

---

## REINFORCEMENT LEARNING

---

### 32 REINFORCEMENT LEARNING [1611](#)

#### 32.1 Preliminaries [1612](#)

##### 32.1.1 Notations [1612](#)

##### 32.1.2 Finite state Markov decision process [1612](#)

##### 32.1.3 Policy iteration and value iteration [1614](#)

###### 32.1.3.1 Policy iteration [1614](#)

###### 32.1.3.2 Value iteration [1618](#)

#### 32.2 Reinforcement learning theory [1622](#)

##### 32.2.1 Overview [1622](#)

##### 32.2.2 State-action Value function (Q function) [1624](#)

##### 32.2.3 Monte-Carlo method [1626](#)

###### 32.2.3.1 On-policy value estimation [1626](#)

###### 32.2.3.2 Off-policy value estimation [1628](#)

###### 32.2.3.3 MC-based reinforcement learning control [1628](#)

##### 32.2.4 TD(o) learning [1629](#)

###### 32.2.4.1 TD(o) for value estimation [1629](#)

###### 32.2.4.2 On-policy reinforcement learning control [1630](#)

###### 32.2.4.3 Off-policy reinforcement learning control [1631](#)

##### 32.2.5 TD(n) learning [1633](#)

###### 32.2.5.1 Motivation and concepts [1633](#)

###### 32.2.5.2 TD(n) for value estimation [1634](#)

###### 32.2.5.3 TD(n) for reinforcement learning control [1636](#)

---

32.2.6	Standing challenges in reinforcement learning	1636
32.2.6.1	Curse of dimensionality	1636
32.2.6.2	Sample efficiency	1637
32.2.6.3	Exploration-exploitation dilemma	1637
32.2.6.4	Deadly triad	1637
32.3	Policy gradient learning	1638
32.3.1	Stochastic policy gradient fundamentals	1638
32.3.1.1	Preliminaries: derivative and expectation	1638
32.3.1.2	Theoretical framework based on finite-horizon trajectories	1639
32.3.1.3	Theoretical framework based on distributions*	1642
32.3.1.4	Estimate policy gradient and basic algorithms	1646
32.3.1.5	Bootstrap and Actor-Critic methods	1648
32.3.1.6	Common stochastic policies and their representations	1650
32.3.2	Advanced methods for policy gradient estimation	1652
32.3.2.1	Stochastic policy gradient with baseline	1652
32.3.2.2	Generalized advantage estimation	1655
32.3.2.3	Summary of stochastic gradient descent forms	1656
32.3.3	Deterministic policy gradient	1657
32.4	Algorithms zoo	1659
32.4.1	Neural Fitted Q Iteration (NFQ)	1659
32.4.2	Canonical deep Q learning	1660
32.4.3	DQN variants	1662
32.4.3.1	Overview	1662
32.4.3.2	Double Q learning	1663
32.4.3.3	Dueling network	1663
32.4.3.4	Deep Recurrent Q network (DRQN)	1664
32.4.3.5	Asynchronous Methods	1664
32.4.4	Universal value function approximator	1666

---

32.4.5	Deep deterministic policy gradient (DDPG) algorithm	1669
32.4.6	Twin-delayed deep deterministic policy gradient (TD3)	1671
32.4.7	Trust Region Policy Optimization (TRPO)	1674
32.4.7.1	TRPO	1674
32.4.7.2	Evaluating Hessian of KL-divergence	1676
32.4.8	Proximal Policy Optimization (PPO)	1677
32.4.9	Soft Actor-Critic(SAC)	1679
32.4.9.1	Entropy regulated reinforcement learning	1679
32.4.9.2	The SAC algorithm	1680
32.4.10	Evolution strategies	1681
32.5	Advanced training strategies	1684
32.5.1	Priority experience replay	1684
32.5.2	Hindsight experience generation	1685
32.5.3	Reverse goal generation	1686
32.5.4	Reverse goal generation on low-dimensional manifolds	1687
32.5.4.1	Key idea	1687
32.5.4.2	Example: navigation on a curved surface	1688
32.6	Notes on bibliography	1689

## 32.1 Preliminaries

### 32.1.1 Notations

$S_t, s_t$	state at time $t$ ; $S_t$ denotes random variable; $s_t$ denotes one realization.
$A_t, a_t$	action at time $t$ ; $A_t$ denotes random variable; $a_t$ denotes one realization.
$R_t$	reward at time $t$ .
$\gamma$	discount rate ( $0 \leq \gamma \leq 1$ ).
$G_t$	discounted return at time $t$ ( $\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ ).
$\mathcal{S}$	set of all states, also known as state space.
$\mathcal{S}^T$	set of all terminal states.
$\mathcal{A}$	set of all actions, also known as action space .
$\mathcal{A}(s)$	set of all actions available in state $s$ .
$p(s' s, a)$	transition probability to reach next state $s'$ , given current state $s$ and current action $a$ .
$\pi, \mu$	policy. if deterministic: $\pi(s) \in \mathcal{A}(s)$ for all $s \in \mathcal{S}$ . if stochastic: $\pi(a s) = \mathbb{P}(A_t = a S_t = s)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ .
$V^\pi$	state-value function for policy $\pi$ ( $v_\pi(s) \doteq E[G_t S_t = s]$ for all $s \in \mathcal{S}$ ).
$Q^\pi$	action-value function for policy $\pi$ ( $q_\pi(s, a) \doteq E[G_t S_t = s, A_t = a]$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ ).
$V^*$	optimal state-value function ( $v_*(s) \doteq \max_\pi V^\pi(s)$ for all $s \in \mathcal{S}$ ).
$Q^*$	optimal action-value function ( $q_*(s, a) \doteq \max_\pi Q^\pi(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ ).

### 32.1.2 Finite state Markov decision process

**Definition 32.1.1 (finite state Markov decision process).** A *finite state Markov decision process (MDP)* is characterized by a tuple  $(\mathcal{S}, \mathcal{A}, \mathcal{P})$  where  $\mathcal{S}$  is the state space,  $\mathcal{A}$  is the action space, and  $\mathcal{P} = \{p(s'|s, a), s, s' \in \mathcal{S}, a \in \mathcal{A}\}$  is the state transition probability. We require  $\mathcal{S}, \mathcal{A}$  to have finite number of elements. The goal is compute an optimal control policy  $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$  such that the expected total reward in the process

$$J = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_{t+1}, a_t)\right]$$

is maximized, where  $R(s, a) : \mathcal{S} \times \mathcal{A} \rightarrow \mathbf{R}$  is the one-step reward function and  $\gamma \in [0, 1)$  is the discount factor.

*Example 32.1.1 (examples of reward functions).*

- In a navigation task, we can set  $r(s_t, a_t) = \mathbf{1}(s_t \in S_{\text{target}})$ .
- In a game, the state of gaining scores has a reward 1 and other states have a reward 0.

**Definition 32.1.2 (value functions).**

- Let  $\pi$  be a given control policy. We can define a **value function** associated with this policy by

$$V^\pi(s) = E^\pi \left[ \sum_{n=0}^N \gamma^n R(s_{t+n+1}, a_t) \mid s_0 = s, \pi \right],$$

which is the expected total rewards by following a given policy  $\pi$  starting from initial state  $s$ .

- Given a value function associated with a policy  $\pi$ , we can obtain  $\pi$  via

$$\pi(s) = \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} p(s'|s, a) (r + \gamma V(s'))$$

- The **optimal value function**  $V^*$  and the **optimal policy**  $\pi^*$  are connected via

$$V^*(s) = \max_{\pi} V^\pi(s), \pi^*(s) = \arg \max_{\pi} V^\pi(s).$$

**Lemma 32.1.1 (recursive relationship of value functions).** Given a value function  $V^\pi$  associated with a control policy  $\pi$ . The value function satisfies the following backward relationship:

$$V^\pi(s) = E_{s' \sim P(s'|s, a=\pi(s))} [R(s', a) + \gamma V^\pi(s')].$$

Particularly, we have the Bellman's equation characterizing the optimal value function by

$$V^*(s) = \max_a E_{s' \sim P(s'|s, a)} [R(s', a) + \gamma V^*(s')].$$

*Proof.* The definition of  $V^\pi$  says

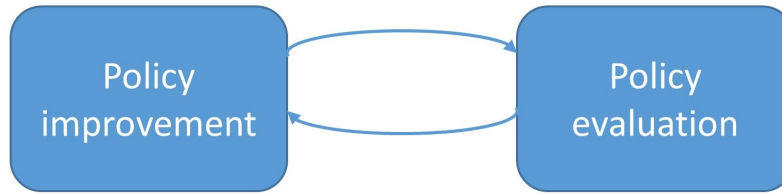
$$\begin{aligned} V^\pi(s) &= E^\pi \left[ \sum_{n=0}^N \gamma^n R(s_{t+n+1}, a_t) \mid s_0 = s, \pi \right] \\ &= E_{s_1 \sim P(s_1|s_0, a=\pi(s))} [R(s_1, a_0) + E^\pi \left[ \sum_{t=1}^N \gamma^t R(s_{t+1}, a_t) \mid s_1 = s', \pi \right] \mid s_0 = s, \pi] \\ &= E_{s_1 \sim P(s_1|s_0, a=\pi(s))} [R(s_1, a_0) + V(s_1) \mid s_0 = s, \pi] \end{aligned}$$

where we have used the tower property of conditional expectation. □

## 32.1.3 Policy iteration and value iteration

## 32.1.3.1 Policy iteration

The core idea underlying policy iteration is to iteratively carry out two procedures: **policy evaluation** and **policy improvement** [Figure 32.1.1]. Given a starting policy  $\pi$ , we perform policy evaluation to estimate the value function  $V^\pi$  associated with this policy; then we improve the policy via dynamic programming principles, or the Bellman Principle.



**Figure 32.1.1:** Policy iteration involves iteratively carrying out policy evaluation and policy improvement procedures.

**Methodology 32.1.1 (iterative policy evaluation procedure).** Given a policy  $\pi$ , we can evaluate the policy via calculating  $V^\pi$  using the following iterative procedure:

$$V^{k+1}(s) = \sum_{s' \in \mathcal{S}} p(s'|s, a = \pi(s))(r + \gamma V^k(s')), \forall s \in \mathcal{S}$$

where superscript  $k$  is the iteration index.

$V^k(s)$  will converge to value function  $V^\pi$  [Theorem 32.1.1].

**Theorem 32.1.1 (convergence property of iterative policy evaluation).** For a finite state MDP, we can write the value function recursive relationship explicitly as

$$V^\pi(s) = \sum_{s' \in \mathcal{S}} P(s'|s, a = \pi(s))[R(s', a) + \gamma V^\pi(s')].$$

We can express the recursive relationship as a matrix form given by

$$V = T(R + \gamma V),$$

where  $R, V \in \mathbb{R}^{|S|}, T \in \mathbb{R}^{|S| \times |S|}$ .

We further define  $H(V) \triangleq T(R + \gamma V)$  as the policy evaluation operator.

We have

- $H$  is a contraction mapping.
- In iterative policy evaluation,  $V^k(s)$  will converge to value function  $V^\pi$ . Or equivalently,  $V^\pi$  is the fixed point of  $H$ , and

$$\lim_{n \rightarrow \infty} H^n(V) = V^\pi.$$

- (error bound) If  $\|H^k(V) - H^{k-1}(V)\|_\infty \leq \epsilon$ , then

$$\|H^k(V) - V^\pi\|_\infty \leq \frac{\epsilon}{1 - \gamma}$$

*Proof.* (1)

$$\begin{aligned} & \|H(\tilde{V}) - H(V)\|_\infty \\ &= \|TR + \gamma T\tilde{V} - TR - \gamma TV\|_\infty \text{ (by definition)} \\ &= \|\gamma T(\tilde{V} - V)\|_\infty \text{ (simplification)} \\ &\leq \gamma \|T\|_\infty \|\tilde{V} - V\|_\infty \text{ (since } \|AB\| \leq \|A\| \|B\|) \\ &= \gamma \|\tilde{V} - V\|_\infty \text{ (since } \max_s \sum_{s'} T(s, s') = 1) \end{aligned}$$

(2) Note that from Fixed point Theorem [Theorem 5.2.3], we have

$$\|H^n(V) - V^\pi\|_\infty \leq \gamma^n \|V - V^\pi\|_\infty.$$

Therefore,

$$\lim_{n \rightarrow \infty} H^n(V) = V^\pi.$$

(3)

$$\begin{aligned}
& \|H^k(V) - V^\pi\|_\infty \\
&= \|H^k(V) - H^\infty(V)\|_\infty \\
&= \left\| \sum_{t=1}^{\infty} H^{t+k}(V) - H^{t+k+1}(V) \right\|_\infty \\
&\leq \sum_{t=1}^{\infty} \|H^{t+k}(V) - H^{t+k+1}(V)\|_\infty \\
&\leq \sum_{t=1}^{\infty} \gamma^t \|H^k(V) - H^{k+1}(V)\|_\infty \\
&\leq \sum_{t=1}^{\infty} \gamma^t \epsilon
\end{aligned}$$

□

**Remark 32.1.1 (error estimation and stopping criterion).** The third property can be used as a stopping criterion during iterations. Suppose the tolerance is  $Tol$ , then we should iterate until the maximum change during consecutive iteration is small than  $(1 - \gamma) \times Tol$ .

Given a learned value function  $V^\pi$  of a policy  $\pi$ , we can derive its  $Q$  function counterpart via

$$Q(s, a) = \sum_{s' \in \mathcal{S}} p(s'|s, a)(r + \gamma V(s')), \forall s, a.$$

$Q$  function offer a convenient way to improve current policy  $\pi$ . Indeed, by relying on following **policy improvement theorem**, we can consistently improve our policy towards the optimal one.

**Theorem 32.1.2 (policy improvement theorem).** [1, p. 78] Define the  $Q$  function associated with a policy  $\pi$  as

$$Q^\pi(s, a) = \sum_{s' \in \mathcal{S}} p(s'|s, a)(r + \gamma V(s')), \forall s, a.$$

Let  $\pi$  and  $\pi'$  be two policies. If  $\forall s \in \mathcal{S}$ ,

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s),$$

then

$$V^{\pi'}(s) \geq V^\pi(s), \forall s \in \mathcal{S}.$$



That is  $\pi'$  is a better policy than  $\pi$ .<sup>a</sup>

<sup>a</sup> Note that  $Q^\pi(s, \pi'(s))$  is the expected process reward for a process where the first step takes action  $\pi'(s)$  and thereafter uses  $\pi(s)$ .

*Proof.*

$$\begin{aligned}
v_\pi(s) &\leq Q^\pi(s, \pi'(s)) \\
&= E[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \pi'(s)] \\
&= E^{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\
&\leq E^{\pi'}[R_{t+1} + \gamma Q^\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \\
&= E^{\pi'}[R_{t+1} + \gamma E^\pi[R_{t+2} + \gamma V^\pi(s_{t+2}) | S_{t+1}, A_{t+1} = \pi'(S_{t+1})] | S_t = s] \\
&= E^{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 V^\pi(S_{t+2}) | S_t = s] \\
&\leq E^{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 r_{t+3} + \gamma^3 v_\pi(S_{t+3}) | S_t = s] \\
&\vdots \\
&\leq E^{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots | S_t = s] \\
&= V^{\pi'}(s)
\end{aligned}$$

□

Based on this algorithm, we can create a better policy via following greedy manner

$$\pi'(s) = \arg \max_{a \in \mathcal{A}(s)} Q(s, a), \forall s.$$

**Methodology 32.1.2 (policy improvement procedure).** Given a value function  $V$ , we can improve the policy implicitly associated with the value function via two steps:

- First calculate the intermediate  $Q$  function

$$Q(s, a) = \sum_{s' \in \mathcal{S}} p(s' | s, a) (r + \gamma V(s')), \forall s, a.$$

- Second improve the policy by

$$\pi'(s) = \arg \max_{a \in \mathcal{A}(s)} Q(s, a), \forall s.$$

Notably, let  $\pi'$  be the improved greedy policy, if  $V^{\pi'} = V^\pi$ , then  $\pi$  is the optimal policy, based on the definition and recursive relation of  $V$  [Lemma 32.1.1].

The following algorithm [algorithm 69] summarizes the policy iteration method[1, p. 80].

---

**Algorithm 69:** The policy iteration algorithm for MDP

---

**Input:** MDP model, small positive number  $\theta$

**Output:** policy  $\pi \approx \pi_*$

```

1 Initialize  $\pi$  arbitrarily (e.g.,  $\pi(a|s) = \frac{1}{|\mathcal{A}(s)|}$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$ )
2 policy-stable  $\leftarrow$  false
3 repeat
4    $V \leftarrow \text{Policy\_Evaluation}(\text{MDP}, \pi, \theta)$ 
5    $\pi' \leftarrow \text{Policy\_Improvement}(\text{MDP}, V)$ 
6   if  $\pi = \pi'$  then
7     policy-stable  $\leftarrow$  true
8   end
9    $\pi \leftarrow \pi'$ 
10 until policy-stable = true;
11 return  $\pi$ 

```

---

### 32.1.3.2 Value iteration

The policy iteration method iterates the two steps of evaluating policy and improving policy. On the other hand, we can directly iteratively estimate the optimal value function, the value function associated the optimal policy, without evaluating the policy associated with value functions. In fact, since policies can be directly calculated from a given value function, having the optimal value function will just give us the optimal policy.

Let  $V^{(0)}$  be an initial value function. Based on the value iteration theorem [Theorem 32.1.3], we can design iteration like

$$V^{(k+1)}(s) = \max_a \sum_{s' \in \mathcal{S}} P(s'|s, a) [R(s', a) + \gamma V^{(k)}(s')], \forall s \in \mathcal{S}.$$

where  $k$  is the iteration number. The following theorem shows that this iteration procedure will lead to convergence to the optimal value function.

More formally, we have



**Theorem 32.1.3 (convergence property of value iteration).** *For a finite state MDP, we can write the optimal value function recursive relationship as*

$$V^*(s) = \max_a \sum_{s' \in \mathcal{S}} P(s'|s, a) [R(s', a) + \gamma V^*(s')], \forall s \in \mathcal{S}.$$

*We can express the recursive relationship as a matrix form given by*

$$V = T(R + \gamma V),$$

*where  $R, V \in \mathbb{R}^{|\mathcal{S}|}, T \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}|}$ .*

*We further define  $H(V) \triangleq \max_a T(R + \gamma V)$  as the value iteration operator.*

*We have*

- *$H$  is a **contraction mapping**.*
- *In iterative policy evaluation,  $V^k(s)$  will converge to the unique optimal value function  $V^*$ . Or equivalently,  $V^*$  is the **fixed point** of the contraction mapping  $H$ , and*

$$\lim_{n \rightarrow \infty} H^n(V) = V^*.$$

- *(error bound) If  $\|H^k(V) - H^{k-1}(V)\|_\infty \leq \epsilon$ , then*

$$\|H^k(V) - V^*\|_\infty \leq \frac{\epsilon}{1 - \gamma}.$$

*Proof.* (1) Without loss of generality, for each  $s$ , we assume  $H(V')(s) \geq H(V)(s)$  and let

$$a_s^* = \arg \max_a \sum_{s' \in \mathcal{S}} P(s'|s, a) [R(s', a) + \gamma V^\pi(s')].$$

Then

$$\begin{aligned} 0 &\leq H(V')(s) - H(V)(s) \\ &\leq \sum_{s' \in \mathcal{S}} P(s'|s, a) (R(s', a) + \gamma V'(s') - R(s', a) - \gamma V(s')) \\ &\leq \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) (V'(s') - V(s')) \\ &\leq \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) \|V'(s') - V(s')\|_\infty \\ &= \gamma \|V'(s') - V(s')\|_\infty. \end{aligned}$$

(2) Because  $H$  is a contraction mapping and  $V^* = HV^*$ ,  $V^*$  is the fixed point of  $H$ . Use Banach Fixed Point Theorem [Theorem 5.2.3], we have

$$\lim_{n \rightarrow \infty} H^n(V) = V^*.$$

(3)

$$\begin{aligned} & \|H^k(V) - V^*\|_\infty \\ &= \|H^k(V) - H^k(V^*)\|_\infty \\ &= \left\| \sum_{t=1}^k H^{t+k}(V) - H^{t+k+1}(V) \right\|_\infty \\ &\leq \sum_{t=1}^k \|H^{t+k}(V) - H^{t+k+1}(V)\|_\infty \quad \text{via triangle inequality} \\ &\leq \sum_{t=1}^k \gamma^t \|H^k(V) - H^{k+1}(V)\|_\infty \quad \text{via contraction mapping} \\ &\leq \sum_{t=1}^k \gamma^t \epsilon \\ &\leq \frac{\epsilon}{1 - \gamma}. \end{aligned}$$

We can further refer to similar proofs regarding that the dynamic programming operator is a contraction mapping. See Theorem 31.6.2 and section 5.2.  $\square$

A direct application of the value iteration theorem gives the following value iteration algorithm [algorithm 70].

---

**Algorithm 70:** Value iteration algorithm for a finite state MDP

---

**Input:** MDP, small positive number  $\epsilon$  as tolerance

**Output:** Value function  $V \approx V^*$  and policy  $\pi \approx \pi^*$ .

```

1 Initialize  $V$  arbitrarily. Set  $(V(s) = 0$  for all  $s \in \mathcal{S}^T$ .)
2 repeat
3    $\Delta = 0$ 
4   for  $s \in \mathcal{S}$  do
5      $v = V(s)$ 
6      $V(s) = \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} P(s'|s, a)(R(s', a) + \gamma V(s'))$ 
7      $\Delta = \max(\Delta, |v - V(s)|)$ 
8   end
9 until  $\Delta < \epsilon$ ;
10 Compute the policy  $\pi(s) = \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} P(s'|s, a)(R(s', a) + \gamma V(s'))$ .
11 return  $V$  and  $\pi$ 
```

---

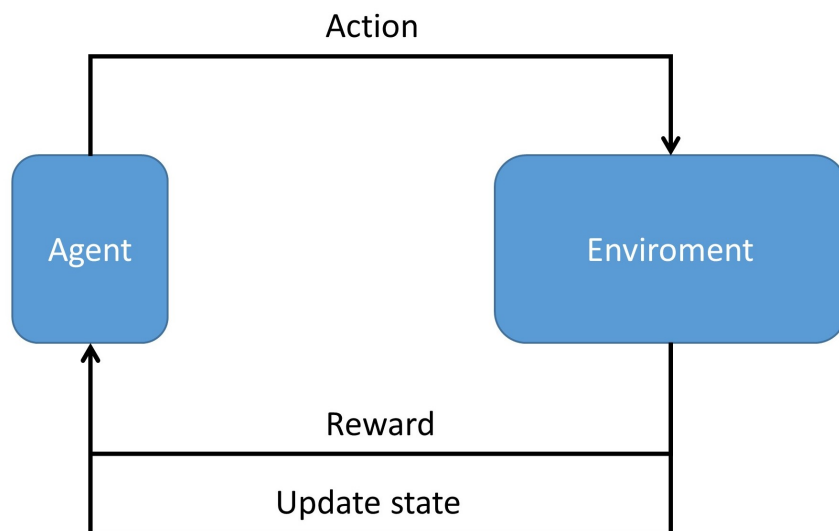
**Remark 32.1.2** (value iteration vs. policy iteration; model-based vs. model-free.).

- At the first glance, it may seem the simplicity of the value iteration method will make the policy iteration method obsolete. It is critical to know that value iteration requires the knowledge of model, which is represented by the transition probabilities  $p(s'|s, a)$ .
- Later we will see that for many complicated real-world decision-making problems, a model is a luxury and often unavailable. In such situations, we usually turn to reinforcement learning, a model-free, data-driven approach to learn control policies. Most reinforcement learning algorithms, from a high-level abstraction, are comprised of the two steps of policy evaluation and policy improvement.

## 32.2 Reinforcement learning theory

### 32.2.1 Overview

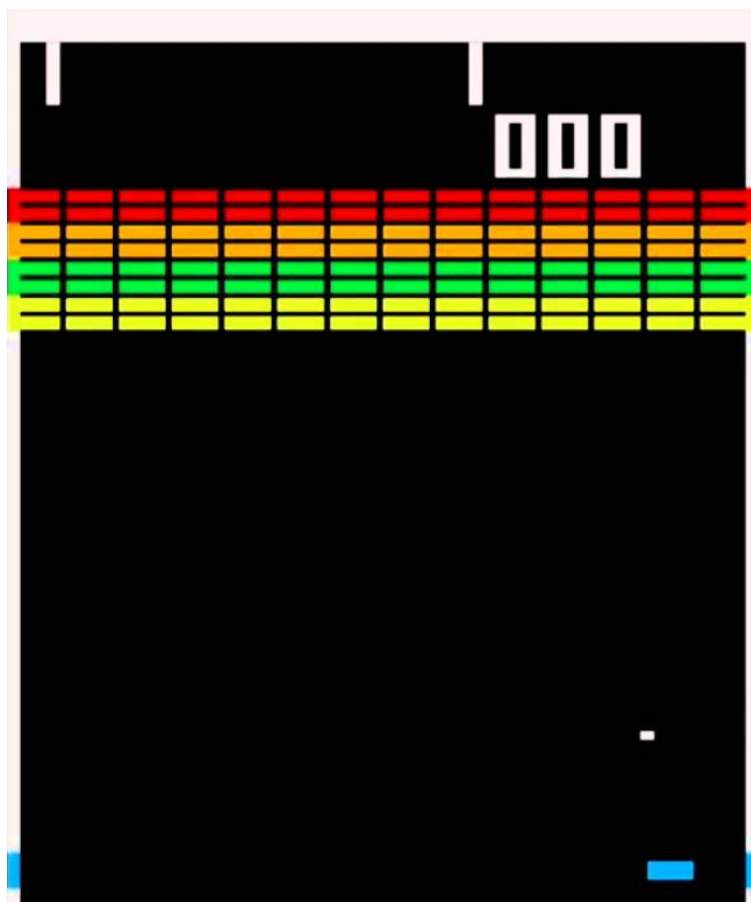
In essence, we can view **reinforcement learning (RL)** as a data-driven way to solve the policy optimization problem in MDP. In RL, we use a setting that contains an agent and an environment [Figure 32.2.1], where data are collected through agent-environment interactions and then further utilized to optimize policies. The agent iteratively takes action specified by a policy, interacts with the environment, and ultimately learn 'knowledge or strategies' from the interactions. At every step, the agent makes an observation of the environment (we called a state or an observation), then it chooses an action to take. The environment will respond to the agent's action by changing its state and providing a reward signal to the agent, which serves as a gauge on the quality of the agent's current action. The goal of the agent, however, is not to maximize the immediate reward of an action, but is to maximize its cumulative reward along the whole process.



**Figure 32.2.1:** One core component in reinforcement learning is agent environment interaction. The agent takes actions based on observations on the environment and a decision-making module that maps observations to action. The environment model updates system state and provides rewards according to the action

For example, in the Atari game *Breakout* [Figure 32.2.2], we assume an agent controls the bat to deflect the ball to destroy the bricks. The actions allowed are moving left and

moving right; the state includes positions of the ball, the bat, and all the bricks; rewards will be given to the agent if bricks are hit. The environment, represented by a physical simulator, will simulate the ball's trajectory and collision between the ball and the bricks. The ultimate goal is learn a control policy that specifies the action to take after observing a state.



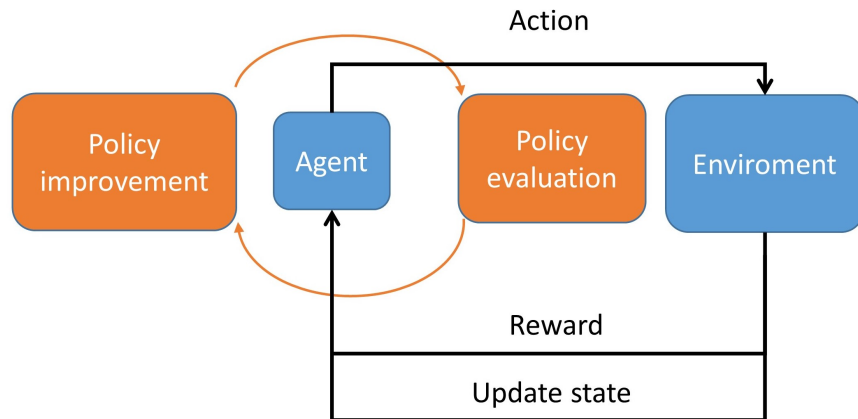
**Figure 32.2.2:** Scheme of the Atari game *breakout*.

In finite state Markov decision process, we introduce the two step iterative framework consisting of policy evaluation and policy improvement to seek optimal control policies. Although the agent-environment interaction paradigm seems to be vastly different from the Markov decision framework, many reinforcement learning algorithms can also be interpreted under this two step framework. We can view the agent-environment interaction process under a specified policy as policy evaluation step based on the rewards received from the environment. With the estimated value functions, we can similarly apply the policy improvement methods [Figure 32.2.3].

The sharp contrast between MDP and reinforcement learning is that reinforcement learning is model-free learning, the knowledge of the environment is from agent-

environment interaction data. On the other hand, MDP is model-based learning, meaning that the agent already know beforehand all the responses and rewards from environment for every action it takes. For complex read-world problems where models are not available, reinforcement learning offers a viable approach to learning optimal control policy via gradually building up the knowledge of the environment.

How much knowledge of the environment should the agent gain before the agent starts to improve the policy? Exploring the environment sufficiently would be prohibitive; on the other hand, improving the policy based on limited knowledge can produce inferior policies. The balance of environment exploration and policy improvement is known as the **exploration-exploitation dilemma**. One common way out is the  $\epsilon$  greedy action selection, where the agent has  $(1 - \epsilon)$  probability to continue to collect more samples based on currently perceived optimal control policy to improve current policy and  $\epsilon$  probability to randomly explore the environment via random actions.



**Figure 32.2.3:** Policy evaluation and policy improvement framework in the context reinforcement learning.

### 32.2.2 State-action Value function ( $Q$ function)

Like value function in a finite state MDP,  $Q$  functions play the same critical role in reinforcement learning. Now we go through their formal definition and their recursive relations.

**Definition 32.2.1 (state-action value function).** [2, p. 16]



- The **state-action value function**  $Q^\pi : S \times A \rightarrow \mathbb{R}$  associated with a policy  $\pi$  is defined as the expected return starting from state  $s$ , taking action  $a$  and thereafter following the policy  $\pi$ , given as

$$Q^\pi(s, a) = E^\pi \left\{ \sum_{k=0}^{\infty} \gamma^k R(s_{t+1}, a_t) \mid s_0 = s, a_t = a \right\}.$$

- The **optimal state-action value function**  $Q^* : S \times A \rightarrow \mathbb{R}$  is defined as the expected return starting from state  $s$ , taking action  $a$  and thereafter following an optimal policy  $\pi^*$ , such that

$$Q^*(s, a) = \max_{\pi} Q^\pi(s, a)$$

- The optimal policy  $\pi^*$  is related to  $Q^*$  as

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

- The value function associated with a policy  $\pi$ ,

$$V^\pi(s) = E^\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{k+1} \mid s_0 = s \right\}.$$

- The value function  $V(s)$  is connected with  $Q(s, a)$  via

$$V^\pi(s) = Q(s, \pi(s)).$$

- The optimal state-action value function is connected to value function via

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

#### Lemma 32.2.1 (recursive relations of the Q function).

- The state-action value function will satisfy

$$Q^\pi(s, a) = E_{s' \sim p(s' | s, \pi(s))}^\pi [r + \gamma Q^\pi(s', \pi(s'))],$$

where the expectation is taken with respect to the distribution of  $s'$  (the state after taking  $a$  at  $s$ ).

- The optimal state-action value function will satisfy

$$Q^*(s, a) = E_{s' \sim p(s' | s, \pi(s))}^{\pi^*} [r + \gamma \max_{a' \in A(s')} Q(s', a')],$$

where the expectation is taken with respect to the distribution of  $s'$  (the state after taking  $a$  at  $s$ ).

*Proof.* (1) From the definition  $Q^\pi(s, a)$ , we have

$$\begin{aligned} Q^\pi(s, a) &= E^\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_0 = s, a_t = a \right\} \\ &= E^\pi \left\{ r_1 + \sum_{k=1}^{\infty} \gamma^k r_{t+k+1} \mid s_0 = s, a_t = a \right\} \\ &= E^\pi \left\{ r_1 + E^\pi \left[ \sum_{k=1}^{\infty} \gamma^k r_{t+k+1} \mid s_1 = s, a_1 = \pi(s_1) \right] \mid s_0 = s, a_t = a \right\} \\ &= E^\pi \{ r_1 \mid s_0 = s, a_t = a \} + Q^\pi(s_1, \pi(s_1)) \mid s_0 = s, a_t = a \} \end{aligned}$$

where we have used the tower property of conditional expectation. (2) From (1) we have

$$Q^*(s, a) = E_{s' \sim p(s' \mid s, \pi(s))}^* [r + \gamma Q^*(s', \pi^*(s'))].$$

Further note that  $\pi^*(s') = \arg \max_{a \in \mathcal{A}(s')} Q(s', a)$  □

**Remark 32.2.1 (recursive relation for value functions).** Recall that in [Lemma 32.1.1](#), we have covered the recursive relation for value functions.

- Given a value function  $V^\pi$  associated with control policy  $\pi$ . The value function satisfies the following backward relationship:

$$V^\pi(s) = E_{s' \sim P(s' \mid s, a = \pi(s))} [R(s', a) + \gamma V^\pi(s')].$$

- Particularly, we have the Bellman's equation saying that

$$V^*(s) = \max_a E_{s' \sim P(s' \mid s, a)} [R(s', a) + \gamma V^*(s')].$$

### 32.2.3 Monte-Carlo method

#### 32.2.3.1 On-policy value estimation

Monte-Carlo method estimation of the value function associated with a policy is based on the straight forward definition:

$$V^\pi(s) = E^\pi \left[ \sum_{n=0}^T \gamma^n R(s_{t+n+1}, a_t) \mid s_0 = s, \pi \right].$$

$$Q^\pi(s, a) = E^\pi \left\{ \sum_{n=0}^T \gamma^n R(s_{t+1}, a_t) \mid s_0 = s, a_t = a \right\}.$$

The key steps are the following trajectory generation and aggregation. For example,

- Generate  $n$  trajectories using  $\pi$ , with each trajectories given by  $s_0, a_0, r_1, \dots, s_T$ .
- Estimate value function based on the definition

$$V(s_0) = \frac{1}{n} \sum_{i=1}^n G_n.$$

In the actual implementation of the algorithm, we can use Markov property of the trajectory and make most use of the data by the following first-visit MC method [algorithm 71]. Note that Monte-Carlo method usually applies to episodic tasks, where each episodes eventually terminate no matter what actions are selected. Since we use averaging as the estimator, the estimation is unbiased and the estimation error falls as  $1/\sqrt{n}$ , where  $n$  is the first-visit samples at state  $s$ .

---

**Algorithm 71:** First-visit MC value function estimation

---

**Input:** policy  $\pi$ , positive integer  $num\_episodes$

**Output:** value function estimates  $V^\pi$  and  $Q^\pi$  (if  $num\_episodes$  is large enough)

```

1 Initialize counter  $N(s) = 0$  and  $N(s, a)$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}$ .
2 Initialize  $returns\_sum(s) = 0$  for all  $s \in \mathcal{S}$ 
3 Initialize  $returns\_sum(s, a) = 0$  for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 
4 for  $i = 1$  to  $num\_episodes$  do
5   Generate an episode  $s_0, a_0, r_1, \dots, s_T$  using  $\pi$ .
6   for  $t = T - 1$  to  $0$  do
7      $G_t = \sum_{i=t}^T r_i$ .
8     if  $S_t$  is a first visit (i.e.,  $s_t$  does not appear in  $s_0, a_0, \dots, s_{t-1}, a_{t-1}$ ) then
9        $N(s_t) = N(s_t) + 1$ 
10       $returns\_sum(s_t) = returns\_sum(s_t) + G_t$ 
11       $N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$ 
12       $returns\_sum(s_t, a_t) \leftarrow returns\_sum(s_t, a_t) + G_t$ 
13   end
14 end
15  $V(s) \leftarrow returns\_sum(s) / N(s)$  for all  $s \in \mathcal{S}$ 
16  $Q(s, a) \leftarrow returns\_sum(s, a) / N(s, a)$  for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 
17 return  $V$ 
```

---

## 32.2.3.2 Off-policy value estimation

In the on-policy Monte-Carlo method, trajectories generated on a policy can only be used to estimate the value functions associated the policy. There are cases where we want to estimate the value function associated with  $\pi$  using trajectories generated under another exploratory policy  $\beta$ . The following theorem gives the theoretical tool to do so.

**Lemma 32.2.2 (MC off-policy estimation via importance-sampling).** *Given a starting state  $s_t$ , the probability of the subsequent state-action trajectory,  $a_t, s_{t+1}, a_{t+1}, \dots, s_T$  generated via following policy  $\pi$ . We have*

- $$Pr\{a_t, s_{t+1}, a_{t+1}, \dots, s_T | s_t, a_{t:T-1} \sim \pi\} = \prod_{k=t}^{T-1} \pi(a_k | s_k) p(s_{k+1} | s_k, a_k).$$
- Suppose we want to use trajectories generated under policy  $\beta$  to estimate value function  $V^\pi$ , we have

$$E^\beta[G_t \rho_{t:T-1} | s_t = s] = V^\pi(s)$$

where  $G_t = \sum_{n=t}^{T-1} R_{n+1}$ , and

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(a_k | s_k) p(s_{k+1} | s_k, a_k)}{\prod_{k=t}^{T-1} \beta(a_k | s_k) p(s_{k+1} | s_k, a_k)} = \prod_{k=t}^{T-1} \frac{\pi(a_k | s_k)}{\beta(a_k | s_k)}$$

*Proof.* (1)

$$\begin{aligned} & Pr\{a_t, s_{t+1}, a_{t+1}, \dots, s_T | s_t, a_{t:T-1} \sim \pi\} \\ &= \pi(a_t | s_t) p(s_{t+1} | s_t, a_t) \pi(a_{t+1} | s_{t+1}) \cdots p(s_T | s_{T-1}, a_{T-1}) \\ &= \prod_{k=t}^{T-1} \pi(a_k | s_k) p(s_{k+1} | s_k, a_k) \end{aligned}$$

(2) Note that

$$V^\pi(s) = E^\pi[G_t | s_t = s] = E^\beta[G_t \rho_{t:T-1} | s_t = s]$$

□

## 32.2.3.3 MC-based reinforcement learning control

Previous algorithms and theorems provide methods to estimate the value functions associated a policy  $\pi$ . Our generally policy iteration procedure suggest that we can combine value estimation with policy improvement to optimize a control policy.

To facilitate the policy improvement, we estimate  $Q$  function in the value estimation, and then improve the policy using greedy procedure  $\pi(s) = \arg \max_a Q(s, a)$ .

An example algorithm is showed in [algorithm 72\[1\]](#). Notably, to address the exploration-exploitation dilemma, the action selection is  $\epsilon$ -greedy, where there is a  $\epsilon$  probability of selecting random action and an  $1 - \epsilon$  probability of selecting  $\pi(s) = \arg \max_a Q(s, a)$ . In addition, the  $Q$  function estimation is different from the aggregation average method; instead, it employs a stochastic gradient descent step to approach an biased target  $G_t$ .

---

**Algorithm 72:** MC-based reinforcement learning control

---

**Input:** positive integer  $num\_episodes$ , learning rate  $\alpha, \epsilon$   
**Output:** policy  $\pi$  ( $\approx \pi_*$  if  $num\_episodes$  is large enough)

```

1 Initialize  $Q$  arbitrarily (e.g.,  $Q(s, a) = 0$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$ )
2 for  $i \leftarrow 1$  to  $num\_episodes$  do
3    $\pi \leftarrow \epsilon$ -greedy( $Q$ )
4   Generate an episode  $s_0, a_0, r_1, \dots, s_T$  using  $\pi$ 
5   for  $t \leftarrow 0$  to  $T - 1$  do
6     if  $(s_t, a_t)$  is a first visit (with return  $G_t$ ) then
7        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(G_t - Q(s_t, a_t))$ 
8     end
9 end
10 return  $\pi$ 

```

---

#### 32.2.4 TD(o) learning

##### 32.2.4.1 TD(o) for value estimation

The Monte-Carlo method estimates value function based on the connection of process return and value function. It is arguably the most straight forward method except for the possibility of high variance of the estimate (i.e., sum of random variables can have large variance).

Temporal-difference learning, also known as  $TD(0)$ , provides a novel route that combines Monte-Carlo and the recursive relations [[Lemma 32.1.1](#), [Lemma 32.2.1](#)] between value function. It updates the value function  $V(s)$  estimate via stochastic gradient descent to approaches its bootstrapped target

$$E[r + \gamma V(s')].$$

In comparison with Monte Carlo methods, TD(o) usually learns faster than Monte-Carlo, at the cost of using a possibly biased bootstrapped target. Monte-Carlo directly learns

the unbiased target, although the return sample is usually of high variance (because the return is the sum of random variables). The algorithm is given in [algorithm 73](#).

---

**Algorithm 73:** TD(0) estimation of a value function.

---

**Input:** A policy  $\pi$  to be evaluated. Learning rate  $\alpha$

**Output:** An estimate of  $V^\pi$

```

1 Initialize policy parameter  $\theta \in \mathbb{R}^d, w \in \mathbb{R}^m$ 
2 repeat
3   Initialize initial state  $s_0$ 
4   repeat
5     Selection action  $a$  according to  $\pi$ , transition to  $s'$ , and observe reward  $R$ .
6     Estimate  $V$  estimate via  $V(s) = V(s) + \alpha(R + \gamma V(s') - V(s))$ 
7     Estimate  $Q$  estimate via  $Q(s, a) = Q(s, a) + \alpha(R + \gamma Q(s', \pi(s')) - Q(s, a))$ 
8      $s = s'$ 
9   until  $S$  is terminal;
10 until sufficient iterations;
```

---

#### 32.2.4.2 On-policy reinforcement learning control

The on-policy Q learning involves two steps: the first step is to learn an state-action value function  $Q^\pi(s, a)$  for the current behavior policy  $\pi$  for all states  $s$  and actions  $a$ . The learning target for  $Q(s_t, a_t)$  is given by

$$(r_{t+1} + \gamma Q(s_{t+1}, \pi(s_{t+1}, a))).$$

The second is a policy improvement step, where we can get an improved policy at state  $s$  via  $\pi(s) = \arg \max_a Q(s, a)$ .

The resulting algorithm, containing  $\epsilon$  greedy exploration, is given by [algorithm 74](#).

---

**Algorithm 74:** SARSA learning

---

**Input:** policy  $\pi$ ,  $\epsilon$   
**Output:** value function  $Q$

- 1 Initialize  $Q$  arbitrarily (e.g.,  $Q(s, a) = 0$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$ , and  $Q(\text{terminal-state}, \cdot) = 0$ )
- 2 **repeat**
- 3     Initialize  $s \notin \mathcal{S}_T$ .
- 4     **repeat**
- 5         Choose an action  $a = \arg \max_a Q(s, a)$  with  $1 - \epsilon$  probability; otherwise choose a random action (i.e.,  $\epsilon$ -greedy).
- 6         Execute the action  $a$ , observe the new state  $s'$  and receive the reward  $r$ .
- 7         Choose an action  $a' = \arg \max_a Q(s', a)$  with  $1 - \epsilon$  probability; otherwise choose a random action (i.e.,  $\epsilon$ -greedy).
- 8          $Q(s, a) = Q(s, a) + \alpha(r_t + \gamma Q(s', a') - Q(s, a))$
- 9          $s = s'$ .
- 10     **until**  $s_t$  is in terminal states  $\mathcal{S}_T$ ;
- 11 **until** sufficient iterations;

---

#### 32.2.4.3 Off-policy reinforcement learning control

The off-policy Q learning involves two iterative steps: the first step is to directly learn/approximate the optimal state-action value function  $Q^*(s, a)$  (based on current estimated  $Q(s, a)$ ) independent of the current behavior policy  $\pi$  for all states  $s$  and actions  $a$ . The learning target is given by

$$(r_{t+1} + \gamma \max_a Q(S_{t+1}, a)).$$

The second is a policy improvement step, where we can get an improved policy at state  $s$  via  $\pi(s) = \arg \max_a Q(s, a)$ .

The resulting algorithm, contains  $\epsilon$  greedy exploration, is given by [algorithm 74](#).

---

**Algorithm 75:** Q-learning algorithm

---

**Input:** *epsilon*, discount factor  $\gamma$ , and learning rate  $\alpha$

```

1 Initialize Q. repeat
2   Initialize  $s \notin \mathcal{S}_T$ . repeat
3     Choose an action  $a = \arg \max_a Q(s, a)$  with  $1 - \epsilon$  probability; otherwise
       choose a random action (i.e.,  $\epsilon$ -greedy).
4     Execute the action  $a$ , observe the new state  $s'$  and receive the reward  $r$ .
5     Update Q function estimate via
           
$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a' \in A(s')} Q(s', a'))$$

6      $s = s'$ 
7   until  $s_t$  is in terminal states  $\mathcal{S}_T$ ;
8 until sufficient iteration;
Output: The Q function.

```

---

**Remark 32.2.2.** The convergence of Q learning algorithm has been addressed at [3, p. 495]. Essentially, there are two requirements:

- All state control pairs  $(s, a)$  must be generated infinitely often within the infinitely long sequence  $\{(s_k, a_k)\}$ .
- The stepsize/learning rate should be diminishing and satisfying the following conditions:

$$\alpha_k > 0, \forall k, \sum_{k=1}^{\infty} \alpha_k = \infty, \sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

One choice would be  $\alpha_k = \frac{1}{n}$ .

**Remark 32.2.3** (on-policy learning (SARSA) vs. off-policy learning (Q learning)).

- The reason that SARSA is on-policy is that it updates its Q-values using the Q-value of the next state  $s'$  and the current policy's action  $a''$ . It estimates the return for state-action pairs assuming the current policy ( $\epsilon$ -greedy) continues to be followed.
- The reason that Q-learning is off-policy is that it updates its Q-values using the Q-value of the next state  $s'$  and the greedy action  $a'$ . In other words, it estimates the return (total discounted future reward) for state-action pairs assuming a greedy policy were followed despite the fact that it's not following a greedy policy.
- In a nutshell, the bootstrapped Q function at  $(s_{t+1}, a_{t+1})$  in SARSA is

$$Q(s_{t+1}, a_{t+1}) = \epsilon \cdot E_a[Q(s_{t+1}, a)] + (1 - \epsilon) \cdot \max_a Q(s_{t+1}, a);$$



in Q-learning it is

$$Q(s_{t+1}, a_{t+1}) = \max_a Q(s_{t+1}, a).$$

### 32.2.5 TD( $n$ ) learning

#### 32.2.5.1 *Motivation and concepts*

One step temporal difference learning, or TD(o) learning usually contains error due to the learning of a bootstrapped target. To see this, consider one-step Sarsa algorithm that uses the target

$$G_t = R_{t+1} + \gamma Q_\theta(s_{t+1}, a_{t+1}).$$

Likewise, in the one-step value estimation, we have target for  $V$  given by

$$G_t = R_{t+1} + \gamma V_\theta(s_{t+1}).$$

Because  $Q_\theta(s_{t+1}, a_{t+1})$  and  $V_\theta(s_{t+1})$  are the estimates of  $Q^\pi$  and  $V^\pi$ , i.e., the target are biased, we propagate error when we use a biased target.

On the other extreme, Monte-Carlo methods like the REINFORCE algorithm uses the entire sequence to estimate values, with the target given by

$$G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^K V(s_{t+K}), s_{t+K} \in \mathcal{S}_T.$$

The target is unbiased, but Monte-Carlo methods tend to learn slowly due to the high variance (because of the summation of multiple random variables).

TD( $n$ ) is a method between the two extremes: the target contains  $n$ -step forward rewards plus a bootstrapped component. TD( $n$ ) combines the two strengths of TD(o) and Monte Carlo and addresses their weaknesses, as TD( $n$ ) produces a target with less biased than TD(o) and of smaller variance than Monte-Carlo.

The construction of the value target in TD( $n$ ), with  $1 \leq n \leq \infty$ , is showed in [Table 32.2.1](#). Clearly, TD( $n$ ) is the generalization of TD(o) and the Monte Carlo method.

$n$	$G_t$
$n = 1$	$G_t^{(1)} = R_{t+1} + \gamma V(s_{t+1})$
$n = 2$	$G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(s_{t+2})$
$\dots$	$\dots$
$n = n$	$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^n V(s_{t+n})$
$\dots$	$\dots$
$n = \infty$	$G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^K V(s_{t+K}), s_{t+K} \in \mathcal{S}_T$

**Table 32.2.1:** Estimating cumulative rewards  $G_t^{(n)}$  of different steps  $n$  as the target for value function  $V$ .  $G_t^{(1)}$  corresponds to temporal-difference TD(o) and  $G_t^{(\infty)}$  corresponds to Monte-Carlo estimation. If the process terminates at  $K$  and  $K < n$ , then we use  $G_t^{(n)} = G_t^{(K)}$ . Trajectories are generated under policy  $\pi$ .

### 32.2.5.2 TD(n) for value estimation

Like TD(o) and Monte-Carlo Methods, we can use TD(n) to estimate the value function associated with a policy  $\pi$ , with the algorithm given in [algorithm 76](#). Note that in our algorithm, we use a buffer of size  $n$  to store the previous  $n$  transitions and rewards in order to construct  $G_t^n$ .

Although this algorithm is estimating  $V$ , we can estimate  $Q(s_t, a_t)$  by changing the target  $G_t^{(n)}$  to

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^n Q(s_{t+n}, a_{t+n}),$$

where the first step action is  $a_t$ , not necessarily  $\pi(s_t)$ .

---

**Algorithm 76:** TD(n) estimation of a value function.

---

**Input:** A policy  $\pi$  to be evaluated. Learning rate  $\alpha$ . A buffer of size  $n$  to store transitions and reward in the last  $n$  steps.

**Output:** An estimate of  $V^\pi$

```

1 repeat
2   Initialize initial state  $s_0$ 
3   repeat
4     Selection action  $a$  according to  $\pi$ , transition to  $s'$ , and observe reward  $R$ .
5     Using the buffer to store  $R$ . Construct  $G_t^n$  using previous stored  $R$  is the
       buffer. If not enough observation for construction, then continue.
       Estimate  $V$  estimate via  $V(s_{-n}) = V(s_{-n}) + \alpha(R + \gamma V(s') - V(s_{-n}))$ 
6     Estimate  $Q$  estimate via
           
$$Q(s_{-n}, a) = Q(s_{-n}, a) + \alpha(R + \gamma Q(s', \pi(s')) - V(s_{-n})),$$

       Note hat  $s_{-n}, a_{-n}$  are state and action  $n$  steps ago, retrieved from the
       buffer.  $s = s'$ .
7   until  $s$  is terminal;
8 until sufficient iterations;
```

---

32.2.5.3 *TD(n) for reinforcement learning control*

Similar to how we turn TD(0) value estimation algorithm to a reinforcement learning control algorithm, by adding a policy improvement steps, we get yield a n-step SARSA algorithm [algorithm 77].

**Algorithm 77:** *n*-step SARSA learning

---

**Input:** policy  $\pi$ ,  $\epsilon_i$ , learning rate  $\alpha$ . A buffer of size  $n$  to store transitions and reward in the last  $n$  steps.

**Output:** value function  $Q$

- 1 Initialize  $Q$  arbitrarily (e.g.,  $Q(s, a) = 0$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$ , and  $Q(\text{terminal-state}, \cdot) = 0$ )
- 2 **repeat**
- 3     Initialize  $s \notin \mathcal{S}_T$ .
- 4     **repeat**
- 5         Choose an action  $a = \arg \max_a Q(s, a)$  with  $1 - \epsilon$  probability; otherwise choose a random action(i.e.,  $\epsilon$ -greedy).
- 6         Execute the action  $a$ , observe the new state  $s'$  and receive the reward  $r$ .
- 7         Construct  $G_t^n$  using previous stored  $R$  is the buffer. If not enough observation for construction, then continue. Choose an action  $a' = \arg \max_a Q(s', a)$  with  $1 - \epsilon$  probability; otherwise choose a random action(i.e.,  $\epsilon$ -greedy).
- 8         
$$Q(s_{-n}, a_{-n}) = Q(s_{-n}, a_{-n}) + \alpha(G_t^{(n)} + \gamma Q(s', a') - Q(s_{-n}, a_{-n}))$$

where  $s_{-n}, a_{-n}$  are state and action  $n$  steps ago, retrieved from the buffer.

$s = s'$ .
- 9     **until**  $s_t$  is in terminal states  $\mathcal{S}_T$ ;
- 10 **until** sufficient iterations;
- 11 **return**  $Q$

---

## 32.2.6 Standing challenges in reinforcement learning

32.2.6.1 *Curse of dimensionality*

Most RL algorithm requires the learning of  $Q(s, a)$  or  $V(s)$  for *all* state  $s$  and action  $a$ . In tabular RL, where  $Q$  and  $V$  are stored as multi-dimensional arrays, increasing dimensionality will exponentially increase the memory storage requirement of value functions. One popular solution is to used parameterized functions to represent  $Q$  and

$V$ , including linear function approximators using predefined basis functions to nonlinear approximators such as neural networks.

#### 32.2.6.2 *Sample efficiency*

Sample efficiency is regarding how many data samples are needed if we want to learn a competitive policy. Although RL have achieved remarkable performance in problems there are sufficient data, many real-world problems are facing the hurdle of data scarcity and hinder the application of RL. How to collect data sample and how to efficiently use sample have received extensive attention. Common strategies include off-policy to reuse data sample, curriculum learning to enable smart exploration of parameter space, etc.

#### 32.2.6.3 *Exploration-exploitation dilemma*

Because of the nature of model-free learning, the knowledge of the environment is from agent-environment interaction data. In one limit, if the agent explores the environment sufficiently to obtain a model of the environment, RL reduces to model-based MDP. Complicated problems are usually characterized by their immense state space or high cost (e.g., time, money) to collect more data and it is prohibitive to construct a high fidelity model by exploring the environment comprehensively. On the other hand, if we rush for the policy based on an substantially incomplete representation of environment, we can possibly end up with a mediocre control policy.

The dilemma stems from need to make better decisions based relatively incomplete information. Different RL algorithms cope with the dilemma in a different fashion, including  $\epsilon$  greedy action selection, stochastic policy with entropy penalty, addition of noise into policy parameter space.

#### 32.2.6.4 *Deadly triad*

In solving complex RL problems, it is almost inevitable to use nonlinear function approximators for state value estimations. When using bootstrapping TD methods with off-policy learning, value estimation error can propagate in a non-controllable way, leading to divergence. The three issues combined, nonlinear approximator, bootstrapping, and off policy, are to known as **the deadly triad** [4].

## 32.3 Policy gradient learning

### 32.3.1 Stochastic policy gradient fundamentals

#### 32.3.1.1 Preliminaries: derivative and expectation

We first review some useful results that will be used repeatedly in the following subsections devoted to the policy gradient theory.

**Lemma 32.3.1 (derivative and expectation identities for stochastic policy gradient).**

*Suppose functions and integrals we consider here are well-behaved such that derivative and integration can be exchanged.*

- Let  $g_\theta(x)$  be a function parameterized by  $\theta$ . We have

$$\nabla_\theta \int f(x)g_\theta(x)dx = \int f(x)\nabla_\theta g_\theta(x)dx$$

- (**log-derivative trick**) Let  $p_\theta$  be a  $\theta$ -parameterized density function for random variable  $x$ , we have

$$\nabla_\theta E_{X \sim p_\theta}[f(X)] = E_{X \sim p_\theta}[f(X)\nabla_\theta \log p_\theta(X)],$$

where  $E_{X \sim p_\theta}$  indicates expectation is taken with respect to random variable  $X$  whose the distribution is given by  $p_\theta(X)$ .

- 

$$\nabla_\theta E_{X \sim p_\theta}[p_\theta(X)] = 0.$$

Moreover, if we let  $b_0$  be a constant and  $b(\theta)$  be a function only depends on  $\theta$ , we have

$$\nabla_\theta E_{X \sim p_\theta}[b_0 p_\theta(X)] = 0, \nabla_\theta E_{X \sim p_\theta}[b(\theta)p_\theta(X)] = 0.$$

*Proof.* (1) Note that  $f(x)$  is not dependent on  $\theta$ . Also check the following example. (2) Using (1), we have

$$\begin{aligned} \nabla_\theta E_{X \sim p_\theta}[f(X)] &= \int f(x)\nabla_\theta p_\theta(x)dx \\ &= \int f(x)\nabla_\theta p_\theta(x)dx \\ &= \int f(x)p_\theta(x)\nabla_\theta \log p_\theta(x)dx \\ &= E_{X \sim p_\theta(\tau)}[f(x)\nabla_\theta \log p_\theta(X)]. \end{aligned}$$

(3) Use (2) in reverse. We have

$$E[\nabla_{\theta} \log p_{\theta}(X)] = \nabla_{\theta} E[1] = 0.$$

For the rest, we see the  $b_0$  and  $b(\theta)$  can be pulled out of the expectation. □

*Example 32.3.1.* Consider  $f(x) = x^2$ ,  $p_{\theta} = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(x-\theta)^2}{2\sigma^2})$ , we have

$$\begin{aligned} & \frac{d}{d\theta} \int f(x) p_{\theta}(x) dx \\ &= \frac{d}{d\theta} \int x^2 \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(x-\theta)^2}{2\sigma^2}) dx \\ &= \int x^2 \frac{d}{d\theta} \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(x-\theta)^2}{2\sigma^2}) dx \\ &= \int x^2 \frac{(x-\theta)}{\sigma} \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(x-\theta)^2}{2\sigma^2}) dx \\ &= \int f(x) \frac{dp_{\theta}}{d\theta} dx \end{aligned}$$

### 32.3.1.2 Theoretical framework based on finite-horizon trajectories

Policy gradient methods aim to learn a parameterized policy to directly select actions without using value functions. Specifically, we will parameterize the policy by  $\theta$  such that a stochastic policy is given by

$$\pi(a|s, \theta) = \Pr(a_t = a | s_t = s, \theta),$$

and a deterministic policy is given by  $\mu(a|s, \theta)$ . The ultimate goal in policy gradient methods boils down to seeking the optimal parameter  $\theta^*$  such that expected cumulative rewards during decision-making processes are maximized.

The mainstream methodology to optimize the policy parameter  $\theta$  is gradient descent. Therefore calculating or estimating the gradient with respect to  $\theta$  is the cornerstone for nearly all policy gradient algorithms.

The pros and cons of policy gradient methods, compared with value based methods, are

- Policy-based methods usually have better convergence properties as they directly work in the policy space.

- Policy gradients are more effective in high dimensional action spaces, especially when using continuous actions.
- Policy gradient method can learn a stochastic policy, while value-function-based method cannot.

This section we will introduce basic definition and procedures regarding how to estimate policy gradient in a finite-horizon trajectory setting. Next section we cover policy gradient concept and calculation in the infinite-horizon setting. The later is theoretically more challenging than the former, but its results is elegant and shed light on the connection to the value based methods we covered in our previous sections.

We start with basic definitions in the finite-horizon setting.

**Definition 32.3.1 (basic concepts).**

- (stochastic policy) A (stochastic) policy  $\pi_\theta$  is a differential function parameterized by  $\theta$ , given by

$$\pi(a|s, \theta) = \Pr(A_t = a, \theta_t = \theta), \theta \in \mathbb{R}^d.$$

- Assume Markov model for the state transition dynamics. Consider a finite-length trajectory generated by carrying out policy  $\pi$  defined by  $\tau = \{s_0, a_0, s_1, \dots, a_{T-1}, s_T\}$ . Then we define the **trajectory probability** by

$$p_\theta(\tau) = \rho_0(s_0) \prod_{t=1}^T \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t).$$

where  $\rho_0$  is the initial state distribution,  $p(s_{t+1}|s_t, a_t)$  is the transition probability specified by the Markov chain.

- Let  $r_1 = R(s_1, a_0), r_2 = R(s_2, a_1), \dots, r_T$  be the rewards associated with above trajectory. We define trajectory reward  $R(\tau)$  as

$$R(\tau) = \sum_{t=1}^T r_t$$

The objective function to optimize  $\theta$  is the expected trajectory rewards, which is given as follows.

**Definition 32.3.2 (objective function of policy gradient method).** The objective function for policy parameter  $\theta$  optimization is

$$\max_{\theta} J(\theta) = \max_{\theta} E_{\tau \sim p_\theta(\tau)}[R(\tau)],$$



where  $\tau$  is a stochastic trajectory of length  $T$  generated from policy  $\pi_\theta(a|s)$ , and  $p_\theta(\tau)$  is the trajectory probability under policy parameter  $\theta$ .

Now we are in a position to introduce the theory underlying calculating the gradient of  $J(\theta)$  with respect to  $\theta$  in the following Th.. Analytical calculation based on this theorem is usually intractable. We postpone numerical approaches to following sections.

**Theorem 32.3.1 (stochastic policy gradient on finite-horizon trajectories).** *With the reward-to-go function given by*

$$J(\theta) = E_{\tau \sim p_\theta(\tau)}[r(\tau)] = \int r(\tau) p_\theta(\tau) d\tau,$$

its gradient is

$$\begin{aligned} \nabla_\theta J(\theta) &= \int r(\tau) \nabla_\theta p_\theta(\tau) d\tau \\ &= E_{\tau \sim p_\theta(\tau)} \left[ \left( \sum_{t=0}^{T-1} \nabla_\theta \ln \pi_\theta(a_t | s_t) \right) \left( \sum_{t=0}^{T-1} R(s_{t+1}, a_t) \right) \right] \end{aligned}$$

*Proof.* First, we have [Lemma 32.3.1]

$$\begin{aligned} \nabla_\theta J(\theta) &= \int R(\tau) \nabla_\theta p_\theta(\tau) d\tau \\ &= \int R(\tau) \nabla_\theta p_\theta(\tau) d\tau \\ &= \int R(\tau) p_\theta(\tau) \nabla_\theta \ln p_\theta(\tau) d\tau \\ &= E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla_\theta \ln p_\theta(\tau)]. \end{aligned}$$

Note that

$$\begin{aligned} p_\theta(\tau) &= p(s_1) \prod_{t=0}^{T-1} \pi_\theta(a_t | s_t) p(s_{t+1} | s_t, a_t) \\ \implies \ln p_\theta(\tau) &= \ln p_0(s_0) + \left( \sum_{t=0}^{T-1} \ln \pi_\theta(a_t | s_t) + \ln p(s_{t+1} | s_t, a_t) \right) \\ \implies \nabla_\theta \ln p_\theta(\tau) &= \sum_{t=0}^{T-1} \nabla_\theta \ln \pi_\theta(a_t | s_t) \end{aligned}$$

Plug into  $\nabla_\theta J(\theta)$ , we have

$$\begin{aligned}\nabla_{\theta} J(\theta) &= E_{\tau \sim p_{\theta}(\tau)} \left[ \left( \sum_{t=0}^{T-1} \nabla_{\theta} \ln \pi_{\theta}(a_t | s_t) \right) (R(\tau)) \right] \\ &= E_{\tau \sim p_{\theta}(\tau)} \left[ \left( \sum_{t=0}^{T-1} \nabla_{\theta} \ln \pi_{\theta}(a_t | s_t) \right) \left( \sum_{t=0}^{T-1} R(s_{t+1}, a_t) \right) \right]\end{aligned}$$

□

### 32.3.1.3 Theoretical framework based on distributions\*

Besides formulating policy gradient based on finite-length trajectories, we can also approach policy gradient in the context of stationary Markov chain specified by system dynamics and control policy. However, the derivation is significantly abstract and difficult. In general, policy gradient theorem based on finite-length trajectories is sufficient to understand most of algorithms.

**Definition 32.3.3.** We assume finite state Markov decision process and the Markov chain can reach equilibrium under arbitrary stochastic policy.

- (stochastic policy) A (stochastic) policy  $\pi_{\theta}$  is a differential function parameterized by  $\theta$ , given by

$$\pi(a|s, \theta) = \Pr(A_t = a, \theta_t = \theta), \theta \in \mathbb{R}^d.$$

- stationary distribution under stochastic policy  $\pi$ , denoted by  $d^{\pi}(s)$ , is the stationary distribution of Markov chain under policy  $\pi$ . Mathematically,

$$d^{\pi}(s) = \lim_{t \rightarrow \infty} P(s_t = s | s_0, \pi).$$

Note that  $d^{\pi}(s)$  is independent of initial state.

- The stationary reward function  $J(\theta)$  is defined by

$$J(\theta) = \sum_{s \in \mathcal{S}} d^{\pi}(s) V^{\pi}(s) = \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} \pi_{\theta}(a|s) Q^{\pi}(s, a),$$

where we use the relation

$$\sum_{a \in \mathcal{A}} \pi_{\theta}(a|s) Q^{\pi}(s, a) = V^{\pi}(s).$$

- Discounted reward-to-go function associated with policy  $\pi$ :

$$J(\pi) = E \left[ \sum_{k=1}^{\infty} \gamma^{k-1} r(s_k, a) | \pi \right].$$

- State-action value function associated with policy  $\pi$ , which is defined as the expected return starting from state  $s$ , taking action  $a$  and **thereafter following** the policy  $\pi$ . It is given by

$$Q^\pi(s, a) = E\left[\sum_{k=1}^{\infty} \gamma^{k-1} r(s_k, a_k) | S_1 = s, A_1 = a; \pi\right].$$

- Value function associated with policy  $\pi$ :

$$V^\pi(s) = E\left[\sum_{k=1}^{\infty} \gamma^{k-1} r(s_k, a_k) | S_1 = s; \pi\right],$$

which the total expected return starting from state  $s$ , taking actions specified by the policy  $\pi$ .

- Discounted state distribution functions

$$\rho(s') = \int_{\mathcal{S}} \sum_{k=1}^{\infty} \gamma^{k-1} p_1(s) p(s \rightarrow s', t, \pi),$$

and we can write

$$J(\theta) = \int_{\mathcal{S}} \rho(s) \int_{\mathcal{A}} \pi(a|s, \theta) r(s, a) da ds = E_{s \sim \rho(s), a \sim \pi(s)}[r(s, a)].$$

#### Definition 32.3.4 (discounted state distribution).

- $\rho_0(s)$  initial distribution over states.
- $k$  step transition probability  $\rho^\pi(s \rightarrow s', k)$  characterizes the probability distribution for a Markov chain, starting from state  $s$ , after moving  $k$  steps according to policy  $\pi$ .
- Discounted state distribution

$$\rho^\pi(s') = \int_{\mathcal{S}} \sum_{k=1}^{\infty} \gamma^{k-1} \rho_0(s) \rho(s \rightarrow s', t, \pi),$$

and we can write

$$J(\theta) = \int_{\mathcal{S}} \rho(s) \int_{\mathcal{A}} \pi(a|s, \theta) r(s, a) da ds = E_{s \sim \rho(s), a \sim \pi(s)}[r(s, a)].$$

- discounted stationary reward function is given by

$$J(\theta) = \int_{s \in \mathcal{S}} \rho^\pi(s) \int_{a \in \mathcal{A}} Q(s, a) \pi_\theta(a|s) ds da = E_{s \sim \rho^\pi} \left[ \int_{a \in \mathcal{A}} Q^\pi(s, a) \pi_\theta(a|s) \right].$$

**Theorem 32.3.2 (policy gradient theorem on distributions).** [1, p. 325]

$$\begin{aligned}\nabla_{\theta} J(\theta) &= \nabla_{\theta} \sum_{s \in S} d^{\pi}(s) \sum_{a \in \mathcal{A}} Q^{\pi}(s, a) \pi_{\theta}(a|s) \\ &\propto \sum_{s \in S} d^{\pi}(s) \sum_{a \in \mathcal{A}} Q^{\pi}(s, a) \nabla_{\theta} \pi_{\theta}(a|s) \\ &= E_{s \sim d^{\pi}} \left[ \sum_{a \in \mathcal{A}} Q^{\pi}(s, a) \nabla_{\theta} \pi_{\theta}(a|s) \right]\end{aligned}$$

*note that both  $Q$  and  $\pi$  are functions of  $\theta$ .*

*Proof.*

$$\begin{aligned}\nabla_{\theta} V^{\pi}(s) &= \nabla_{\theta} \left( \sum_{a \in \mathcal{A}} \pi_{\theta}(a|s) Q^{\pi}(s, a) \right) \\ &= \sum_{a \in \mathcal{A}} \left( \nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) + \pi_{\theta}(a|s) \nabla_{\theta} Q^{\pi}(s, a) \right) \\ &= \sum_{a \in \mathcal{A}} \left( \nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) + \pi_{\theta}(a|s) \sum_{s'} P(s'|s, a) \nabla_{\theta} V^{\pi}(s') \right) \\ &= \sum_{a \in \mathcal{A}} \left( \nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) + \pi_{\theta}(a|s) \sum_{s'} P(s'|s, a) \nabla_{\theta} V^{\pi}(s') \right)\end{aligned}$$

where in the third line we remove  $r$  since it does not depend on  $\theta$ .

Therefore, the gradient has the following recursive form

$$\nabla_{\theta} V^{\pi}(s) = \sum_{a \in \mathcal{A}} \left( \nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) + \pi_{\theta}(a|s) \sum_{s'} P(s'|s, a) \nabla_{\theta} V^{\pi}(s') \right).$$

Denote  $\phi(s) = \sum_{a \in \mathcal{A}} \nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a)$ , we have

$$\begin{aligned}
 & \nabla_{\theta} V^{\pi}(s) \\
 &= \phi(s) + \sum_a \pi_{\theta}(a|s) \sum_{s'} P(s'|s, a) \nabla_{\theta} V^{\pi}(s') \\
 &= \phi(s) + \sum_{s'} \sum_a \pi_{\theta}(a|s) P(s'|s, a) \nabla_{\theta} V^{\pi}(s') \\
 &= \phi(s) + \sum_{s'} \rho^{\pi}(s \rightarrow s', 1) \nabla_{\theta} V^{\pi}(s') \\
 &= \phi(s) + \sum_{s'} \rho^{\pi}(s \rightarrow s', 1) \left[ \phi(s') + \sum_{s''} \rho^{\pi}(s' \rightarrow s'', 1) \nabla_{\theta} V^{\pi}(s'') \right] \\
 &= \phi(s) + \sum_{s'} \rho^{\pi}(s \rightarrow s', 1) \left[ \phi(s') + \sum_{s''} \rho^{\pi}(s' \rightarrow s'', 1) \nabla_{\theta} V^{\pi}(s'') \right] \\
 &= \phi(s) + \sum_{s'} \rho^{\pi}(s \rightarrow s', 1) \phi(s') + \sum_{s''} \rho^{\pi}(s \rightarrow s'', 2) \nabla_{\theta} V^{\pi}(s'') \\
 &= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \rho^{\pi}(s \rightarrow x, k) \phi(x)
 \end{aligned}$$

Note that we interpret  $\sum_{k=0}^{\infty} \rho^{\pi}(s \rightarrow x, k)$  as a scalar proportional to the stationary distribution at  $x$ , independent of  $s$ . Therefore

$$\nabla_{\theta} J(\theta) = \nabla_{\theta} V^{\pi}(s) \propto \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} Q^{\pi}(s, a) \nabla_{\theta} \pi_{\theta}(a|s)$$

□

**Lemma 32.3.2.** Let  $\rho(s)$  be the equilibrium distribution under policy  $\pi$ . Let  $\pi$  be parameterized by  $\theta$ . Let the reward-to-go function be

$$J(\theta) = \int_{s \in \mathcal{S}} \rho^{\pi}(s) \int_{a \in \mathcal{A}} Q(s, a) \pi_{\theta}(a|s) ds da = E_{s \sim \rho^{\pi}} \left[ \int_{a \in \mathcal{A}} Q^{\pi}(s, a) \pi_{\theta}(a|s) \right].$$

We have

$$\nabla_{\theta} J(\theta) = E_{s \sim \rho(s), a \in \pi(s)} [\nabla_{\theta} \ln \pi(a|s, \theta) Q^{\pi}(s, a)]$$

*Proof.*

$$\begin{aligned}
 \nabla_{\theta} J(\theta) &= \int_{\mathcal{S}} \rho(s) \int_{\mathcal{A}} \nabla_{\theta} \pi(a|s, \theta) Q^{\pi}(s, a) da ds \\
 &= \int_{\mathcal{S}} \rho(s) \int_{\mathcal{A}} \pi(a|s, \theta) \frac{\nabla_{\theta} \pi(a|s, \theta)}{\pi(a|s, \theta)} Q^{\pi}(s, a) da ds \\
 &= \int_{\mathcal{S}} \rho(s) \int_{\mathcal{A}} \pi(a|s, \theta) \nabla_{\theta} \ln \pi(a|s, \theta) Q^{\pi}(s, a) da ds \\
 &= E_{s \sim \rho(s), a \in \pi(s)} [\nabla_{\theta} \ln \pi(a|s, \theta) Q^{\pi}(s, a)]
 \end{aligned}$$

□

#### 32.3.1.4 Estimate policy gradient and basic algorithms

In [Theorem 32.3.1](#) and [Theorem 32.3.2](#), we have established that

$$\nabla_{\theta} J(\theta) = E_{\tau \sim p_{\theta}(\tau)} \left[ \left( \sum_{t=0}^{T-1} \nabla_{\theta} \ln \pi_{\theta}(a_t | s_t) \right) \left( \sum_{t=0}^{T-1} R(s_{t+1}, a_t) \right) \right].$$

We can use Monte Carlo to estimate the expectation in the following way.

**Methodology 32.3.1 (unbiased estimator of total expected rewards).** Suppose we have used policy  $\pi_{\theta}$  to generate  $N$  sample trajectories  $\tau_1, \dots, \tau_N$  of length  $T$ . Let  $s_{i,t}, a_{i,t}$  represent the sample value of state and action at trajectory  $i$  and time  $t$ . Then we can estimate

$$E_{\tau \sim p_{\theta}(\tau)} \left[ \sum_{t=0}^{T-1} R(s_{t+1}, a_t) \right]$$

via

$$\frac{1}{N} \sum_{i=1}^N \sum_{t=0}^{T-1} R(s_{i,t+1}, a_{i,t})$$

**Remark 32.3.1 (connection maximum likelihood estimation).** In the policy gradient method, we have

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} R(\tau_i) \ln p_{\theta}(\tau_i).$$

Similarly, the gradient for the log-likelihood function based on trajectories  $\tau_1, \dots, \tau_N$  is given by

$$\nabla_{\theta} L(\theta) \approx \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \ln p_{\theta}(\tau_i).$$

Therefore, policy gradient is similar to a weighted version of log-likelihood function. The weight is given by the trajectory reward. The policy gradient method thus maximizes the likelihood of trajectories with high rewards and minimizes the likelihood of trajectories with low rewards.

Now we have our first policy gradient algorithm based on batch trajectory update, given by [algorithm 78](#).

---

**Algorithm 78:** A generic batch policy-gradient algorithm (REINFORCE)

---

**Input:** A policy differential function parameterized  $\pi(a|s, \theta)$ , learning rate  $\alpha > 0$ .

**Output:** Improved policy  $\pi(a|s, \theta^*)$  parameterized by  $\theta^*$

1 Initialize policy parameter  $\theta \in \mathbb{R}^d$

2 **repeat**

3     Using current policy  $\pi(\cdot|\cdot, \theta)$  to generate  $N$  sample trajectories.

4     Estimate policy gradient

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N [(\nabla_{\theta} \ln \pi_{\theta}(a_{i,t}|s_{i,t})) (\sum_{t=1}^T R(s_{i,t}, a_{i,t}))].$$

    Update  $\theta = \theta + \alpha \nabla_{\theta} J(\theta)$ .

5 **until** *sufficient iterations*;

6 **return**  $V$

---

**Remark 32.3.2** (interpretation and issues with poor credit assignment and high variance).

- In the algorithm, we are taking gradient steps to maximize the probability of taking certain actions that lead to higher trajectory reward  $R(\tau)$ . Particularly, if  $R(\tau)$  is high, we will increase the probability of actions observed in  $\tau$ , and vice versa.
- However, this method has a severe drawback: since we only compute  $R(\tau)$  at the end of an episode, if some of the actions taken in  $\tau$  are truly bad then  $R(\tau)$  is large, we will still promote the probability of these bad actions.
- Another draw back is the high variance of the stochastic quantity  $R(\tau)$ , which is the summation of a possibly very long sequence of random variables  $\sum_t R(s_{t+1}, a_t)$ .

One way to improve the credit assignment and reduce the variance of gradient estimates is to simplify the gradient estimation via **causal relationship**. We have

$$\begin{aligned} \nabla_{\theta} J(\theta) &= E_{\tau \sim p_{\theta}(\tau)} \left[ \left( \sum_{t=0}^{T-1} \nabla_{\theta} \ln \pi_{\theta}(a_t|s_t) \right) \left( \sum_{t=0}^{T-1} R(s_{t+1}, a_t) \right) \right] \\ &= E_{\tau \sim p_{\theta}(\tau)} \left[ \left( \sum_{t=0}^{T-1} \nabla_{\theta} \ln \pi_{\theta}(a_t|s_t) \right) \left( \sum_{t'=t}^T R(s_{t'+1}, a_{t'}) \right) \right] \end{aligned}$$

This is because for every  $t$ , the quantity

$$E_{\tau \sim p_{\theta}(\tau)}[(\nabla_{\theta} \ln \pi_{\theta}(a_t|s_t))(\sum_{t'=1}^{t-1} R(s_{t'+1}, a_{t'}))]$$

does not depend on  $\theta$ . Intuitively, past trajectories does not depend on  $\pi_{\theta}(a_t|s_t)$ . Because we have fewer summation terms, the variance will in general decrease.

These improvements give us a basic Monte-Carlo policy gradient algorithm [algorithm 79].

---

**Algorithm 79:** A basic Monte-Carlo policy-gradient algorithm

---

**Input:** A policy differential function parameterized  $\pi(a|s, \theta)$ , learning rate  $\alpha > 0$ .

**Output:** Improved policy  $\pi(a|s, \theta^*)$  parameterized by  $\theta^*$

```

1 Initialize policy parameter  $\theta \in \mathbb{R}^d$ 
2 repeat
3   Using current policy  $\pi(\cdot|\cdot, \theta)$  to generate an episode  $(s_1, a_1, r_1), \dots, (s_T, a_T, r_T)$ .
4   for each step in the episode  $t \in \{1, \dots, T\}$  do
5      $G =$  return from step  $t$ 
6      $\theta = \theta + \alpha \gamma^t G \nabla_{\theta} \ln \pi(a_t|s_t, \theta)$ 
7   end
8 until sufficient iterations;
9 return  $V$ 
    
```

---

### 32.3.1.5 Bootstrap and Actor-Critic methods

In our previous section, we can see that we can estimate the gradient with respect to  $\theta$  from trajectories via

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=0}^{T-1} [(\nabla_{\theta} \ln \pi_{\theta}(a_{i,t}|s_{i,t})) \hat{Q}_{i,t}],$$

where  $\hat{Q}_{i,t} = \sum_{t'=t}^{T-1} R(s_{i,t'+1}, a_{i,t'})$ . In the REINFORCE algorithm and the Monte-Carlo algorithm, we estimate the cumulative rewards  $\sum_{t=0}^{T-1} R(s_{i,t}, a_{i,t-1})$  directly from samples, which in general come with large variance (also see subsection 32.2.5]. Similar to how we cope with the large variance issue in value based reinforcement learning, we can use bootstrap to reduce variance at the price of some bias. First, we can view  $\hat{Q}_{i,t}$  is one sample from state-action value function defined by

$$Q^{\pi}(s_t, a_t) = \sum_{t'=t}^{T-1} E_{\pi_{\theta}}[r(s_{t'+1}, a_{t'})|s_t, a_t]$$



In the REINFORCE algorithm and the Monte-Carlo algorithm, these samples are only used once and then discarded. Alternatively, if we maintain a function approximator that continuously improve approximate  $Q$  or  $V$ , we can write our gradient estimate by

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=0}^{T-1} [(\nabla_{\theta} \ln \pi_{\theta}(a_{i,t}|s_{i,t})) \hat{Q}(s_{i,t}, a_{i,t})],$$

Now we can decompose the estimation of policy gradient into two components, first component  $\nabla \pi$  and second component is  $\hat{Q}(s, a)$  or  $\hat{V}(s)$ . The Actor-Critic is the implementation of above idea, where we use a function approximator, called Actor, responsible to generate and update the policy, and a function approximator, called Critic, to approximate the value function that will assist in the estimation of policy gradients. The critic parameters will be updated based on bootstrapping and Bellman's equation and the Actor will be updated using policy gradient.

Below is a basic Actor-Critic algorithm [algorithm 80]. Note that we update the Critic by least square optimization of

$$\frac{1}{2}(y - Q(s, a|\theta^C))^2,$$

where  $y$  is the target given by  $y = R(s, a) + Q(s', a'; \theta^Q), a' \sim \pi(a|s', \theta^A)$ . If  $s \in \mathcal{S}_T, y = R(s, a)$  and update the Actor via

$$\theta^A = \theta^A + \alpha \hat{Q}(s, a; \theta^C) \nabla_{\theta} \pi(a|s, \theta^A).$$

**Algorithm 80:** Actor-Critic policy-gradient method

**Input:** A policy differential function parameterized  $\pi(a|s, \theta^A)$ , a differentiable state-value parameterization  $Q(s, a; \theta^C)$ , learning rate  $\alpha_1 > 0, \alpha_2 > 0$ .

**Output:** Improved policy  $\pi(a|s, \theta^{A*})$

```

1 Initialize Actor and Critic parameters.
2 repeat
3   Initialize initial state  $s_0$ 
4   repeat
5     Take action  $a \sim \pi(\cdot|S, \theta)$ , observe  $s', r$ .
6     Construct target for Critic:  $y = R(s, a) + Q(s', a'; \theta^C), a' \sim \pi(a|s', \theta^A)$ . If
        $s \in \mathcal{S}_T, y = R(s, a)$  Perform gradient descent update on  $\theta^C$  with learning
       rate  $\alpha_1$  on the objective function
          
$$\frac{1}{2}(y - Q(s, a|\theta^C))^2.$$

       Perform gradient descent update on  $\theta^A$  with learning rate  $\alpha_1$  using
          
$$\theta^A = \theta^A + \alpha \hat{Q}(s, a; \theta^C) \nabla_{\theta} \pi(a|s, \theta^A)$$

        $s = s'$ 
7   until  $s$  is terminal;
8 until sufficient iterations;
9 return  $V$ 

```

**Remark 32.3.3 (interpretation).** We can interpret the algorithm in the policy evaluation and policy improvement framework. The Actor-Critic algorithm is performing on-policy update. The improvement of the policy is achieved by gradient descent on policy parameter  $\theta^A$  and policy evaluation is achieved by estimating  $Q(s, a; \theta^C)$ .

## 32.3.1.6 Common stochastic policies and their representations

Commonly used stochastic policies and their neural network realization for discrete and continuous action spaces are given below.

**Definition 32.3.5 (categorical policy).** A categorical policy samples a set of discrete actions based on some probabilities. Its neural network architecture could be a feed-forward network with softmax layer on the final output layer, similar to neural network architecture for classification problems. The output of the network is the probability over the actions, denoted by  $\pi_{\theta}(a|s) = \Pr_{\theta}(a)$ .

**Definition 32.3.6 (multivariate Gaussian policy).** A multivariate Gaussian policy draws actions from a multivariate Gaussian distribution, characterized by a mean vector  $\mu$  and a covariance matrix  $\Sigma$ . For simplicity, diagonal  $\Sigma$  is often assumed.

A diagonal Gaussian policy always has a neural network that maps from observations to mean actions,  $\mu_\theta(s)$ . There are two different ways that the covariance matrix is typically represented.

Often, we employ feed-forward neural networks [Figure 32.3.1(a)] to directly map state  $s$  to mean vector and the log standard deviation, respectively. The mean network and the log std network might share some layers.

The final action is synthesized by

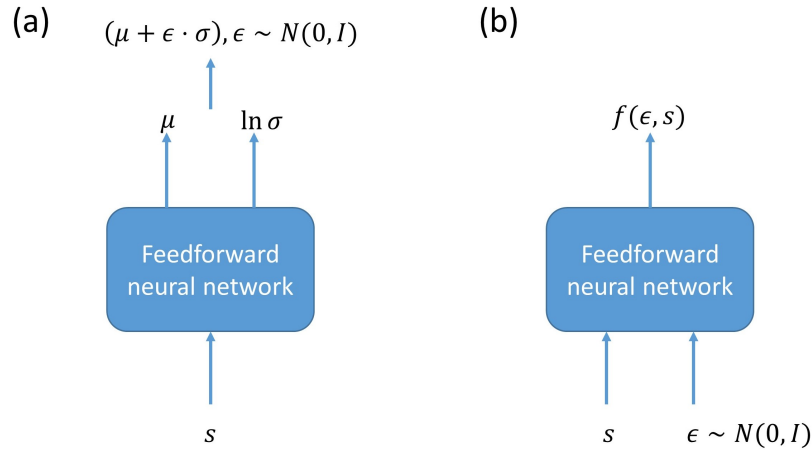
$$a = \mu_\theta(s) + \sigma_\theta(s) \odot z,$$

where  $\odot$  denotes the element-wise product of two vectors,  $z$  is a standard Gaussian random vector.

Note that given a policy  $\pi_{\mu,\sigma}$ , the probability of sampling  $a$  is given by

$$\log \pi_{\mu,\theta}(a|s) = -\frac{1}{2} \left( \sum_{i=1}^k \left( \frac{(a_i - \mu_i)^2}{\sigma_i^2} + 2 \log \sigma_i \right) + k \log 2\pi \right).$$

**Definition 32.3.7 (generic stochastic policy).** Harnessing the universal representation power of neural networks, we can further parameterize a stochastic policy via Figure 32.3.1(b). In the network, we feed a state  $s$  and a Gaussian noise vector  $\epsilon$  into the network. We can view the output of the network as a random variable transformed from  $z$ , conditioning on the information  $s$ .



**Figure 32.3.1:** Neural network parameterization for a Gaussian policy (a) and (b) a generic stochastic policy.

### 32.3.2 Advanced methods for policy gradient estimation

#### 32.3.2.1 Stochastic policy gradient with baseline

In practice, the gradient estimate from trajectories can have large variance. One way to remedy is to subtract the cumulative reward statistics by a baseline function, which is a function or depending on the trajectories. To understand how it works, let's first look at the variance of policy gradient when we subtract a baseline. Let  $b$  be the baseline, we have the modified gradient given by

$$\nabla_{\theta} J(\theta) E_{\tau \sim p_{\theta}(\tau)} \nabla_{\theta} \log \pi_{\theta}(\tau) [R(\tau) - b].$$

It is critical to note that subtracting the baseline function will not introduce bias into the policy gradient.

**Lemma 32.3.3 (gradient of baseline function).** *Let  $b(s)$  be a baseline differential function that it is not a function of the trajectory  $\tau$ . Let  $\pi(a|s, \theta)$  be a  $\theta$  parameterized stochastic policy. We have*

$$\begin{aligned}\nabla_{\theta} J(\theta) &= E_{\tau \sim p_{\theta}(\tau)} [\nabla_{\theta} \log \pi(a|s, \theta) (R(\tau) - b)] \\ &= E_{\tau \sim p_{\theta}(\tau)} [\nabla_{\theta} \log \pi(a|s, \theta) R(\tau)]\end{aligned}$$

*Proof.* Use [Lemma 32.3.1](#). Note that

$$E_{\tau \sim p_{\theta}(\tau)} [\nabla_{\theta} \log \pi(a|s, \theta) b] = b \cdot E_{\tau \sim p_{\theta}(\tau)} [\nabla_{\theta} \log \pi(a|s, \theta)] = b \cdot 0 = 0.$$

□

To see how the baseline can modify the variance of the policy gradient estimate, we have

$$\begin{aligned}\text{Var}[\nabla_{\theta} J(\theta)] &= \text{Var}_{\tau \sim p_{\theta}(\tau)} [\nabla_{\theta} \log \pi_{\theta}(\tau) [R(\tau) - b]] \\ &= E_{\tau \sim p_{\theta}(\tau)} [(g(\tau) [R(\tau) - b])^2] - [E_{\tau \sim p_{\theta}(\tau)} [(g(\tau) [R(\tau) - b])] ]^2 \\ &= E_{\tau \sim p_{\theta}(\tau)} [(g(\tau) [R(\tau) - b])^2] - [E_{\tau \sim p_{\theta}(\tau)} [(g(\tau) R(\tau))] ]^2\end{aligned}$$

Clearly, the variance will be minimize if we choose  $b$  properly such that  $E_{\tau \sim p_{\theta}(\tau)} [(g(\tau) [R(\tau) - b])^2]$  is minimized (which is just a weighted least square optimization problem). With straight forward calculus, we can see the minimizer  $b^*$  is given by

$$b^* = \frac{E_{\tau \sim p_{\theta}(\tau)} [g(\tau) R(\tau)]}{E_{\tau \sim p_{\theta}(\tau)} [g(\tau)]}.$$

In practice, we simply choose  $b = E_{\tau \sim p_{\theta}(\tau)} [R(\tau)]$  as the baseline function.

In the REINFORCE type of batch learning algorithms, we can estimate gradient with baseline by

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \log \pi_{\theta}(\tau_i) [R(\tau_i) - b], b = \frac{1}{N} \sum_{i=1}^N R(\tau_i).$$

where  $b$  can be view as an estimate of  $E_{\tau \sim p_{\theta}(\tau)} [R(\tau)]$  and thus is a function of  $\theta$  instead of  $\tau$ .

In the Actor-Critic type of algorithms, one common choice of baseline is a state-value function estimate  $\hat{V}^{\pi}(s)$ , which can be approximated by Critic. Then the policy gradient becomes

$$\nabla_{\theta} J(\theta) = E_{s \sim \rho(s), a \in \pi_{\theta}(s)} [(Q^{\pi}(s, a) - V^{\pi}(s)) \nabla_{\theta} \ln \pi(a|s, \theta)]$$

Notably, we usually define an **Advantage function** associated with policy  $\pi$  by

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s),$$

where  $A^\pi(s, a)$  characterize the advantage of choosing  $a$  at state  $s$  over choose  $\pi(a|s)$  at state  $s$ . That is the  $A^\pi(s, a)$  provides the direction to update policy  $\pi$ .

An Actor-Critic type of algorithm with the advantage function is showed in [algorithm 81](#), where  $\hat{V}(s; w)$  will also be updated based on least-squared methods.

**Remark 32.3.4 (more intuitive interpretation).** To understand why subtracting mean values (i.e., state value function as the baseline) can help learning more effectively, we look at the policy gradient step that aims to increase probability of actions that receive positive action and reduce probability of actions that receive negative reward. For control tasks that involves positive rewards and negative rewards (as a form of penalty), the gradient step makes sense. However, if all rewards received are positive, but more larger than the rest, then the gradient step will promote both actions, although some at a smaller extent. By subtracting the mean, the gradient step will explicitly make actions receiving larger reward more frequently and action receiving smaller rewards less frequently.

**Lemma 32.3.4 (log-derivative trick).** Let  $x$  be a random variable following a  $\theta$  parameterized distribution given by  $p_\theta$ . We have

- $$\nabla_\theta E[f(x)] = E[f(x) \nabla_\theta \log p_\theta(x)].$$
- $$E[\nabla_\theta \log p_\theta(x)] = 0.$$

*Proof.* (1)

$$\begin{aligned} \nabla_\theta E[f(x)] &= \nabla_\theta \int p_\theta(x) f(x) dx \\ &= \int \frac{p_\theta(x)}{p_\theta(x)} \nabla_\theta p_\theta(x) f(x) dx \\ &= \int p_\theta(x) \nabla_\theta \log p_\theta(x) f(x) dx \\ &= E[f(x) \nabla_\theta \log p_\theta(x)] \end{aligned}$$

(2) Note that  $f(x) = 1$ . □

One common choice of baseline is a state-value function estimate  $\hat{V}(s)$ . Then the policy gradient becomes

$$\nabla_\theta J(\theta) = E_{s \sim \rho(s), a \in \pi_\theta(s)} [(Q^\pi(s, a) - V^\pi(s)) \nabla_\theta \ln \pi(a|s, \theta)]$$

Notably, we usually define an **Advantage function** associated with policy  $\pi$  by

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s),$$

where  $A^\pi(s, a)$  characterize the advantage of choosing  $a$  at state  $s$  over choose  $\pi(a|s)$  at state  $s$ . That is the  $A^\pi(s, a)$  provides the direction to update policy  $\pi$ .

An Actor-Critic type of algorithm with the advantage function is showed in [algorithm 81](#), where  $\hat{V}(s; w)$  will also be updated based on least-squared methods.

---

**Algorithm 81:** Policy-gradient method with a value function baseline

---

**Input:** A policy differential function parameterized  $\pi(a|s, \theta)$ , a differentiable state-value parameterization  $\hat{V}(s, w)$ , learning rate  $\alpha_1, \alpha_2 > 0$ .  
**Output:** Improved policy  $\pi(a|s, \theta^*)$  parameterized by  $\theta^*$

```

1 Initialize policy parameter  $\theta \in \mathbb{R}^d$ 
2 repeat
3   Using current policy  $\pi(\cdot|\cdot, \theta)$  to generate an episode
    $(s_0, a_0, r_1), \dots, (s_{T-1}, a_{T-1}, r_T)$ .
4   for each step in the episode  $t \in \{1, \dots, T\}$  do
5     Calculate reward-to-go  $G_t = \sum_{n=t}^T \gamma^{n-t} r_n$ . Note that depending on if the
     task is infinite horizon process, we need to add  $\hat{V}(s_T)$  to  $G_t$  if  $s_T$  is not
     terminal.
6     Calculate advantage function  $\delta = G_t - \hat{V}(s_t, w)$ 
7     Perform gradient descent on policy  $\pi_\theta$ 
           
$$\theta = \theta + \alpha \gamma^t \delta \nabla_\theta \ln \pi(a_t|s_t, \theta).$$

8     Perform a gradient descent step to improve value function estimation
           
$$w = w + \alpha_2 (R_t - \hat{V}(s_t; w)) \delta \nabla_w (r_t - \hat{V}(s_t; w))$$

9   end
10 until sufficient iterations;
11 return  $V$ 
```

---

### 32.3.2.2 Generalized advantage estimation

In [subsubsection 32.3.2.1](#), we establish that we can obtain a reduced-variance but biased policy gradient via

$$\nabla_\theta J(\theta) = E_{s \sim \rho(s), a \in \pi_\theta(s)} [(Q^\pi(s, a) - V^\pi(s)) \nabla_\theta \ln \pi(a|s, \theta)].$$

It turns out we can reduce the variance and bias even more by improving the estimation accuracy of  $A$  via  $n$  step estimation using a method similar to TD( $n$ ) method [[subsection 32.2.5](#)] and by taking averages[5].

We first have the  $n$  step estimates for advantage function, given by

$$\begin{aligned}
 \hat{A}_t^{(1)} &\triangleq \delta_t^V = -V(s_t) + r_t + \gamma V(s_{t+1}) \\
 \hat{A}_t^{(2)} &\triangleq \delta_t^V + \gamma \delta_{t+1}^V = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \\
 \hat{A}_t^{(3)} &\triangleq \delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+2}^V = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+3}) \\
 \hat{A}_t^{(k)} &\triangleq \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma r_{t+1} + \dots + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k}) \\
 \hat{A}_t^{(\infty)} &\triangleq \sum_{l=0}^{\infty} \gamma^l \delta_{t+l}^V = -V(s_t) + \sum_{l=0}^{\infty} \gamma^l r_{t+l}
 \end{aligned}$$

Our way to reduce variance is by taking averages. ... proposed generalized advantage estimate to be the exponential average of  $n$  step advantage estimates, given by

$$\begin{aligned}
 \hat{A}_t^{GAE} &= (1 - \lambda) \left( \hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \dots \right) \\
 &= (1 - \lambda) \left( \delta_t^V + \lambda \left( \delta_t^V + \gamma \delta_{t+1}^V \right) + \lambda^2 \left( \delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+2}^V \right) + \dots \right) \\
 &= (1 - \lambda) \left( \delta_t^V + \lambda \left( \delta_t^V + \gamma \delta_{t+1}^V \right) + \lambda^2 \left( \delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+2}^V \right) + \dots \right) \\
 &= (1 - \lambda) \left( \delta_t^V \left( 1 + \lambda + \lambda^2 + \dots \right) + \gamma \delta_{t+1}^V \left( \lambda + \lambda^2 + \lambda^3 + \dots \right) \right. \\
 &\quad \left. + \gamma^2 \delta_{t+2}^V \left( \lambda^2 + \lambda^3 + \lambda^4 + \dots \right) + \dots \right) \\
 &= (1 - \lambda) \left( \delta_t^V \left( \frac{1}{1 - \lambda} \right) + \gamma \delta_{t+1}^V \left( \frac{\lambda}{1 - \lambda} \right) + \gamma^2 \delta_{t+2}^V \left( \frac{\lambda^2}{1 - \lambda} \right) + \dots \right) \\
 &= \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l}^V
 \end{aligned}$$

### 32.3.2.3 Summary of stochastic gradient descent forms

The policy gradient can be written by

$$g = E \left[ \sum_{t=0}^{\infty} \Psi_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right],$$

where  $\Psi_t$  can take following forms [5]:

- total reward of the trajectory

$$\sum_{t=0}^{\infty} \gamma^t r_t$$



- total reward following action  $a_t$

$$\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'}.$$

- total reward following action  $a_t$  with baseline

$$\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} - b(s_t).$$

- $Q$  function

$$Q^{\pi}(s_t, a_t).$$

- Advantage function

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t).$$

- TD residual

$$r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t).$$

### 32.3.3 Deterministic policy gradient

Suppose now the control policy  $\pi$  is a  $\theta$  parameterized deterministic policy given by  $\mu_{\theta}$ <sup>1</sup>. And we further assume the stationary distribution of the state  $s$  under such policy is given by  $\rho^{\mu}(s)$ . The objective function we are trying to maximize now becomes

$$J(\mu_{\theta}) = \int_S \rho^{\mu}(s) Q(s, \mu_{\theta}(s)) ds.$$

The gradient of this objective function is given by the following theorem.

**Theorem 32.3.3 (deterministic policy gradient theorem).** [6] *If  $\nabla_{\theta} \mu_{\theta}(s)$  and  $\nabla_a Q^{\mu}(s, a)$  exist, then*

$$\begin{aligned} \nabla_{\theta} J(\theta) &= \int_S \rho^{\mu}(s) \nabla_a Q^{\mu}(s, a) \nabla_{\theta} \mu_{\theta}(s) \Big|_{a=\mu_{\theta}(s)} ds \\ &= E_{s \sim \rho^{\mu}} [\nabla_a Q^{\mu}(s, a) \nabla_{\theta} \mu_{\theta}(s) \Big|_{a=\mu_{\theta}(s)}] \end{aligned}$$

**Remark 32.3.5 (interpretation).**

<sup>1</sup> In the stochastic policy gradient setting, the action function is a parameterized probability distribution over the action space by  $\pi_{\theta}(a|s)$ .

- As a comparison, the gradient of the objective function in stochastic policy setting is given by

$$\begin{aligned}\nabla_{\theta} J(\pi_{\theta}) &= \int_{\mathcal{S}} \rho^{\pi}(s) \int_{\mathcal{A}} \nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s, a) da ds \\ &= E_{s \sim \rho^{\pi}, a \sim \pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi}(s, a)]\end{aligned}$$

which integrate over both the action and the state space. In deterministic policy gradient, only state space is being integrated over.

- Computing the stochastic policy gradient may require more samples, especially if the action space has many dimensions. Therefore, using deterministic policy can potentially be faster.

Because the policy is deterministic, another attractive feature in the gradient is that we will not need to adjust the importance weight as we do in stochastic policy gradient. More formally,

$$\begin{aligned}\nabla_{\theta} J_{\beta}(\mu_{\theta}) &\approx \int_{\mathcal{S}} \rho^{\beta}(s) \nabla_{\theta} \mu_{\theta}(a|s) Q^{\mu}(s, a) ds \\ &= E_{s \sim \rho^{\beta}} [\nabla_{\theta} \mu_{\theta}(s) \nabla_a Q^{\mu}(s, a)|_{a=\mu_{\theta}(s)}]\end{aligned}$$

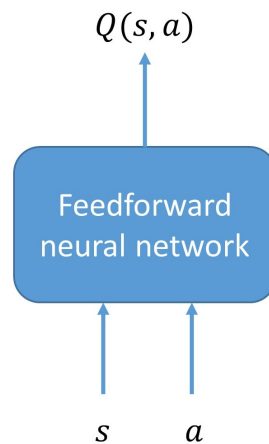
where  $\beta$ , the policy used to sample the data, is different from current policy  $\mu_{\theta}$  we optimize over.

## 32.4 Algorithms zoo

### 32.4.1 Neural Fitted Q Iteration (NFQ)

Value functions are the cornerstones of reinforcement learning. In small problems, value functions are represented by a table. When tackling real-world problems involving large action and state space dimensionality, one faces the problem of an appropriate, efficient method to represent value functions. In NFQ[7], artificial neural networks, thanks to their universal approximation capability, are used to represent value functions [Figure 32.4.1]. Another key idea underlying NFQ is updating the neural value function off-line considering an entire set of transition experiences. Updating network on-line with non-linear value function approximators can lead to instability and divergence in the training. Updating offline allows the application of advanced supervised learning methods, that enables faster and reliable convergence than online gradient descent methods.

The full algorithm is showed in [algorithm 82](#).



**Figure 32.4.1:** A typical feed-forward neural network used in NFQ to approximate the  $Q$  function.

**Algorithm 82:** Neural Fitted Q Iteration (NFQ)

---

**Input:** A set of transition sample  $D = \{(s, a, s', r)\}$ , step to learn  $N$

- 1 Initialize network  $Q_0$ .
- 2 **for**  $n = 1, \dots, N$  **do**
- 3     Generate pattern set  $P = (input, output)$ , where
 
$$input = (s, u)$$

$$target = r + \gamma \max_b Q_n(s, b)$$
- 4     Stochastic gradient descent training  $Q$  network and get  $Q_{n+1}$
- 5 **end**

**Output:**  $Q$  value function

---

## 32.4.2 Canonical deep Q learning

In the DQN framework[8], we use deep neural network approximate the optimal action-value function

$$Q^*(s, a) = \max_{\pi} E \left[ r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t = s, a_t = a, \pi \right],$$

which is maximizing accumulated discounted rewards in the process over all possible policy  $\pi$  after achieving state  $s$  and taking an action  $a$ , where  $r_t$  is the one step reward function,  $\gamma$  is the discount factor. With the optimal  $Q^*$ , the optimal action at state  $s$  is given by  $a^* = \arg \max_a Q^*(s, a)$ .

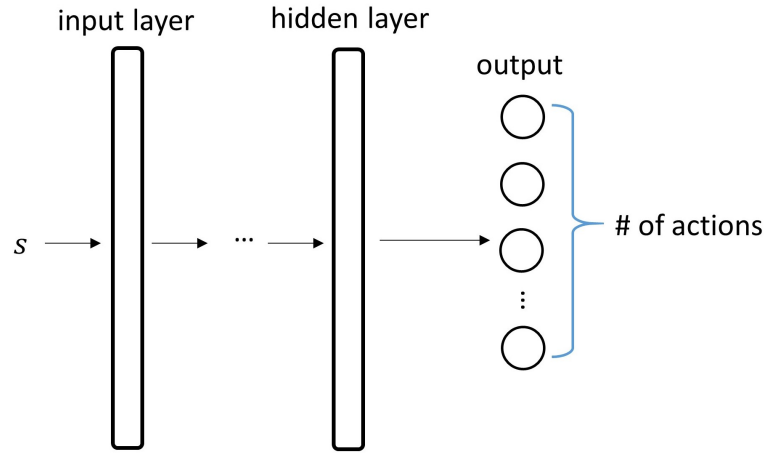
Alternatively, using  $\pi^*$  to denote the optimal policy  $\pi^*(s) = \arg \max_a Q^*(s, a)$ , we can re-write optimal  $Q$  function by

$$Q^*(s, a) = E \left[ r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t = s, a_t = a, \pi^* \right],$$

which is further connected to optimal state-value function by

$$V^*(s) = E \left[ r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t = s, \pi^* \right] = \max_a Q^*(s, a).$$

A typical neural network for deep Q learning is showed in [Figure 32.4.2](#), which takes a state or an observation, denoted by  $s$ , as the input, and outputs multiple values corresponding  $Q(s, a)$ . The number of outputs is the same number as the number of actions.



**Figure 32.4.2:** The network architecture for canonical deep Q learning. The network takes a state or an observation, denoted by  $s$ , as the input, and outputs multiple values corresponding  $Q(s, a)$ . The number of outputs

The core of the training procedure is gradient decent. The authors in [8] proposed multiple improvements for stability and convergence:

**Remark 32.4.1** (procedures to improve the stability of training).

- Before deep Q learning, target value used to train Q network was

$$r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta),$$

which leads to unstable training process. In deep Q learning, the target use is generated by a target network  $Q'$ , whose parameters are slowly updated, given by,

$$r_j + \gamma \max_{a'} Q'(s_{j+1}, a'; \theta^-).$$

- Before deep Q learning, directly using consecutively generated samples also cause instability in the training process. This is because these samples are correlated and correlated samples applied to stochastic gradient method can lead to divergence. By using a replay buffer to randomly sample experience to learn, we decorrelate the sample and reuse samples.
- A helpful trick is to clip the error term

$$y - Q(s, a; \theta)$$

between  $[-1, 1]$  or other more proper intervals. Such clipping corresponding to a more robust error function, like Huber loss function.

The final training algorithm is showed in [algorithm 83](#).

---

**Algorithm 83:** Deep Q-learning with experience replay
 

---

```

1 Initialize replay memory  $\mathcal{B}$  to capacity  $N$ .
2 Initialize action-value function  $Q$  with random weights  $\theta$ .
3 Initialize target action-value function  $Q'$  with weights  $\theta^- = \theta$ .
4 for  $episode = 1, M$  do
5   Initialize sequence  $s_1 = \{x_1\}$ .
6   for  $t = 1, T$  do
7     With probability  $\epsilon$  select a random action  $a_t$ , otherwise select
        $a_t = \arg \max_a Q(s_t, a; \theta)$ .
8     Execute action  $a_t$  in emulator and observe reward  $r_t$  and state  $x_{t+1}$ .
9     Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $s_{t+1}$ .
10    Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $\mathcal{B}$ .
11    Sample random minibatch of transitions  $(s_j, a_j, r_j, s_{j+1})$  from  $\mathcal{B}$ .
12    Set
        
$$y_j = \begin{cases} r_j, & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q'(s_{j+1}, a'; \theta^-), & \text{otherwise} \end{cases}$$

        Every  $C$  steps reset  $Q' = Q$ .
13   end
14 end

```

---

### 32.4.3 DQN variants

#### 32.4.3.1 Overview

Although Deep Q learning algorithm has achieved remarkable performance in various challenging benchmark tasks, there has been extensive discussion on its possible weaknesses.

For example,

- The Q value is calculated using bootstrapped approach. Is such method accurate or biased?
- Is random sampling of experience is best option? Some experiences might be more important in contributing to the learning processes.
- Q network is directly learning state-action function. Will it be better to learn state value and action value separately?
- Is there any better approach to explore the state space beside  $\epsilon$  greedy approach?

- The network structure is not suitable to handle tasks with long memory state dynamics. How to equip Q network with RNN structure?
- Deep Q learning usually takes a long time to converge.

Since the invention of the Deep Q learning algorithm, there have been various approaches to improve the canonical Deep Q learning algorithm.

### 32.4.3.2 Double Q learning

In the canonical deep Q learning, the target of the Q network is given by

$$y_t^Q = r_{t+1} + \gamma \max_{a'} Q'(\phi_{j+1}, a'; \theta^-),$$

the maximization operator can propagate noise and overestimate Q values. In double Q-learning ([9]), we use two networks to generate the target value

$$y_t^{DoubleQ} = r_{t+1} + \gamma Q'(s_{t+1}, \arg \max_a Q(s_{t+1}, a; \theta_t); \theta'_t)$$

where  $Q'$  is used to generate target and  $Q$  is used to generate action. Because  $Q$  and  $Q'$  are not fully correlated, such method reduce the tendency of overestimation.

### 32.4.3.3 Dueling network

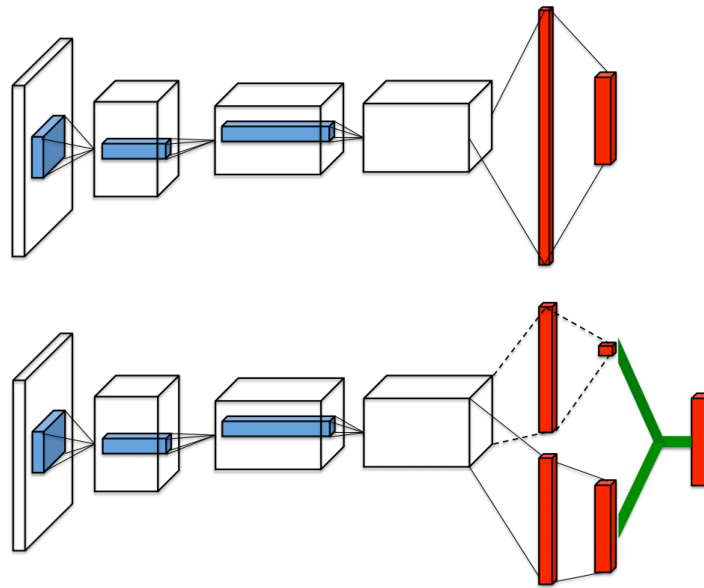
Given the Q function and the state value function  $V$  associated with a policy  $\pi$ , we can define the **advantage function** by

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s).$$

The advantage function shows the advantage of selecting an action with respect to the selection given by policy  $\pi$ .

In DQN dueling architecture [10], as showed in [Figure 32.4.3](#), we explicitly separates the representation of state values and state-dependent action advantages via two separate subnetworks and synthesize the final Q function via  $Q^\pi = V^\pi + A^\pi$ .

The main advantage of explicitly separating two estimators is that the network can be focused on learning state functions without having to learn the state-action value Q function first. After all, the agent are learning to choose better action leading to states with higher values.



**Figure 32.4.3:** A typical single stream Q-network (top) in deep Q learning and a dueling Q-network (bottom). The dueling network has two streams to separately estimate (scalar) state-value and the advantage function for each action; the green output module synthesizes the Q value from two streams. Both networks output Q-values for each action. [10].

#### 32.4.3.4 Deep Recurrent Q network (DRQN)

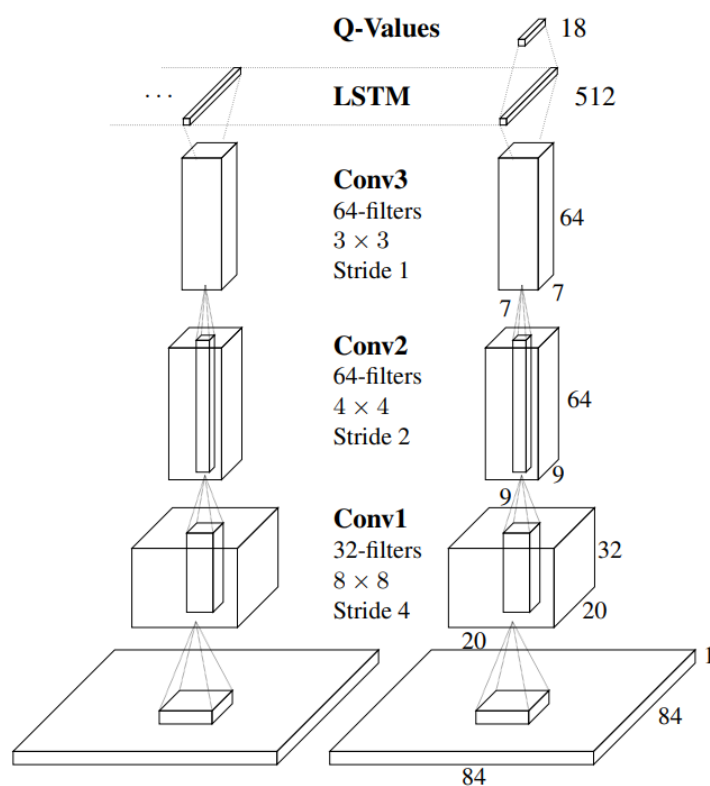
The main idea in DRQN[11] is to incorporate recurrent modules into learn representations from both current and past observations, as a attempt for Markov decision process with partial observability. Many sequential decision problems requires high-dimensional observation from the past, yet classical DQN mostly optimize decision-making using current and immediate past observation. An example architecture will something like Figure 32.4.4, where sequence of past observation are fed into a neural network with recurrent layers to learn the temporal relationship between observation or paint the full pictures from relatively local observations.

Notably, the usage of recurrent layers requires the input to be sequences, we usually sample episodes from replay memory and update the network from the episode start or a random start point of the episode and proceed towards the conclusion of the episode.

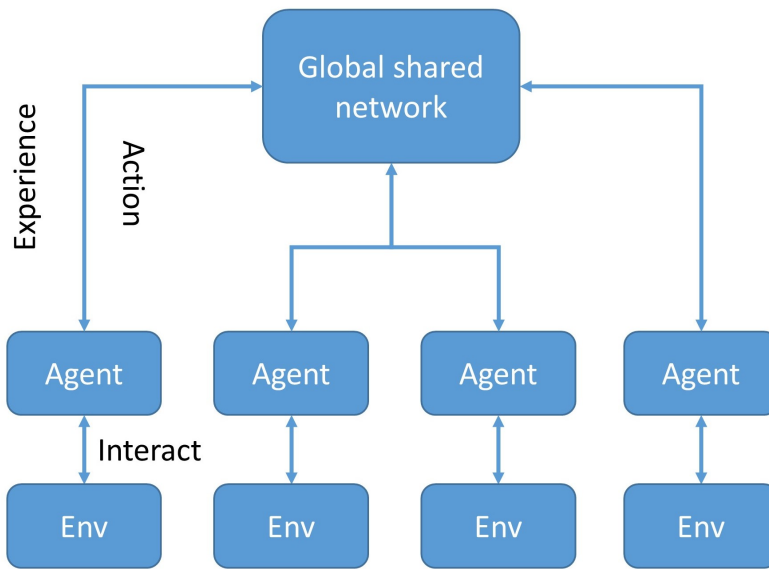
#### 32.4.3.5 Asynchronous Methods

Deep Q learning relies on experience replay to reduce the correlation between samples and stabilize the training process. An alternative paradigm for deep reinforcement





**Figure 32.4.4:** In a DQRN, recurrent layers are usually placed on the last layer before output. Unlike canonical DQN, we need to feed sequence of observation into the network and finally output Q values. Above scheme only unfolds to two steps.[\[11\]](#)



**Figure 32.4.5:** An example architecture that implements the asynchronous reinforcement learning paradigm. Multiple agents interact with multiple instances of environments in parallel. Agents collect experiences to train a globally shared network that learns control policies.

learning to asynchronously use multiple agents to in parallel interact with on multiple instances of the environment[12], as showed in Figure 32.4.5. This parallelism offer several critical benefits:

- Since agents are running different exploration policies at different state sub-regions, experiences are less likely to be correlated. Experiences collected online can be used for on-policy learning.
- For some DQN learning where environment simulation is prohibitable, it speeds up training processes by generating experiences in parallel.

#### 32.4.4 Universal value function approximator

Many relatively simple, primitive sequential decision-making tasks can be abstractly viewed as taking actions to move from arbitrary starting points in a high-dimensional state to one prescribed target point. More complex, sophisticated tasks can be solved by decomposing them to a series of simpler tasks. For example, a long distance robot navigation task can be decomposed into a series of short-ranged navigation tasks to a set of goals leading to the ultimate goal. To enable such decomposition, it is necessary to learn

**Algorithm 84:** Asynchronous Deep Q-Learning for each thread

---

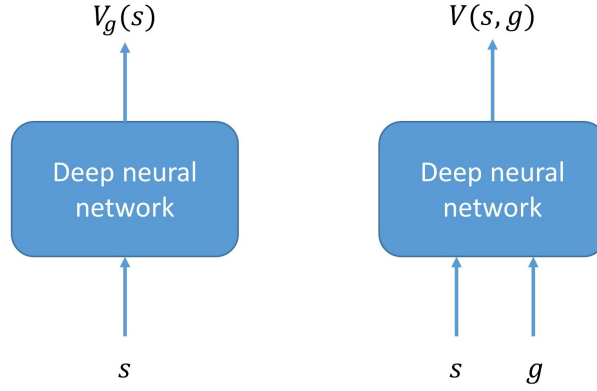
**Input:** minibatch  $k$ , step-size  $\eta$ , replay period  $K$  and size  $N$ , exponents  $\alpha$  and  $\beta$ , budget  $T$

- 1 For each thread, initialize thresh step counter  $t = 0$
- 2 Initialize target network parameters  $\theta' = \theta$
- 3 Get initial state  $s$ .
- 4 **repeat**
- 5     With probability  $\epsilon$  select a random action  $a_t$ , otherwise select  
 $a_t = \arg \max_a Q(s_t, a; \theta)$ .
- 6     Execute action  $a_t$  and observe reward  $r_t$  and state  $s_{t+1}$ .
- 7     Set  $s_{t+1} = s_t, a_t, s_{t+1}$ .
- 8     Set
 
$$y_j = \begin{cases} r_j, & \text{if episode terminates at step } j + 1 \\ r_j + \gamma \arg \max_a Q(s_{j+1}, a^*; \theta'), & \text{otherwise} \end{cases}$$
- 9     Every  $C$  steps reset  $Q' = Q$ .
- 10 **until**  $episode = 1, M$ ;

---

goal-parameterized policies. In the single goal setting, control policies are encoded by state value function  $V$  or the  $Q$  function. Similarly, we can encode goal-parameterized policies by goal-parameterized value function or  $Q$  function, known as universal value function. In canonical deep Q learning, we use a neural network to learn the control policy and the value function for a single goal [Figure 32.4.6]; in universal value function approximator, we use goals as the inputs to the neural network, which the aim to learn control policies and value function parameterized by goals, and more importantly generalizing to unseen goals.

From a broader perspective, universal value function approximator is closer to general intelligence, in which the intelligent agent can accomplish tasks of similar goals without relearning.



**Figure 32.4.6:** A comparison between a typical deep Q learning network (left) and a typical universal value function approximator network (right).

A modified DQN algorithm with universal value approximator is given by [algorithm 85](#).

---

**Algorithm 85:** Deep Q-learning with universal value function approximator

---

```

1 Initialize replay memory  $\mathcal{B}$  to capacity  $N$ .
2 Initialize action-value function  $Q$  with random weights  $\theta$ .
3 Initialize target action-value function  $Q'$  with weights  $\theta^- = \theta$ .
4 for  $episode = 1, M$  do
5   Initialize state  $s_1$ .
6   Sample a goal from  $G$ . for  $t = 1, T$  do
7     With probability  $\epsilon$  select a random action  $a_t$ , otherwise select
        $a_t = \arg \max_a Q(s_t, g, a; \theta)$ .
8     Execute action  $a_t$  in the simulator and observe reward  $r_t$  and state  $s_{t+1}$ .
9     Store transition  $(s_t, a_t, r_t, s_{t+1}, g)$  in  $\mathcal{B}$ .
10    Sample random minibatch of transitions  $(s_j, a_j, r_j, s_{j+1}, g)$  from  $\mathcal{B}$ .
11    Set
      
$$y_j = \begin{cases} r_j, & \text{if episode terminates at step } j + 1 \\ r_j + \gamma \max_{a'} Q'(s_{j+1}, a', g; \theta^-), & \text{otherwise} \end{cases}$$

      Every  $C$  steps reset  $Q' = Q$ .
12  end
13 end

```

---

## 32.4.5 Deep deterministic policy gradient (DDPG) algorithm

Deep Deterministic Policy Gradient (DDPG) is one of the most important result the deterministic policy gradient theorem. DDPG employs an actor-critic network architecture [Figure 32.4.7], but there are couple of differences between DDPG and an ordinary actor-critic algorithm

- Ordinary actor-critic algorithms will use stochastic policy, where DDPG uses deterministic policy.
- The critic in an ordinary actor-critic algorithm learns the value function, while the critic in DDPG learns the Q function.

DDPG further combines characteristics of DQN such as off-line learning and experience replay. Because it applies to decision-making problems with continuous action space, it greatly expand the application regime of deep Q learning. By contrast, DQN applies only to decision-making problems with discrete action space or small continuous action space that can be properly discretized.

In DDPG, the Q network update is based on Bellman principle and is similar to that in DQN. The actor network learns a deterministic control policy function  $\mu$ , based on gradients given by

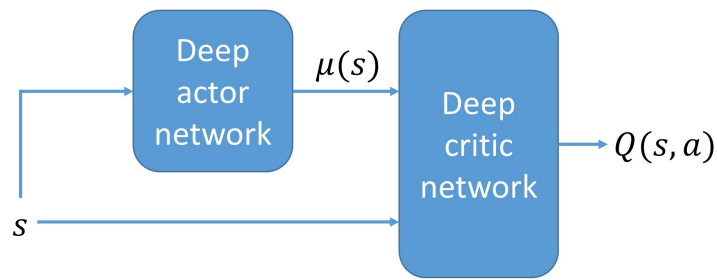
$$\begin{aligned}\nabla_{\theta} J(\theta) &= \int_S \rho^{\mu}(s) \nabla_a Q^{\mu}(s, a) \nabla_{\theta} \mu_{\theta}(s) \Big|_{a=\mu_{\theta}(s)} ds \\ &= E_{s \sim \rho^{\mu}} [\nabla_a Q^{\mu}(s, a) \nabla_{\theta} \mu_{\theta}(s) \Big|_{a=\mu_{\theta}(s)}]\end{aligned}$$

Similar to DQN, DDPG employs experience replay to increase sample efficiency and target networks to stabilize the training process. Additionally, DDPG employs an OU process for exploration, analogous to the  $\epsilon$  greedy rule in DQN. One critical difference between DDPG and DQN is the how to improve policy based on estimated Q.

**Remark 32.4.2 (policy improvement in DDPG).**

- In DQN and many other Q learning algorithms, policy improvement is achieved by taken actions via  $\pi(s) = \arg \max_a Q(s, a)$ . But the maximizing operation is difficult in continuously action space. Instead, DDPG relies on deterministic policy gradient and take gradient steps to improve parameterized policies.

The complete algorithm is showed in [algorithm 86](#).



**Figure 32.4.7:** A typical deep neural network architecture for DDPG reinforcement learning. The actor network outputs control policy; the critic network outputs estimate of Q function. Through back-propagation, the critic network improves estimation accuracy and the actor network improves policy.

**Algorithm 86:** Deep deterministic policy gradient algorithm (DDPG)

**Input:** A randomly initialized critic network  $Q(s, a|\theta^Q)$  and an actor network  $\mu(s|\theta^\mu)$  with weights  $\theta^Q, \theta^\mu$ .

**Output:** Optimal critic and actor networks

```

1 Initialize target networks  $Q'$  and  $\mu'$  with weights  $\theta^{Q'} = \theta^Q$  and  $\theta^{\mu'} = \theta^\mu$ .
2 Initialize replay buffer  $\mathcal{B}$ . Initialize policy parameter  $\theta \in \mathbb{R}^d, w \in \mathbb{R}^m$ .
3 repeat
4   Initialize a random process  $\mathcal{N}$  for action exploration.
5   Initialize initial state  $s_1$ .
6   for  $t = 1, T$  do
7     Select action  $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$  according to current actor network and
      exploration noise.
8     Execute action  $a_t$  and observe reward  $r_t$  and new state  $s_{t+1}$ .
9     Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $\mathcal{B}$ .
10    Sample a random minibatch of  $N$  transitions  $(s_i, a_i, r_i, s_{i+1})$  from  $\mathcal{B}$ .
11    Set  $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$ .
12    Update the critic network by minimizing the loss


$$L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2.$$
\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_{i=1}^N \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}.\theta^{Q'} = \tau \theta^Q + (1 - \tau) \theta^{Q'}

$$\theta^{\mu'} = \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$$

14  end
15 until sufficient iterations;
```

## 32.4.6 Twin-delayed deep deterministic policy gradient (TD3)

Q-learning algorithms have been found to overestimate of the value function[9]. This overestimation can propagate through the training iterations and lead to inferior policies.

In DQN learning for discrete control, this issue is remedied by using Double Q-learning [9].

Twin Delayed Deep Deterministic policy gradient (short for TD3) [13] employs a couple of techniques to address the value function overestimation in DDPG.

- Clipped Double Q-learning. We use the minimum estimation among two independent network estimates as the target. This can favor underestimation bias which is hard to propagate through training:

$$y = r + \gamma \min_{i=1,2} Q_{w_i} \left( s', \mu_{\theta_1}(s') \right)$$

- Delayed update of Policy Networks. By updating policy network slower than the value network, we can possibly reduce the detrimental effect of value overestimation when the policy is poor. Value overestimation of a poor policy can lead to the learning of even poorer policies.
- Target Policy Smoothing. Note that deterministic policies can overfit to spiking peaks in the value function. If the actor network exploits too much the spikes in the value network, it can cause divergence or inferior policies. TD3 add noise to the selected action to make it harder for exploitation of spikes in learned value function. This approach mimics the idea of SARSA update and enforces the intuition that similar actions should have similar values.



The final algorithm is given by [algorithm 87](#).

---

**Algorithm 87:** Twin-delayed deep deterministic policy gradient (TD3)

---

**Input:** A randomly initialized two critic networks  $Q(s, a | \theta_i^Q)$  and one actor network  $\mu(s | \theta_i^\mu)$  with weights  $\theta_i^Q, \theta_i^\mu, i = 1, 2$

**Output:** Optimal critic and actor networks

- 1 Initialize two target Q networks  $Q'$  and a one target actor network  $\mu'$  with weights  $\theta_i^{Q'} = \theta_i^Q$  and  $\theta_i^{\mu'} = \theta_i^\mu, i = 1, 2$ .
  - 2 Initialize replay buffer  $\mathcal{B}$ . Initialize policy parameter  $\theta \in \mathbb{R}^d, w \in \mathbb{R}^m$
  - 3 **repeat**
  - 4     Initialize a random process  $\mathcal{N}$  for action exploration.
  - 5     Initialize initial state  $s_1$ .
  - 6     **for**  $t = 1, T$  **do**
  - 7         Select action  $a_t = \mu(s_t | \theta^\mu) + \mathcal{N}_t$  according to current actor network and exploration noise.
  - 8         Execute action  $a_t$  and observe reward  $r_t$  and new state  $s_{t+1}$ .
  - 9         Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $\mathcal{B}$ .
  - 10         Sample a random minibatch of  $N$  transitions  $(s_i, a_i, r_i, s_{i+1})$  from  $\mathcal{B}$ .
  - 11         Compute actions using target actor network for state  $s_{i+1}$ , add noise and probably clip it to get  $a'$ .
  - 12         Set  $y_i = r_i + \gamma \min_{i=1,2} Q'(s_{i+1}, a' | \theta^{Q'})$ .
  - 13         Update the critic network by minimizing the loss

$$L = \frac{1}{N} \sum_i^N (y_i - Q(s_i, a_i | \theta^Q))^2.$$
  - 14         Update the actor policy network **less frequently** using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_{i=1}^N \nabla_a Q(s, a | \theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s | \theta^\mu)|_{s_i}.$$
  - 15         Update the target network

$$\theta_i^{Q'} = \tau \theta_i^Q + (1 - \tau) \theta_i^{Q'}, i = 1, 2$$

$$\theta_i^{\mu'} = \tau \theta_i^\mu + (1 - \tau) \theta_i^{\mu'}$$
  - 16     **end**
  - 17 **until** *sufficient iterations*;
-

## 32.4.7 Trust Region Policy Optimization (TRPO)

## 32.4.7.1 TRPO

TRPO [14] is an improved policy gradient algorithm that aims to achieve stable and robust improvement on each gradient step. Its key idea is to introduce a KL divergence based constraint on the gradient step size at each iteration.

Suppose now we are optimize the policy parameter  $\theta$  based on its previous iterate  $\theta_k$ . The gradient of the objective function is the same as the gradient following objective function(also see [subsubsection 32.3.2.3](#)):

$$J(\theta) = E_{s,a \sim \pi_{\theta_k}} \left[ \frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a) \right].$$

In seeking the next iterate  $\theta_k$  that can improve  $J(\theta)$ , we use following optimization formulation:

$$\theta_{k+1} = \arg \min_{\theta} L(\theta_k, \theta), \quad s.t., \bar{D}_{KL}(\theta || \theta_k) \leq \delta,$$

where  $L(\theta_k, \theta)$  is given by

$$L(\theta_k, \theta) = E_{s,a \sim \pi_{\theta_k}} \left[ \frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a) \right],$$

and  $D_{KL}(\theta || \theta_k)$  is an **average** KL-divergence between the two policies averaging over all states, given by

$$\bar{D}_{KL}(\theta || \theta_k) = E_{s \sim \pi_{\theta_k}} \left[ D_{KL}(\pi_{\theta}(\cdot|s) || \pi_{\theta_k}(\cdot|s)) \right]$$

The theoretical TRPO update quite complicated to work with; instead, TRPO uses Taylor expand the objective and constraint to leading order around  $\theta_k$

$$\begin{aligned} L(\theta_k, \theta) &\approx g_k^T (\theta - \theta_k) \\ \bar{D}_{KL}(\theta || \theta_k) &\approx \frac{1}{2} (\theta - \theta_k)^T H_k (\theta - \theta_k) \end{aligned}$$

where  $g_k$  and  $H_k$  are the gradient and Hessian at  $\theta_k$ . which gives the following approximate optimization problem

$$\theta_{k+1} = \arg \max_{\theta} g_k^T (\theta - \theta_k), \quad s.t., \frac{1}{2} (\theta - \theta_k)^T H_k (\theta - \theta_k) \leq \delta.$$

This approximate optimization problem has an iterative solution <sup>2</sup>given by

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g_k^T H_k^{-1} g_k}} H_k^{-1} g_k.$$

A problem is that, due to the approximation errors introduced by the Taylor expansion, this may not satisfy the KL constraint, or actually improve the surrogate advantage. TRPO uses backtracking line search,

$$\theta_{k+1} = \theta_k + \alpha \sqrt{\frac{2\delta}{g_k^T H_k^{-1} g_k}} H_k^{-1} g_k$$

such that  $\alpha \in (0, 1)$  is chosen to ensure  $\pi_{\theta_{k+1}}$  to satisfy the KL constraint and produce a positive surrogate advantage.

---

<sup>2</sup> To see this, the constraint such be active since objective function is linear.

Finally, the gradient  $g_k$  can be obtained via automatic differentiation, the Hessian  $H_k$  via Fisher information matrix [subsubsection 32.4.7.2], and the  $H^{-1}g$  can be evaluated by coordinate descent without explicitly evaluating  $H^{-1}$ .

---

**Algorithm 88:** Trust Region Policy Optimization
 

---

**Input:** A randomly initialized policy parameter  $\theta_0$ , value function parameter  $\phi_0$ , KL-divergence threshold  $\delta$ .

**Output:** Optimal critic and actor networks

1 Initialize iteration count  $k = 0$ .

2 **repeat**

3     Collect a set of trajectories  $D = \{\tau_1, \dots\}$  by carrying out policy  $\pi_{\theta_k}$  and compute their rewards-to-go for each state.

4     Compute advantage estimate  $\hat{A}_t$  based on the current value function  $V_{\phi_k}$ .

5     Estimate policy gradient by

$$\hat{g}_k = \frac{1}{|D|} \sum_{\tau \in D_k} \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \Big|_{\theta_k} \hat{A}_t$$

Update policy by

$$\theta_{k+1} = \theta_k + \alpha \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g.$$

6     Estimate value function via least-squared minimization of

$$\phi_{k+1} = \arg \min_{\phi} \frac{1}{|D| T} \sum_{\tau \in D} \sum_{t=0}^T \left( V_{\phi}(s_t) - \hat{R}_t \right)^2$$

7 **until** sufficient iterations;

---

### 32.4.7.2 Evaluating Hessian of KL-divergence

Based on the following [Lemma 32.4.1](#), the Hessian of KL divergence is equivalent to Fisher matrix.

Note that given a  $\theta$  parameterized distribution  $p(x|\theta)$ , its Fisher information matrix is given by

$$F = E_{p(x|\theta)} \left[ \nabla \log p(x|\theta) \nabla \log p(x|\theta)^T \right].$$

With data samples  $\{x_1, x_2, \dots, x_N\}$ ,  $F$  can be evaluated empirically via

$$F = \frac{1}{N} \sum_{i=1}^N \nabla \log p(x_i|\theta) \nabla \log p(x_i|\theta)^T$$

**Lemma 32.4.1 (Fisher matrix and Hessian of KL divergence).** Consider two distributions  $p(x|\theta)$  and  $p(x|\theta')$  parameterized by  $\theta$  and  $\theta'$  vectors. The Fisher information for  $p(x|\theta)$  with respect to vector  $\theta$  is given by

$$F = E_{p(x|\theta)} \left[ \nabla \log p(x|\theta) \nabla \log p(x|\theta)^T \right]$$

Fisher information matrix  $F$  is the Hessian of KL divergence between two distributions, with respect to  $\theta'$ , evaluated at  $\theta' = \theta$ .

*Proof.* Based on the definition of KL divergence, we can write

$$KL[p(x|\theta) \| p(x|\theta')] = E_{p(x|\theta)} [\log p(x|\theta)] - E_{p(x|\theta)} [\log p(x|\theta')]$$

where the first term is entropy and the second term is cross-entropy.

$$\begin{aligned} \nabla_{\theta'} KL[p(x|\theta) \| p(x|\theta')] &= \nabla_{\theta'} E_{p(x|\theta)} [\log p(x|\theta)] - \nabla_{\theta'} \lim_{p(x|\theta)} [\log p(x|\theta')] \\ &= -E_{p(x|\theta)} [\nabla_{\theta'} \log p(x|\theta')] \\ &= - \int p(x|\theta) \nabla_{\theta'} \log p(x|\theta') dx \end{aligned}$$

The second derivative is

$$\nabla_{\theta'}^2 KL[p(x|\theta) \| p(x|\theta')] = - \int p(x|\theta) \nabla_{\theta'}^2 \log p(x|\theta') dx$$

The Hessian with respect to  $\theta'$  evaluated at  $\theta' = \theta$  is

$$H = - \int p(x|\theta) \nabla_{\theta}^2 \log p(x|\theta) dx = F$$

where we used the basic property of Fisher Information matrix [Theorem 13.2.1] that the Fisher information is the negative Hessian.  $\square$

## 32.4.8 Proximal Policy Optimization (PPO)

PPO simplifies the gradient step size regularization in TRPO by using a modified surrogate objective function. Instead of optimizing an KL related constraint in TRPO, a typical PPO formulation updates policies via

$$\theta_{k+1} = \arg \max_{\theta} E_{s,a \sim \pi_{\theta_k}} [L(s, a, \theta_k, \theta)]$$

where  $L$  is given by

$$L(s, a, \theta_k, \theta) = \min \left( \frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \text{clip} \left( \frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A^{\pi_{\theta_k}}(s, a) \right)$$

in which  $\epsilon$  is a (small) hyperparameter which roughly says how far away the new policy is allowed to go from the old.

Although the objective function seems complicated at the first glance, it aims for two purposes similar to TRPO: first, it helps maintain the new policy close to the old policy; second, the new policy is better the old policy.

The simplified form has more straight forward interpretation:

**Remark 32.4.3 (interpretation).**

- If the advantage  $A(s, a) > 0$  for  $(s, a)$ , then its contribution to the objective reduces to

$$L(s, a, \theta_k, \theta) = \min \left( \frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)}, (1 + \epsilon) \right) A^{\pi_{\theta_k}}(s, a)$$

Because  $A(s, a) > 0$ , increasing  $L$  will make  $a$  more likely, that is, it will make  $\pi_{\theta}(a|s)$  increase.

- If the advantage  $A(s, a) < 0$  for  $(s, a)$ , then its contribution to the objective reduces to

$$L(s, a, \theta_k, \theta) = \min \left( \frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)}, (1 - \epsilon) \right) A^{\pi_{\theta_k}}(s, a)$$

Because  $A(s, a) < 0$ , increasing  $L$  will make  $a$  less likely, that is, it will make  $\pi_{\theta}(a|s)$  decrease.

- The min operator limits how much the objective can increase. For example, if  $\pi_{\theta}(a|s) > (1 + \epsilon)\pi_{\theta_k}(a|s)$ , the min will put a ceiling of  $(1 + \epsilon)A^{\pi_{\theta_k}}(s, a)$ .

Combination the valuation function estimation similar in TRPO, we have an example PPO algorithm in [algorithm 89](#).

---

**Algorithm 89:** Proximal Policy Optimization
 

---

**Input:** A randomly initialized policy parameter  $\theta_0$ , value function parameter  $\phi_0$ , KL-divergence threshold  $\delta$ .

**Output:** Optimal critic and actor networks

1 Initialize iteration count  $k = 0$ .

2 **repeat**

3     Collect a set of trajectories  $\mathcal{D} = \{\tau_1, \dots\}$  by carrying out policy  $\pi_{\theta_k}$  and compute their rewards-to-go for each state.

4     Compute advantage estimate  $\hat{A}_t$  based on the current value function  $V_{\phi_k}$ .

5     Estimate policy gradient via

$$\hat{g}_k = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta} (a_t | s_t) \Big|_{\theta_k} \hat{A}_t$$

Update policy by stochastic gradient on

$$\theta_{k+1} = \arg \max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \min \left( \frac{\pi_{\theta} (a_t | s_t)}{\pi_{\theta_k} (a_t | s_t)} A^{\pi_{\theta_k}} (s_t, a_t), \text{clip} \left( \epsilon, A^{\pi_{\theta_k}} (s_t, a_t) \right) \right)$$

6     Estimate value function via least-squared minimization of

$$\phi_{k+1} = \arg \min_{\phi} \frac{1}{|\mathcal{D}| T} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^T \left( V_{\phi} (s_t) - \hat{R}_t \right)^2$$

Set  $k = k + 1$ .

7 **until** *sufficient iterations*;

---

### 32.4.9 Soft Actor-Critic(SAC)

#### 32.4.9.1 Entropy regulated reinforcement learning

SAC algorithm [15] aims to extend DDPG to stochastic policies. Stochastic policies can often explore the action and state space better than deterministic policies. As SAC works on stochastic policies, SAC uses entropy penalty to learn diverse, more exploration stochastic policies.

The objective function SAC optimizes is the expected process reward plus entropy penalty. By adjusting the penalty strength, we can control the trade-off of exploration-exploitation, which eventually determines the quality of learned policies. Similar to TD3, SAC also contains additional techniques to address value estimation issue. The stochasticity in the action selection also alleviate the exploitation of poorly learned value function spikes.

We first start with a set of definitions involving entropy penalties.

**Definition 32.4.1 (entropy regulated reinforcement learning).**

- The value function associated with a stochastic policy  $\pi(\cdot|s)$  is defined by

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

where  $H$  is the entropy of distribution  $\pi(\cdot|s)$  and  $\alpha > 0$  control the penalty magnitude.

- The value function associated with a stochastic policy  $\pi(\cdot|s)$  is defined by

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

Note that  $Q$  function does not have entropy penalty at  $t = 0$  when action  $a$  is known.

- The objective is seek a control policy  $\pi$  that maximize value function:

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

It is straight forward to see that  $V$  and  $Q$  are connected by

$$V^\pi(s) = E_{a \sim \pi} [Q^\pi(s, a)] + \alpha H(\pi(\cdot|s));$$

that is,  $V$  is the sum of action averaged  $Q$  and the entropy penalty.

To use TD method to learn  $Q$  or  $V$ , it is necessary to know the Bellman principle under the entropy penalty framework. We have

$$\begin{aligned} Q^\pi(s, a) &= E_{\substack{s' \sim P \\ a' \sim \pi}} \left[ R(s, a, s') + \gamma \left( Q^\pi(s', a') + \alpha H(\pi(\cdot|s')) \right) \right] \\ &= E_{s' \sim P} \left[ R(s, a, s') + \gamma V^\pi(s') \right] \end{aligned}$$



## 32.4.9.2 The SAC algorithm

The SAC algorithm uses different neural networks to represent stochastic policy  $\pi_\theta$ ,  $Q_\phi$  function, and  $V_\psi$  function.

The learning of  $V$  will use  $Q$  to provide the target and take gradient descent on loss function given by

$$J_V(\psi) = E_{s_t \sim \mathcal{D}} \left[ \frac{1}{2} \left( V_\psi(s_t) - E_{a_t \sim \pi_\phi} \left[ Q_\theta(s_t, a_t) - \log \pi_\phi(a_t | s_t) \right] \right)^2 \right],$$

where  $\mathcal{D}$  is the sample distribution. Further, we can use two twin  $Q$  functions, similar to TD3, to reduce the overestimation bias.

The learning of  $Q$  will use  $V$  to provide the target and take gradient descent on loss function given by

$$J_Q(\theta) = E_{(s_t, a_t) \sim \mathcal{D}} \left[ \frac{1}{2} \left( Q_\theta(s_t, a_t) - \hat{Q}(s_t, a_t) \right)^2 \right],$$

where

$$\hat{Q}(s_t, a_t) = r(s_t, a_t) + \gamma E_{s_{t+1} \sim p} [V_\psi(s_{t+1})]$$

To facilitate the learning of policy  $\pi$ , we parameterize  $a_t = f_\theta(\epsilon_t; s_t)$  via a neural network transformation, where  $\epsilon_t$  is an input noise vector, sampled from a fixed distribution (e.g., multivariate Gaussian). We then improve the policy by taking gradient descent steps on the objective function

$$J_\pi(\phi) = E_{s_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}} \left[ \log \pi_\phi(f_\phi(\epsilon_t; s_t) | s_t) - Q_\theta(s_t, f_\phi(\epsilon_t; s_t)) \right].$$

It can be showed [15] that the gradient for the policy is given by

$$\nabla_\theta J_\pi(\theta) = \nabla_\theta \log \pi_\theta(a_t | s_t) + \left( \nabla_{a_t} \log \pi_\theta(a_t | s_t) - \nabla_{a_t} Q(s_t, a_t) \right) \nabla_\theta f_\theta(\epsilon_t; s_t)$$

The complete algorithm is given in [algorithm 90](#).

---

**Algorithm 90:** Soft Actor-Critic (SAC) policy optimization

---

**Input:** A randomly initialized policy parameter  $\theta_0$ , Q function parameter  $\phi_0$ , and value function parameter  $\psi_0$ .

**Output:** Optimal policy network and value networks.

```

1 Initialize iteration count  $k = 0$ .
2 repeat
3   for  $t = 1, T$  do
4     Select action  $a_t = \mu(s_t | \theta^\mu)$  according to current policy network.
5     Execute action  $a_t$  and observe reward  $r_t$  and new state  $s_{t+1}$ .
6     Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $\mathcal{B}$ .
7     Randomly sample a batch of transition  $B$  from  $\mathcal{B}$ .
8     Estimate value function via least-squared minimization of
9
10      
$$\frac{1}{|B|} \sum_{s \in B} \left( Q_{\phi,1}(s, a_\theta(s)) - \alpha \log \pi_\theta(a_\theta(s) | s) \right)$$

11
12      where  $y_v(s) = \min_{i=1,2} Q_{\phi_i}(s, a) - \alpha \log \pi_\theta(a | s)$ .
13      Estimate Q function via least-squared minimization of
14
15      
$$y_q(r, s', d) = r + \gamma(1 - d)V_{\psi_{\text{targ}}}(s')$$

16
17      Update policy by taking stochastic gradient ascent by
18
19      
$$\frac{1}{|B|} \sum_{s \in B} \left( Q_{\phi,1}(s, a_\theta(s)) - \alpha \log \pi_\theta(a_\theta(s) | s) \right)$$

20   end
21 until sufficient iterations;
```

---

### 32.4.10 Evolution strategies

Evolution Strategies (ES) is an iterative black-box heuristic search optimization methods proven successful in solving complex problems in many domains. It is inspired by natural evolution process and mathematically belongs to a type of stochastic optimization method. Given a parameter vector  $\theta$  to be optimized, each iteration perturbations will be added to  $\theta$  to yield different modified copies of  $\theta$ . A fitness function  $F(\theta)$  will be then used gauge the quality of each copy. Good copies are recombined in certain way to generate a new  $\theta$  for next iteration. Recently, an ES type algorithm was developed in the

context of reinforcement learning[16].  $\theta$  is policy parameter, and the fitness value can be value functions associated with this policy  $\pi_\theta$  or other proxy functions of similar nature.

More formally, let  $\theta$  be a random vector drawn from a distribution parameterized by  $\psi$ . Our goal is to optimize  $\psi$  (instead of  $\theta$ ), such that fitness scores averaging from  $\theta \sim p_\psi(\theta)$  can be maximize. In [16], gradient types of optimization is used and the gradient the expected fitness with respect to parameter  $\psi$  is given by

**Lemma 32.4.2.**

$$\nabla_\psi E_{\theta \sim p_\psi(\theta)}[F(\theta)] = E_{\theta \sim p_\psi(\theta)}[F(\theta) \nabla_\psi \log p_\psi(\theta)]$$

*Proof.*

$$\begin{aligned} \nabla_\psi E_{\theta \sim p_\psi(\theta)}[F(\theta)] &= \nabla_\psi \int F(\theta) p_\psi(\theta) d\theta \\ &= \int F(\theta) \nabla_\psi p_\psi(\theta) d\theta \\ &= \int F(\theta) p_\psi(\theta) \nabla_\psi \log p_\psi(\theta) d\theta \\ &= E_{\theta \sim p_\psi(\theta)}[F(\theta) \nabla_\psi \log p_\psi(\theta)] \end{aligned}$$

□

A commonly used parameterized distribution is isotropic multivariate Gaussian  $MN(\mu, \sigma^2 I)$ ,  $\mu \in \mathbb{R}^d$ , where  $\psi$  is  $(\mu, \sigma)$ . Then we have

**Corollary 32.4.0.1.**

$$\nabla_\mu E_{\theta \sim p_\psi(\theta)}[F(\theta)] = E_{\theta \sim p_\psi(\theta)}[F(\theta) \frac{(\theta - \mu)}{\sigma}]$$

*Proof.* Note that

$$p_\psi(\theta) = \frac{1}{(2\pi)^{d/2} \sigma^d} \exp\left(-\frac{(\theta - \mu)^T (\theta - \mu)}{2\sigma^2}\right).$$

Taking gradient with respect to  $\mu$  we will get the result.

□

The gradient can be estimated the procedures: first generate perturbed copies  $\theta_i = \mu + \sigma \epsilon_i$ ,  $\epsilon_i \sim MN(0, I)$ ,  $i = 1, 2, \dots, N$ , then the estimate for the gradient is given by

$$\frac{1}{N\sigma} \sum_{i=1}^N F(\theta_i) \epsilon_i,$$

where  $F_i = F(\theta_i)$ .

Wrapping together, we have one basic ES algorithm for reinforcement learning [algorithm 91].

---

**Algorithm 91:** Isotropic multivariate Gaussian evolution strategies for reinforcement learning

---

**Input:** Learning rate  $\alpha$ , noise standard deviation  $\sigma$ , and initial policy distribution parameters  $\mu_0$ .

```

1 for  $t = 1, 2, \dots$  do
2   Sample  $\epsilon_1, \dots, \epsilon_n \sim N(0, I)$ .
3   for  $n = 1, \dots, n$  do
4     Using perturbed policy parameter  $\theta_i = \mu_t + \sigma \epsilon_i$  to generate episodes and
       get process returns, denoted by  $F(\theta_i)$ .
5     Setting  $\mu_{t+1} = \mu_t + \frac{\alpha}{\sigma n} \sum_{i=1}^n F_i \epsilon_i$ 
6   end
7 end
Output:  $\theta$ 

```

---

We can also implement ES algorithm in a distributed fashion, as we show in [algorithm 92](#).

---

**Algorithm 92:** Isotropic multivariate Gaussian parallelized evolution strategies for reinforcement learning

---

**Input:** Learning rate  $\alpha$ , noise standard deviation  $\sigma$ , and initial policy distribution parameters  $\mu_0$ .

```

1 Initialize  $n$  workers for known random seeds and same initial parameter  $\mu_0$  for  $t = 1, 2, \dots$  do
2   for each worker  $n = 1, \dots, n$  do
3     Sample  $\epsilon_1, \dots, \epsilon_n \sim N(0, I)$ .
4     Using perturbed policy parameter  $\theta_i = \mu_t + \sigma \epsilon_i$  to generate episodes and
       get process returns, denoted by  $F(\theta_i)$ .
5   end
6   Send all the scalar  $F_i$  from each worker to every other worker.
7   for each worker  $n = 1, \dots, n$  do
8     Reconstruct all perturbations  $\epsilon_j, j = 1, n$  from known random seeds.
9     Setting  $\mu_{t+1} = \mu_t + \frac{\alpha}{\sigma n} \sum_{i=1}^n F_i \epsilon_i$ 
10  end
11 end
Output:  $\theta$ 

```

---

ES have several appealing properties compared to RL algorithm discussed before, including indifference to the distribution of rewards (sparse or dense), no need for backpropagating gradients, and tolerance of potentially arbitrarily long time horizons.

## 32.5 Advanced training strategies

Our previous section [section 32.4] presents a collection of core algorithms that address difficult aspects of the reinforcement learning challenges in tackling complicated decision-making problems[subsection 32.2.6]. These algorithms have been used in multiple domains and have achieved remarkable performance in many long-standing challenging problems.

Despite its remarkable success, training a DRL agent usually requires substantial human effort on hyper-parameter tuning (e.g., learning rate, neural network architecture, exploration-exploitation balance, etc.), particularly for high-dimensional problems, temporally extended, and sparse reward tasks. For these goal-oriented control tasks, deep reinforcement learning can be time-consuming and sample inefficiency because iterative improvement of the agent's behavioral policy requires determination of the long term consequences of their actions using sparse and delayed rewards.

In response to these challenges, this section we continue to present additional advanced training strategies that can speed up convergence and stabilize training.

### 32.5.1 Priority experience replay

In data-driven reinforcement learning, not all transitions are born equal: some transitions are of more importance because either the action receives large rewards or there is a large TD error associated with the transition. The key idea of prioritized experience replay [12] is to replay these critical experience tuples more often to speed up the Q function approximation learning process. The importance of a experience can measured by absolute TD-error.

After associate a scalar priority  $p$  with each transition tuple, they are sampled according to a probability proportional to  $p^\alpha$ , where the parameter  $\alpha$  determines how much priority is used (if we set  $\alpha = 0$ , then we will get uniform samples). To property correct the sampling weight, we further perform weighted stochastic gradient descent, which the sampling weight proportional to  $1/p^\beta$ , where the parameter  $\beta$  determines the impact of importance-sampling weight(To see this, if we set  $\alpha = 0$ , then we will not use any importance sampling).

Further, to ensure new transitions to be more likely sampled at least once, we assign the highest priority to new samples. Finally, reply experiences in a stochastic manner

is preferred over deterministic replay since it can help avoid the algorithm to exploit approximation error and provide robustness. The algorithm is showed in [algorithm 93](#).

---

**Algorithm 93:** Q learning with prioritized experience replay
 

---

**Input:** exponents  $\alpha$  and  $\beta$ .

- 1 Initialize network  $Q_0$  and replay memory buffer  $\mathcal{B}$ . **repeat**
  - 2     Selection action  $a_t$  and observe transition tuple  $(s_t, a_t, s_{t+1}, r_{t+1})$ . Store transition  $(s_t, a_t, s_{t+1}, r_{t+1})$  in  $\mathcal{B}$  with priority  $p_t = \max_{i \in \mathcal{B}} p_i$ . Sample a minibatch of samples, denoted by tuple  $(s_j, a_j, s_{j+1}, r_{j+1})$  with probability  $Pr(j) = \frac{p_j^\alpha}{\sum_i p_i^\alpha}$ .
  - 3     Compute importance-sampling weight  $w_j \propto [\frac{1}{p_j}]^\beta$
  - 4     Perform weight stochastic gradient descent to update Q network.
  - 5     Compute TD-error  $\delta_j = r_{j+1} + \gamma \max_a Q_{target}(s_{j+1}, a) - Q(s_j, a_j)$ .
  - 6     Adjust the priorities of these samples by  $p_j = |\delta_j|$ .
  - 7 **until** sufficient iterations;
- Output:** Q value function
- 

### 32.5.2 Hindsight experience generation

Hindsight experience replay[17] can be viewed as data-sharing strategy for universal value function approximator learning, where the goal is to learn value functions and control policies that are parameterized by task goals. It exploits the fact that a given experience tuple  $(s_t || g, a_t, s_{t+1} || g, r_t || g)$  for a goal  $g$  can be used to construct a experience tuple  $(s_t || g', a_t, s_{t+1} || g', r'_t || g')$  for another goal  $g'$ , where  $r'$  indicates reward can be goal-dependent. In an off-policy learning algorithm, such as DQN and DDPG, these additional experiences almost come free of charge.

The algorithm is showed in [algorithm 94](#).

---

**Algorithm 94:** Deep Q-learning with hindsight experience replay
 

---

```

1 Initialize replay memory  $\mathcal{B}$  to capacity  $N$ .
2 Initialize action-value function  $Q$  with random weights  $\theta$ .
3 Initialize target action-value function  $Q'$  with weights  $\theta^- = \theta$ .
4 for  $episode = 1, M$  do
5   Initialize sequence  $s_1 = \{x_1\}$ .
6   Sample a goal  $g \in G$  for  $t = 1, T$  do
7     With probability  $\epsilon$  select a random action  $a_t$ , otherwise select
        $a_t = \arg \max_a Q(s_t, g, a; \theta)$ .
8     Execute action  $a_t$  in emulator and observe reward  $r_t$  and state  $x_{t+1}$ .
9     Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $s_{t+1}$ .
10    Store transition  $(s_t || g, a_t, r_t || g, s_{t+1} || g)$  in  $\mathcal{B}$ .
11    For every  $H$  steps, sample an additional  $g' \in G$ , set  $r' = r(s_{t+1}, a_t, g')$ , and
      store the transition  $(s_t || g', a_t, r' || g', s_{t+1} || g)$  in  $\mathcal{B}$ .
12    Sample random minibatch of transitions  $(s_j || g, a_j, r_j || g, s_{j+1} || g)$  from  $\mathcal{B}$ .
13    Set
      
$$y_j = \begin{cases} r_j, & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q'(s_{j+1}, g, a'; \theta^-), & \text{otherwise} \end{cases}$$

      Every  $C$  steps reset  $Q' = Q$ .
14   end
15 end

```

---

### 32.5.3 Reverse goal generation

There has been considerable efforts to address the training brittleness and sample inefficiency for these goal-oriented control tasks. One simple yet effective approach is curriculum learning. Curriculum learning[18] first introduced by Bengio to speed up the training in the supervised learning, have also been applied to DRL. The core idea is to schedule the training process such that the agent learns to accomplish simpler tasks before learning to accomplish more complex tasks. For example, the reverse curriculum generation[19], or reverse goal generation, generates curriculum by expanding the start state set via random actions from the goal state and states where the agent has good chances to reach the goal state. The agent will finally learn the control strategies when the start state set covers the desired start state region. Besides the simple heuristics of reverse curriculum generation, there are more sophistic approaches that involves adaptive generation of curriculum from a trainable model[20].



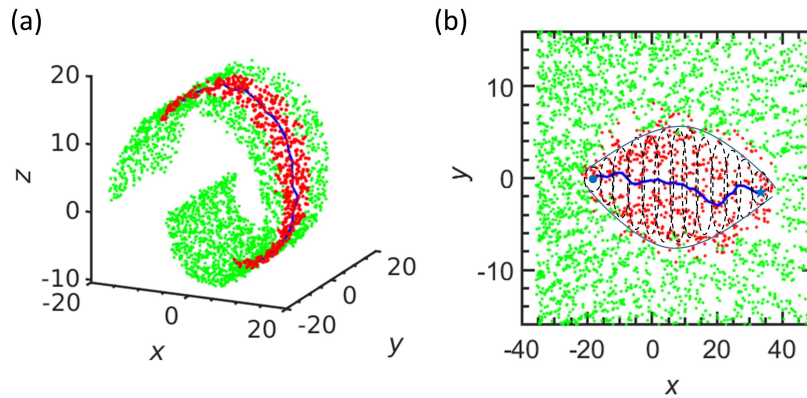
## 32.5.4 Reverse goal generation on low-dimensional manifolds

32.5.4.1 *Key idea*

The success of the curriculum generated to cover the desired start state region relies on the Markov train induced under random action can sufficiently sample the whole state spaces. This assumption can be satisfied easily for deterministic system such as robotics. For high dimensional dynamics in nano-scale science and engineering, the target state is usually low energy state and there exists various adsorbing state or states with every small escape probability the Markov chain. Use random action to produce starting state can fail to produce diverse and gradually difficulties level curriculum.

To address such limitation, we can use manifold learning to help planning curriculum to accelerate the training of DRL agent for goal-oriented tasks [Figure 32.5.1]. Manifold learning is a subset of nonlinear dimensional reduction algorithms that aim to discover the low-dimensional intrinsic structures from high-dimensional data.

The low dimensional manifold can be used to incorporate prior knowledge to avoid trapping regions during exploration and explore regions that a solution can be found. Knowing the global structure can help design curriculum to effective explore the state space that improves sample efficiency.



**Figure 32.5.1:** Illustration of planning curriculum on a swiss roll manifold. Red targets can generating along the path connecting from an intended start point to the target goal position.

32.5.4.2 *Example: navigation on a curved surface*

In our first example, we consider navigating a micro/nano-sized colloidal robot on a curved surface. Colloidal robots have demonstrated great potentials for applications

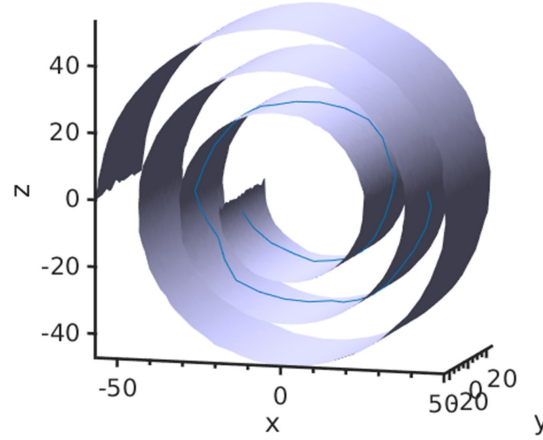
like precision surgery and targeted drug delivery [21][22][23] inside the patient's body and environmental remediation. Navigating the colloidal robot on curved surfaces is a simplified model of navigating on curved interfaces such as bubbles. The equation of motion of such particle is given by

$$\partial_t r = u + \xi(t) \quad (10)$$

$$s.t. \quad c(r) = 0 \quad (11)$$

where  $r \in \mathbb{R}^3$  is the position of the particle, and  $c(r) = 0$  is the constraint that the particle has to stay on the surface, and  $u \in \mathbb{R}^3$  is the external force field as the control input. We allow six discrete actions, corresponding to  $(5, 0, 0)$ ,  $(-5, 0, 0)$ ,  $(0, 5, 0)$ ,  $(0, -5, 0)$ ,  $(5, 0, 0)$ ,  $(5, 0, 0)$ . The  $\xi$  is a three-dimensional zero mean white noise stochastic process satisfying  $E[\xi(t)\xi(t')] = 2DI\delta(t - t')$ , where  $D$  is the transnational diffusivity coefficients. The numerical simulation of equation of motion is carried out using a constrained Brownian dynamics algorithm developed in our previous work[24].

Figure 32.5.2 shows the a representative trajectory on the curved surface under a control policy learned via curriculum learning on a low-dimensional manifold.



**Figure 32.5.2:** One representative trajectory on the curved surface via a control policy learned via curriculum learning on a low-dimensional manifold.

## 32.6 Notes on bibliography

For an overview on Markov decision process, see [25].

For an overview of reinforcement learning, see [1].

Excellent online resources include [Spinning Up](#), [Lil'sLog](#).

---

## BIBLIOGRAPHY

---

1. Sutton, R. S. & Barto, A. G. *Reinforcement learning: An introduction* (MIT press, 2018).
2. Wiering, M. & Van Otterlo, M. Reinforcement learning. *Adaptation, Learning, and Optimization* **12** (2012).
3. Bertsekas, D. *Dynamic Programming and Optimal Control Athena Scientific optimization and computation series v. 2*. ISBN: 9781886529441 (Athena Scientific, 2012).
4. Achiam, J., Knight, E. & Abbeel, P. Towards Characterizing Divergence in Deep Q-Learning. *arXiv preprint arXiv:1903.08894* (2019).
5. Schulman, J., Moritz, P., Levine, S., Jordan, M. & Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).
6. Silver, D. *et al.* Deterministic policy gradient algorithms in ICML (2014).
7. Riedmiller, M. Neural fitted Q iteration—first experiences with a data efficient neural reinforcement learning method in *European Conference on Machine Learning* (2005), 317–328.
8. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518**, 529 (2015).
9. Van Hasselt, H., Guez, A. & Silver, D. Deep Reinforcement Learning with Double Q-Learning. in *AAAI* **2** (2016), 5.
10. Wang, Z. *et al.* Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581* (2015).
11. Hausknecht, M. & Stone, P. Deep recurrent q-learning for partially observable mdps. *CoRR, abs/1507.06527* **7** (2015).
12. Mnih, V. *et al.* Asynchronous methods for deep reinforcement learning in *International conference on machine learning* (2016), 1928–1937.
13. Fujimoto, S., van Hoof, H. & Meger, D. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477* (2018).
14. Schulman, J., Levine, S., Abbeel, P., Jordan, M. & Moritz, P. Trust region policy optimization in *International conference on machine learning* (2015), 1889–1897.
15. Haarnoja, T., Zhou, A., Abbeel, P. & Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290* (2018).

16. Salimans, T., Ho, J., Chen, X., Sidor, S. & Sutskever, I. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864* (2017).
17. Andrychowicz, M. *et al.* *Hindsight experience replay* in *Advances in Neural Information Processing Systems* (2017), 5048–5058.
18. Bengio, Y., Louradour, J., Collobert, R. & Weston, J. *Curriculum learning* in *Proceedings of the 26th annual international conference on machine learning* (ACM, 2009), 41–48. ISBN: 1605585165.
19. Florensa, C., Held, D., Wulfmeier, M., Zhang, M. & Abbeel, P. Reverse Curriculum Generation for Reinforcement Learning, 1–14 (2017).
20. Matiisen, T., Oliver, A., Cohen, T. & Schulman, J. Teacher-Student Curriculum Learning (2017).
21. Li, J., Ávila, B. E.-f. D., Gao, W., Zhang, L. & Wang, J. Micro / nanorobots for biomedicine : Delivery , surgery , sensing , and detoxification (2017).
22. Mallouk, T. E. & Sen, A. Powering Nanorobots. *Scientific American* **300**, 72–77 (2009).
23. Yang, Y., Bevan, M. A. & Li, B. Efficient Navigation of Active Particles in an Unseen Environment via Deep Reinforcement Learning, 1–14 (2019).
24. Yang, Y. & Li, B. A Simulation Algorithm for Brownian Dynamics on Complex Curved Surfaces. *arXiv preprint arXiv:1908.07166* (2019).
25. Puterman, M. L. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming* (John Wiley & Sons, 2014).

## Part VII

## APPENDIX