{E}_{X\sim p_1}[f(X)]$? Let

$$\bar{f}\triangleq\frac{1}{n}\sum_{i=1}^{n}f(x_i)$$

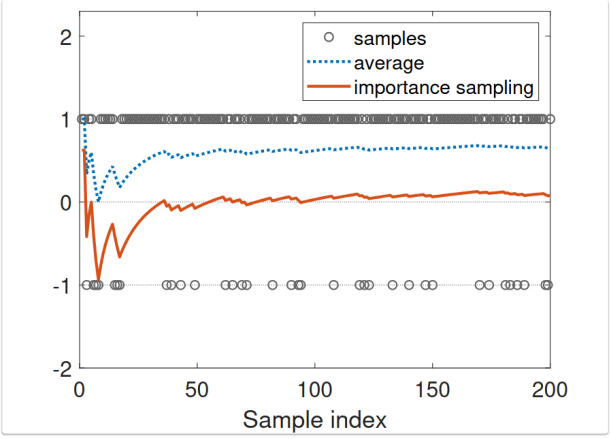
where $x_i\sim p_1$, then,

$$\begin{aligned}\mathbb{E}_{X\sim p_1}[\bar{f}]&=\mathbb{E}_{X\sim p_1}[f(X)]\\ \mathrm{var}_{X\sim p_1}[\bar{f}]&=\frac{1}{n}\mathrm{var}_{X\sim p_1}[f(X)]\end{aligned}$$

Therefore, $\bar{f}$ is a good approximation for $\mathbb{E}_{X\sim p_1}[f(X)]=\mathbb{E}_{X\sim p_0}[X]$:

$$\begin{aligned}\bar{x}&=\tfrac{1}{n}\textstyle\sum_{i=1}^{n}x_i\to\mathbb{E}_{X\sim p_1}[X]\\ \bar{f}&=\tfrac{1}{n}\textstyle\sum_{i=1}^{n}\frac{p_0(x_i)}{p_1(x_i)}x_i\to\mathbb{E}_{X\sim p_0}[X]\end{aligned}$$

- $\frac{p_0(x_i)}{p_1(x_i)}$ is called the *importance weight*.



Off-policy policy gradient

Like the previous on-policy case, we need to derive the policy gradient in the off-policy case.

Suppose β is the behavior policy that generates experience samples. Our aim is to use these samples to update a target policy π that can minimize the metric

$$J(\theta) = \sum_{s \in \mathcal{S}} d_\beta(s) v_\pi(s) = \mathbb{E}_{S \sim d_\beta}[v_\pi(S)]$$

where d_β is the stationary distribution under policy β.

> ***Off-policy policy gradient theorem***
>
> *In the discounted case where $\gamma \in (0,1)$, the gradient of $J(\theta)$ is*
>
> $$\nabla_\theta J(\theta) = \mathbb{E}_{S \sim \rho, A \sim \beta}\left[\frac{\pi(A|S,\theta)}{\beta(A|S)} \nabla_\theta \ln \pi(A|S,\theta) q_\pi(S,A)\right]$$
>
> *where β is the behavior policy and ρ is a state distribution.*

Off-policy actor-critic algorithm

The corresponding stochastic gradient-ascent algorithm is

$$\theta_{t+1} = \theta_t + \alpha_\theta \frac{\pi(a_t|s_t,\theta_t)}{\beta(a_t|s_t)} \nabla_\theta \ln \pi(a_t|s_t,\theta_t)\left(q_t(s_t,a_t) - v_t(s_t)\right)$$

Similar to the on-policy case,

$$q_t(s_t,a_t) - v_t(s_t) \approx r_{t+1} + \gamma v_t(s_{t+1}) - v_t(s_t) \doteq \delta_t(s_t,a_t)$$

Then, the algorithm becomes

$$\theta_{t+1} = \theta_t + \alpha_\theta \frac{\pi(a_t|s_t,\theta_t)}{\beta(a_t|s_t)} \nabla_\theta \ln \pi(a_t|s_t,\theta_t) \delta_t(s_t,a_t)$$

and hence

$$\theta_{t+1}=\theta_t+\alpha_\theta\left(\frac{\delta_t\left(s_t,a_t\right)}{\beta\left(a_t|s_t\right)}\right)\nabla_\theta\pi\left(a_t|s_t,\theta_t\right)$$

Pseudocode: Off-policy actor-critic based on importance sampling

Initialization: A given behavior policy $\beta(a|s)$. A target policy $\pi(a|s,\theta_0)$ where θ_0 is the initial parameter vector. A value function $v(s,w_0)$ where w_0 is the initial parameter vector.

Aim: Search for an optimal policy maximizing $J(\theta)$.

At time step t in each episode, **do**

Generate a_t following $\beta(s_t)$ and then observe r_{t+1}, s_{t+1}.

TD error (advantage function):

$$\delta_t=r_{t+1}+\gamma v_t\left(s_{t+1},w_t\right)-v_t\left(s_t,w_t\right)$$

Critic (value update):

$$w_{t+1}=w_t+\alpha_w\frac{\pi(a_t|s_t,\theta_t)}{\beta(a_t|s_t)}\delta_t\nabla_w v\left(s_t,w_t\right)$$

Actor (policy update):

$$\theta_{t+1}=\theta_t+\alpha_\theta\frac{\pi(a_t|s_t,\theta_t)}{\beta(a_t|s_t)}\delta_t\nabla_\theta\ln\pi\left(a_t|s_t,\theta_t\right)$$

Deterministic actor-critic (DPG)

Deterministic policy gradient

Up to now, the policies used in the policy gradient methods are all stochastic since $\pi(a|s,\theta)>0$ for every (s,a), and the policy gradient theorems introduced before are merely valid for stochastic policies.

If the policy must be deterministic, we must derive a new policy gradient theorem.

Benefit for deterministic policies in the policy gradient methods: it can handle continuous action.

The ways to represent a policy:

- Up to now, a general policy is denoted as $\pi(a|s,\theta)\in[0,1]$, which can be either stochastic or deterministic.
- Now, the deterministic policy is specifically denoted as

$$a=\mu(s,\theta)\triangleq\mu(s)$$

where μ is a mapping from S to A, e.g. a neural network with the input as s, the output as a, and the parameter θ.

Consider the metric of average state value in the discounted case:

$$J(\theta) = \mathbb{E}\left[v_\mu(s)\right] = \sum_{s\in\mathcal{S}} d_0(s) v_\mu(s)$$

where $d_0(s)$ is a probability distribution satisfying $\sum_{s\in\mathcal{S}} d_0(s) = 1$.

- d_0 is selected to be independent of μ. The gradient in this case is easier to calculate.
- There are two special yet important cases of selecting d_0.
  The first special case is that $d_0\left(s_0\right) = 1$ and $d_0\left(s \neq s_0\right) = 0$, where s_0 is a specific starting state of interest.
  The second special case is that d_0 is the stationary distribution of a behavior policy that is different from the μ.

> *In the discounted case where $\gamma \in (0,1)$, the gradient of $J(\theta)$ is*
>
> $$\begin{aligned}\nabla_\theta J(\theta) &= \sum_{s\in\mathcal{S}} \rho_\mu(s) \nabla_\theta \mu(s) \left(\nabla_a q_\mu(s,a)\right)\Big|_{a=\mu(s)} \\ &= \mathbb{E}_{S\sim\rho_\mu}\left[\nabla_\theta \mu(S)\left(\nabla_a q_\mu(S,a)\right)|_{a=\mu(S)}\right]\end{aligned}$$
>
> *Here ρ_μ is a state distribution.*

The gradient does not involve the distribution of the action A. As a result, the deterministic policy gradient method is off-policy.

Deterministic actor-critic algorithm

Based on the policy gradient, the gradient-ascent algorithm for maximizing $J(\theta)$ is:

$$\theta_{t+1} = \theta_t + \alpha_\theta \mathbb{E}_{S\sim\rho_\mu}\left[\nabla_\theta \mu(S)\left(\nabla_a q_\mu(S,a)\right)|_{a=\mu(S)}\right]$$

The corresponding stochastic gradient-ascent algorithm is

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \mu\left(s_t\right)\left(\nabla_a q_\mu\left(s_t,a\right)\right)|_{a=\mu(s_t)}$$

Pseudocode: Deterministic actor-critic algorithm

Initialization: A given behavior policy $\beta(a|s)$. A deterministic target policy $\mu\left(s,\theta_0\right)$ where θ_0 is the initial parameter vector. A value function $v\left(s,w_0\right)$ where w_0 is the initial parameter vector.

Aim: Search for an optimal policy by maximizing $J(\theta)$.

At time step t in each episode, **do**

Generate a_t following β and then observe r_{t+1}, s_{t+1}.

TD error:

$$\delta_t = r_{t+1} + \gamma q\left(s_{t+1}, \mu\left(s_{t+1}, \theta_t\right), w_t\right) - q\left(s_t, a_t, w_t\right)$$

Critic (value update):

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w q\left(s_t, a_t, w_t\right)$$

Actor (policy update):

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \mu\left(s_t, \theta_t\right)\left(\nabla_a q\left(s_t, a, w_{t+1}\right)\right)|_{a=\mu(s_t)}$$

> *Remarks:*
>
> - *This is an off-policy implementation where the behavior policy β may be different from μ.*
> - *β can also be replaced by μ+noise.*
> - *How to select the function to represent $q(s,a,w)$?*
>   ***Linear function:*** *$q(s,a,w) = \phi^T(s,a)w$ where $\phi(s,a)$ is the feature vector. Details can be found in the DPG paper.*
>   ***Neural networks:*** *deep deterministic policy gradient (DDPG) method.*