Reinforcement Learning Notes

J S

2025.06.26

Preface

This is a personal study note of Reinforcement Learning. The main reference is *Mathematical Foundations of Reinforcement Learning* [Zhao, 2025]. Other book resources include Garcia and Rachelson [2013], Horn and Johnson [2012], Sutton and Barto [1998] and Lambert [2024].

Notation	Meaning
$\mathcal{S}$	The set of all possible states
$\mathcal{A}$	The set of all possible actions
$\mathcal{T}$	The set of all time steps where a decision need to be made
$\mathbb{R}$	The set of read numbers
$\mathbb{N}$	The set of natural numbers (0 included)
$[x]_i$	i-th element of vector x
$[A]_{ij}$	Element of the i-th row and j-th column of matrix A
$p(x\|y)$	Conditional probability distribution of x given y
$a := b$	a is defined as b

Table 0.1: Notations

Contents

1 Fundamentals of Reinforcement Learning

1.1 Markov Decision Process

Reinforcement learning (RL) is an interdisciplinary area of machine learning and optimal control that is concerned with **how an intelligent agent should take actions in a dynamic environment**. Reinforcement learning is one of the three basic machine learning paradigms, alongside supervised learning and unsupervised learning. The **purpose** of reinforcement learning is for the agent to **learn an optimal (or near-optimal) policy that maximizes the reward function**.

Basic reinforcement learning is modeled as a **Markov decision process**:

Definition 1.1 (Markov Decision Process)**.** [Garcia and Rachelson, 2013] Markov decision process (MDP) is defined as *controlled stochastic processes* which satisfies the Markov property and assigns reward values to state transitions. Formally, they are described by a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, p, r)$ where:

$\mathcal{S}$ is the set of all possible states in which the process' evolution takes place;
- $\mathcal{A}$ is the set of all possible actions which control the state dynamics;
- $\mathcal{T}$ is the set of time steps where decisions need to be made, which is a subset of $\mathbb{N}$ and can either be finite or infinite;
- $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the state transition probability function;
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function defined on state transitions.

At every time step $t \in \mathcal{T}$, an action a_t is applied in the current state s_t, affecting the process in its transition to the next state s_{t+1}:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \cdots .$$

The transition probability is given as[a] $p(s_{t+1}|s_t, a_t)$. The state transition satisfies the **Markov property**: $p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = p(s_{t+1}|s_t, a_t)$. This indicates that the next state depends merely on the current state and action.

A reward[b] $r_{t+1} = r(s_t, a_t)$ is obtained for this transition. The important feature is that this reward only depends on the simple input of the current state s and the current action a.[c][d]

[a] $p(s_{t+1}|s_t, a_t)$ is a conditional marginal distribution which can be derived from the joint probability $p(s_t, a_t, s_{t+1})$.

[b] We use r_{t+1} instead of r_t to denote the reward because it emphasizes that the next reward r_{t+1} and next state s_{t+1} are jointly determined [Sutton and Barto, 1998].

[c] Notably, the reward can also be modeled in a stochastic way. The relationship is given by the conditional probability distribution: $r_{t+1} \sim p_r(r|s_t, a_t)$. The distribution of r also follows the Markov property:

$$p_r(r|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = p_r(r|s_t, a_t).$$

However, in most cases, we don't explicitly consider random reward, but instead model the reward r_{t+1} as a deterministic function of s_t and a_t, i.e., $r_{t+1} = r(s_t, a_t)$.

[d] Why the reward is a function of s_t and a_t, but the next state s_{t+1} isn't involved as an input? This seems counterintuitive, since it is usually the next state that affects the reward.

Here we consider the general case where the reward is not a deterministic function of state and action but a random variable R. We initially set s_t, a_t, s_{t+1} as the input, so the distribution of reward is

$$R_{t+1} \sim p_r(r|s_t, a_t, s_{t+1}).$$

The key is that the distribution of s_{t+1} also depends on s_t and a_t by the transition probability $p(s_{t+1}|s_t, a_t)$. So we can eliminate s_{t+1}'s existence by taking the expectation over the next state s_{t+1}:

$$\tilde{R}_{t+1} \sim p_r(r|s_t, a_t) = \mathbb{E}_{s_{t+1}\sim p(s_{t+1}|s_t,a_t)} p_r(r|s_t, a_t, s_{t+1}).$$

Moreover, we usually consider deterministic reward given s_t and a_t, so we can further take the expectation of the distribution of reward, which yields

$$r_{t+1} = r(s_t, a_t) = \mathbb{E}[\tilde{R}_{t+1}] = \int_{-\infty}^{\infty} p_r(r|s_t, a_t) r dr.$$

The transition and reward functions can vary across time. The process is said to be **stationary** when these functions do not change across all time steps. In the scope of this article, we will keep this stationarity hypothesis by default.

A Markov decision process defines an environment $\mathcal{E}$, which is the system the agent interacts with. In this sense, the goal of reinforcement learning is to learn a good **policy** to optimize some target in this dynamic environment.

In this article, we assume discrete state space $\mathcal{S}$ and action space $\mathcal{A}$ by default. It is easy to generalize to continuous cases.

1.2 Action Policy

Markov decision processes allow us to model the state evolution dynamics of a stochastic system when this system is controlled by an agent choosing and applying the actions a_t at every time step t. The procedure of choosing such actions is called an action **policy**, or **strategy**, and is written as π.

A policy can be deterministic or stochastic, Markov or history-dependent. For a history-dependent policy, the action is based on the whole history h_t of the process; for a Markov policy, only the current state s_t is considered. For a deterministic policy, $\pi_t(s_t)$ or $\pi_t(h_t)$ defines the chosen action a_t; for a stochastic policy, $\pi_t(a, s_t)$ or $\pi_t(a, h_t)$ represents the joint probability distribution of a_t and state information.

Policy π_t	Deterministic	Stochastic
Markov	$s_t \to a_t$	$a_t, s_t \to \mathbb{R}$
History-dependent	$h_t \to a_t$	$h_t, s_t \to \mathbb{R}$

Table 1.1: Different policy families for MDPs

Therefore, we obtain four main families of policies, as shown in Table 1.1. These sets of policies are included in each other, from the most general case of stochastic, history-dependent policies, to the very specific case of deterministic, Markov policies, as shown in Figure 1.1.

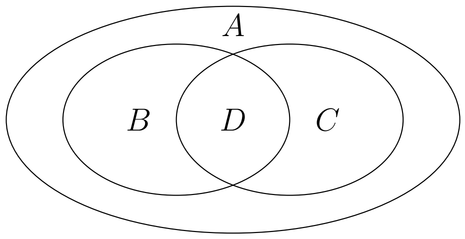

Figure 1.1: Veen diagram of different types of action policies. A: stochastic, history-dependent; B: deterministic, history-dependent; C: stochastic, Markov; D: deterministic, Markov.

Once the policy in an MDP is fixed, the MDP degenerates into an ordinary Markov process.

1.3 Returns

After executing an action at a state at time t, the agent obtains a **reward**, denoted as r_t, as feedback from the environment. Usually the reward is modeled as a function of the state s_t and action a_t, denoted as $r_t = r(s_t, a_t)$.

A reward can be interpreted as a human-machine interface, with which we can guide the agent to behave as we expect. If we expect the agent to act in a specific way, we should design a reward function where the expected action gives a higher reward and unexpected action gives a lower reward.

To determine a good policy, we must consider the *total reward* obtained in the long run, rather than an *immediate reward*. An action with the greatest immediate reward may not lead to the greatest total reward.

A **trajectory** is a state-action-reward chain: $((s_0, a_0, r_1), (s_1, a_1, r_2), (s_2, a_2, r_3), \ldots)$. When interacting with the environment by following a policy, the agent may stop at some terminal states. The resulting trajectory is called an **episode** (or a *trial*), which is a finite trajectory.

The **return** of a trajectory is defined as the sum of all the rewards collected along the trajectory. Returns are also called *total rewards* or *cumulative rewards*.

A return consists of an *immediate reward* and *future rewards*. Here, the immediate reward is the reward obtained after taking an action at the initial state; the future rewards refer to the rewards obtained after leaving the initial state. It is possible that the immediate reward is negative while the future reward is positive. Thus, which actions to take should be determined by the return (i.e., the total reward) rather than the immediate reward to avoid short-sighted decisions.

Return can also be defined for infinitely long trajectories. We must introduce the *discounted return* concept for infinitely long trajectories:

$$\text{discounted return} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{i=0}^{\infty} \gamma^i r_{t+1+i},$$

where $\gamma \in (0, 1)$ is called the **discount rate**.

The discount rate can be used to adjust the emphasis placed on near- or far-future rewards. In particular, if is close to 0, then the agent places more emphasis on rewards obtained in the near future. The resulting policy would be short-sighted. If is close to 1, then the agent places more emphasis on the far future rewards. The resulting policy is far-sighted and dares to take risks of obtaining negative rewards in the near future.

2 State Values and Bellman Equation

2.1 State Values

Returns can be used to evaluate policies. However, they are inapplicable to stochastic systems because starting from one state may lead to different returns. Motivated by this problem, the concept of **state value** is introduced.

First, we need to introduce some necessary notations. Consider a sequence of time steps $t = 0, 1, 2, \ldots$. At time t, the agent is in state S_t, and the action taken following a policy π is A_t. The next state is S_{t+1}, and the immediate reward obtained is R_{t+1}. This process can be expressed concisely as

$$S_t \xrightarrow{A_t} S_{t+1}, R_{t+1}.$$

Note that $S_t, S_{t+1}, A_t, R_{t+1}$ are all random variables. Moreover, $S_t, S_{t+1} \in \mathcal{S}$, $A_t \in \mathcal{A}$, and $R_{t+1} = r(S_t, A_t)$.

Starting from t, we can obtain a state-action-reward trajectory:

$$S_t \xrightarrow{A_t} S_{t+1}, R_{t+1} \xrightarrow{A_{t+1}} S_{t+2}, R_{t+2} \xrightarrow{A_{t+2}} S_{t+3}, R_{t+3} \cdots$$

By definition, the discounted return along the trajectory is

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots,$$

where $\gamma \in (0, 1)$ is the discount rate.

G_t is also a random variable, and we can calculate its conditional expectation:

$$v_\pi(s) := \mathbb{E}[G_t | S_t = s].$$

Here, $v_\pi(s)$ is called the **state-value function** or simply the **state value** of s. Some important remarks are given below.

- $v_\pi(s)$ depends on s. This is because it is a conditional expectation with the condition that the agent starts from $S_t = s$.
- $v_\pi(s)$ depends on π. This is because the trajectories are generated by following the policy π. For a different policy, the state value may be different.
- $v_\pi(s)$ does not depend on t. As long as the environment is stationary (do not vary with time), the value of a state is determined once the policy is given, whenever the current time step is.

2.2 Bellman Equation

Bellman equation is a crucial mathematical tool for analyzing state values. In a nutshell, the Bellman equation is a set of linear equations that describe the relationships between the values of all the states. We next derive the Bellman equation. First, note that the discounted return G_t can be rewritten as

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \\ &= R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \dots) \\ &= R_{t+1} + \gamma G_{t+1}, \end{aligned}$$

This equation establishes the relationship between G_t and G_{t+1}. Then, the state value can be written as

$$\begin{aligned} v_\pi(s) &= \mathbb{E}[G_t | S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \mathbb{E}[R_{t+1} | S_t = s] + \gamma \mathbb{E}[G_{t+1} | S_t = s]. \end{aligned} \tag{2.1}$$

The two terms are analyzed below.

- The first term, $\mathbb{E}[R_{t+1}|S_t = s]$, is the expectation of the immediate rewards. By using the law of total expectation, it can be calculated as

$$\begin{aligned} \mathbb{E}[R_{t+1}|S_t = s] &= \sum_{a \in \mathcal{A}} \pi(a|s) \mathbb{E}[R_{t+1}|S_t = s, A_t = a] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) r(s,a). \end{aligned} \tag{2.2}$$

- The second term, $\mathbb{E}[G_{t+1}|S_t = s]$, is the expectation of the future rewards. It can be calculated as

$$\begin{aligned} \mathbb{E}[G_{t+1}|S_t = s] &= \sum_{s' \in \mathcal{S}} \mathbb{E}[G_{t+1}|S_t = s, S_{t+1} = s'] p(s'|s) \\ &= \sum_{s' \in \mathcal{S}} \mathbb{E}[G_{t+1}|S_{t+1} = s'] p(s'|s) \quad \text{(due to the Markov property)} \\ &= \sum_{s' \in \mathcal{S}} v_\pi(s') p(s'|s) \\ &= \sum_{s' \in \mathcal{S}} v_\pi(s') \sum_{a \in \mathcal{A}} p(s'|s,a) \pi(a|s). \end{aligned} \tag{2.3}$$

Substituting (2.2) and (2.3) into (2.1) yields

$$\begin{aligned} v_\pi(s) &= \mathbb{E}[R_{t+1}|S_t = s] + \gamma \mathbb{E}[G_{t+1}|S_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) r(s,a) + \gamma \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} p(s'|s,a) v_\pi(s') \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \Big[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v_\pi(s') \Big], \quad \forall s \in \mathcal{S} \end{aligned} \tag{2.4}$$

(2.4) is the **Bellman equation**, which characterizes the relationships of state values. It is a fundamental tool for designing and analyzing reinforcement learning algorithms.
Some remarks of Bellman equation are given as follows.

- $v_\pi(s)$ and $v_\pi(s')$ are unknown state values to be calculated.
- $\pi(a|s)$ gives the probability of taking action a under state s by policy π.

- $p(s'|s,a)$ gives the system model, which is the transition probability from state s to state s' with action a.

Note that $\sum_{s'\in\mathcal{S}} p(s'|s,a) = 1$, so (2.4) can be rewritten as

$$\begin{aligned} v_\pi(s) &= \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} p(s'|s,a)\Big[r(s,a) + \gamma v_\pi(s')\Big] \\ &= \mathbb{E}_{a\sim\pi(a|s)}\mathbb{E}_{s'\sim p(s'|s,a)}\Big[r(s,a) + \gamma v_\pi(s')\Big] \end{aligned} \tag{2.5}$$

2.3 Matrix-vector Form of the Bellman Equation

The Bellman equation in (2.4) is in an elementwise form. Since it is valid for every state, we can combine all these equations and write them concisely in a matrix-vector form, which will be frequently used to analyze the Bellman equation.

To derive the matrix-vector form, we first rewrite the Bellman equation in (2.4) as

$$v_\pi(s) = r_\pi(s) + \gamma \sum_{s'\in\mathcal{S}} p_\pi(s'|s) v_\pi(s'), \tag{2.6}$$

where

- $r_\pi(s) := \sum_{a\in\mathcal{A}} \pi(a|s) r(s,a)$ denotes the mean of the immediate rewards in state s under policy π,
- $p_\pi(s'|s) := \sum_{a\in\mathcal{A}} \pi(a|s) p(s'|s,a)$ is the probability of transitioning from s to s' under policy π.

Suppose that the states are indexed as s_i with $i = 1,\dots,n$, where $n = |\mathcal{S}|$. For all states s_i ($i = 1,2,\dots,n$), (2.6) can be written in matrix form as

$$\underbrace{\begin{bmatrix} v_\pi(s_1) \\ v_\pi(s_2) \\ \vdots \\ v_\pi(s_n) \end{bmatrix}}_{:=V_\pi} = \underbrace{\begin{bmatrix} r_\pi(s_1) \\ r_\pi(s_2) \\ \vdots \\ r_\pi(s_n) \end{bmatrix}}_{:=R_\pi} + \gamma \underbrace{\begin{bmatrix} p_\pi(s_1|s_1) & p_\pi(s_2|s_1) & \cdots & p_\pi(s_n|s_1) \\ p_\pi(s_1|s_2) & p_\pi(s_2|s_2) & \cdots & p_\pi(s_n|s_2) \\ \vdots & \vdots & \ddots & \vdots \\ p_\pi(s_1|s_n) & p_\pi(s_2|s_n) & \cdots & p_\pi(s_n|s_n) \end{bmatrix}}_{:=P_\pi} \underbrace{\begin{bmatrix} v_\pi(s_1) \\ v_\pi(s_2) \\ \vdots \\ v_\pi(s_n) \end{bmatrix}}_{:=V_\pi}, \tag{2.7}$$

$$V_\pi = R_\pi + \gamma P_\pi V_\pi, \tag{2.8}$$

where V_π is the unknown to be solved, and R_π, P_π are known.

The matrix P_π has some interesting properties. First, it is a nonnegative matrix, meaning that all its elements are equal to or greater than zero. Second, the sum of the values in every row of P_π is equal to one. This property is denoted as $P_\pi \mathbf{1} = \mathbf{1}$, where $\mathbf{1} = [1,\dots,1]^T \in \mathbb{R}^n$.

2.4 Solving State Values from the Bellman Equation

Since $V_\pi = R_\pi + \gamma P_\pi V_\pi$ ((2.8)) is a simple linear equation, its **closed-form solution** can be easily obtained as

$$V_\pi = (I - \gamma P_\pi)^{-1} R_\pi. \tag{2.9}$$

Some properties of $(I - \gamma P_\pi)^{-1}$ are given below.

1. $I - \gamma P_\pi$ is invertible.[1]
2. $(I-\gamma P_\pi)^{-1} \geq I$, where "$\geq$" represents elementwise comparison. This is because all the entries of P_π are nonnegative, and hence $(I - \gamma P_\pi)^{-1} = I + \gamma P_\pi + \gamma^2 P_\pi^2 + \cdots \geq I$.
3. For any vector $r \geq 0$, it holds that $(I - \gamma P_\pi)^{-1} r \geq r \geq 0$. This property follows from the second property because $[(I - \gamma P_\pi)^{-1} - I]r \geq 0$. As a consequence, if $r_1 \geq r_2$, we have $(I - \gamma P_\pi)^{-1} r_1 \geq (I - \gamma P_\pi)^{-1} r_2$.

Although the closed-form solution is useful for theoretical analysis purposes, it is not applicable in practice because it involves a matrix inversion operation, which still needs other numerical algorithms. In fact, we can directly solve the Bellman equation using the following **iterative algorithm**:

$$V_{k+1} = R_\pi + \gamma P_\pi V_k, \quad k = 0, 1, 2, \dots \tag{2.10}$$

This algorithm generates a sequence of values $\{V_0, V_1, V_2, \dots\}$, where $V_0 \in \mathbb{R}^n$ is an (arbitrary) initial guess of V_π. It holds that[2]

$$V_k \to V_\pi = (I - \gamma P_\pi)^{-1} R_\pi, \quad \text{as } k \to \infty. \tag{2.11}$$

So if the number of iteration is high enough, the generated V_k is approximately V_π

2.5 Action Values

To evaluate the goodness of an action, we introduce **action value**. The action value of a state-action pair (s, a) is defined as

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

As is shown in the above expression, the action value is defined as the expected return that can be obtained after taking an action at a state.

What is the relationship between action values and state values?

1. First, it follows from the properties of conditional expectation that

$$\mathbb{E}[G_t|S_t = s] = \mathbb{E}_{a\sim\pi(a|s)}\mathbb{E}[G_t|S_t = s, A_t = a] = \sum_{a\in\mathcal{A}} \pi(a|s)\mathbb{E}[G_t|S_t = s, A_t = a]$$

$$\implies \quad v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s, a). \tag{2.12}$$

As a result, a state value is the expectation of the action values associated with that state.

[1] According to the Gershgorin circle theorem [Horn and Johnson, 2012], every eigenvalue of $I - \gamma P_\pi$ lies within at least one of the Gershgorin circles. For the i-th Gershgorin circle, the center is at $[I - \gamma P_\pi]_{ii} = 1 - \gamma P_\pi(s_i|s_i)$, and the radius equals to $\sum_{j\neq i}[I - \gamma P_\pi]_{ij} = -\sum_{j\neq i} \gamma P_\pi(s_j|s_i)$. Since $\gamma < 1$, we know that the radius is less than the magnitude of the center: $\sum_{j\neq i} \gamma P_\pi(s_j|s_i) < 1 - \gamma P_\pi(s_i|s_i)$. Therefore, all Gershgorin circles do not encircle the origin, and hence no eigenvalue of $I - \gamma P_\pi$ is zero.

[2] Define the error as $\delta_k = v_k - V_\pi$. We only need to show that $\delta_k \to 0$. Substituting $v_{k+1} = \delta_{k+1} + V_\pi$ and $v_k = \delta_k + V_\pi$ into (2.10) gives

$$\delta_{k+1} + V_\pi = R_\pi + \gamma P_\pi(\delta_k + V_\pi),$$

which can be rewritten as

$$\begin{aligned}\delta_{k+1} &= -V_\pi + R_\pi + \gamma P_\pi \delta_k + \gamma P_\pi V_\pi \\ &= \gamma P_\pi \delta_k - V_\pi + (R_\pi + \gamma P_\pi V_\pi) \\ &= \gamma P_\pi \delta_k.\end{aligned}$$

As a result,

$$\delta_{k+1} = \gamma P_\pi \delta_k = \gamma^2 (P_\pi)^2 \delta_{k-1} = \cdots = \gamma^{k+1}(P_\pi)^{k+1}\delta_0.$$

Since every entry of P_π is nonnegative and no greater than one, we have that $0 \leq (P_\pi)^k \leq 1$ for any k. On the other hand, since $\gamma < 1$, we know that $\gamma^k \to 0$, and hence $\delta_{k+1} = \gamma^{k+1}(P_\pi)^{k+1}\delta_0 \to 0$ as $k \to \infty$.

2. Second, since the state value is given by

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)\Big[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a)v_\pi(s')\Big],$$

comparing it with (2.12) leads to

$$\begin{aligned} q_\pi(s,a) &= r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a)v_\pi(s') \\ &= r(s,a) + \gamma\mathbb{E}_{s'\sim p(s'|s,a)}[v_\pi(s')]. \end{aligned} \tag{2.13}$$

It can be seen that the action value consists of two terms. The first term corresponds to the immediate reward, and the second term correspond to the expectation of future rewards.

(2.12) and (2.13) are the two sides of the same coin: (2.12) shows how to obtain state values from action values, whereas (2.13) shows how to obtain action values from state values.
The Bellman equation can also be expressed in action values. Substituting (2.12) into (2.13) yields

$$q_\pi(s,a) = r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a) \sum_{a'\in\mathcal{A}} \pi(a'|s')q_\pi(s',a') \tag{2.14}$$

The above equation is valid for every state-action pair. If we put all these equations together, their matrix-vector form is

$$\underbrace{\begin{bmatrix} Q_1 \\ Q_2 \\ \vdots \\ Q_n \end{bmatrix}}_{:=Q_\pi} = \underbrace{\begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_n \end{bmatrix}}_{:=\tilde{R}} + \gamma \underbrace{\begin{bmatrix} P_{11} & P_{21} & \cdots & P_{1n} \\ P_{21} & P_{22} & \cdots & P_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ P_{n1} & P_{nn} & \cdots & P_{nn} \end{bmatrix}}_{:=P} \underbrace{\begin{bmatrix} \Pi_{11} & 0 & \cdots & 0 \\ 0 & \Pi_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \Pi_{nn} \end{bmatrix}}_{:=\Pi} \underbrace{\begin{bmatrix} Q_1 \\ Q_2 \\ \vdots \\ Q_n \end{bmatrix}}_{:=Q_\pi}, \tag{2.15}$$

where

$$Q_i = \begin{bmatrix} q_\pi(s_i,a_1) \\ q_\pi(s_i,a_2) \\ \vdots \\ q_\pi(s_i,a_m) \end{bmatrix} \in \mathbb{R}^{m\times 1}, \quad R_i = \begin{bmatrix} r(s_i,a_1) \\ r(s_i,a_2) \\ \vdots \\ r(s_i,a_n) \end{bmatrix} \in \mathbb{R}^{m\times 1},$$

$$P_{ij} = \begin{bmatrix} p(s_i|s_j,a_1) \\ p(s_i|s_j,a_2) \\ \vdots \\ p(s_i|s_j,a_m) \end{bmatrix} \in \mathbb{R}^{m\times 1}, \quad \Pi_{ii} = \begin{bmatrix} \pi(a_1|s_i) \\ \pi(a_2|s_i) \\ \cdots \\ \pi(a_m|s_i) \end{bmatrix}^T \in \mathbb{R}^{1\times m}.$$

This can be written concisely as

$$Q_\pi = \tilde{R} + \gamma P\Pi Q_\pi. \tag{2.16}$$

Here $Q_\pi \in \mathbb{R}^{(n\times m)\times 1}$ is the vector of action values; $\tilde{R} \in \mathbb{R}^{(n\times m)\times 1}$ is the vector of immediate rewards; $P \in \mathbb{R}^{(n\times m)\times n}$ is the transition probability matrix whose row is indexed by the state-action pairs and whose column is indexed by the states; $\Pi \in \mathbb{R}^{n\times(n\times m)}$ is the policy matrix which is block diagonal.

(2.16) is the Bellman equation in terms of action values. Compared to the Bellman equation in terms of state values ((2.8)), (2.16) has some unique features. For example, $\tilde{R}$ and P are merely determined by the system model, independent of the policy. The policy is embedded in Π. It can be verified that (2.16) is also a contraction mapping and has a unique solution that can be iteratively solved.

3 Optimal State Values and Bellman Optimality Equation

3.1 Optimal State Values and Optimal Policy

The ultimate goal of reinforcement learning is to seek **optimal policies**. The definition is based on state values. In particular, consider two given policies π_1 and π_2. If the state value of π_1 is greater than or equal to that of π_2 for any state, i.e., $v_{\pi_1}(s) \geq v_{\pi_2}(s), \forall s \in \mathcal{S}$, then π_1 is said to be better than π_2. Furthermore, if a policy is better than all the other possible policies, then this policy is optimal. This is formally stated below.

Definition 3.1 (Optimal Policy and Optimal State Value)**.** A policy π^* is optimal if $v_{\pi^*}(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$ and for any other policy π. The state values of π are the optimal state values.

The above definition indicates that an optimal policy has the greatest state value for every state compared to all the other policies. This definition also leads to many questions:

- Existence: Does the optimal policy exist?
- Uniqueness: Is the optimal policy unique?
- Stochasticity: Is the optimal policy stochastic or deterministic?
- Algorithm: How to obtain the optimal policy and the optimal state values?

These questions will be answered in the remainder of this section.

3.2 Bellman Optimality Equation

The tool for analyzing optimal policies and optimal state values is the **Bellman optimality equation (BOE)**.

For every $s \in \mathcal{S}$, the elementwise expression of the BOE is

$$\begin{aligned} v(s) &= \max_{\pi(s)\in\Pi(s)} \sum_{a\in\mathcal{A}} \pi(a|s)\left[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a)v(s')\right] \\ &= \max_{\pi(s)\in\Pi(s)} \sum_{a\in\mathcal{A}} \pi(a|s) q(s,a), \end{aligned} \tag{3.1}$$

where $v(s), v(s')$ are unknown variables to be solved and

$$q(s,a) \doteq r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a)v(s').$$

Here, $\pi(s)$ denotes a policy for state s, and $\Pi(s)$ is the set of all possible policies for s.

| Matrix-vector Form of BOE

The BOE refers to a set of equations defined for all states. If we combine these equations, we can obtain a concise matrix-vector form:

$$\underbrace{\begin{bmatrix} v(s_1) \\ v(s_2) \\ \vdots \\ v(s_n) \end{bmatrix}}_{:=v} = \max_{\pi\in\Pi} \left(\underbrace{\begin{bmatrix} r_\pi(s_1) \\ r_\pi(s_2) \\ \vdots \\ r_\pi(s_n) \end{bmatrix}}_{:=R_\pi} + \gamma \underbrace{\begin{bmatrix} p_\pi(s_1|s_1) & p_\pi(s_2|s_1) & \cdots & p_\pi(s_n|s_1) \\ p_\pi(s_1|s_2) & p_\pi(s_2|s_2) & \cdots & p_\pi(s_n|s_2) \\ \vdots & \vdots & \ddots & \vdots \\ p_\pi(s_1|s_n) & p_\pi(s_2|s_n) & \cdots & p_\pi(s_n|s_n) \end{bmatrix}}_{:=P_\pi} \underbrace{\begin{bmatrix} v(s_1) \\ v(s_2) \\ \vdots \\ v(s_n) \end{bmatrix}}_{:=v} \right), \tag{3.2}$$

$$v = \max_{\pi \in \Pi} (R_\pi + \gamma P_\pi v). \tag{3.3}$$

Here v, $R_\pi \in \mathbb{R}^n$, $P_\pi \in \mathbb{R}^{n \times n}$, where $n = |\mathcal{S}|$. $\max_\pi(\cdot)$ is performed in an elementwise manner. The structures of R_π and P_π are the same as those in the matrix-vector form of the normal Bellman equation ((2.7)).

The solution of Bellman optimality equation ((3.1)) is done in a two step manner: (1) maximize its right hand side; (2) solve the equality. Below are details of the operations.

| Maximization of the Right Hand Side of BOE

Example 3.1 (Inspiring Example of Maximum Weighted Sum)**.**

- **Problem**: Given $q_1, q_2, \dots, q_n \in \mathbb{R}$, we would like to find the optimal values of weights $c_1, c_2, \dots, c_n$ to maximize

$$\sum_{i=1}^{n} c_i q_i = c_1 q_1 + c_2 q_2 + \cdots + c_n q_n,$$

  where $\sum_{i=1}^{n} c_i = 1$ and $c_i \geq 0$.

- **Solution**: Denote $q_k = \max_{i \in \{1,2,\dots,n\}} q_i$. It is easy to find that

$$\sum_{i=1}^{n} c_i q_i \leq \sum_{i=1}^{n} c_i q_k = q_k.$$

  So the optimal solution is $c_k^* = 1$ and $c_i^* = 0$ for all $i \neq k$.

Inspired by the above example, since $\sum_a \pi(a|s) = 1$, we have

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

where equality is achieved when

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

Here, $a^* = \arg\max_a q(s,a)$. In summary, for any state s, the optimal policy $\pi(s)$ is the one that selects the action a that has the greatest value of $q(s,a)$. And by maximizing the right hand side of (3.1), the BOE is transformed as

$$v(s) = \max_{a \in \mathcal{A}} q(s,a)$$

| Solution of the Equality of BOE

Since the optimal value of π is determined by v, the right-hand side of (3.1) is a function of v, denoted as

$$f(v) = \max_{\pi \in \Pi} (R_\pi + \gamma P_\pi v).$$

And the BOE can be expressed in a concise form as

$$v = f(v).$$

This equation can be solved by contraction mapping theorem. The contraction property is guaranteed by in Proposition 3.1. The solution is given in Proposition 3.2.

Proposition 3.1 (Contraction Property of the Right Hand Side of BOE)**.** The function $f(v) = \max_{\pi\in\Pi}(R_\pi + \gamma P_\pi v)$ on the right-hand side of the BOE is a contraction mapping. In particular, for any $v_1, v_2 \in \mathbb{R}^{|\mathcal{S}|}$, it holds that

$$\|f(v_1) - f(v_2)\|_\infty \le \gamma \|v_1 - v_2\|_\infty$$

where $\gamma \in (0,1)$ is the discount rate, and $\|\cdot\|_\infty$ is the maximum norm, which is the maximum absolute value of the elements of a vector.

Proof of Proposition 3.1 Consider any two vectors $v_1, v_2 \in \mathbb{R}^{|\mathcal{S}|}$, and suppose that $\pi_1^* := \arg\max_\pi(r_\pi + \gamma P_\pi v_1)$ and $\pi_2^* := \arg\max_\pi(r_\pi + \gamma P_\pi v_2)$. Then,

$$\begin{aligned}
f(v_1) &= \max_\pi(r_\pi + \gamma P_\pi v_1) = r_{\pi_1^*} + \gamma P_{\pi_1^*} v_1 \ge r_{\pi_2^*} + \gamma P_{\pi_2^*} v_1, \\
f(v_2) &= \max_\pi(r_\pi + \gamma P_\pi v_2) = r_{\pi_2^*} + \gamma P_{\pi_2^*} v_2 \ge r_{\pi_1^*} + \gamma P_{\pi_1^*} v_2,
\end{aligned}$$

where $\ge$ is an elementwise comparison. As a result,

$$\begin{aligned}
f(v_1) - f(v_2) &= r_{\pi_1^*} + \gamma P_{\pi_1^*} v_1 - (r_{\pi_2^*} + \gamma P_{\pi_2^*} v_2) \\
&\le r_{\pi_1^*} + \gamma P_{\pi_1^*} v_1 - (r_{\pi_1^*} + \gamma P_{\pi_1^*} v_2) \\
&= \gamma P_{\pi_1^*}(v_1 - v_2).
\end{aligned}$$

Similarly, it can be shown that $f(v_2) - f(v_1) \le \gamma P_{\pi_2^*}(v_2 - v_1)$. Therefore,

$$\gamma P_{\pi_2^*}(v_1 - v_2) \le f(v_1) - f(v_2) \le \gamma P_{\pi_1^*}(v_1 - v_2).$$

Define

$$z := \max\left\{|\gamma P_{\pi_2^*}(v_1 - v_2)|, |\gamma P_{\pi_1^*}(v_1 - v_2)|\right\} \in \mathbb{R}^{|\mathcal{S}|},$$

where $\max(\cdot)$, $|\cdot|$, and $\ge$ are all elementwise operators. By definition, $z \ge 0$. On the one hand, it is easy to see that

$$-z \le \gamma P_{\pi_2^*}(v_1 - v_2) \le f(v_1) - f(v_2) \le \gamma P_{\pi_1^*}(v_1 - v_2) \le z,$$

which implies

$$|f(v_1) - f(v_2)| \le z.$$

It then follows that

$$\|f(v_1) - f(v_2)\|_\infty \le \|z\|_\infty, \tag{3.4}$$

where $\|\cdot\|_\infty$ is the maximum norm.

On the other hand, suppose that z_i is the i-th entry of z, and p_i^T and q_i^T are the i-th row of $P_{\pi_1^*}$ and $P_{\pi_2^*}$, respectively. Then,

$$z_i = \max\{\gamma|p_i^T(v_1 - v_2)|, \gamma|q_i^T(v_1 - v_2)|\}.$$

Since p_i is a vector with all nonnegative elements and the sum of the elements is equal to one, it follows that

$$|p_i^T(v_1 - v_2)| \le p_i^T|v_1 - v_2| \le \|v_1 - v_2\|_\infty.$$

Similarly, we have $|q_i^T(v_1 - v_2)| \leq \|v_1 - v_2\|_\infty$. Therefore, $z_i \leq \gamma\|v_1 - v_2\|_\infty$ and hence

$$\|z\|_\infty = \max_i |z_i| \leq \gamma\|v_1 - v_2\|_\infty. \tag{3.5}$$

Combining (3.4) and (3.5) yields

$$\|f(v_1) - f(v_2)\|_\infty \leq \gamma\|v_1 - v_2\|_\infty,$$

which concludes the proof of the contraction property of $f(v)$.

3.3 Optimality Property of BOE Solution

| Solving v^*

Proposition 3.2 (Existence, Uniqueness, and Algorithm for Bellman Optimality Equation)**.** For the BOE $v = f(v) = \max(R_\pi + P_\pi v)$, there always exists a unique solution v^*, which can be solved iteratively by

$$v_{k+1} = f(v_k) = \max_{\pi\in\Pi}(R_\pi + \gamma P_\pi v_k), \quad k = 0, 1, 2, \ldots.$$

The value of v_k converges to v^* exponentially fast as $k \to \infty$ given any initial guess v_0.

The proof of this theorem directly follows from the contraction mapping theorem since $f(v)$ is a contraction mapping. This theorem is important because it answers some fundamental questions.

- Existence of v^* : The solution of the BOE always exists.
- Uniqueness of v^*: The solution v^* is always unique.
- Algorithm for solving v^*: The value of v^* can be solved by the iterative algorithm suggested by the above theorem. This iterative algorithm has a specific name called *value iteration*, which will be introduced later.

| Solving π^*

Once the value of v^* has been obtained, we can easily obtain π^* by solving

$$\pi^* = \arg\max_{\pi\in\Pi}(R_\pi + \gamma P_\pi v^*). \tag{3.6}$$

Substituting (3.6) into the BOE ((3.3)) yields

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

Therefore, $v^* = v_{\pi^*}$ is the state value of π^*, and the BOE is a special Bellman equation whose corresponding policy is π^*.

| Optimality of v^* and π^*

Up to now, we can solve v^* and π^* from the Bellman optimality equation. However, it is still unclear whether this solution is the optimal one. The optimality is guaranteed by the following theorem.

Proposition 3.3 (Optimality of v^* and π^*)**.** The solution v^* is the optimal state value, and π^* is an optimal policy. That is, for any policy π, it holds that

$$v^* = v_{\pi^*} \geq v_\pi,$$

where v_π is the state value of π, and $\geq$ is an elementwise comparison.

Now, it is clear why we must study the BOE: its solution corresponds to optimal state values and optimal policies. The proof of the above theorem is given in the following box.

Proof of proposition 3.3 For any policy π, it holds that

$$v_\pi = R_\pi + \gamma P_\pi v_\pi.$$

Since

$$v^* = \max_\pi (R_\pi + \gamma P_\pi v^*) = R_{\pi^*} + \gamma P_{\pi^*} v^* \geq R_\pi + \gamma P_\pi v^*,$$

we have

$$v^* - v_\pi \geq (R_\pi + \gamma P_\pi v^*) - (R_\pi + \gamma P_\pi v_\pi) = \gamma P_\pi (v^* - v_\pi).$$

Repeatedly applying the above inequality gives

$$v^* - v_\pi \geq \gamma P_\pi (v^* - v_\pi) \geq \gamma^2 P_\pi^2 (v^* - v_\pi) \geq \cdots \geq \gamma^n P_\pi^n (v^* - v_\pi).$$

It follows that

$$v^* - v_\pi \geq \lim_{n\to\infty} \gamma^n P_\pi^n (v^* - v_\pi) = 0,$$

where the last equality is true because $\gamma < 1$ and P_π^n is a nonnegative matrix with all its elements less than or equal to 1 (because $P_\pi^n \mathbf{1} = \mathbf{1}$). Therefore, $v^* \geq v_\pi$ for any π.

We next examine the π^* in (3.6). In particular, the following theorem shows that there always exists a deterministic greedy policy that is optimal.

Proposition 3.4 (Optimality of Greedy Policy)**.** For any $s \in \mathcal{S}$, the deterministic greedy policy

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

is an optimal policy for solving the BOE. Here,

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

where

$$q^*(s,a) \coloneqq r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v^*(s').$$

Proof of Proposition 3.4 As defined in Definition 3.1, the optimal policy is the one that has the maximum state values among all policies for any state. According to Bellman Equation

(2.4), the state value of a policy is

$$\begin{aligned} v_\pi(s) &= \sum_{a\in\mathcal{A}} \pi(a|s)\Big[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a)v_\pi(s')\Big] \\ &\le \sum_{a\in\mathcal{A}} \pi(a|s)\Big[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a)v^*(s')\Big] \\ &= \sum_{a\in\mathcal{A}} \pi(a|s)q^*(s,a) \\ &\le \max_{a\in\mathcal{A}} q^*(s,a). \end{aligned}$$

The first inequality is obtained since $\max v_\pi(s') = v^*(s')$, and the equality condition is $\pi = \pi^*$ by definition. The second inquality is obtained since the expression is a sum with positive weights (inspired by Example 3.1), and the equality condition is

$$\pi(a|s) = \begin{cases} 1, & a = \arg\max_a q^*(s,a), \\ 0, & a \ne \arg\max_a q^*(s,a). \end{cases}$$

The two equality conditions indicate that to maximize $v_\pi(s)$, we obtain an optimal policy π^*, which deterministically chooses the action with the highest action value.

The policy in Proposition 3.4 is called *greedy* because it seeks the actions with the greatest $q^*(s,a)$.

Finally, there are two important properties of π^*.

- **Uniqueness of optimal policies**: Although the value of v^* is unique, the optimal policy that corresponds to v^* may not be unique.
- **Stochasticity of optimal policies**: An optimal policy can be either stochastic or deterministic. However, it is certain that there always exists a deterministic optimal policy according to Proposition 3.4

3.4 Factors that Influence Optimal Policies

For every $s \in \mathcal{S}$, the elementwise expression of the BOE is

$$v(s) = \max_{\pi(s)\in\Pi(s)} \sum_{a\in\mathcal{A}} \pi(a|s)\Big[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a)v(s')\Big]$$

It is clear that the optimal state value and optimal policy are determined by the following parameters: (1) the immediate reward r; (2) the discount rate γ; (3) the system model $p(s'|s,a)$. While the system model is fixed, we focus on the impact of γ and r.

| Impact of the Discount Rate

The discount rate γ influences the scale of future rewards. When the discount rate is decreased, the optimal policy becomes *short-sighted* due to the relatively small value of future rewards. In the extreme case where $\gamma = 0$, the policy does not dare to take risks anymore, and only favors the action with the highest immediate reward. In contrast, when the discount rate is high, the policy is *far-sighted* and attach more importance to the cumulative reward of a trajectory.

It is notable that the policy would not result in meaningless detours (redundant intermediate states). This is because the discount rate would reduce the scale of the reward of far away future steps. Given optimal intermediate states, it is clear that the shorter the trajectory is, the greater the return is. Therefore, although the immediate reward of every step may not explicitly encourage the agent to approach the target as quickly as possible, the discount rate does encourage it to do so.

| Impact of the Reward Values

One important fact is that optimal policies are **invariant to affine transformations of the rewards**. In other words, if we scale all the rewards by a constant or add the same value to all the rewards, the optimal policy remains the same.

Proposition 3.5 (Optimal Policy Invariance)**.** Consider a Markov decision process with $v^* \in \mathbb{R}^{|\mathcal{S}|}$ as the optimal state value satisfying $v^* = \max_{\pi\in\Pi}(r_\pi + \gamma P_\pi v^*)$. If every reward $r \in \mathcal{R}$ is changed by an affine transformation to $\alpha r + \beta$, where $\alpha, \beta \in \mathbb{R}$ and $\alpha > 0$, then the corresponding optimal state value v' is also an affine transformation of v^*:

$$v' = \alpha v^* + \frac{\beta}{1-\gamma}\mathbf{1}$$

where $\gamma \in (0,1)$ is the discount rate and $\mathbf{1} = [1,\ldots,1]^T$. Consequently, the optimal policy derived from v' is invariant to the affine transformation of the reward values.

4 Value Iteration and Policy Iteration

4.1 Value Iteration

Value iteration is exactly the algorithm suggested by the contraction mapping theorem for solving the Bellman optimality equation.

This algorithm is iterative and has two steps in every iteration.

1. **Policy update**: Find a policy that can solve the following optimization problem:
$$\pi_{k+1} = \arg\max_\pi (R_\pi + \gamma P_\pi v_k),$$
where v_k is obtained in the previous iteration.
2. **Value update**: Calculate a new value v_{k+1} by
$$v_{k+1} = R_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_k,$$
where v_{k+1} will be used in the next iteration.

The value iteration algorithm introduced above is in a matrix-vector form. To implement this algorithm, we need to further examine its elementwise form.

1. **Policy update**:
$$\begin{aligned}\pi_{k+1} &= \arg\max_\pi \sum_a \pi(a|s)\Big[r(s,a) + \gamma\sum_{s'} p(s'|s,a)v_k(s')\Big] \\ &= \arg\max_\pi \sum_a \pi(a|s)q_k(s,a)\end{aligned}$$
As shown before, the optimal policy that can solve the above optimization problem is
$$\pi_{k+1}(a|s) = \begin{cases}1, & a = \arg\max_a q_k(s,a), \\ 0, & a \neq \arg\max_a q_k(s,a),\end{cases}$$

If $a_k^* = \arg\max_a q_k(s,a)$ has multiple solutions, we can select any of them without affecting the convergence of the algorithm.

2. **Value update**:

$$\begin{aligned} v_{k+1}(s) &= \sum_a \pi_{k+1}(a|s)\Big[r(s,a) + \gamma \sum_{s'} p(s'|s,a)v_k(s')\Big] \\ &= \max_a q_k(s,a). \end{aligned}$$

In summary, the above steps can be illustrated as

$$v_k(s) \to q_k(s,a) \to \underset{\text{new greedy policy}}{\pi_{k+1}} \to \underset{\text{new state value}}{v_{k+1}(s)} = \max_a q_k(s,a)$$

A formal algorithm is given below.

Algorithm 1 Value Iteration Algorithm

```
Require: The system model $p(s'|s,a)$
Require: The reward model $r(s,a)$
  Initialize: A guess of initial state values $v_0$
  Initialize: Initial action value function $q_0(s,a)$; initial policy function $\pi_0(a|s)$
  Initialize: $k = 0$
  while $v_k$ has not converged do
      for all state $s \in \mathcal{S}$ do
          for all action $a \in \mathcal{A}$ do
              $q_k(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a)v_k(s')$        ▷ Q-value update
          end for
          $a_k^* = \arg\max_a q_k(s,a)$        ▷ Optimal action
          $\pi_{k+1}(a|s) = \begin{cases} 1, & \text{if } a = a_k^* \\ 0, & \text{otherwise} \end{cases}$        ▷ Policy update
          $v_{k+1}(s) = \max_a q_k(s,a)$        ▷ Value update
      end for
      $k = k + 1$
  end while
```

It is notable that $v_k(s)$ is not the true state value, and $q_k(s,a)$ is not the true action value. They are just approximations.

The convergence criterion can be set as $\|v_k - v_{k-1}\|_2 < \varepsilon$, where ε is the prespecified threshold, a small positive value.

4.2 Policy Iteration

Policy iteration is another iterative algorithm to solve the optimal state values and optimal policy. Each iteration has two steps.

1. **Policy evaluation**: Evaluate a given policy by calculating the corresponding state value. This can be done by solving the following Bellman equation:

$$v_{\pi_k} = R_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k},$$

where π_k is the policy obtained in the last iteration and v_{π_k} is the state value to be calculated. The values of R_{π_k} and P_{π_k} can be obtained from the system model.

2. **Policy improvement**: Once v_{π_k} has been calculated in the first step, a new policy π_{k+1} can be obtained as
$$\pi_{k+1} = \arg\max_{\pi}(R_\pi + \gamma P_\pi v_{\pi_k})$$

The algorithm pseudocode is given below.

Algorithm 2 Policy Iteration Algorithm

```
Require: The system model $p(s'|s,a)$
Require: The reward model $r(s,a)$
  Initialize: A guess of initial policy function $\pi_0(a|s)$; $k = 0$.
  while $v_{\pi_k}$ has not converged do
      $\boxed{\textbf{Policy Evaluation:}}$
      Initialize: An arbitrary guess $v_{\pi_k}^{(0)}$; $j = 0$
      while $v_{\pi_k}^{(j)}$ has not converged do
          for all state $s \in \mathcal{S}$ do
              $v_{\pi_k}^{(j+1)}(s) = \sum_a \pi_k(a|s)\left[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a) v_{\pi_k}^{(j)}(s')\right]$
          end for
          $j = j + 1$
      end while
      $\boxed{\textbf{Policy Improvement:}}$
      for all state $s \in \mathcal{S}$ do
          for all action $a \in \mathcal{A}$ do
              $q_{\pi_k}(s,a) = r(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a) v_{\pi_k}(s')$
          end for
          $a_k^* = \arg\max_a q_{\pi_k}(s,a)$
          $\pi_{k+1}(a|s) = \begin{cases} 1, & \text{if } a = a_k^* \\ 0, & \text{otherwise} \end{cases}$
      end for
      $k = k + 1$
  end while
```

| How to Calculate v_{π_k}?

According to the Bellman equation $v_{\pi_k} = R_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}$, a natural closed-form solution is $v_{\pi_k} = (I - \gamma P_{\pi_k})^{-1} r_{\pi_k}$. However, this is inefficient to implement since it requires other numerical algorithms to compute the matrix inverse. A more adopted algorithm is an iterative one that can be easily implemented:
$$v_{\pi_k}^{(j+1)} = R_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}^{(j)}, \quad j = 0, 1, 2, \ldots$$
where $v_{\pi_k}^{(j)}$ denotes the j-th estimate of v_{π_k}. Starting from any initial guess $v_{\pi_k}^{(0)}$, it is ensured that $v_{\pi_k}^{(j)} \to v_{\pi_k}$ as $j \to \infty$.

| Why Is π_{k+1} Better Than π_k?

The policy improvement step can truly improve the given policy. This is guaranteed by the following theorem.

Proposition 4.1 (Policy Improvement)**.** If $\pi_{k+1} := \arg\max_\pi(r_\pi + \gamma P_\pi v_{\pi_k})$, then $v_{\pi_{k+1}} \geq v_{\pi_k}$.

Here, $v_{\pi_{k+1}} \geq v_{\pi_k}$ means that $v_{\pi_{k+1}}(s) \geq v_{\pi_k}(s)$ for all s.

Proof of Proposition 4.1 Since $v_{\pi_{k+1}}$ and v_{π_k} are state values, they satisfy the Bellman equations:

$$\begin{aligned} v_{\pi_{k+1}} &= R_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}, \\ v_{\pi_k} &= R_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}. \end{aligned}$$

Since $\pi_{k+1} = \arg\max_\pi(R_\pi + \gamma P_\pi v_{\pi_k})$, we know that

$$R_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k} \geq R_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}.$$

It then follows that

$$\begin{aligned} v_{\pi_k} - v_{\pi_{k+1}} &= (R_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}) - (R_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}) \\ &\leq (R_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k}) - (R_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}) \\ &\leq \gamma P_{\pi_{k+1}} (v_{\pi_k} - v_{\pi_{k+1}}). \end{aligned}$$

Therefore,

$$\begin{aligned} v_{\pi_k} - v_{\pi_{k+1}} &\leq \gamma^2 P^2_{\pi_{k+1}} (v_{\pi_k} - v_{\pi_{k+1}}) \leq \dots \\ &\leq \gamma^n P^n_{\pi_{k+1}} (v_{\pi_k} - v_{\pi_{k+1}}) \\ &\leq \lim_{n\to\infty} \gamma^n P^n_{\pi_{k+1}} (v_{\pi_k} - v_{\pi_{k+1}}) = 0. \end{aligned}$$

The limit is due to the facts that $\gamma^n \to 0$ as $n \to \infty$ and $P^n_{\pi_{k+1}}$ is a matrix where all entries are positive and all rows sum to one.

| Why Can the Policy Iteration Algorithm Eventually Find an Optimal Policy?

The policy iteration algorithm generates a sequence of state values: $\{v_{\pi_0}, v_{\pi_1}, \dots, v_{\pi_k}, \dots\}$. Suppose that v^* is the optimal state value. Then, $v_{\pi_k} \leq v^*$ for all k. Since the policies are continuously improved according to Proposition 4.1, we know that

$$v_{\pi_0} \leq v_{\pi_1} \leq v_{\pi_2} \leq \cdots \leq v_{\pi_k} \leq \cdots \leq v^*.$$

Since v_{π_k} is nondecreasing and always bounded from above by v^*, it follows from the monotone convergence theorem that v_{π_k} converges to a constant value, denoted as v_∞, when $k \to \infty$. The fact that $v_\infty = v^*$ is guaranteed by the following theorem.

Proposition 4.2 (Convergence of Policy Iteration)**.** The state value sequence $\{v_{\pi_k}\}_{k=0}^\infty$ generated by the policy iteration algorithm converges to the optimal state value v^*. As a result, the policy sequence $\{\pi_k\}_{k=0}^\infty$ converges to an optimal policy.

4.3 Truncated Policy Iteration

The value iteration and policy iteration algorithms are two special cases of the **truncated policy iteration algorithm**.

| Comparing Value Iteration and Policy Iteration

- **Policy iteration**: Select an arbitrary initial policy π_0. In the k-th iteration, do the following two steps.
  1. **Policy evaluation (PE)**: Given π_k, solve v_{π_k} from
  $$v_{\pi_k} = R_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}.$$
  2. **Policy improvement (PI)**: Given v_{π_k}, solve π_{k+1} from
  $$\pi_{k+1} = \arg\max_{\pi}(R_\pi + \gamma P_\pi v_{\pi_k}).$$
- **Value iteration**: Select an arbitrary initial value v_0. In the kth iteration, do the following two steps.
  1. **Policy update (PU)**: Given v_k, solve π_{k+1} from
  $$\pi_{k+1} = \arg\max_{\pi}(R_\pi + \gamma P_\pi v_k).$$
  2. **Value update (VU)**: Given π_{k+1}, solve v_{k+1} from
  $$v_{k+1} = R_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_k.$$

The above steps of the two algorithms can be illustrated as

$$\begin{tikzcd}
\text{Policy iteration:} & \pi_0 \arrow[r, "PE"] & v_{\pi_0} \arrow[r, "PI"] & \pi_1 \arrow[r, "PE"] & v_{\pi_1} \arrow[r, "PI"] & \pi_2 \arrow[r, "PE"] & v_{\pi_2} \arrow[r, "PI"] & \cdots \\
\text{Value iteration:} & & v_0 \arrow[r, "PU"] & \pi_1' \arrow[r, "VU"] & v_1 \arrow[r, "PU"] & \pi_2' \arrow[r, "VU"] & v_2 \arrow[r, "PU"] & \cdots
\end{tikzcd}$$

	Policy iteration algorithm	**Value iteration algorithm**	**Comments**
1) Policy:	π_0	N/A	
2) Value:	$v_{\pi_0} = r_{\pi_0} + \gamma P_{\pi_0} v_{\pi_0}$	$v_0 \doteq v_{\pi_0}$	
3) Policy:	$\pi_1 = \arg\max_\pi(r_\pi + \gamma P_\pi v_{\pi_0})$	$\pi_1 = \arg\max_\pi(r_\pi + \gamma P_\pi v_0)$	The two policies are the same
4) Value:	$v_{\pi_1} = r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}$	$v_1 = r_{\pi_1} + \gamma P_{\pi_1} v_0$	$v_{\pi_1} \geq v_1$ since $v_{\pi_1} \geq v_{\pi_0}$
5) Policy:	$\pi_2 = \arg\max_\pi(r_\pi + \gamma P_\pi v_{\pi_1})$	$\pi_2' = \arg\max_\pi(r_\pi + \gamma P_\pi v_1)$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 4.1: A comparison between the implementation steps of policy iteration and value iteration.

It can be seen that the procedures of the two algorithms are very similar.

The following observations can be obtained from the above process.

- If the iteration is run *only once*, then $v_{\pi_1}^{(1)}$ is actually v_1, as calculated in the value iteration algorithm.
- If the iteration is run *an infinite number of times*, then $v_{\pi_1}^{(\infty)}$ is actually v_{π_1}, as calculated in the policy iteration algorithm.
- If the iteration is run *a finite number of times* (denoted as j_{truncate}), then such an algorithm is called *truncated policy iteration*.

As a result, the value iteration and policy iteration algorithms can be viewed as two extreme cases of the truncated policy iteration algorithm: value iteration terminates at $j_{\text{truncate}} = 1$, and policy iteration terminates at $j_{\text{truncate}} = \infty$.

It should be noted that, although the above comparison is illustrative, it is based on the condition that $v_{\pi_1}^{(0)} = v_0 = v_{\pi_0}$. The two algorithms cannot be directly compared without this condition.

| Truncated Policy Iteration Algorithm

Algorithm 3 Truncated Policy Iteration Algorithm

```
Require: The system model $p(s'|s,a)$
Require: The reward model $r(s,a)$
  Initialize: A guess of initial policy function $\pi_0(a|s)$; $k = 0$.
  while $v_k$ has not converged do
      Policy Evaluation:
      Initialize: Set $v_k^{(0)} = v_{k-1}$
      Initialize: Set the maximum number of iterations $j_{\text{truncate}}$; $j = 0$
      while $j < j_{\text{truncate}}$ do
          for all state $s \in \mathcal{S}$ do
              $v_k^{(j+1)}(s) = \sum_a \pi_k(a|s)\left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v_k^{(j)}(s')\right]$
          end for
          $j = j + 1$
      end while
      $v_k = v_k^{j_{\text{truncate}}}$
      Policy Improvement:
      for all state $s \in \mathcal{S}$ do
          for all action $a \in \mathcal{A}$ do
              $q_{\pi_k}(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v_{\pi_k}(s')$
          end for
          $a_k^* = \arg\max_a q_k(s,a)$
          $\pi_{k+1}(a|s) = \begin{cases} 1, & \text{if } a = a_k^* \\ 0, & \text{otherwise} \end{cases}$
      end for
      $k = k + 1$
  end while
```

Up to now, the advantages of truncated policy iteration are clear.

- Compared to the policy iteration algorithm, the truncated one merely requires a finite number of iterations in the policy evaluation step and hence is more computationally efficient.
- Compared to value iteration, the truncated policy iteration algorithm can speed up its convergence rate by running for a few more iterations in the policy evaluation step.

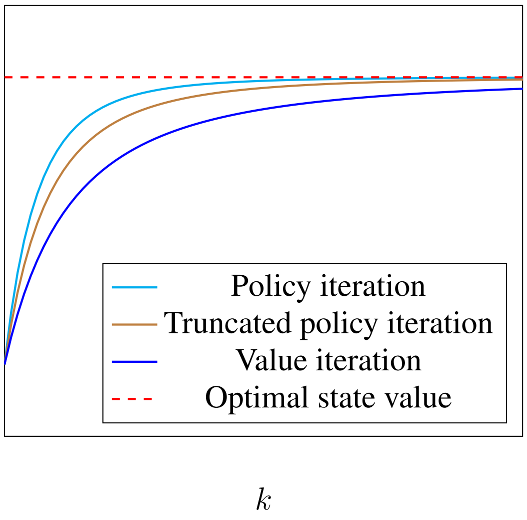

Figure 4.1: Illustration of the convergence speed of value iteration, policy iteration, and truncated policy iteration.

Finally, the idea of interaction between value and policy updates widely exists in reinforcement learning algorithms. This idea is named *generalized policy iteration*.

5 Monte Carlo Based Model-free Reinforcement Learning

In the previous sections, the algorithms to find optimal policies are based on the *system model*. In this section, we start topics on **model-free reinforcement learning** algorithms that do not presume system models.

How can we find optimal policies without models? The philosophy is simple: If we do not have a model, we must have some data. If we do not have data, we must have a model. If we have neither, then we are not able to find optimal policies. The data in reinforcement learning usually refers to the agent's interaction experiences with the environment.

5.1 Basic MC-based Algorithm

| Converting Policy Iteration to be Model-free

There are two steps in every iteration of the policy iteration algorithm (Algorithm 2). The first step is *policy evaluation*, which aims to compute v_{π_k} by solving $v_{\pi_k} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}$. The second step is *policy improvement*, which aims to compute the greedy policy $\pi_{k+1} = \arg\max_\pi (r_\pi + \gamma P_\pi v_{\pi_k})$. The elementwise form of the policy improvement step is

$$\begin{aligned}\pi_{k+1}(s) &= \arg\max_\pi \sum_a \pi(a|s)\Big[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_{\pi_k}(s')\Big] \\ &= \arg\max_\pi \sum_a \pi(a|s) q_{\pi_k}(s,a).\end{aligned}$$

It must be noted that the action values lie in the core of these two steps. Specifically, in the first step, the state values are calculated for the purpose of calculating the action values. In the second step, the new policy is generated based on the calculated action values. Let us reconsider how we can calculate the action values. Two approaches are available.

1. **Model-based approach**: This is the approach adopted by the policy iteration algorithm. In particular, we can first calculate the state value v_{π_k} by solving the Bellman equation. Then, we can calculate the action values by using

$$q_{\pi_k}(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) v_{\pi_k}(s').] \tag{5.1}$$

   This approach requires the system model $\{p(r|s,a), p(s'|s,a)\}$ to be known.

2. **Model-free Approach**: Recall that the definition of an action value is

$$\begin{aligned} q_{\pi_k}(s,a) &= \mathbb{E}[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots |S_t = s, A_t = a], \end{aligned}$$

   which is the expected return obtained when starting from (s,a). Since $q_{\pi_k}(s,a)$ is an expectation, it can be estimated by Monte Carlo methods. To do that, starting from (s,a), the agent can interact with the environment by following policy π_k and then obtain a certain number of episodes. Suppose that there are n episodes and that the return of the ith episode is $g^{(i)}_{\pi_k}(s,a)$. Then, $q_{\pi_k}(s,a)$ can be approximated as

$$q_{\pi_k}(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a] \approx \frac{1}{n}\sum_{i=1}^{n} g^{(i)}_{\pi_k}(s,a). \tag{5.2}$$

   If the number of episodes (n) is sufficiently large, the approximation will be sufficiently accurate according to the law of large numbers.

The fundamental idea of MC-based reinforcement learning is to use a model-free method for estimating action values, as shown in (5.2), to replace the model-based method in the policy iteration algorithm, as shown in (5.1).

We are now ready to present the first MC-based reinforcement learning algorithm. Starting from an initial policy π_0, the algorithm has two steps in the k-th iteration ($k = 0, 1, 2, \ldots$).

1. **Policy evaluation**: This step is used to estimate $q_{\pi_k}(s,a)$ for all (s,a). Specifically, for every (s,a), we collect sufficiently many episodes and use the average of the returns, denoted as $q_k(s,a)$, to approximate $q_{\pi_k}(s,a)$.

2. **Policy improvement**: This step solves $\pi_{k+1}(s) = \arg\max_\pi \sum_a \pi(a|s) q_k(s,a)$ for all $s \in \mathcal{S}$. The greedy optimal policy is $\pi_{k+1}(a^*_k|s) = 1$ where $a^*_k = \arg\max_a q_k(s,a)$.

Algorithm 4 MC Basic Algorithm (a model-free variant of policy iteration)

```
Initialize: A guess of initial policy function $\pi_0(a|s)$; $k = 0$.
for the $k$-th iteration do
    for all state $s \in \mathcal{S}$ do
        $\boxed{\textbf{Policy Evaluation:}}$
        for all action $a \in \mathcal{A}$ do
            Collect sufficiently many episodes starting from $(s, a)$ by following $\pi_k$
            Compute the returns of each episode, $g^{(i)}_{\pi_k}(s, a)$
            $q_{\pi_k}(s, a) = \frac{1}{n}\sum_{i=1}^{n} g^{(i)}_{\pi_k}(s, a)$
        end for
        $\boxed{\textbf{Policy Improvement:}}$
        $a^*_k = \arg\max_a q_{\pi_k}(s, a)$
        $\pi_{k+1}(a|s) = \begin{cases} 1, & \text{if } a = a^*_k \\ 0, & \text{otherwise} \end{cases}$
    end for
    $k = k + 1$
end for
```

We name the above algorithm as MC Basic. Since policy iteration is convergent, MC Basic is also convergent when given sufficient samples. That is, for every (s, a), suppose that there are sufficiently many episodes starting from (s, a). Then, the average of the returns of these episodes can accurately approximate the action value of (s, a).

In practice, we usually do not have sufficient episodes for every (s, a). As a result, the approximation of the action values may not be accurate. Nevertheless, the algorithm usually can still work. This is similar to the truncated policy iteration algorithm, where the action values are neither accurately calculated.

The MC basic is very similar to policy iteration algorithm. The only difference is that it calculates action values directly from experience samples, whereas policy iteration calculates state values first and then calculates the action values based on the system model. It should be noted that the model-free algorithm directly estimates action values. Otherwise, if it estimates state values instead, we still need to calculate action values from these state values using the system model.

| Episode Length and Sparse Rewards

In some reinforcement learning settings, the reward is *sparse*. The **sparse reward** refers to the scenario in which no positive rewards can be obtained unless the target is reached.

Sparse rewards may downgrade the efficiency of model-free reinforcement learning. When using Monte Carlo method, we are generating multiple episodes and averaging their results. In the scenario of sparse reward, if the episode is not long enough, the agent is less likely to reach the target, so most episodes would not receive meaningful feedback, and no information could be gained. However, when the state space is large, it is challenging to generate episodes that is long enough due to computation efficiency.

One simple solution is to design nonsparse rewards. For instance, we can redesign the reward setting so that the agent can obtain a small positive reward when reaching the states near the target. In this way, an *"attractive field"* can be formed around the target state so that the agent can gain information about the target more easily.

| Utilizing Samples More Efficiently

An important aspect of MC-based reinforcement learning is how to use samples more efficiently. Specifically, suppose that we have an episode of samples obtained by following a policy :

$$s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_4} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \cdots \tag{5.3}$$

where the subscripts refer to the state or action indexes rather than time steps. Every time a state-action pair appears in an episode, it is called a **visit** of that state-action pair. Different strategies can be employed to utilize the visits.

The first and simplest strategy is to use the *initial visit*. That is, an episode is only used to estimate the action value of the initial state-action pair. For the example in (5.3), the initial-visit strategy merely estimates the action value of (s_1, a_2). The MC Basic algorithm utilizes the initial-visit strategy.

However, this strategy is *not sample-efficient* because the episode also visits many other state-action pairs such as (s_2, a_4), (s_2, a_3), and (s_5, a_1). These visits can also be used to estimate the corresponding action values. In particular, we can decompose the episode in (5.3) into multiple subepisodes:

$$\begin{aligned}
s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_4} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \cdots \quad & \text{[original episode]} \\
s_2 \xrightarrow{a_4} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_3 \xrightarrow{a_1} \cdots \quad & \text{[subepisode starting from } (s_2, a_4)\text{]} \\
s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \cdots \quad & \text{[subepisode starting from } (s_1, a_2)\text{]} \\
s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \cdots \quad & \text{[subepisode starting from } (s_2, a_3)\text{]} \\
s_5 \xrightarrow{a_1} \cdots \quad & \text{[subepisode starting from } (s_5, a_1)\text{]}
\end{aligned}$$

The trajectory generated after the visit of a state-action pair can be viewed as a new episode, which can be used to estimate more action values. In this way, the samples in the episode can be utilized more efficiently.

Moreover, a state-action pair may be visited multiple times in an episode. For exam ple, (s_1, a_2) is visited twice in the episode in (5.3). If we only count the first-time visit, this is called a *first-visit strategy*. If we count every visit of a state-action pair, such a strategy is called *every-visit*. In terms of sample usage efficiency, the every-visit strategy is the best. If an episode is sufficiently long such that it can visit all the state-action pairs many times, then this single episode may be sufficient for estimating all the action values using the every-visit strategy.

It is worth noting that the samples obtained by the every-visit strategy are correlated because the trajectories overlap. Nevertheless, the correlation would not be strong if the two visits are far away from each other in the trajectory.

| Updating Policies More Efficiently

Another aspect of MC-based reinforcement learning is when to update the policy. Two strategies are available.

1. The first strategy is, in the policy evaluation step, to collect all the episodes starting from the same state-action pair and then approximate the action value using the average return of these episodes. This strategy is adopted in the MC Basic algorithm.

   The drawback of this strategy is that the agent must wait until all the episodes have been collected before the estimate can be updated.

2. The second strategy is to use the return of a single episode to approximate the corresponding action value. In this way, we can immediately obtain a rough estimate when we receive an episode. Then, the policy can be improved in an episode-by-episode fashion.

Since the return of a single episode cannot accurately approximate the corresponding action value, one may wonder whether the second strategy is good. In fact, this strategy falls into the scope of *generalized policy iteration*. That is, we can still update the policy even if the value estimate is not sufficiently accurate.

| MC Exploring Starts Algorithm

Combining the techniques of boosting efficiency, we get the **MC exploring starts** algorithm, as shown in 5.

Algorithm 5 MC Exploring Starts (an efficient variant of MC Basic)

```
Initialize: Initial policy $\pi_0(a|s)$; initial action value $q(s,a)$ for all $(s,a)$.
Initialize: $\text{Returns}(s,a) = 0$, $\text{Nums}(s,a) = 0$, for all $(s,a)$.
Goal: Search for an optimal policy.
for each episode do
    Episode generation:
    Select a starting state-action pair $(s_0, a_0)$
    Ensure all state-action pairs can be possibly selected.        ▷ exploring-starts condition
    Follow the curreny policy, generate an episode of length $T$:
        $(s_0, a_0, r_1), (s_1, a_1, r_2), \dots, (s_{T-1}, a_{T-1}, r_T)$.
    Initialize for each episode: $g \leftarrow 0$
    for each step of the episode, $t = T-1, T-2, \dots, 0$ do        ▷ Iterate in an inverse order
        $g \leftarrow \gamma g + r_{t+1}$
        $\text{Returns}(s_t, a_t) \leftarrow \text{Returns}(s_t, a_t) + g$        ▷ Total return starting from $(s_t, a_t)$
        $\text{Nums}(s_t, a_t) \leftarrow \text{Nums}(s_t, a_t) + 1$        ▷ Number of episodes starting from $(s_t, a_t)$
        Policy Evaluation:
        $q(s_t, a_t) \leftarrow \text{Returns}(s_t, a_t)/\text{Nums}(s_t, a_t)$
        Policy Improvement:
        $a^* = \arg\max_a q(s_t, a)$
        $\pi(a|s_t) = \begin{cases} 1, & \text{if } a = a^* \\ 0, & \text{otherwise} \end{cases}$
    end for
end for
```

Interestingly, when calculating the discounted return obtained by starting from each state-action pair, the procedure starts from the ending states and travels back to the starting state.

Note that the *exploring starts condition* requires sufficiently many episodes starting from every state-action pair. Only if every state-action pair is well explored, can we accurately estimate their action values and hence successfully find optimal policies.

5.2 ϵ-greedy MC Algorithm

The MC Exploring Starts algorithm can be extended by removing the exploring starts condition. This condition actually requires that every state-action pair can be visited sufficiently many times, which can also be achieved based on *soft policies*[3]. One type of common soft policies is ϵ-greedy policies[4].

[3]A policy is *soft* if it has a positive probability of taking any action at any state.

[4]An ϵ-greedy policy is a stochastic policy that has a higher chance of choosing the greedy action and the same nonzero probability of taking any other action. Here, the greedy action refers to the action with the greatest action value.

In particular, suppose $\epsilon \in [0,1]$. The corresponding ϵ-greedy policy has the following form[5]:

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

where $|\mathcal{A}|$ denotes the number of actions.

To integrate ϵ-greedy policies into MC learning, we only need to change the policy improvement step from greedy to ϵ-greedy.

In particular, the policy improvement step in MC Basic or MC Exploring Starts aims to solve

$$\pi_{k+1} = \arg\max_{\pi \in \Pi} \sum_a \pi(a|s) q_{\pi_k}(s,a),$$

where Π denotes the set of all possible policies. We know that the solution is a greedy policy:

$$\pi_{k+1}(a|s) = \begin{cases} 1, & a = \arg\max_a q_{\pi_k}(s,a), \\ 0, & a \neq \arg\max_a q_{\pi_k}(s,a), \end{cases}$$

Now, the policy improvement step is changed to solve

$$\pi_{k+1} = \arg\max_{\pi \in \Pi_\epsilon} \sum_a \pi(a|s) q_{\pi_k}(s,a),$$

where Π_ϵ denotes the set of all ϵ-greedy policies with a given value of ϵ. In this way, we force the policy to be ϵ-greedy. The solution is

$$\pi_{k+1}(a|s) = \begin{cases} 1 - \dfrac{|\mathcal{A}(s)| - 1}{|\mathcal{A}(s)|}\epsilon, & a = \arg\max_a q_{\pi_k}(s,a), \\ \dfrac{1}{|\mathcal{A}(s)|}\epsilon, & a \neq \arg\max_a q_{\pi_k}(s,a), \end{cases}$$

With the above change, we obtain a new algorithm called **MC ϵ-Greedy** (Algorithm 6).

[5]The probability of taking the greedy action is always greater than that of taking any other actions because

$$1 - \frac{\epsilon}{|\mathcal{A}|}(|\mathcal{A}| - 1) = 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} \geq \frac{\epsilon}{|\mathcal{A}|}$$

for any $\epsilon \in [0,1]$.

Algorithm 6 MC-Greedy (a variant of MC Exploring Starts)

```
Initialize: Initial policy $\pi_0(a|s)$; initial action value $q(s,a)$ for all $(s,a)$.
Initialize: $\mathrm{Returns}(s,a) = 0$, $\mathrm{Nums}(s,a) = 0$, for all $(s,a)$.
Require: $\epsilon \in [0,1]$
Goal: Search for an optimal policy.
for each episode do
    Episode generation:
    Select a starting state-action pair $(s_0, a_0)$    ▷ The exploring-starts condition is not required
    Follow the curreny policy, generate an episode of length $T$:
      $(s_0, a_0, r_1), (s_1, a_1, r_2), \dots, (s_{T-1}, a_{T-1}, r_T)$.
    Initialize for each episode: $g \leftarrow 0$
    for each step of the episode, $t = T-1, T-2, \dots, 0$ do    ▷ Iterate in an inverse order
        $g \leftarrow \gamma g + r_{t+1}$
        $\mathrm{Returns}(s_t, a_t) \leftarrow \mathrm{Returns}(s_t, a_t) + g$    ▷ Total return starting from $(s_t, a_t)$
        $\mathrm{Nums}(s_t, a_t) \leftarrow \mathrm{Nums}(s_t, a_t) + 1$    ▷ Number of episodes starting from $(s_t, a_t)$
        Policy Evaluation:
        $q(s_t, a_t) \leftarrow \mathrm{Returns}(s_t, a_t)/\mathrm{Nums}(s_t, a_t)$
        Policy Improvement:
        $a^* = \arg\max_a q(s_t, a)$
        $\pi(a|s_t) = \begin{cases} 1 - \frac{|\mathcal{A}|-1}{\mathcal{A}}\epsilon, & \text{if } a = a^* \\ \frac{1}{|\mathcal{A}|}\epsilon, & \text{otherwise} \end{cases}$    ▷ The only change from MC Exploring Starts
    end for
end for
```

If greedy policies are replaced by ϵ-greedy policies in the policy improvement step, can we still guarantee to obtain optimal policies? When given sufficient samples, the algorithm can converge to an ϵ-greedy policy that is optimal in the set Π_ϵ. However, the policy may not be optimal in Π. Nevertheless, if ϵ is sufficiently small, the optimal policies in Π_ϵ are close to those in Π.

| Exploration and Exploitation

Exploration and exploitation constitute a fundamental tradeoff in reinforcement learning. **Exploration** means that the policy can possibly take as many actions as possible. In this way, all the actions can be visited and evaluated well. **Exploitation** means that the improved policy should take the greedy action that has the greatest action value. However, since the action values obtained at the current moment may not be accurate due to insufficient exploration, we should keep exploring while conducting exploitation to avoid missing optimal actions.

ϵ-greedy policies provide one way to balance exploration and exploitation. On the one hand, an ϵ-greedy policy has a higher probability of taking the greedy action so that it can exploit the estimated values. On the other hand, the ϵ-greedy policy also has a chance to take other actions so that it can keep exploring.

Exploitation is related to optimality because optimal policies should be greedy. The fundamental idea of ϵ-greedy policies is to enhance exploration by sacrificing optimality/exploitation. If we would like to enhance exploitation and optimality, we need to reduce the value of ϵ. However, if we would like to enhance exploration, we need to increase the value of ϵ.

6 Stochastic Approximation

6.1 Mean Estimation Problem

Consider a random variable X that takes values from a finite set $\mathcal{X}$. Our goal is to estimate $\mathbb{E}[X]$. Suppose that we have a sequence of i.i.d. samples $\{x_i\}_{i=1}^n$. The expected value of X can be approximated by

$$\mathbb{E}[X] \approx \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{6.1}$$

The approximation in (6.1) is the basic idea of Monte Carlo estimation. According to the law of large numbers, $\bar{x} \to \mathbb{E}[X]$ as $n \to \infty$. The above method is a *non-incremental* method, which collects all the samples first and then calculates the average. The drawback is that if the number of samples is large, we may have to wait for a long time until all of the samples are collected. To avoid this drawback, an *incremental* method can be applied. Specifically, suppose that

$$w_{k+1} \coloneqq \sum_{i=1}^{k} x_i, \quad k = 1, 2, \ldots,$$

and hence

$$w_k \coloneqq \sum_{i=1}^{k-1} x_i, \quad k = 2, 3, \ldots.$$

Therefore, we obtain the following incremental algorithm[6]

$$w_{k+1} = w_k - \frac{1}{k}(w_k - x_k) \tag{6.2}$$

Thus, by (6.2), $\bar{x}$ can be calculated in an incremental manner. Every time we receive a sample, the mean estimation can be immediately updated. As more samples are obtained, the estimation accuracy can be gradually improved according to the law of large numbers.

Furthermore, consider an algorithm with a more general expression:

$$w_{k+1} = w_k - \alpha_k(w_k - x_k) \tag{6.3}$$

It can be shown that if α_k satisfies some mild conditions, $w_k \to \mathbb{E}[X]$ as $k \to \infty$. The details are discussed in the following sections.

6.2 Robbins-Monro Algorithm

Stochastic approximation refers to a broad class of stochastic iterative algorithms for solving root-nding or optimization problems. Compared to many other root-finding algorithms such as gradient-based ones, stochastic approximation is powerful in the sense that it does not require the expression of the objective function or its derivative.

- **Black-box root-finding problem**: Suppose that we would like to find the root of the equation

$$g(w) = 0,$$

[6]
$$w_{k+1} = \frac{1}{k}\sum_{i=1}^{k} x_i = \frac{1}{k}\Big(\sum_{i=1}^{k-1} x_i + x_k\Big) = \frac{1}{k}\Big((k-1)w_k + x_k\Big) = w_k - \frac{1}{k}(w_k - x_k)$$

where $w \in \mathbb{R}$ is the unknown variable and $g : \mathbb{R} \to \mathbb{R}$ is a function. If the expression of g or its derivative is known, there are many numerical algorithms that can be used.

In practice, the expression of the function g is usually unknown. Moreover, we can only obtain a noisy observation of $g(w)$:

$$\tilde{g}(w, \eta) = g(w) + \eta,$$

where $\eta \in \mathbb{R}$ is the observation error, which may or may not be Gaussian.

In summary, what we have is a black-box system where only the input w and the noisy output $\tilde{g}(w, \eta)$ are known. Our aim is to solve $g(w) = 0$ merely from the inputs and outputs.

- **Robbins-Monro algorithm**: To solve the black-box root-finding problem, we have the following iterative algorithm:

$$w_{k+1} = w_k - a_k \tilde{g}(w_k, \eta_k), \quad k = 1, 2, 3, \ldots. \tag{6.4}$$

  Here, w_k is the k-th estimate of the root, $\tilde{g}(w_k, \eta_k)$ is the k-th noisy observation, and a_k is a positive coefficient. As can be seen, the RM algorithm does not require any information about the function. It only requires the inputs and outputs. Detailed analyses are given in the following sections.

| Convergence Analysis

Proposition 6.1 (Robbins-Monro Theorem)**.** For the Robbins-Monro algorithm in (6.4), w_k almost surely converges to the root w^* satisfying $g(w^*) = 0$ if the following conditions are satisfied:

(a)

$$0 < c_1 \le \nabla_w g(w) \le c_2 \text{ for all } w;$$

(b)

$$\sum_{k=1}^{\infty} a_k = \infty, \qquad \sum_{k=1}^{\infty} a_k^2 < \infty;$$

(c)

$$\mathbb{E}[\eta_k | \mathcal{H}_k] = 0, \qquad \mathbb{E}[\eta_k^2 | \mathcal{H}_k] < \infty;$$

where $\mathcal{H}_k = \{w_k, w_{k-1}, \ldots\}$.

Analysis of the three conditions is given below.

(a) In the first condition, $0 < c_1 \le \nabla_w g(w)$ indicates that $g(w)$ is a monotonically increasing function. This condition ensures that the root of $g(w) = 0$ exists and is unique.[7][8]

The inequality $\nabla_w g(w) \le c_2$ indicates that the gradient of $g(w)$ is bounded from above.

(b) Some insightful intuitions about the second condition are given below.

(1) $\sum_{k=1}^{\infty} a_k^2 < \infty$ indicates that $a_k \to 0$ as $k \to \infty$, which is a necessary condition for convergence. Suppose that the observation $\tilde{g}(w_k, \eta_k)$ is always bounded. Since

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

[7] If $g(w)$ is monotonically decreasing, we can simply treat $-g(w)$ as a new function that is monotonically increasing.

[8] As an application, we can formulate an optimization problem in which the objective function is $J(w)$ as a root-finding problem: $g(w) := \nabla_w J(w) = 0$. In this case, the condition that $g(w)$ is monotonically increasing indicates that $J(w)$ is convex, which is a commonly adopted assumption in optimization problems.

if $a_k \to 0$, then $a_k\tilde{g}(w_k, \eta_k) \to 0$ and hence $w_{k+1} - w_k \to 0$, indicating that w_{k+1} and w_k approach each other when $k \to \infty$. Otherwise, if a_k does not converge, then w_k may still fluctuate when $k \to \infty$.[9]

(2) $\sum_{k=1}^{\infty} a_k = \infty$ indicates that a_k should not converge to zero too fast. Summarizing both sides of the equations of $w_2 - w_1 = -a_1\tilde{g}(w_1, \eta_1)$, $w_3 - w_2 = -a_2\tilde{g}(w_2, \eta_2)$, $w_4 - w_3 = -a_3\tilde{g}(w_3, \eta_3)$, . . ., we have

$$w_1 - w_\infty = \sum_{k=1}^{\infty} a_k\tilde{g}(w_k, \eta_k).$$

If $\sum_{k=1}^{\infty} a_k < \infty$, then $|\sum_{k=1}^{\infty} a_k\tilde{g}(w_k, \eta_k)|$ is also bounded. Let b denote the finite upper bound such that

$$|w_1 - w_\infty| = \left|\sum_{k=1}^{\infty} a_k\tilde{g}(w_k, \eta_k)\right| \le b.$$

If the initial guess w_1 is selected far away from w^* such that $|w_1 - w^*| > b$, then it is impossible to have $w_\infty = w^*$. Therefore, the condition $\sum_{k=1}^{\infty} a_k = \infty$ is necessary to ensure convergence given an arbitrary initial guess.

(c) The third condition is mild. It does not require the observation error η_k to be Gaussian. An important special case is that $\{\eta_k\}$ is an i.i.d. stochastic sequence satisfying $\mathbb{E}[\eta_k] = 0$ and $\mathbb{E}[\eta_k^2] < \infty$. In this case, the third condition is valid because η_k is independent of $\mathcal{H}_k$ and hence we have $\mathbb{E}[\eta_k|\mathcal{H}_k] = \mathbb{E}[\eta_k] = 0$ and $\mathbb{E}[\eta_k^2|\mathcal{H}_k] = \mathbb{E}[\eta_k^2]$.

| Application to Mean Estimation

The Robbins-Monro theorem can be applied to the mean estimation problem. Recall that

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

is the mean estimation algorithm in (6.3). When $\alpha_k = 1/k$, we can obtain the analytical expression of w_{k+1} as $w_{k+1} = \frac{1}{k}\sum_{i=1}^{k} x_i$. For general values of α_k, we can show that the algorithm in this case is a special RM algorithm and hence its convergence naturally follows.

In particular, define a function as

$$g(w) := w - \mathbb{E}[X].$$

The original problem is to obtain the value of $\mathbb{E}[X]$. This problem is formulated as a root-finding problem to solve $g(w) = 0$. Given a value of w, the noisy observation that we can obtain is $\tilde{g} := w - x$, where x is a sample of X. Note that $\tilde{g}$ can be written as

$$\begin{aligned}\tilde{g}(w, \eta) &= w - x \\ &= w - x + \mathbb{E}[X] - \mathbb{E}[X] \\ &= (w - \mathbb{E}[X]) + \underbrace{(\mathbb{E}[X] - x)}_{:=\eta} := g(w) + \eta,\end{aligned}$$

The RM algorithm for solving this problem is

$$w_{k+1} = w_k - \alpha_k\tilde{g}(w_k, \eta_k) = w_k - \alpha_k(w_k - x_k),$$

which is exactly the algorithm in (6.3). As a result, it is guaranteed by Theorem 6.1 that w_k converges to $\mathbb{E}[X]$ almost surely if $\sum_{k=1}^{\infty} \alpha_k = \infty$, $\sum_{k=1}^{\infty} \alpha_k^2 < \infty$, and $\{x_k\}$ is i.i.d.[10]

[9]In the RM algorithm, a_k is often selected as a sufficiently small constant in many applications. Although the second condition is not satisfied anymore in this case because $\sum_{k=1}^{\infty} a_k^2 = \infty$ rather than $\sum_{k=1}^{\infty} a_k^2 < \infty$, the algorithm can still converge in a certain sense.

[10]It is worth mentioning that the convergence property does not rely on any assumption regarding the distribution of X.

6.3 Stochastic Gradient Descent

This section introduces stochastic gradient descent (SGD) algorithms. We will see that SGD is a special RM algorithm, and the mean estimation algorithm is a special SGD algorithm.
Consider the following optimization problem:

$$\min_w J(w) = \mathbb{E}[f(w, X)] \tag{6.5}$$

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

A straightforward method for solving (6.5) is gradient descent:

$$\begin{aligned} w_{k+1} &= w_k - \alpha_k \nabla_w J(w_k) \\ &= w_k - \alpha_k \nabla_w \mathbb{E}[f(w, X)] \\ &= w_k - \alpha_k \mathbb{E}[\nabla_w f(w, X)]. \end{aligned} \tag{6.6}$$

This gradient descent algorithm can find the optimal solution w^* under some mild conditions such as the convexity of f.

The gradient descent algorithm requires the expected value $\mathbb{E}[\nabla_w f(w_k, X)]$. One way to obtain the expected value is based on the probability distribution of X. However, the distribution is often unknown in practice. Another way is to collect a large number of i.i.d. samples $\{x_i\}_{i=1}^n$ of X so that the expected value can be approximated as

$$\mathbb{E}[\nabla_w f(w_k, X)] \approx \frac{1}{n} \sum_{i=1}^n \nabla_w f(w_k, x_i).$$

Then, (6.6) becomes

$$w_{k+1} = w_k - \alpha_k \cdot \frac{1}{n} \sum_{i=1}^n \nabla_w f(w_k, x_i). \tag{6.7}$$

One problem of the algorithm in (6.7) is that it requires all the samples in each iteration. In practice, if the samples are collected one by one, it is favorable to update w every time a sample is collected. To that end, we can use the following algorithm:

$$w_{k+1} = w_k - \alpha_k \nabla_w f(w_k, x_k), \tag{6.8}$$

where x_k is the sample collected at time step k. This is the well-known **stochastic gradient descent**[11] algorithm.

Compared to the gradient descent algorithm in (6.5), SGD replaces the true gradient $\mathbb{E}[\nabla_w f(w, X)]$ with the stochastic gradient $\nabla_w f(w_k, x_k)$. Since $\nabla_w f(w_k, x_k) \neq \mathbb{E}[\nabla_w f(w, X)]$, can such a replacement still ensure $w_k \to w^*$ as $k \to \infty$? The answer is yes. We next present an intuitive explanation and postpone the rigorous proof of the convergence to Section 6.4.5.

In particular, since

$$\nabla_w f(w_k, x_k) = \mathbb{E}[\nabla_w f(w_k, X)] + \underbrace{\Big(\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w_k, X)]\Big)}_{:=\eta_k},$$

the SGD algorithm in (6.8) can be rewritten as

$$w_{k+1} = w_k - \alpha_k \mathbb{E}[\nabla_w f(w_k, X)] - \alpha_k \eta_k.$$

[11]This algorithm is called "stochastic" because it relies on stochastic samples $\{x_k\}$.

Therefore, the SGD algorithm is the same as the regular gradient descent algorithm except that it has a perturbation term $\alpha_k\eta_k$. Since $\{x_k\}$ is i.i.d., we have $\mathbb{E}_x[\nabla_w f(w_k, x_k)] = \mathbb{E}_x[\nabla_w f(w_k, X)]$. As a result,

$$\mathbb{E}[\eta_k] = \mathbb{E}\Big[\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w_k, X)]\Big] = \mathbb{E}_x[\nabla_w f(w_k, x_k)] - \mathbb{E}_x[\nabla_w f(w_k, X)] = 0.$$

Therefore, the perturbation term η_k has a zero mean, which intuitively suggests that it may not jeopardize the convergence property. A rigorous proof of the convergence of SGD is given in Section 6.4.5.

| Application to Mean Estimation

SGD can also be applied to analyze the mean estimation problem. We formulate the mean estimation problem as an optimization problem:

$$\min_w J(w) \text{ where } J(\omega) = \mathbb{E}\Big[\frac{1}{2}\|w - X\|^2\Big] =: \mathbb{E}[f(w, X)] \tag{6.9}$$

where $f(w, X) = \|w - X\|^2/2$ and the gradient is $\nabla_w f(w, X) = w - X$. It can be verified that the optimal solution is $w^* = \mathbb{E}[X]$ by solving $\nabla_w J(w) = 0$. Therefore, this optimization problem is equivalent to the mean estimation problem.

- The gradient descent algorithm for solving (6.9) is

$$\begin{aligned} w_{k+1} &= w_k - \alpha_k \nabla_w J(w_k) \\ &= w_k - \alpha_k \mathbb{E}[\nabla_w f(w_k, X)] \\ &= w_k - \alpha_k \mathbb{E}[w_k - X]. \end{aligned}$$

  This gradient descent algorithm is not applicable since $\mathbb{E}[w_k - X]$ or $\mathbb{E}[X]$ on the right-hand side is unknown (in fact, it is what we need to solve).
- The SGD algorithm for solving (6.9) is

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

  where x_k is a sample obtained at time step k. Notably, this SGD algorithm is the same as the iterative mean estimation algorithm in (6.3). Therefore, (6.3) is an SGD algorithm designed specifically for solving the mean estimation problem.

| Convergence Pattern of SGD

The idea of the SGD algorithm is to replace the true gradient with a stochastic gradient. An analysis of the impact of random gradient on SGD convergence speed is given below.

The relative error between the stochastic and true gradients defined as

$$\delta_k := \frac{|\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w_k, X)]|}{|\mathbb{E}[\nabla_w f(w_k, X)]|}. \tag{6.10}$$

For the sake of simplicity, we consider the case where w and $\nabla_w f(w, x)$ are both scalars. Since w^* is the optimal solution, it holds that $\mathbb{E}[\nabla_w f(w^*, X)] = 0$. Then, the relative error can be rewritten as

$$\begin{aligned} \delta_k &= \frac{|\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w_k, X)]|}{|\mathbb{E}[\nabla_w f(w_k, X)] - \mathbb{E}[\nabla_w f(w^*, X)]|} \\ &= \frac{|\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w_k, X)]|}{|\mathbb{E}[\nabla_w^2 f(\tilde{w}_k, X)(w_k - w^*)]|} \end{aligned} \tag{6.11}$$

where the last equality is obtained by the mean value theorem and $\tilde{w}_k \in [w_k, w^*]$. Suppose that f is strictly convex such that $\nabla_w^2 f \geq c > 0$ for all w, X. Then, the denominator becomes

$$|\mathbb{E}[\nabla_w^2 f(\tilde{w}_k, X)(w_k - w^*)]| = |\mathbb{E}[\nabla_w^2 f(\tilde{w}_k, X)]||w_k - w^*| \geq c|w_k - w^*|. \tag{6.12}$$

Substituting the above inequality into (6.11) yields

$$\delta_k \leq \frac{|\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w_k, X)]|}{c|w_k - w^*|}.$$

The above inequality depicts the convergence pattern of SGD: the relative error δ_k is inversely proportional to $|w_k - w^*|$. As a result,

- when w_k is far from w^*, the relative error δ_k is small, and the SGD algorithm behaves like the gradient descent algorithm so that w_k quickly converges to w^*;
- when w_k is close to w^*, the relative error δ_k is large, and the convergence exhibits more randomness.

| A Deterministic Formulation of SGD

The formulation of SGD involves random variables. One may often encounter a deterministic formulation of SGD without involving any random variables.

In particular, consider a set of real numbers $\{x_i\}_{i=1}^n$, where x_i does not have to be a sample of any random variable. The optimization problem to be solved is to minimize the average:

$$\min_w J(w) = \frac{1}{n}\sum_{i=1}^n f(w, x_i),$$

where $f(w, x_i)$ is a parameterized function, and w is the parameter to be optimized. The gradient descent algorithm for solving this problem is

$$w_{k+1} = w_k - \alpha_k \nabla_w J(w_k) = w_k - \alpha_k \frac{1}{n}\sum_{i=1}^n \nabla_w f(w_k, x_i).$$

Suppose that the set $\{x_i\}_{i=1}^n$ is large and we can only fetch a single number each time. In this case, it is favorable to update w_k in an incremental manner:

$$w_{k+1} = w_k - \alpha_k \nabla_w f(w_k, x_k). \tag{6.13}$$

It must be noted that x_k here is the number fetched at time step k instead of the kth element in the set $\{x_i\}_{i=1}^n$.

The algorithm in (6.13) is very similar to SGD, but its problem formulation is subtly different because it does not involve any random variables or expected values. Then, many questions arise. For example, is this algorithm SGD? How should we use the finite set of numbers $\{x_i\}_{i=1}^n$? Should we sort these numbers in a certain order and then use them one by one, or should we randomly sample a number from the set?

A quick answer to the above questions is that, although no random variables are involved in the above formulation, we can convert the deterministic formulation to the stochastic formulation by introducing a random variable. In particular, let X be a random variable defined on the set $\{x_i\}_{i=1}^n$. Suppose that its probability distribution is uniform such that $p(X = x_i) = 1/n$. Then, the deterministic optimization problem becomes a stochastic one:

$$\min_w J(w) = \frac{1}{n}\sum_{i=1}^n f(w, x_i) = \mathbb{E}[f(w, X)]. \tag{6.14}$$

The last equality in the above equation is strict instead of approximate. Therefore, the algorithm in (6.13) is SGD, and the estimate converges if x_k is uniformly and independently sampled from $\{x_i\}_{i=1}^n$. Note that x_k may repeatedly take the same number in $\{x_i\}_{i=1}^n$ since it is sampled randomly.

| Convergence of SGD

Proposition 6.2 (Convergence of SGD)**.** For the SGD algorithm in (6.8), if the following conditions are satisfied, then w_k converges to the root of $\nabla_w \mathbb{E}[f(w, X)] = 0$ almost surely.

(a)

$$0 < c_1 \le \nabla_w^2 f(w, X) \le c_2;$$

(b)

$$\sum_{k=1}^{\infty} a_k = \infty \text{ and } \sum_{k=1}^{\infty} a_k^2 < \infty;$$

(c)

$$\{x_k\}_{k=1}^{\infty} \text{ are i.i.d.}$$

Proof of Proposition 6.2 The problem to be solved by SGD is to minimize $J(w) = \mathbb{E}[f(w, X)]$. This problem can be converted to a root-finding problem. That is, to find the root of $\nabla_w J(w) = \mathbb{E}[\nabla_w f(w, X)] = 0$. Let

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

Then, SGD aims to find the root of $g(w) = 0$. This is exactly the problem solved by the RM algorithm. The quantity that we can measure is $\tilde{g} = \nabla_w f(w, x)$, where x is a sample of X. Note that $\tilde{g}$ can be rewritten as

$$\tilde{g}(w, \eta) = \nabla_w f(w, x) = \mathbb{E}[\nabla_w f(w, X)] + \nabla_w f(w, x) - \mathbb{E}[\nabla_w f(w, X)].$$

Then, the RM algorithm for solving $g(w) = 0$ is

$$w_{k+1} = w_k - a_k \tilde{g}(w_k, \eta_k) = w_k - a_k \nabla_w f(w_k, x_k),$$

which is the same as the SGD algorithm in (6.8). As a result, the SGD algorithm is a special RM algorithm. We next show that the three conditions in Robbins-Monro Theorem (Theorem 6.1) are satisfied. Then, the convergence of SGD naturally follows from Theorem 6.1.

- Since $\nabla_w g(w) = \nabla_w \mathbb{E}[\nabla_w f(w, X)] = \mathbb{E}[\nabla_w^2 f(w, X)]$, it follows from $c_1 \le \nabla_w^2 f(w, X) \le c_2$ that $c_1 \le \nabla_w g(w) \le c_2$. Thus, the first condition in Theorem 6.1 is satisfied.
- The second condition in Theorem 6.1 is the same as the second condition in this theorem.
- The third condition in Theorem 6.1 requires $\mathbb{E}[\eta_k|\mathcal{H}_k] = 0$ and $\mathbb{E}[\eta_k^2|\mathcal{H}_k] < \infty$. Since $\{x_k\}$ is i.i.d, we have $\mathbb{E}_{x_k}[\nabla_w f(w, x_k)] = \mathbb{E}[\nabla_w f(w, X)]$ for all k. Therefore,

  $$\mathbb{E}[\eta_k|\mathcal{H}_k] = \mathbb{E}[\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w_k, X)]|\mathcal{H}_k].$$

  Since $\mathcal{H}_k = \{w_k, w_{k-1}, \ldots\}$ and x_k is independent of $\mathcal{H}_k$, the first term on the right-hand side becomes $\mathbb{E}[\nabla_w f(w_k, x_k)|\mathcal{H}_k] = \mathbb{E}_{x_k}[\nabla_w f(w_k, x_k)]$. The second term becomes $\mathbb{E}[\mathbb{E}[\nabla_w f(w_k, X)]|\mathcal{H}_k] = \mathbb{E}[\nabla_w f(w_k, X)]$ because $\mathbb{E}[\nabla_w f(w_k, X)]$ is a function of w_k. Therefore,

$$\mathbb{E}[\eta_k|\mathcal{H}_k] = \mathbb{E}_{x_k}[\nabla_w f(w_k, x_k)] - \mathbb{E}[\nabla_w f(w_k, X)] = 0.$$

Similarly, it can be proven that $\mathbb{E}[\eta_k^2|\mathcal{H}_k] < \infty$ if $|\nabla_w f(w, x)| < \infty$ for all w given any x.

Given the three conditions in Theorem 6.1 are satisfied, the convergence of the SGD algorithm follows.

7 Temporal-Difference Methods

This chapter introduces **temporal-difference (TD)** methods for reinforcement learning. Similar to Monte Carlo (MC) learning, TD learning is also model-free, but it has some advantages due to its incremental form. In fact, TD learning algorithms can be viewed as special stochastic algorithms for solving the Bellman or Bellman optimality equations.

7.1 TD Learning of State Values

TD learning often refers to a broad class of reinforcement learning algorithms. This subsection introduces the classic TD algorithm that estimate state values.

| Algorithm Description

Given a policy π, our goal is to estimate $v_\pi(s)$ for all $s \in \mathcal{S}$. Suppose that we have some experience samples $(s_0, r_1, s_1, \ldots, s_t, r_{t+1}, s_{t+1}, \ldots)$ generated by following π (here t denotes the time step). The following TD algorithm can estimate the state values using these samples:

$$v_{t+1}(s_t) = v_t(s_t) - \alpha_t(s_t)\Big[v_t(s_t) - \big(r_{t+1} + \gamma v_t(s_{t+1})\big)\Big], \quad \text{if } s = s_t \tag{7.1}$$

$$v_{t+1}(s) = v_t(s), \quad \text{for all } s \neq s_t, \tag{7.2}$$

where $t = 0, 1, 2, \ldots$. Here, $v_t(s_t)$ represents the estimation of $v_\pi(s_t)$ at time t; $\alpha_t(s_t)$ is the learning rate for s_t at time t.

It should be noted that, at time t, only the value of the visited state s_t is updated. The values of the unvisited states $s \neq s_t$ remain unchanged as shown in the second equation.

| Mathematical Derivation

In fact, TD learning algorithm can be viewed as a special stochastic approximation algorithm for solving the Bellman equation. To see that, first recall that the definition of the state value is

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s], \quad s \in \mathcal{S}. \tag{7.3}$$

We can rewrite this as

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s], \quad s \in \mathcal{S}. \tag{7.4}$$

That is because $\mathbb{E}[G_{t+1}|S_t = s] = \sum_a \pi(a|s) \sum_{s'} p(s'|s, a) v_\pi(s') = \mathbb{E}[v_\pi(S_{t+1})|S_t = s]$. This equation is another expression of the Bellman equation and is sometimes called the *Bellman expectation equation*.

The TD algorithm can be derived by applying the Robbins-Monro algorithm to solve the Bellman equation in (7.4). The details are given in the box below.

Derivation of the TD Algorithm For state s_t, we define a function as

$$g(v_\pi(s_t)) \doteq v_\pi(s_t) - \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s_t].$$

Then, (7.4) is equivalent to

$$g(v_\pi(s_t)) = 0.$$

Our goal is to solve the above equation to obtain $v_\pi(s_t)$ using the Robbins-Monro algorithm. Since we can obtain r_{t+1} and s_{t+1}, which are the samples of R_{t+1} and S_{t+1}, the noisy observation of $g(v_\pi(s_t))$ that we can obtain is

$$\begin{aligned}\tilde{g}(v_\pi(s_t)) &= v_\pi(s_t) - [r_{t+1} + \gamma v_\pi(s_{t+1})] \\ &= \underbrace{\Big(v_\pi(s_t) - \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s_t]\Big)}_{g(v_\pi(s_t))} \\ &\quad + \underbrace{\Big(\mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s_t] - [r_{t+1} + \gamma v_\pi(s_{t+1})]\Big)}_{\eta}.\end{aligned} \tag{7.5}$$

Therefore, the Robbins-Monro algorithm for solving $g(v_\pi(s_t)) = 0$ is

$$\begin{aligned}v_{t+1}(s_t) &= v_t(s_t) - \alpha_t(s_t)\tilde{g}(v_t(s_t)) \\ &= v_t(s_t) - \alpha_t(s_t)\Big(v_t(s_t) - \big[r_{t+1} + \gamma v_\pi(s_{t+1})\big]\Big),\end{aligned}$$

where $v_t(s_t)$ is the estimate of $v_\pi(s_t)$ at time t, and $\alpha_t(s_t)$ is the learning rate.

The algorithm in (7.5) has a similar expression to that of the TD algorithm in (7.1). The only difference is that the right-hand side of (7.5) contains $v_\pi(s_{t+1})$, whereas (7.1) contains $v_t(s_{t+1})$. That is because (7.5) is designed to merely estimate the state value of s_t by assuming that the state values of the other states are already known. If we would like to estimate the state values of all the states, then $v_\pi(s_{t+1})$ on the right-hand side should be replaced with $v_t(s_{t+1})$. Then, (7.5) is exactly the same as (7.1). However, can such a replacement still ensure convergence? The answer is yes, and it will be proven later in Theorem 7.1.

| Property Analysis

Some important properties of the TD algorithm are discussed as follows.

First, we give a further interpretation of the TD algorithm expression. In particular, (7.1) can be described as

$$v_{t+1}(s_t) = v_t(s_t) - \alpha_t(s_t)\Big[v_t(s_t) - \Big(r_{t+1} + \gamma v_t(s_{t+1})\Big)\Big], \tag{7.6}$$

where

$$\bar{v}_t \doteq r_{t+1} + \gamma v_t(s_{t+1})$$

is called the **TD target** and

$$\delta_t \doteq v(s_t) - \bar{v}_t = v_t(s_t) - (r_{t+1} + \gamma v_t(s_{t+1}))$$

is called the **TD error**. It can be seen that the new estimate $v_{t+1}(s_t)$ is a combination of the current estimate $v_t(s_t)$ and the TD error δ_t.

- Why is $\bar{v}_t$ called the TD target?

This is because $\bar{v}_t$ is the target value that the algorithm attempts to drive $v(s_t)$ to. To see that, subtracting $\bar{v}_t$ from both sides of (7.6) gives

$$v_{t+1}(s_t) - \bar{v}_t = [v_t(s_t) - \bar{v}_t] - \alpha_t(s_t)[v_t(s_t) - \bar{v}_t] = \Big[1 - \alpha_t(s_t)\Big][v_t(s_t) - \bar{v}_t].$$

Taking the absolute values of both sides of the above equation gives

$$\Big|v_{t+1}(s_t) - \bar{v}_t\Big| = \Big|1 - \alpha_t(s_t)\Big|\Big|v_t(s_t) - \bar{v}_t\Big|.$$

Since $\alpha_t(s_t)$ is a small positive number, we have $0 < 1 - \alpha_t(s_t) < 1$. It then follows that

$$\Big|v_{t+1}(s_t) - \bar{v}_t\Big| < \Big|v_t(s_t) - \bar{v}_t\Big|.$$

The above inequality is important because it indicates that the new value $v_{t+1}(s_t)$ is closer to $\bar{v}_t$ than the old value $v_t(s_t)$. Therefore, this algorithm mathematically drives $v_t(s_t)$ toward $\bar{v}_t$. This is why $\bar{v}_t$ is called the TD target.

- What is the interpretation of the TD error?

  First, this error is called temporal-difference because $\delta_t = v_t(s_t) - (r_{t+1} + \gamma v_t(s_{t+1}))$ reflects the discrepancy between two time steps t and $t+1$. Second, the TD error is zero in the expectation sense when the state value estimate is accurate. To see that, when $v_t = v_\pi$, the expected value of the TD error is

$$\begin{aligned}\mathbb{E}[\delta_t|S_t = s_t] &= \mathbb{E}[v_\pi(S_t) - (R_{t+1} + \gamma v_\pi(S_{t+1}))|S_t = s_t]\\ &= v_\pi(s_t) - \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s_t]\\ &= 0.\end{aligned}$$

  Therefore, the TD error reflects not only the discrepancy between two time steps but also, more importantly, the discrepancy between the estimate v_t and the true state value v_π.

Second, the TD algorithm in (7.1) can only estimate the state values of a given policy. To find optimal policies, we still need to further calculate the action values and then conduct policy improvement. This will be introduced in Section 7.2.

Third, while both TD learning and MC learning are model-free, what are their advantages and disadvantages? The answers are summarized in Table 7.1.

TD learning	**MC learning**
Incremental: TD learning is incremental. It can update the state/action values immediately after receiving an experience sample.	*Non-incremental:* MC learning is non-incremental. It must wait until an episode has been completely collected. That is because it must calculate the discounted return of the episode.
Continuing tasks: Since TD learning is incremental, it can handle both episodic and continuing tasks. Continuing tasks may not have terminal states.	*Episodic tasks:* Since MC learning is non-incremental, it can only handle episodic tasks where the episodes terminate after a finite number of steps.
Bootstrapping: TD learning bootstraps because the update of a state/action value relies on the previous estimate of this value. As a result, TD learning requires an initial guess of the values.	*Non-bootstrapping:* MC is not bootstrapping because it can directly estimate state/action values without initial guesses.
Low estimation variance: The estimation variance of TD is lower than that of MC because it involves fewer random variables. For instance, to estimate an action value $q_\pi(s_t, a_t)$, Sarsa merely requires the samples of three random variables: $R_{t+1}, S_{t+1}, A_{t+1}$.	*High estimation variance:* The estimation variance of MC is higher since many random variables are involved. For example, to estimate $q_\pi(s_t, a_t)$, we need samples of $R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$. Suppose that the length of each episode is L. Assume that each state has the same number of actions as $\|\mathcal{A}\|$. Then, there are $\|\mathcal{A}\|^L$ possible episodes following a soft policy. If we merely use a few episodes to estimate, it is not surprising that the estimation variance is high.

Table 7.1: A comparison between TD learning and MC learning.

| Convergence Analysis

The convergence analysis of the TD algorithm in (7.1) is given below.

Proposition 7.1 (Convergence of TD learning)**.** Given a policy π, by the TD algorithm in (7.1), $v_t(s)$ converges almost surely to $v_\pi(s)$ as $t \to \infty$ for all $s \in \mathcal{S}$ if $\sum_t \alpha_t(s) = \infty$ and $\sum_t \alpha_t^2(s) < \infty$ for all $s \in \mathcal{S}$.

Some remarks about α_t are given below.

- First, the condition of $\sum_t \alpha_t(s) = \infty$ and $\sum_t \alpha_t^2(s) < \infty$ must be valid for all $s \in \mathcal{S}$. Note that, at time t, $\alpha_t(s) > 0$ if s is being visited and $\alpha_t(s) = 0$ otherwise. The condition $\sum_t \alpha_t(s) = \infty$ requires the state s to be visited an infinite (or sufficiently many) number of times. This requires either the condition of exploring starts or an exploratory policy so that every state-action pair can possibly be visited many times.
- Second, the learning rate α_t is often selected as a small positive constant in practice. In this case, the condition that $\sum_t \alpha_t^2(s) < \infty$ is no longer valid. When α is constant, it can still be shown that the algorithm converges in the sense of expectation.

7.2 TD Learning of Action Values: SARSA

The TD algorithm introduced in Section 7.1 can only estimate state values. This section introduces another TD algorithm called **SARSA** that can directly estimate action values. Estimating action values is important because it can be combined with a policy improvement step to learn optimal policies.

| Algorithm Description

Given a policy π, our goal is to estimate the action values. Suppose that we have some experience samples generated following π: $(s_0, a_0, r_1, s_1, a_1, \ldots, s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, \ldots)$. We can use the following Sarsa algorithm to estimate the action values:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)\Big[q_t(s_t, a_t) - (r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1})\Big], \tag{7.7}$$

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

where $t = 0, 1, 2, \ldots$ and $\alpha_t(s_t, a_t)$ is the learning rate. Here, $q_t(s_t, a_t)$ is the estimate of $q_\pi(s_t, a_t)$. At time t, only the q-value of (s_t, a_t) is updated, whereas the q-values of the others remain the same.

Some important properties of the Sarsa algorithm are discussed as follows.

- Why is this algorithm called "Sarsa"? That is because each iteration of the algorithm requires $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. Sarsa is an abbreviation for state-action-reward-state-action.
- Why is Sarsa designed in this way? One may have noticed that Sarsa is similar to the TD algorithm in (7.1). In fact, Sarsa can be easily obtained from the TD algorithm by replacing state value estimation with action value estimation.
- What does Sarsa do mathematically? Similar to the TD algorithm in (7.1), Sarsa is a stochastic approximation algorithm for solving the Bellman equation of a given policy:

$$q_\pi(s, a) = \mathbb{E}[R + \gamma q_\pi(S', A')|s, a] \quad \text{for all } (s, a). \tag{7.8}$$

Equation 7.8 is the Bellman equation expressed in terms of action values.

Is Sarsa convergent? Since Sarsa is the action-value version of the TD algorithm in (7.1), the convergence result is similar to Theorem 7.1 and given below.

Convergence of Sarsa Given a policy π, by the Sarsa algorithm in (7.7), $q_t(s, a)$ converges almost surely to the action value $q_\pi(s, a)$ as $t \to \infty$ for all (s, a) if $\sum_t \alpha_t(s, a) = \infty$ and $\sum_t \alpha_t^2(s, a) < \infty$ for all (s, a).

The proof is similar to that of Theorem 7.1 and is omitted here. The condition of $\sum_t \alpha_t(s, a) = \infty$ and $\sum_t \alpha_t^2(s, a) < \infty$ should be valid for all (s, a). In particular, $\sum_t \alpha_t(s, a) = \infty$ requires that every state-action pair must be visited an infinite (or sufficiently many) number of times. At time t, if $(s, a) = (s_t, a_t)$, then $\alpha_t(s, a) > 0$; otherwise, $\alpha_t(s, a) = 0$.

| Optimal Policy Learning via Sarsa

The Sarsa algorithm in (7.7) can only estimate the action values of a given policy. To find optimal policies, we can combine it with a policy improvement step. The combination is also often called Sarsa, and its implementation procedure is given in Algorithm 7.

Algorithm 7 Optimal policy learning by Sarsa

```
Require: $\alpha_t(s,a) = \alpha > 0$ for all $(s,a)$ and all $t$; $\epsilon \in (0,1)$.
Require: Initial $q_0(s,a)$ for all $(s,a)$.
Require: Initial $\epsilon$-greedy policy $\pi_0$ derived from $q_0$.
Goal: Learn an optimal policy that can lead the agent to the target state from an initial state $s_0$.
  for each episode do
      Generate $a_0$ at $s_0$ following $\pi_0(s_0)$
      while $s_t$ $(t = 0,1,2,\ldots)$ is not the target state do
          Collect an experience sample $(r_{t+1}, s_{t+1}, a_{t+1})$ given $(s_t, a_t)$:
          Generate $r_{t+1}, s_{t+1}$ by interacting with the environment; generate $a_{t+1}$ following $\pi_t(s_{t+1})$.
          Update q-value for $(s_t, a_t)$:
              $q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - (r_{t+1} + \gamma q_t(s_{t+1},a_{t+1}))\Big]$
          Update policy for $s_t$:
              $\pi_{t+1}(a|s_t) = \begin{cases} 1 - \frac{\epsilon}{|\mathcal{A}(s_t)|}(|\mathcal{A}(s_t)| - 1) \text{ if } a = \arg\max_a q_{t+1}(s_t,a) \\ \frac{\epsilon}{|\mathcal{A}(s_t)|} \text{ otherwise} \end{cases}$
          $s_t \leftarrow s_{t+1}, a_t \leftarrow a_{t+1}$
      end while
  end for
```

As shown in Algorithm 7, each iteration has two steps. The first step is to update the q-value of the visited state-action pair. The second step is to update the policy to an ϵ-greedy one.

The q-value update step only updates the single state-action pair visited at time t. Afterward, the policy of s_t is immediately updated. Therefore, we do not evaluate a given policy sufficiently well before updating the policy.

| Expected Sarsa

Given a policy π, its action values can be evaluated by **Expected Sarsa**, which is a variant of Sarsa. The Expected Sarsa algorithm is

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

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

where

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

is the expected value of $q_t(s_{t+1},a)$ under policy π_t. The expression of the Expected Sarsa algorithm is very similar to that of Sarsa. They are different only in terms of their TD targets. In particular, the TD target in Expected Sarsa is $r_{t+1}+\gamma\mathbb{E}[q_t(s_{t+1},A)]$, while that of Sarsa is $r_{t+1}+\gamma q_t(s_{t+1},a_{t+1})$. Since the algorithm involves an expected value, it is called Expected Sarsa. Although calculating the expected value may increase the computational complexity slightly, it is beneficial in the sense that it reduces the estimation variances because it reduces the random variables in Sarsa from $\{s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}\}$ to $\{s_t, a_t, r_{t+1}, s_{t+1}\}$.

Similar to the TD learning algorithm in (7.1), Expected Sarsa can be viewed as a stochastic approximation algorithm for solving the following equation:

$$q_\pi(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \mathbb{E}[q_\pi(S_{t+1},A_{t+1})|S_t = s, A_t = a]\right], \quad \text{for all } s, a. \tag{7.9}$$

The above equation may look strange at first glance. In fact, it is another expression of the Bellman equation. To see that, substituting

$$\mathbb{E}[q_\pi(S_{t+1}, A_{t+1})|S_{t+1}] = \sum_{A'} q_\pi(S_{t+1}, A')\pi(A'|S_{t+1}) = v_\pi(S_{t+1})$$

into (7.9) gives

$$q_\pi(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a\Big],$$

which is clearly the Bellman equation.

7.3 TD Learning of Action Values: n-step Sarsa

This section introduces n-step Sarsa, an extension of Sarsa. Sarsa and MC learning are two extreme cases of n-step Sarsa.

Recall that the definition of the action value is

$$q_\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a], \tag{7.10}$$

where G_t is the discounted return satisfying

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots .$$

In fact, G_t can also be decomposed into different forms:

$$\begin{aligned}
\text{Sarsa} \longleftarrow\ G_t^{(1)} &= R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}),\\
G_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, A_{t+2}),\\
&\ \vdots\\
n\text{-step Sarsa} \longleftarrow\ G_t^{(n)} &= R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n q_\pi(S_{t+n}, A_{t+n}),\\
&\ \vdots\\
\text{MC} \longleftarrow\ G_t^{(\infty)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} \cdots
\end{aligned}$$

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

Substituting different decompositions of $G_t^{(n)}$ into $q_\pi(s,a)$ in (7.10) results in different algorithms.

- When $n = 1$, we have

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

  The corresponding stochastic approximation algorithm for solving this equation is

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

  which is the Sarsa algorithm in (7.7).

- When $n = \infty$, we have

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

  The corresponding algorithm for solving this equation is

  $$q_{t+1}(s_t,a_t) = g_t \doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots,$$

  where g_t is a sample of G_t. In fact, this is the MC learning algorithm, which approximates the action value of (s_t,a_t) using the discounted return of an episode starting from (s_t,a_t).

- For a general value of n, we have

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

  The corresponding algorithm for solving the above equation is

$$q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - (r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n q_t(s_{t+n}, a_{t+n}))\Big]. \quad (7.11)$$

  This algorithm is called **n-step Sarsa**.

To summarize, n-step Sarsa is a more general algorithm because it becomes the (one-step) Sarsa algorithm when $n = 1$ and the MC learning algorithm when $n = \infty$ (by setting $\alpha_t = 1$). If n is selected as a large number, the estimate has a relatively high variance but a small bias; if n is selected to be small, the estimate has a relatively large bias but a low variance.

To implement the n-step Sarsa algorithm in (7.11), we need the experience samples $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, \ldots, r_{t+}$ Since $(r_{t+n}, s_{t+n}, a_{t+n})$ has not been collected at time t, we have to wait until time $t+n$ to update the q-value of (s_t, a_t). To that end, (7.11) can be rewritten as

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

where $q_{t+n}(s_t, a_t)$ is the estimate of $q_\pi(s_t, a_t)$ at time $t+n$.

7.4 TD Learning of Optimal Action Values: Q-learning

Q-learning is one of the most classic reinforcement learning algorithms. Recall that Sarsa can only estimate the action values of a given policy, and it must be combined with a policy improvement step to nd optimal policies. By contrast, Q-learning can directly estimate optimal action values and find optimal policies.

| Algorithm Description

The Q-learning algorithm is

$$q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - \Big(r_{t+1} + \gamma \max_{a\in\mathcal{A}(s_{t+1})} q_t(s_{t+1}, a)\Big)\Big], \quad (7.12)$$

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

where $t = 0, 1, 2, \ldots$. Here, $q_t(s_t, a_t)$ is the estimate of the optimal action value of (s_t, a_t) and $\alpha_t(s_t, a_t)$ is the learning rate for (s_t, a_t).

The expression of Q-learning is similar to that of Sarsa. They are different only in terms of their TD targets: the TD target of Q-learning is $r_{t+1} + \gamma \max_a q_t(s_{t+1}, a)$, whereas that of Sarsa is $r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1})$. Moreover, given (s_t, a_t), Sarsa requires $(r_{t+1}, s_{t+1}, a_{t+1})$ in every iteration, whereas Q-learning merely requires (r_{t+1}, s_{t+1}).

Why is Q-learning designed as the expression in (7.12), and what does it do mathematically? Q-learning is a stochastic approximation algorithm for solving the following equation:

$$q(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma \max_a q(S_{t+1}, a)|S_t = s, A_t = a\Big]. \quad (7.13)$$

This is the Bellman optimality equation expressed in terms of action values. The convergence analysis of Q-learning is similar to Theorem 7.1 and omitted here.

| Off-policy and On-policy

Two policies exist in any reinforcement learning task: a behavior policy and a target policy. The behavior policy is the one used to generate experience samples. The target policy is the one that is constantly updated to converge to an optimal policy. When the behavior policy is the same as the target policy, such a learning process is called **on-policy**. Otherwise, when they are different, the learning process is called **off-policy**.

Another concept that may be confused with on-policy/off-policy is **online/offline**. Online learning refers to the case where the agent updates the values and policies while interacting with the environment. Offline learning refers to the case where the agent updates the values and policies using pre-collected experience data without interacting with the environment. If an algorithm is on-policy, then it can be implemented in an online fashion, but cannot use pre-collected data generated by other policies. If an algorithm is off-policy, then it can be implemented in either an online or offine fashion.

Sarsa and MC learning are on-policy, while Q-learning is off-policy.

Algorithm 8 Optimal policy learning via Q-learning (on-policy version)

```
Require: $\alpha_t(s,a) = \alpha > 0$ for all $(s,a)$ and all $t$; $\epsilon \in (0,1)$.
  Initialize: A guess of initial action values, $q_0(s,a)$.
  Initialize: $\epsilon$-greedy policy $\pi_0(a|s)$ derived from $q_0$.
  Goal: Learn an optimal path that can lead the agent to the target state from an initial state $s_0$.
  for all episode do
    if $s_t$ $(t = 0,1,2,\ldots)$ is not the target state then
      Collect the experience sample $(a_t, r_{t+1}, s_{t+1})$ given $s_t$.
      Generate $a_t$ following $\pi_t(s_t)$.
      Generate $r_{t+1}, s_{t+1}$ by interacting with the environment.
      Update q-value for $(s_t, a_t)$:
        $q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - (r_{t+1} + \gamma \max_a q_t(s_{t+1},a))\Big]$
      Update policy for $s_t$:
        $\pi_{t+1}(a|s_t) = \begin{cases} 1 - \frac{\epsilon}{|\mathcal{A}|}(|\mathcal{A}|-1), & \text{if } a = \arg\max_a q_{t+1}(s_t,a) \\ \frac{\epsilon}{|\mathcal{A}|}, & \text{otherwise} \end{cases}$
    end if
  end for
```

Algorithm 9 Optimal policy learning via Q-learning (off-policy version)

```
Require: $\alpha_t(s,a) = \alpha > 0$ for all $(s,a)$ and all $t$; $\epsilon \in (0,1)$.
  Initialize: A guess of initial action values, $q_0(s,a)$.
  Initialize: Behavior policy $\pi_b(a|s)$.
  Goal: Learn an optimal target policy $\pi_T$ for all states from the samples generated by $\pi_b$.
  for all episode $\{(s_0,a_0,r_1),(s_1,a_1,r_2),\ldots\}$ generated by behavior policy $\pi_b$ do
    for each step $t = 0,1,2,\ldots$ of the episode do
      Update q-value for $(s_t, a_t)$:
        $q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - (r_{t+1} + \gamma \max_a q_t(s_{t+1},a))\Big]$
      Update target policy for $s_t$:
        $\pi_{T,t+1}(a|s_t) = \begin{cases} 1, & \text{if } a = \arg\max_a q_{t+1}(s_t,a) \\ 0, & \text{otherwise} \end{cases}$
    end for
  end for
```

7.5 A Unified Viewpoint of TD Algorithms

Up to now, we have introduced different TD algorithms such as Sarsa, n-step Sarsa, and Q-learning. These algorithms, together with MC learning, can be included in a unified framework.
In particular, the TD algorithms (for action value estimation) can be expressed in a unified expression:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)[q_t(s_t, a_t) - \bar{q}_t], \tag{7.14}$$

where $\bar{q}_t$ is the TD target. Different TD algorithms have different $\bar{q}_t$. See Table 7.2 for a summary. The MC learning algorithm can be viewed as a special case of (7.14): we can set $\alpha_t(s_t, a_t) = 1$ and then (7.14) becomes $q_{t+1}(s_t, a_t) = \bar{q}_t$.

Algorithm	Expression of the TD target $\bar{q}_t$
Sarsa	$\bar{q}_t = r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1})$
n-step Sarsa	$\bar{q}_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n q_t(s_{t+n}, a_{t+n})$
Q-learning	$\bar{q}_t = r_{t+1} + \gamma \max_a q_t(s_{t+1}, a)$
Monte Carlo	$\bar{q}_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots$

Algorithm	Equation to be solved
Sarsa	BE: $q_\pi(s, a) = \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \vert S_t = s, A_t = a]$
n-step Sarsa	BE: $q_\pi(s, a) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n q_\pi(S_{t+n}, A_{t+n}) \vert S_t = s, A_t = a]$
Q-learning	BOE: $q(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_a q(S_{t+1}, a) \vert S_t = s, A_t = a\right]$
Monte Carlo	BE: $q_\pi(s, a) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \vert S_t = s, A_t = a]$

Table 7.2: A unified point of view of TD algorithms. Here, BE and BOE denote the Bellman equation and Bellman optimality equation, respectively.

Algorithm (7.14) can be viewed as a stochastic approximation algorithm for solving a unified equation: $q(s, a) = \mathbb{E}[\bar{q}_t | s, a]$. This equation has different expressions with different $\bar{q}_t$. These expressions are summarized in Table 7.2. As can be seen, all of the algorithms aim to solve the Bellman equation except Q-learning, which aims to solve the Bellman optimality equation.

References

S. Zhao. *Mathematical Foundations of Reinforcement Learning*. Springer Nature Press, 2025. URL https://github.com/MathFoundationRL/Book-Mathematical-Foundation-of-Reinforcement-Learning.

Frédérick Garcia and Emmanuel Rachelson. *Markov Decision Processes*, chapter 1, pages 1–38. John Wiley & Sons, Ltd, 2013. ISBN 9781118557426. doi: https://doi.org/10.1002/9781118557426.ch1. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118557426.ch1.

Roger A. Horn and Charles R. Johnson. *Matrix analysis*. Cambridge university press, 2012. URL https://www.anandinstitute.org/pdf/Roger_A.Horn.%20_Matrix_Analysis_2nd_edition(BookSee.org).pdf.

R.S. Sutton and A.G. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 9(5):1054–1054, 1998. doi: 10.1109/TNN.1998.712192. URL https://ieeexplore.ieee.org/document/712192.

Nathan Lambert. *Reinforcement Learning from Human Feedback*. Online, 2024. URL https://rlhfbook.com/book.pdf.