Chapter 2
Introduction to Reinforcement Learning

Zihan Ding, Yanhua Huang, Hang Yuan, and Hao Dong

Abstract In this chapter, we introduce the fundamentals of classical reinforcement learning and provide a general overview of deep reinforcement learning. We first start with the basic definitions and concepts of reinforcement learning, including the agent, environment, action, and state, as well as the reward function. Then, we describe a classical reinforcement learning problem, the bandit problem, to provide the readers with a basic understanding of the underlying mechanism of traditional reinforcement learning. Next, we introduce the Markov process, together with the Markov reward process and the Markov decision process. These notions are the cornerstones in formulating reinforcement learning tasks. The combination of the Markov reward process and value function estimation produces the core results used in most reinforcement learning methods: the Bellman equations. The optimal value functions and optimal policy can be derived through solving the Bellman equations. Three main approaches for solving the Bellman equations are then introduced: dynamic programming, Monte Carlo method, and temporal difference learning. We further introduce deep reinforcement learning for both policy and value function approximation in policy optimization. The contents in policy optimization are introduced in two main categories: value-based optimization and policy-based optimization. In value-based optimization, the gradient-based methods are introduced for leveraging deep neural networks, like Deep Q-Networks. In policy-based optimization, the deterministic policy gradient and stochastic policy gradient are

Z. Ding (✉)
Imperial College London, London, UK
e-mail: zhding@mail.ustc.edu.cn

Y. Huang
Xiaohongshu Technology Co., Ltd., Shanghai, China

H. Yuan
Oxford University, Oxford, UK
e-mail: hang.yuan@keble.ox.ac.uk

H. Dong
Peking University, Beijing, China
e-mail: hao.dong@pku.edu.cn

H. Dong et al. (eds.), *Deep Reinforcement Learning*,
https://doi.org/10.1007/978-981-15-4095-0_2

introduced in detail with sufficient mathematical proofs. The combination of value-based and policy-based optimization produces the popular actor-critic structure, which leads to a large number of advanced deep reinforcement learning algorithms. This chapter will lay a foundation for the rest of the book, as well as providing the readers with a general overview of deep reinforcement learning.

Keywords Reinforcement learning · Multi-armed bandit · Markov process · Bellman equation · Dynamic programming · Monte Carlo method · Temporal difference learning · Value-based optimization · Deterministic policy gradient · Stochastic policy gradient

2.1 Introduction

This chapter introduces the basic knowledge of reinforcement learning (RL) as well as deep reinforcement learning (DRL), including the definitions and explanations of basic concepts, as well as the theoretical proofs of some theorems in reinforcement learning domain, which are the fundamentals of advanced topics in (deep) reinforcement learning. Therefore, we encourage the readers to read through and understand well about the contents in this chapter before moving on to the following chapters. We will start with the basic concepts in reinforcement learning.

The **agent** and **environment** are the basic components of reinforcement learning, as shown in Fig. 2.1. The environment is an entity that the agent can interact with. For example, an environment can be a Pong game, which is shown on the right-hand side of Fig. 2.2. The agent controls the paddle to hit the ball back and forth. An agent can "interact" with the environment by using a predefined **action set** $\mathcal{A} = \{A_1, A_2 \ldots\}$. The action set describes all possible actions. In Pong, the action set can be {moveUp, moveDown}. The goal of reinforcement learning algorithms is to teach the agent how to interact "well" with the environment so that the agent is able to obtain a good score under a predefined evaluation metric. In Pong, the metric could just be the score that a player gets. An agent will receive a **reward** r of 1 when the ball hits the wall on the opposite side. In other words, if the agent misses

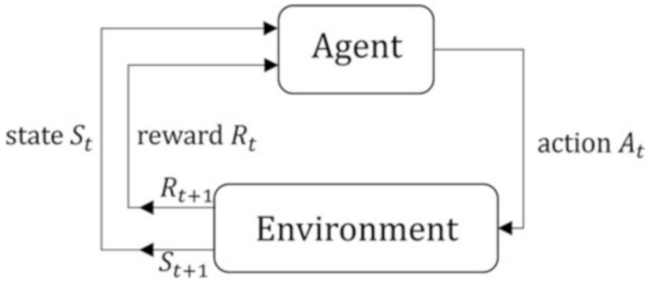

Fig. 2.1 Agent and environment

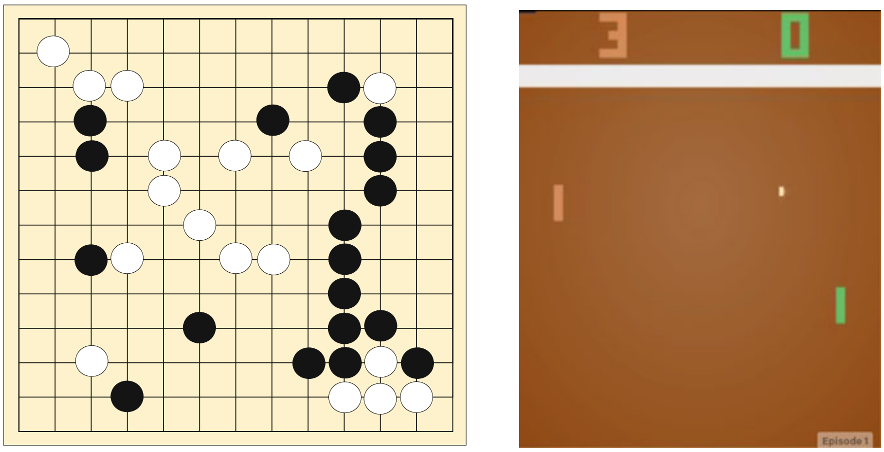

Fig. 2.2 Two types of gaming environments: on the left is the game of Go, where the observation contains the complete state of the environment, and thus the environment is fully observable. On the right is the Atari Pong game, where the observation of a single frame does not contain the velocity of the ball, and thus the environment is partially observable

the ball and lets the ball hit the wall on its side, then its opponent will receive a reward of 1.

Now, let us take a closer look at the relationship between the agent and the environment as depicted in Fig. 2.1. At an arbitrary time step, t, the agent first observes the current state of the environment, S_t, and the corresponding reward value, R_t. The agent then decides what to do next based on the state and reward information. The action the agent intends to perform, A_t, gets fed back into the environment such that we can obtain the new state S_{t+1} and reward R_{t+1}. The observation of the environment state s (s is a general representation of state regardless of time step s) from the agent's perspective does not always contain all the information about the environment. If the observation only contains partial state information, the environment is **partially observable**. Nevertheless, if the observation contains the complete state information of the environment, the environment is **fully observable**. In practice, the observation is usually a function of the state, which makes it difficult to differentiate the observations that contain all the state information from the ones that do not. A better understanding would be from the information perspective that a fully observable environment should not miss any information in the function from the underlying state of the whole environment to the observation of the agent.

The board game Go, shown on the left-hand side of Fig. 2.2, is a typical example of a fully observable environment, where all the placement information of Go pieces is fully observable for both of the players. The Atari game of Pong with a single frame as observation is a partially observable environment, where the velocity of the ball is important for the decision but not available from the stationary frame.

In the literature of reinforcement learning, the action a (a is a general representation of action regardless of time step t) is usually conditioned on the state s to represent the behavior of the agent, under the assumption of fully observable environments. If the environment is partially observable for the agent, the agent usually cannot have direct access to the underlying state, therefore the action has to be conditioned on the observation, without advanced process.

To provide feedback from the environment to the agent, a **reward function** R generates an **immediate reward** R_t according to the environment status and sends it to the agent at every time step. In some cases, the reward function depends on the current state only, i.e., $R_t = R(S_t)$. For instance, in Pong, one of the players will receive the reward immediately, if the ball hits one side of the wall. In this case, the reward function only depends on the current state. However, sometimes the reward function can depend on not only the current state and action but also the states and actions at other time steps. An example would be if in an environment one agent is asked to memorize a sequence of actions done by another player and then repeat the same sequence of actions. So the reward will depend on not just one state-action pair but also the sequence of state-action pairs during the other player's movement and this player's movement. A reward function based on the current state, or even the current state and action will not be indicative for the agent when mimicking the whole sequence.

In reinforcement learning, a **trajectory** is a sequence of states, actions, and rewards:

$$\tau = (S_0, A_0, R_0, S_1, A_1, R_1, \dots)$$

which records how the agent interacts with the environment. The initial state in a trajectory, S_0, is randomly sampled from the **start-state distribution**, denoted by ρ_0, in which:

$$S_0 \sim \rho_0(\cdot) \tag{2.1}$$

For example, the Atari Pong game always starts with a ball in the center of the screen and the game of GO usually starts with a chess piece on a random location of the chessboard.

The transition from a state to the next state can be either a **deterministic transition process** or a **stochastic transition process**. For the deterministic transition, the next state S_{t+1} is governed by a deterministic function:

$$S_{t+1} = f(S_t, A_t), \tag{2.2}$$

where a unique next state S_{t+1} can be found. For the stochastic transition process, the next state S_{t+1} is described as a probabilistic distribution:

$$S_{t+1} \sim p(S_{t+1}|S_t, A_t) \tag{2.3}$$

A trajectory, being referred to also as an **episode**, is a sequence that goes from the initial state to the terminal state (for finite cases). For example, playing one entire game can be considered as an episode. The terminal state is reached when the agent wins or loses the game. Sometimes, one episode can have several sub-games rather than only one. For example, an episode can contain 21 sub-games for the Gym Pong game.

Finally, we shall discuss two important concepts before the end of the section, exploitation and exploration, as well as the well-known exploration-exploitation trade-off. **Exploitation** means maximizing the agent performance using the existing knowledge, and its performance is usually evaluated by the expected reward. For example, a gold digger now has an ore providing him two grams of gold per day, and he knows that the largest gold ore can give him five grams of gold per day. However, he also knows that finding a new ore will not only force him to stop exploiting the current ore but also costs him extra time with a risk of not finding anything at all in the end. Having these in mind he decides to dig the current ore until it is exhausted to maximize the reward (gold in this case) via exploitation and give up exploration, given the large risks of exploration based on his current knowledge. The policy he took here is the **greedy** policy, which means the agent constantly performs the action that yields the highest expected reward based on current information, rather than taking risky trials which may lead to lower expected rewards.

Exploration means increasing existing knowledge by taking actions and interacting with the environment. Back to the example of the gold digger, he wishes to spend some time to find new ore, and if he finds a bigger gold ore, he can get more reward per day. To have a larger long-term return, the short-term return may be sacrificed. The gold digger is always facing the exploitation and exploration dilemma as he needs to decide how much the yield a gold mine has for him to stay and how little the yield a gold mine has for him to keep exploring. Maybe, he also wants to see enough ores before he can make a well-informed decision. From the above descriptions, the readers may already have a primary understanding about the exploration-exploitation trade-off. The **exploration-exploitation trade-off** describes the balance between how much efforts the agent makes on exploration and exploitation, respectively. The trade-off between exploration and exploitation is a central theme of reinforcement learning research and reinforcement learning algorithm development. We will explain it with the following bandit problem.

2.2 Bandits

Single-Armed Bandit is a simple gambling machine as shown on the left-hand side of Fig. 2.3. The agent (player) interacts with the environment (machine) by pulling a single arm down, and receives a reward when the machine hits the jackpot. In a casino, there will usually be many bandits lining up in a row. The agent can choose to pull an arm of any of these slot machines. The distributions of the reward values r conditioned on the actions a, $P(r|a)$, for different bandits are different but fixed. The agent, however, does not know the distributions in the beginning, and the knowledge is acquired through the trials of the agents. The goal is to maximize

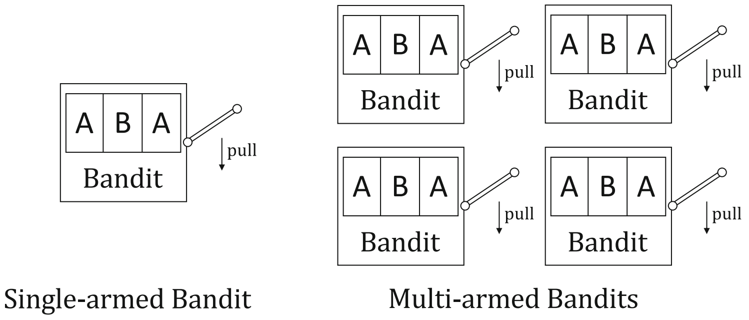

Fig. 2.3 Single-armed bandit (left) and multi-armed bandits (right)

the payoffs after some number of selections. The agent will have to choose among various slot machines at each time step and we refer this game as **multi-armed bandit** (MAB) which is shown on the right-hand side of Fig. 2.3. MAB gives the agent the freedom to make strategic choices of which arm to pull.

We try to solve the MAB problem with standard reinforcement learning methods. The action a of the agent is to choose which arm to pull. A reward will be given right after the action is conducted. Formally, at time step t, we are trying to maximize the expected action value defined as follows:

$$q(a) = \mathbb{E}[R_t | A_t = a]$$

If we already know the true value $q(a)$ of each action a, then the problem will be trivial to solve because we can always choose the action that would yield the best q value. However, in reality, we typically need to estimate the q value and we denote the estimate as $Q(a)$, which should be as close to $q(a)$ as possible.

The MAB problem is an excellent example to illustrate the exploration-exploitation trade-off. After one has estimated the q values for some states, if the agent is always going to take the action that has the greatest Q value, then this agent is considered to be greedy and is exploiting the already estimated q values. If the agent takes on any action that does not have the best Q value, then this agent is considered to be exploring different options. Neither doing only exploration or exploitation is a good way to improve the policy of the agent in most cases.

A simple action-value based method is to estimate $Q_t(a)$ using the ratio between the total rewards received by choosing one action by time t and the total number of times that this specific action has been chosen:

$$Q_t(a) = \frac{\text{sum of the rewards by choosing } a \text{ before } t}{\text{number of times } a \text{ was chosen before } t} = \frac{\sum_{i=0}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=0}^{t-1} \mathbb{1}_{A_i=a}}$$

$\mathbb{1}_{\text{predicate}}$ is one when the predicate is true otherwise it is zero. The greedy approach can be thus written as

$$A_t = \arg\max_a Q_t(a) \tag{2.4}$$

We can, however, easily convert this greedy method into one that also explores other states with probability ϵ. We call this method ϵ-greedy as it randomly chooses an action with probability ϵ and most of the time behaves in a greedy fashion. If we have an infinite number of time steps, we are guaranteed to have $Q_t(a)$ converge to $q(a)$. Moreover, the simple action-value based method is also an online learning approach which we will explain in detail in the next section.

2.2.1 *Online Prediction and Online Learning*

Online prediction problems are the class of problems where the agent has to make predictions about the future. For instance, imagine you have been in Hawaii for a week, and are asked to predict whether it will rain in the next days. Another example can be predicting afternoon oil prices based on the observed fluctuations in oil prices in the morning. Online prediction problems need to be solved with online methods. Online learning is distinguished from traditional statistic learning in the following aspects:

- The sample data is presented in an ordered sequence instead of an unordered batch.
- We will often have to consider the worst case rather than the average case because we do not want things to go out of control in the learning stage.
- The learning objective can be different as online learning often tries to minimize regret whereas statistical learning tries to minimize empirical risk. We will explain what regret is later.

Let us take look at a trivial example in the context of the MAB problem. Let us say, we observe a reward R_t at each time step t. An easy solution to find out what the best action is to update the estimate of the q value using R_t and A_t. A traditional way to compute the mean is to sum up all the previous rewards when A_t has been selected and divide that sum by the count of A_t. This approach is more like batch learning as it involves recomputing every time using a batch of data points. The online alternative would be to use a running average by doing the new estimate using the previous estimate: $Q_i(A_t) = Q_i(A_t) - Q_i(A_t)/N$; $Q_{i+1}(A_t) = Q_i(A_t) + R_t/N$. Q_i is the q value after A_t has been selected i times, and N is the number of times that A_t has been selected.

2.2.2 Stochastic Multi-Armed Bandit

Concretely, if we have $K \geq 2$ arms, we will need to select an arm to pull for each time step $t = 1, 2, \cdots, T$. At each step t, we can observe a reward R_t^i by selecting the ith arm.

Algorithm 1 Multi-armed bandit learning

```
Initialize $K$ arms;
Number of time steps $T$;
Each arm is associated with $v_i \in [0, 1]$. The reward being returned is drawn i.i.d from $v_i$
for $t = 1, 2, \ldots, T$ do
    The agent selects $A_t = i$ from the $K$ arms.
    The environment returns the reward vector $R_t = (R_t^1, R_t^2, \cdots, R_t^K)$.
    The agent observes reward $R_t^i$.
end for
```

In a traditional sense, one often tends to maximize the rewards. However, for a stochastic MAB, we will focus on another metric, regret. The regret after n steps is defined as:

$$RE_n = \max_{j=1,2,\ldots,K} \sum_{t=1}^{n} R_t^j - \sum_{t=1}^{n} R_t^i$$

The first term in the subtraction is the total reward that we accumulate until time n for always receiving the maximized rewards and the second term is the actual accumulated rewards in a trial that has gone through n time steps.

In order to select the best action, we should try to minimize the expected regret because of the stochasticity introduced by our actions and rewards. We will differentiate two different types of regret, the expected regret and pseudo-regret. The expected regret is defined as:

$$\mathbb{E}[RE_n] = \mathbb{E}\left[\max_{j=1,2,\ldots,T} \sum_{t=1}^{n} R_t^j - \sum_{t=1}^{n} R_t^i\right] \tag{2.5}$$

The pseudo-regret is defined as:

$$\overline{RE}_n = \max_{j=1,2,\ldots,T} \mathbb{E}\left[\sum_{t=1}^{n} R_t^j - \sum_{t=1}^{n} R_t^i\right] \tag{2.6}$$

The key distinction between the above two regrets is the order of the maximization and expectation. The expected regret is harder to compute. This is because for the pseudo-regret we only need to find the action that optimizes the regret in expectation, however, for the expected regret, we will have to find the expected regret that is optimal over actions across different trials. Concretely, we have $\mathbb{E}[RE_n] \geq \overline{RE}_n$.

Let μ_i be the mean of v_i, where v_i is the reward value of the $i-$th arm, $\mu^* = \max_{i=1,2,\ldots,T} \mu_i$. In the stochastic setting, we can rewrite Eq. (2.6) as:

$$\overline{RE}_n = n\mu^* - \mathbb{E}\left[\sum_{t=1}^{n} R_t^i\right] \tag{2.7}$$

One way to minimize the pseudo-regret is to select the best arm to pull given the observed sample pulls using ϵ-greedy which we already talked about before. A more sophisticated method is called Upper Confidence Bound (UCB) algorithm. UCB makes use of Hoeffding's lemma to derive an upper confidence bound estimate and chooses the arm whose sample mean has been the greatest so far.

We now introduce the concept of UCB strategy. The exact treatment of using UCB on stochastic MAB for regret optimization can be found in Bubeck et al. (2012). We will explain UCB for the situation when we optimize the policy with respect to the reward. In stochastic MAB, even though the rewards are drawn from a distribution, the reward function is still stationary over time. Let us refer back to the ϵ-greedy method. The ϵ-greedy method explores non-optimal states with a probability ϵ, but the issue is that it considers all the non-optimal states the same and does not make any differentiation. If we want to thoroughly visit every state, we should certainly prioritize the states that have not been visited yet or the states with fewer visits. UCB helps resolve this issue by rewriting Eq. (2.4) into:

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}\right] \tag{2.8}$$

$N_t(a)$ is the number of times action a has been selected till time t and c is a positive real number that determines how much exploration needs to be done. Eq. (2.8) is how we would select an action if we have a non-stationary reward function. When $N_t(a)$ is zero, we consider action a to have the max value. To understand how UCB works, let us focus on the square-root term, which reflects the amount of uncertainty we have for the q-value estimate for a. As the number of times that a is selected increases, the uncertainty decreases as the denominator decreases. Also as more actions other than a is being selected, the uncertainty increases because the $\ln t$ increases but $N_t(a)$ does not. The log operator on t is saying that the impact of incoming new time steps decays as we have more time steps in total. UCB gives some form of an upper bound of the q value for a, and c is the confidence level.

2.2.3 *Adversarial Multi-Armed Bandit*

Stochastic MAB's reward functions are determined by the probabilistic distributions that are usually not changed. In reality, this might not be the case. Imagine that if in a casino, a player wants to make a large profit by playing a group of slot machines and he has found out which machines are more likely to yield better returns, the casino owner will change the behavior of the machines so that the casino does not lose money. This is exactly why adversarial MAB modeling is needed when the rewards are no longer governed by a stationary probabilistic distributions but arbitrarily determined by some adversary. Formally, in adversarial MAB, the reward for the ith arm at time t will be denoted by $R_t^i \in [0, 1]$, at the same time, the player will select an arm at time t, which is denoted by $I_t \in \{1, \dots, K\}$.

One might wonder what if an adversary simply sets all the rewards to zero. This situation could happen, but there would be little point in playing this game if the player can get nothing in return. As a matter of fact, even when the opponent can freely decide what the rewards are going to look like, he would not do so because if all the rewards are zeros, no one will want to play the game anymore. His job is actually to give the player enough rewards just to trick the player in believing that he might have a chance of winning in the long term.

Algorithm 2 Adversarial multi-armed bandit

```
Initialize $K$ arms;
for $t = 1, 2, \dots, T$ do
    The agent selects $I_t$ from the $K$ arms.
    The adversary selects a reward vector $\boldsymbol{R}_t = (R_t^1, R_t^2, \dots, R_t^K) \in [0, 1]^K$
    The agent observes reward $R_t^{I_t}$ and maybe also observes the rest of the reward vector
    depending on the specific problem set up.
end for
```

Algorithm 2 describes the general setup for adversarial MAB. At each time step, the agent will choose an arm I_t to pull and the adversary will decide the reward vector $\boldsymbol{R}_t$ for this time step. The agent might only be able to observe the reward for the arm he selects $R_t^{I_t}$ or the possible rewards for all the machines, $\boldsymbol{R}_t(\cdot)$. We still need two more pieces of information for the problem formulation. The first one is how much the adversary knows about the player's past behavior. It matters because, for some casino owners, they might adapt their reward strategies based on the player's behavior for more benefit. We will call the adversary who sets the rewards independent of the past history an **oblivious adversary** and the one that sets the rewards based on the past history a **non-oblivious adversary**. The second piece of information we have to specify is how much the player knows about the reward vector. We call the game in which a player has full knowledge of the reward vector a **full-information game** and the game with the knowledge of the reward for the action being played a **partial-information game**.

The difference between the oblivious and non-oblivious adversaries only starts to matter when we have a non-deterministic player. If we have a deterministic player, a player whose game strategy does not change, it is fairly easy to show that an adversary can always lower the regret to $\overline{RE} \geq n/2$ where n is the number of pulls the player takes. Therefore, let us focus on a non-deterministic player with full information in the first place. We can make use of the hedge algorithm to tackle this problem. In Algorithm 3, we first set G function to be

Algorithm 3 Hedge for adversarial multi-armed bandit

```
Initialize $K$ arms;
$G_i(0)$ for $i = 1, 2, \dots, K$;
**for** $t = 1, 2, \dots, T$ **do**
    The agent selects $A_t = i_t$ from the distribution $p(t)$, where
        $$p_i(t) = \frac{exp(\eta G_i(t-1))}{\sum_j^K exp(\eta G_j(t-1))}$$
    The agent observes reward vector $g_t$.
    Let $G_i(t) = G(t-1) + g_t^i$, $\forall i \in [1, K]$.
**end for**
```

zero for all the arms, and use the softmax function to obtain the probability density function for the new action. η is a parameter greater than zero to control the temperature. The G function is updated by adding all the new rewards received for all the arms to allow the arm that has received the greatest reward being the most likely one to be selected. We refer this algorithm as Hedge. Hedge is also a building block for the method used under a partial-information game. If we want to limit the agent's observation to only R_t^i, then we will need to expand our reward scalar to a vector such that it can be passed to hedge. Exponential-weight algorithm for Exploration and Exploitation (Exp3) is the method that builds on Hedge for the partial information game. It further utilizes a blending of $p(t)$ and a uniform distribution to ensure that all the machines will get selected and hence the name exploration and exploitation. Auer et al. (1995) have more details on how Exp3 can be used and offer analysis on the confidence bound for regret.

2.2.4 Contextual Bandits

Contextual bandits are also sometimes called associative search tasks. The associative search tasks are best explained in comparison with the non-associative search tasks, the MAB tasks that we just described. When the reward function for the task is stationary, we only need to find the best action, otherwise, when the task is non-stationary we will try to keep track of the changes. This is the case for the non-associative search tasks, but reinforcement learning problems can become a lot more complicated. For instance, if we have several MAB tasks to play, and we will have to choose one at each time step. Even though we can still estimate the general expected reward, the performance is unlikely to be optimal. For cases like this, it would be useful to associate certain features of the slot machine with the learned expected reward. Imagine, for each slot machine, there is an LED light shining different colors at different times. Let us say if the machines with the red light always yield greater reward than the ones with the blue light, then we will be able to associate that information with our action selection policy, i.e. selecting the machines with red lights more.

Contextual bandit tasks are an intermediate between MAB tasks and the full reinforcement learning problems. They are similar to the MAB tasks because, for both situations, their actions only impact the immediate rewards. It is also similar to the full reinforcement learning setting because both require learning a policy function. To convert contextual bandits tasks to full reinforcement learning tasks, we will need to allow the actions to influence not just the intermediate rewards but also the future environment states.

2.3 Markov Decision Process

2.3.1 Markov Process

A Markov process (MP) is a discrete stochastic process with Markov property, which simplifies the simulation of the world in continuous space. Figure 2.4 shows an example of MP. Each circle represents a state and each edge represents a state transition. This graph simulates how a person works on two tasks and goes to bed in the end. To understand how this diagram works, let us look at this example together. Imagine, we are currently doing "Task1", and then with a probability of 0.7 we continue to execute "Task2", after which if we manage to pass with a probability of 0.6, we will pass the exam and then go straight to bed.

Figure 2.5 shows the **probabilistic graphical model** of MP in a probabilistic inference view, which will be frequently mentioned in later sections. In probabilistic graphical models, specifically the ones that we use in this book, a circle indicates a variable, and the arrow with a single direction indicates the relationship between two variables. For example, "$a \rightarrow b$" indicates that variable b is dependent on variable a. The variable in a circle in white denotes a normal variable, while the variable in a circle with a shade of gray denotes an observed variable (shown in later figures of Sect. 2.7), which provides information for taking an inference process of other normal variables. A solid black square with variables inside indicates those variables are iterative, which will be shown in later figures as well. The probabilistic graphical model can help us to have a more intuitive sense of the relationships between

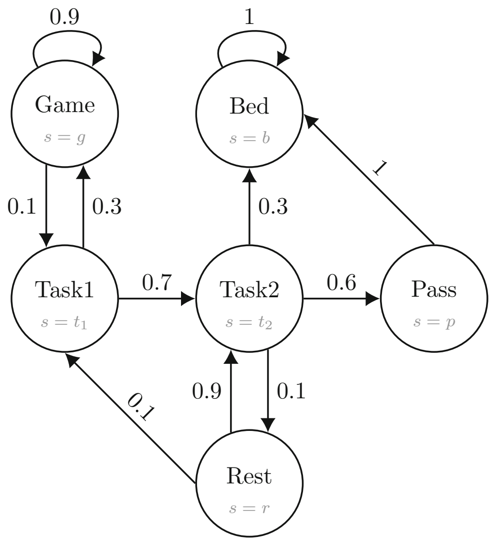

Fig. 2.4 A Markov process example. s denotes the current state and the values on the edges denote the probabilities of moving from one state to another

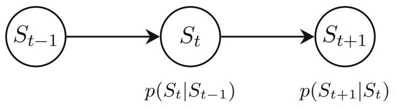

Fig. 2.5 Graphical model of Markov process: a finite representation with t indicating the time step and $p(S_{t+1}|S_t)$ indicating the state transition probability

variables in reinforcement learning, as well as providing rigorous references when we derive the gradients with respect to different variables along the MP chains.

MP follows the assumption of **Markov chain** where the next state S_{t+1} is only dependent on the current state S_t, with the probability of a state jumping to the next state described as follows:

$$p(S_{t+1}|S_t) = p(S_{t+1}|S_0, S_1, S_2, \ldots, S_t) \tag{2.9}$$

This formula describes the "memoryless" property, i.e. **Markov property**, of the Markov chain. Also, if $p(S_{t+2} = s'|S_{t+1} = s) = p(S_{t+1} = s'|S_t = s)$ holds for any time step t and for all possible states, then it is a stationary transition function along the time axis, which is called the **time-homogeneous** property, and the corresponding Markov chain is time-homogeneous Markov chain.

We also frequently use s' to represent the next state, in which the probability that state s at time t will lead to state s' at time $t + 1$ is as following in a time-homogeneous Markov chain:

$$p(s'|s) = p(S_{t+1} = s'|S_t = s) \tag{2.10}$$

The time-homogeneous property is a basic assumption for most of the derivations in the book, and we will not mention it but follow it as default in most cases. However, in practice, the time-homogeneous may not always hold, especially for non-stationary environments, multi-agent reinforcement learning, etc., which concerns with time-inhomogeneous/non-homogeneous cases.

Given a finite **state set** $\mathcal{S}$, we can have a **state transition matrix** $\boldsymbol{P}$. The $\boldsymbol{P}$ for Fig. 2.4 is as follows:

$$\boldsymbol{P} = \begin{array}{c} \begin{array}{cccccc} g & t_1 & t_2 & r & p & b \end{array} \\ \left[\begin{array}{cccccc} 0.9 & 0.1 & 0 & 0 & 0 & 0 \\ 0.3 & 0 & 0.7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.1 & 0.6 & 0.3 \\ 0 & 0.1 & 0.9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array} \begin{array}{c} \\ g \\ t_1 \\ t_2 \\ r \\ p \\ b \end{array}$$

where $\boldsymbol{P}_{i,j}$ represents the probability of transferring the current state S_i to the next state S_j. For example, in Fig. 2.4, state $s = r$ will jump to state $s = t_1$ with a probability of 0.1, and to state $s = t_2$ with 0.9. The sum of each row is equal to 1 and the $\boldsymbol{P}$ is always a square matrix. These probabilities indicate the whole process is stochastic. Markov process can be represented by a tuple of $< \mathcal{S}, \boldsymbol{P} >$. Many simple processes in our world can be represented by this random process as an approximation, which is also a foundation of reinforcement learning methods. Mathematically, the next state is sampled from $\boldsymbol{P}$ as follows:

$$S_{t+1} \sim \boldsymbol{P}_{S_t,\cdot}. \tag{2.11}$$

where symbol $\sim$ represents the next state S_{t+1} is randomly sampled according to the categorical distribution of $\boldsymbol{P}_{S_t,\cdot}$.

For infinite state set or continuous case, a finite matrix cannot be used to represent the transition relationship anymore. Therefore the transition function $p(s'|s)$ is applied as before, with a corresponding relationship $p(s'|s) = \boldsymbol{P}_{s,s'}$ for finite cases.

2.3.2 *Markov Reward Process*

Even though the agent can interact with the environment via the state transition matrix $\boldsymbol{P}_{s,s'} = p(s'|s)$, there is no way for MP to provide reward feedback from the environment to the agent. To provide the feedback, Markov reward process (MRP) extends MP from $< \mathcal{S}, \boldsymbol{P} >$ to $< \mathcal{S}, \boldsymbol{P}, R, \gamma >$. The R and γ represent the **reward function** and **reward discount factor**, respectively. An example of MRP is shown in Fig. 2.6. Figure 2.7 shows the graphical model of MRP in a probabilistic inference

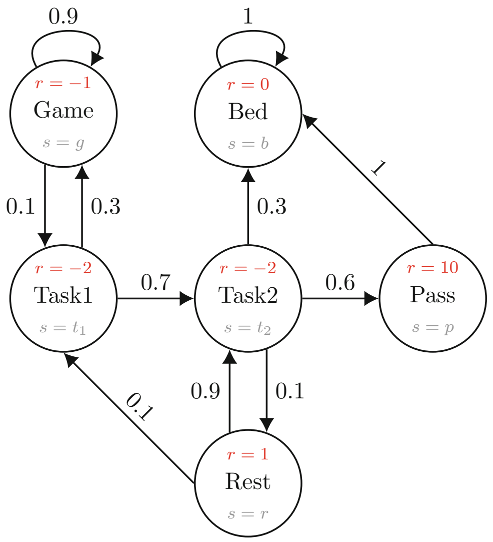

Fig. 2.6 A Markov reward process example. The s denotes the current state and the r denotes the immediate reward for each state. The values on the edges denote the probabilities of moving from one state to the next state

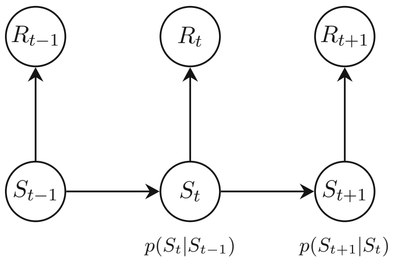

Fig. 2.7 Graphical model of a Markov reward process: a finite representation with t indicating the time step

view. The reward function depends on the current state:

$$R_t = R(S_t) \tag{2.12}$$

But the reward is also a result of the previous action based on the previous state. To better understand the reward as a function of the state, let us take a look at this example. If the agent passes the exam, the agent can obtain an immediate reward of ten, and taking a rest can obtain a reward of one, but if the agent works on a task, the reward of two will be lost. Given a list of immediate reward r for each time step in a single trajectory τ, **a return is the cumulative reward of a trajectory**, in which the **undiscounted return** of finite process with T time steps (not counting the initial one) is as follows:

$$G_{t=0:T} = R(\tau) = \sum_{t=0}^{T} R_t \tag{2.13}$$

where R_t is the immediate reward at time step t, and T represents the time step of the terminal state, or the total number of steps in a finite episode. For example, the trajectory (g, t_1, t_2, p, b) has an undiscounted return of $5 = -1 - 2 - 2 + 10$. Note that some other literature may use G to represent the return, and R to represent the immediate reward, but in this book, we use R as the reward function, therefore $R_t = R(S_t)$ gives the immediate reward at time step t, while $R(\tau) = G^{(T)}_{t=0}$ represents the return along the trajectory $\tau_{0:T}$, and r as a general representation of immediate reward value.

Often, the steps that are closer have a greater impact than the distant ones. We introduce the concept of **discounted return**. The discounted return is a weighted sum of rewards which gives more weights to the closer time steps. We define the discounted return as follows:

$$G_{t=0:T} = R(\tau) = \sum_{t=0}^{T} \gamma^t R_t. \tag{2.14}$$

where a **reward discount factor** $\gamma \in [0, 1]$ is used to reduce the weights as the time step increases. For example in Fig. 2.6, given $\gamma = 0.9$, the trajectory (g, t_1, t_2, p, b) has a return of $2.87 = -1 - 2 \times 0.9 - 2 \times 0.9^2 + 10 \times 0.9^3$. If $\gamma = 0$, the return is only related to the current immediate reward; if $\gamma = 1$, it is the undiscounted return. The discounted factor is especially critical when handling with infinite MRP cases, as it can prevent the return from going to infinite as the time step goes to infinite. Therefore it makes the infinite MRP process evaluative.

Another view of discount factor γ: for conciseness, the reward discount factor γ is sometimes omitted in literature (Levine 2018) in a discrete-time finite-horizon MRP. The discount factor can also be incorporated into the process by simply modifying the transition dynamics, such that any action produces a transition into an absorbing state with probability $1 - \gamma$, and all standard transition probabilities are multiplied by γ.

The **value function** $V(s)$ represents the **expected return from the state** s. For example, if there are two different next states S_1 and S_2, the values estimated with the current policy are $V^\pi(S_1)$ and $V^\pi(S_2)$. The agent policy usually selects the next state with higher value. If the agent acts according to the policy π, we denote the value function as $V^\pi(s)$:

$$V(s) = \mathbb{E}[R_t | S_0 = s] \tag{2.15}$$

A simple way to estimate the $V(s)$ is **Monte Carlo method**, we can randomly sample a large number of trajectories starting from state s according to the given state transition matrix $\boldsymbol{P}$. Take Fig. 2.6 as an example, given $\gamma = 0.9$ and $\boldsymbol{P}$, to estimate $V^\pi(s = t_2)$, we can randomly sample four trajectories as follows and compute the returns of all trajectories individually (Note that, in practice, the number of trajectories is usually far larger than four, but for demonstration purposes

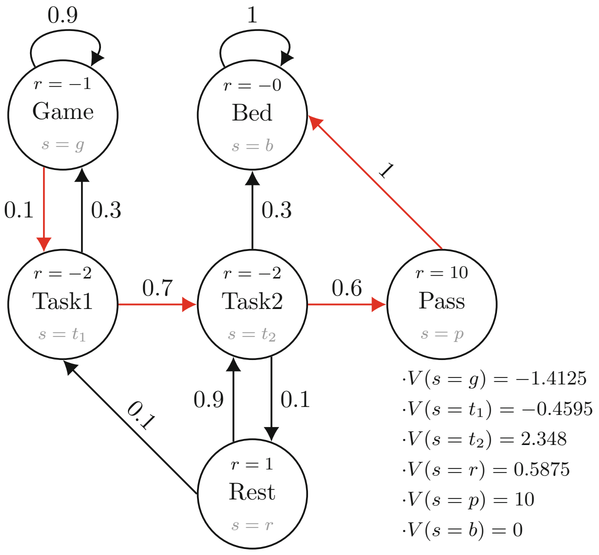

Fig. 2.8 Markov reward process and the estimated value function $V(s)$ by randomly choosing four trajectories for each state i.e., Monte Carlo method. The red edges indicate the learned policy

we simply sample four trajectories here.):

- $s = (t_2, b)$, $R = -2 + 0 \times 0.9 = -2$
- $s = (t_2, p, b)$, $R = -2 + 10 \times 0.9 + 0 \times 0.9^2 = 7$
- $s = (t_2, r, t_2, p, b)$, $R = -2 + 1 \times 0.9 - 2 \times 0.9^2 + 10 \times 0.9^3 + 0 \times 0.9^4 = 4.57$
- $s = (t_2, r, t_1, t_2, b)$, $R = -2 + 1 \times 0.9 - 2 \times 0.9^2 - 2 \times 0.9^3 + 0 \times 0.9^4 = -0.178$

Given the returns of all trajectories, the estimated expected return under state $s = t_2$ is $V(s = t_2) = (-2 + 7 + 4.57 - 0.178)/4 = 2.348$. Figure 2.8 shows all estimated expected returns for all states. Given the expected returns under all states, the simplest policy for the agent is to jump to the next state that has the highest expected return. The actions that can maximize the expected return are highlighted by red in Fig. 2.8. Apart from Monte Carlo methods, there are many other methods to compute $V(s)$, such as Bellman expectation equation and inverse matrix method, etc., which will be introduced later.

2.3.3 Markov Decision Process

Markov decision processes (MDPs) have been studied since the 1950s and have been widely used in modeling disciplines such as economics, control theory, and robotics. To model the process of sequential decision making, MDP is better than MR and MRP. As Fig. 2.9 shows, different from MRP that the immediate rewards are conditioned on the state only (reward values on nodes), the immediate rewards of MDP are associated with the action and state (reward values on edges). Likewise,

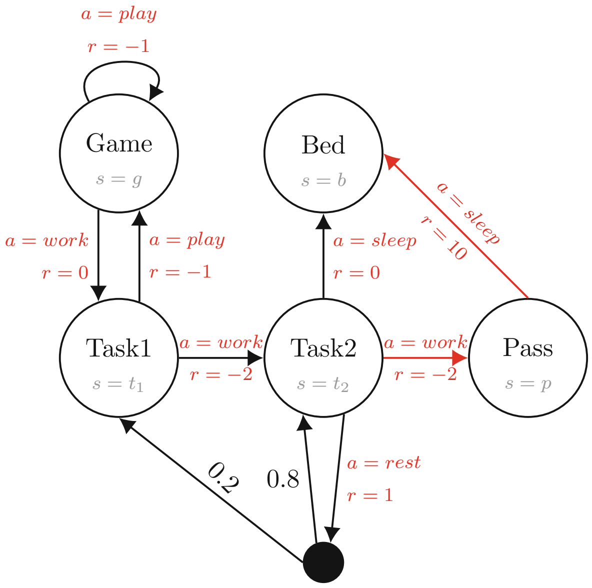

Fig. 2.9 A Markov decision process example, different from MRP in which the immediate rewards are associated with the state only. The immediate rewards of MDP is associated with the current state and the action just taken. The black solid node is an initial state

given an action under a state, the next state is not fixed. For example, if the agent acts "rest" under state $s = t_2$, the next state can be either $s = t_1$ or t_2. Fig. 2.10 shows the graphical model of MDP in a probabilistic inference view.

As mentioned above, MP can be defined as the tuple $< \mathcal{S}, \boldsymbol{P} >$, and MRP is defined as the tuple $< \mathcal{S}, \boldsymbol{P}, R, \gamma >$, where the element of state transition matrix is $\boldsymbol{P}_{s,s'} = p(s'|s)$. This representation extends the finite-dimension state transition matrix to an infinite-dimension probability function. Here, MDP is defined as a tuple of $< \mathcal{S}, \mathcal{A}, \boldsymbol{P}, R, \gamma >$. The element of state transition matrix becomes:

$$p(s'|s, a) = p(S_{t+1} = s'|S_t = s, A_t = a) \tag{2.16}$$

For instance, most of the edges in Fig. 2.9 have a state transition probability of one, e.g., $p(s' = t_2|s = t1, a = work) = 1$, except that $p(s'|s = t_2, a = rest) = [0.2, 0.8]$ which means if the agent performs action $a =$ rest at state $s = t_2$, it has 0.2 probability will transit to state $s' = t_1$, and 0.8 probability will keep the current state. The non-existing edges have a state transition probability of zero e.g., $p(s' = t_2|s = t_1, a = rest) = 0$.

$\mathcal{A}$ represents the **finite action set** $\{a_1, a_2, \ldots\}$, and the immediate reward becomes:

$$R_t = R(S_t, A_t) \tag{2.17}$$

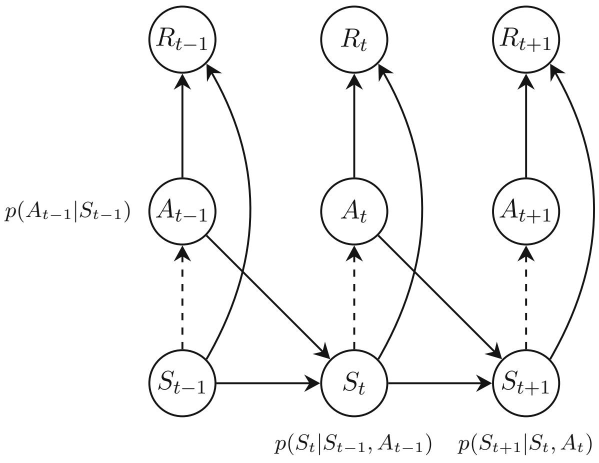

Fig. 2.10 Graphical model of Markov decision process: a finite representation with t indicating the time step, $p(A_t|S_t)$ as the action choice based on current state, and $p(S_{t+1}|S_t, A_t)$ as the state transition probability based on current state and action. The dashed lines indicate the decision process made by the agent

A **policy** π represents the way in which the agent behaves based on its observations of the environment. Specifically, the policy is a mapping from the each state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$ to the probability distribution $\pi(a|s)$ for taking action a in state s, where the distribution is:

$$\pi(a|s) = p(A_t = a|S_t = s), \exists t \tag{2.18}$$

Expected return is the expectation of returns over all possible trajectories under a policy. Therefore, **the goal of reinforcement learning is to find the higher expected return by optimizing the policy**. Mathematically, given the start-state distribution ρ_0 and the policy π, the probability of a T-step trajectory for MDP is:

$$p(\tau|\pi) = \rho_0(S_0) \prod_{t=0}^{T-1} p(S_{t+1}|S_t, A_t)\pi(A_t|S_t) \tag{2.19}$$

Given the reward function R and all possible trajectories τ, the **expected return** $J(\pi)$ is defined as follows:

$$J(\pi) = \int_\tau p(\tau|\pi)R(\tau) = \mathbb{E}_{\tau\sim\pi}[R(\tau)] \tag{2.20}$$

where p here means that the trajectory with higher probability will have a higher weight to the expected return. The **RL optimization problem** is to improve the policy for maximizing the expected return with optimization methods. The **optimal policy** π^*can be expressed as:

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{2.21}$$

where the $*$ symbol means "optimal" for the rest of the book.

Given policy π, the **value function** $V(s)$, the expected return under the state, can be defined as:

$$\begin{aligned} V^{\pi}(s) =& \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s] \\ =& \mathbb{E}_{A_t\sim\pi(\cdot|S_t)}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t)|S_0 = s\right] \end{aligned} \tag{2.22}$$

where $\tau \sim \pi$ means the trajectories τ are sampled given the policy π, $A_t \sim \pi(\cdot|S_t)$ means the action under a state is sampled from the policy, the next state is determined by the state transition matrix $\boldsymbol{P}$ given state s and action a.

In MDP, given an action, we have the **action-value function**, which depends on both the state and the action just taken. It gives an expected return under a state and an action. If the agent acts according to a policy π, we denote it as $Q^{\pi}(s, a)$, which is defined as:

$$\begin{aligned} Q^{\pi}(s, a) &= \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s, A_0 = a] \\ &= \mathbb{E}_{A_t\sim\pi(\cdot|S_t)}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t)|S_0 = s, A_0 = a\right] \end{aligned} \tag{2.23}$$

We need to keep in mind that the $Q^{\pi}(s, a)$ depends on π, as the estimation of the value is an expectation over the trajectories by the policy π. This also indicates if the π changes, the corresponding $Q^{\pi}(s, a)$ will also change accordingly. We therefore usually call the value function estimated with a specific policy the **on-policy value function**, for the distinction from the **optimal value function** estimated with the optimal policy.

We can observe the relation between $v_{\pi}(s)$ and $q_{\pi}(s, a)$:

$$q_{\pi}(s, a) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s, A_0 = a] \tag{2.24}$$

$$v_{\pi}(s) = \mathbb{E}_{a\sim\pi}[q_{\pi}(s, a)] \tag{2.25}$$

There are two simple ways to compute the value function $v_{\pi}(s)$ and action-value function $q_{\pi}(s, a)$: The first is the **exhaustive method** follows Eq. (2.19), it first computes the probabilities of all possible trajectories that start from a state, and then follows Eqs. (2.22) and (2.23) to compute the $V^{\pi}(s)$ and $Q^{\pi}(s, a)$ for this state.

The exhaustive method computes the $V^{\pi}(s)$ for each state individually. However, in practice, the number of possible trajectories would be large and even infinite. Instead of using all possible trajectories, we can use **Monte Carlo method** as described in the previous MRP section to estimate the $V^{\pi}(s)$ by randomly sampling a large number of trajectories. In reality, the estimation formulas of value functions can be simplified, by leveraging the Markov property in MRP, which leads to the Bellman equations in the next section.

2.3.4 Bellman Equation and Optimality

Bellman Equation

The Bellman equation, also known as the Bellman expectation equation, is used to compute the expectation of value function given policy π, over the sampled trajectories guided by the policy. We call this "on-policy" manner as in reinforcement learning the policy is usually changing, and the value function is conditioned on or estimated by current policy.

Recall that the definitions of a value function or an action-value function are $v_{\pi}(s) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s]$ and $q_{\pi}(s, a) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s, A_0 = a]$. We can derive the Bellman equation for on-policy state-value function in a recursive relationship:

$$\begin{aligned}
v_{\pi}(s) &= \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}[R(\tau_{t:T})|S_t = s] \\
&= \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots + \gamma^T R_T|S_t = s] \\
&= \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}[R_t + \gamma(R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T)|S_0 = s] \\
&= \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}[R_t + \gamma R_{\tau_{t+1:T}}|S_t = s] \\
&= \mathbb{E}_{A_t\sim\pi(\cdot|S_t),S_{t+1}\sim p(\cdot|S_t,A_t)}[R_t + \gamma\mathbb{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}[R_{\tau_{t+1:T}}]|S_t = s] \\
&= \mathbb{E}_{A_t\sim\pi(\cdot|S_t),S_{t+1}\sim p(\cdot|S_t,A_t)}[R_t + \gamma v_{\pi}(S_{t+1})|S_t = s] \\
&= \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}[r + \gamma v_{\pi}(s')]
\end{aligned} \tag{2.26}$$

The final formula above holds because s, a are general representations of states and actions, while S_t, A_t are state and action at time step t only. S_t, A_t are sometimes separated from the general representations s, a to show more clearly over whom the expectation is taken over in some of above formulas.

Note that in the above derivation we show the Bellman equation for MDP process, however, the Bellman equation for MRP can be derived by simply removing the action from it:

$$v(s) = \mathbb{E}_{s'\sim p(\cdot|s)}[r + \gamma v(s')] \tag{2.27}$$

There is also Bellman equation for on-policy action-value function: $q_\pi(s,a) = \mathbb{E}_{s'\sim p(\cdot|s,a)}[R(s,a) + \gamma \mathbb{E}_{a'\sim\pi(\cdot|s')}[q_\pi(s',a')]]$, which can be derived as follows:

$$\begin{aligned}
&q_\pi(s,a)\\
&\quad= \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}[R(\tau_{t:T})|S_t = s, A_t = a]\\
&\quad= \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots + \gamma^T R_T|S_t = s, A_t = a]\\
&\quad= \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}[R_t + \gamma(R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T)|S_t = s, A_t = a]\\
&\quad= \mathbb{E}_{S_{t+1}\sim p(\cdot|S_t,A_t)}[R_t + \gamma \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}[R_{\tau_{t+1:T}}]|S_t = s]\\
&\quad= \mathbb{E}_{S_{t+1}\sim p(\cdot|S_t,A_t)}[R_t + \gamma \mathbb{E}_{A_{t+1}\sim\pi(\cdot|S_{t+1})}[q_\pi(S_{t+1},A_{t+1})]|S_t = s]\\
&\quad= \mathbb{E}_{s'\sim p(\cdot|s,a)}[R(s,a) + \gamma \mathbb{E}_{a'\sim\pi(\cdot|s')}[q_\pi(s',a')]]
\end{aligned}$$

The above derivation is based on the finite MDP with maximal length of T, however, these formulas still hold when in the infinite MDP, simply with T replaced by "∞". The two Bellman equations also do not depend on the formats of policy, which means they work for both stochastic policies $\pi(\cdot|s)$ and deterministic policies $\pi(s)$. The usage of $\pi(\cdot|s)$ is for simplicity here. Also, in deterministic transition processes, we have $p(s'|s,a) = 1$.

Solutions of Bellman Equation

The Bellman equation for MRP as in Eq. (2.27) can be solved directly if the transition function/matrix is known, which is called the **inverse matrix method**. We rewrite Eq. (2.27) in a vector form for cases with discrete and finite state space as:

$$\boldsymbol{v} = \boldsymbol{r} + \gamma \boldsymbol{P}\boldsymbol{v} \tag{2.28}$$

where $\boldsymbol{v}$ and $\boldsymbol{r}$ are vectors with their elements $v(s)$ and $R(s)$ for all $s \in \mathcal{S}$, and $\boldsymbol{P}$ is the transition probability matrix with elements $p(s'|s)$ for all $s, s' \in \mathcal{S}$.

Given $\boldsymbol{v} = \boldsymbol{r} + \gamma \boldsymbol{P}\boldsymbol{v}$, we can directly solve it with:

$$\boldsymbol{v} = (\mathbf{1} - \gamma \boldsymbol{P})^{-1}\boldsymbol{r} \tag{2.29}$$

the complexity of the solution is $O(n^3)$, where n is the number of states. Therefore this method does not work for a large number of states, meaning it may not be feasible for large-scale or continuous-valued problems. Fortunately, there are some iterative methods for solving the large-scale MRP problems in practice, like dynamic programming, Monte Carlo estimation, and temporal-difference learning, which will be introduced in detail in later sections.

Optimal Value Functions

Since on-policy value functions are estimated with respect to the policy itself, different policies will lead to different value functions, even for the same set of states and actions. Among all those different value functions, we define the optimal value function as:

$$v_*(s) = \max_\pi v_\pi(s), \forall s \in \mathcal{S}, \tag{2.30}$$

which is actually the **optimal state-value function**. We also have the **optimal action-value function** as:

$$q_*(s,a) = \max_\pi q_\pi(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}, \tag{2.31}$$

And they have the relationship:

$$q_*(s,a) = \mathbb{E}[R_t + \gamma v_*(S_{t+1})|S_t = s, A_t = a], \tag{2.32}$$

which can be derived easily by taking the maximization in the last formula of Eq. (2.26) and plugging in Eqs. (2.25) and (2.30):

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R(s,a) + \gamma \max_\pi \mathbb{E}\left[q_\pi\left(s',a'\right)\right]\right] \\ &= \mathbb{E}\left[R(s,a) + \gamma \max_\pi v_\pi\left(s'\right)\right] \\ &= \mathbb{E}[R_t + \gamma v_*(S_{t+1})|S_t = s, A_t = a]. \end{aligned} \tag{2.33}$$

Another relationship between the two is:

$$v_*(s) = \max_{a \sim \mathcal{A}} q_*(s,a) \tag{2.34}$$

which is obvious by simply maximizing the two sides of Eq. (2.25).

Bellman Optimality Equation

In the above sections we introduced on-policy Bellman equations for normal value functions, as well as the definitions of optimal value functions. So we can apply the Bellman equation on the pre-defined optimal value functions, which gives us the **Bellman optimality equation**, or called Bellman equation for optimal value functions, as follows.

The Bellman equation for optimal state-value function is:

$$v_*(s) = \max_a \mathbb{E}_{s' \sim p(\cdot|s,a)}[R(s,a) + \gamma v_*(s')], \tag{2.35}$$

which can be derived as follows:

$$\begin{aligned}
v_*(s) &= \max_a \mathbb{E}_{\pi^*, s' \sim p(\cdot|s,a)}[R(\tau_{t:T})|S_t = s] \\
&= \max_a \mathbb{E}_{\pi^*, s' \sim p(\cdot|s,a)}\Big[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots + \gamma^T R_T | S_t = s\Big] \\
&= \max_a \mathbb{E}_{\pi^*, s' \sim p(\cdot|s,a)}[R_t + \gamma R_{\tau_{t+1:T}} | S_t = s] \\
&= \max_a \mathbb{E}_{s' \sim p(\cdot|s,a)}\Big[R_t + \gamma \max_{a'} \mathbb{E}_{\pi^*, s' \sim p(\cdot|s,a)}\left[R_{\tau_{t+1:T}}\right] | S_t = s\Big] \\
&= \max_a \mathbb{E}_{s' \sim p(\cdot|s,a)}[R_t + \gamma v_*(S_{t+1}) | S_t = s] \\
&= \max_a \mathbb{E}_{s' \sim p(\cdot|s,a)}[R(s,a) + \gamma v_*(s')]
\end{aligned} \tag{2.36}$$

Bellman equation for optimal action-value function is:

$$q_*(s,a) = \mathbb{E}_{s' \sim p(\cdot|s,a)}[R(s,a) + \gamma \max_{a'} q_*(s',a')], \tag{2.37}$$

which can be derived similarly. Readers can take a practice by finishing the proof.

2.3.5 Other Important Concepts

Deterministic and Stochastic Policies

In the previous sections, the policy is represented as a probability distribution as $\pi(a|s) = p(A_t = a|S_t = s)$, where the action of the agent is sampled from the distribution. A policy with action sampled from the probability distribution is actually called the **stochastic policy distribution**, with the action:

$$a = \pi(\cdot|s) \tag{2.38}$$

However, if we reduce the variance of the probability distribution of a stochastic policy and narrow down its range to the limit, we will get a Dirac delta function (δ function) as a distribution, which is the **deterministic policy** $\pi(s)$. Deterministic policy $\pi(s)$ also means given a state there is only one unique action as follows:

$$a \sim \pi(s) \tag{2.39}$$

Note that the deterministic policy is no longer a mapping from a state and action to the conditional probability distribution, but rather a mapping from a state to an action directly. This slight difference will lead to some different derivations in the policy gradient method introduced in later sections. More detailed categories of policies in reinforcement learning, especially for deep reinforcement learning with parameterized policies are introduced in Sect. 2.7.3.

Partially Observed Markov Decision Process

As mentioned in previous sections, when the state in reinforcement learning environment is not fully represented by the observation for the agent, the environment is partially observable. For a Markov decision process, it is called the partially observed Markov decision process (POMDP), which forms a challenge for improving the policy without complete information of the environment states.

2.3.6 Summary of Terminology in Reinforcement Learning

Apart from the terminology in mathematics notations at the beginning of the book, the summary of terminology for common contents in reinforcement learning is provided as follows:

- R the reward function, $R_t = R(S_t)$ as the reward of state S_t for MRP, $R_t = R(S_t, A_t)$ for MDP, $S_t \in \mathcal{S}$.
- $R(\tau)$ the γ-discounted return of a trajectory τ, $R(\tau) = \sum_{t=0}^{\infty} \gamma^t R_t$.
- $p(\tau)$ the probability of a trajectory:
  - $p(\tau) = \rho_0(S_0) \prod_{t=0}^{T-1} p(S_{t+1}|S_t)$ for MP and MRP, $\rho_0(S_0)$ as start-state distribution.
  - $p(\tau|\pi) = \rho_0(S_0) \prod_{t=0}^{T-1} p(S_{t+1}|S_t, A_t)\pi(A_t|S_t)$ for MDP, $\rho_0(S_0)$ as start-state distribution.
- $J(\pi)$ the expected return of policy π, $J(\pi) = \int_\tau p(\tau|\pi)R(\tau) = \mathbb{E}_{\tau\sim\pi}[R(\tau)]$
- π^* optimal policy: $\pi^* = \arg\max_\pi J(\pi)$
- $v_\pi(s)$ value of state s under policy π (expected return)
- $v_*(s)$ value of state s under the optimal policy
- $q_\pi(s, a)$ value of taking action a in state s under policy π
- $q_*(s, a)$ value of taking action a in state s under the optimal policy
- $V(s)$ the estimates of state-value function for MRP starting from state s:

- $V^{\pi}(s)$ the estimates of on-policy state-value function for MDP, given a policy π, we have expected return:
  - $V^{\pi}(s) \approx v_{\pi}(s) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s]$
- $Q^{\pi}(s, a)$ the estimates of on-policy action-value function for MDP, given a policy π, we have expected return:
  - $Q^{\pi}(s, a) \approx q_{\pi}(s, a) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s, A_0 = a]$
- $V^{*}(s)$ the estimates of optimal state-value function for MDP, we have expected return according to the optimal policy:
  - $V^{*}(s) \approx v_{*}(s) = \max_{\pi} \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s]$
- $Q^{*}(s, a)$ the estimates of on-policy action-value function for MDP, we have expected return according to the optimal policy:
  - $Q^{*}(s, a) \approx q_{*}(s, a) = \max_{\pi} \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s, A_0 = a]$
- $A^{\pi}(s, a)$ the estimated advantage function of state s and action a:
  - $A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s)$
- Relationship of on-policy state-value function $v_{\pi}(s)$ and on-policy action-value function $q_{\pi}(s, a)$:
  - $v_{\pi}(s) = \mathbb{E}_{a\sim\pi}[q_{\pi}(s, a)]$
- Relationship of optimal state-value function $v_{\pi}(s)$ and optimal action-value function $q_{\pi}(s, a)$:
  - $v_{*}(s) = \max_{a} q_{*}(s, a)$
- $a_{*}(s)$ the optimal action for state s according to optimal action-value function:
  - $a_{*}(s) = \arg\max_{a} q_{*}(s, a)$
- Bellman equations for on-policy state-value function:
  - $v_{\pi}(s) = \mathbb{E}_{a\sim\pi(\cdot|s), s'\sim p(\cdot|s,a)}[R(s, a) + \gamma v_{\pi}(s')]$
- Bellman equations for on-policy action-value function:
  - $q_{\pi}(s, a) = \mathbb{E}_{s'\sim p(\cdot|s,a)}[R(s, a) + \gamma \mathbb{E}_{a'\sim\pi(\cdot|s')}[q_{\pi}(s', a')]]$
- Bellman equations for optimal state-value function:
  - $v_{*}(s) = \max_{a} \mathbb{E}_{s'\sim p(\cdot|s,a)}[R(s, a) + \gamma v_{*}(s')]$
- Bellman equations for optimal action-value function:
  - $q_{*}(s, a) = \mathbb{E}_{s'\sim p(\cdot|s,a)}[R(s, a) + \gamma \max_{a'} q_{*}(s', a')]$

2.4 Dynamic Programming

Dynamic Programming (DP) was first introduced by Richard E. Bellman in the 1950s (Bellman et al. 1954) and has been successfully applied to a range of challenging fields. In this term, "dynamic" means that the problem has sequential or temporal components, and "programming" refers to an optimizing policy. DP provides a general framework for complex dynamic problems by breaking them down into sub-problems. For example, each number in the Fibonacci sequence is the sum of the two preceding ones, starting from 0 and 1. One can calculate the 4th number $F_4 = F_3 + F_2$ by $F_4 = (F_2 + F_1) + F_2$ by reusing the solution of the preceding sub-problem $F_2 = F_1 + F_0$. However, DP requires full knowledge of the environment, such as the reward model and the transition model, of which we often have limited knowledge in reinforcement learning. Nonetheless, it does provide a fundamental framework for learning to interact with the MDP incrementally, as most of the reinforcement learning algorithms attempt to achieve.

There are two properties that a problem must have for DP to be applicable: **optimal substructure** and **overlapping sub-problems**. Optimal substructure means that the optimal solution of a given problem can be decomposed into solutions to its sub-problems. Overlapping sub-problems implies that the number of sub-problems is finite and sub-problems occur recursively so that the sub-solutions can be cached and reused. While MDPs with finite actions and states satisfy both properties, the Bellman equation gives the recursive decomposition and value functions store the optimal solution of sub-problems. So in this section, we assume that state set and action set are both finite, and a perfect model of the environment is given.

2.4.1 Policy Iteration

Policy Iteration aims to manipulate the policy directly. Starting from arbitrary policy π, we can evaluate it by applying the Bellman equation recursively:

$$v_\pi(s) = \mathbb{E}_\pi[R_t + \gamma v_\pi(S_{t+1}) \,|\, S_t = s] \tag{2.40}$$

where the expectation is over all possible transitions based on full knowledge of the environment. A natural idea to obtain a better policy is acting greedily with respect to v_π:

$$\pi'(s) = \text{greedy}(v_\pi) = \arg\max_{a \in \mathcal{A}} q_\pi(s, a). \tag{2.41}$$

The improvement can be proved by:

$$\begin{aligned}
v_\pi(s) &= q_\pi(s, \pi(s)) \\
&\leq q_\pi(s, \pi'(s)) \\
&= \mathbb{E}_{\pi'}[R_t + \gamma v_\pi(S_{t+1}) \,|S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_t + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \,|S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_t + \gamma R_{t+1} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2})) \,|S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots |S_t = s] = v_{\pi'}(s).
\end{aligned} \tag{2.42}$$

Apply policy evaluation and greedy improvement above successively until $\pi = \pi'$ forms the policy iteration. Generally, the procedure of policy iteration can be summarized as follows. Given an arbitrary policy π_t, for each state s in each iteration t, we first evaluate $v_{\pi_t}(s)$ and then find a better policy π_{t+1}. We call the former stage **policy evaluation** and the later stage **policy improvement**. Furthermore, we use the term **generalized policy iteration** (**GPI**) to refer to the general interaction of policy evaluation and policy improvement, as shown in Fig. 2.11.

One fundamental question is whether the process of policy iteration converges on the optimal value v_*. At each iteration in policy evaluation, for fixed and deterministic policy π, the value function update can be rewritten by the **Bellman expectation backup operator** $\mathcal{T}^\pi$:

$$(\mathcal{T}^\pi V)(s) = (\mathcal{R}^\pi + \gamma \mathcal{P}^\pi V)(s) = \sum_{r,s'} (r + \gamma V(s')) P(r, s'|s, \pi(s)). \tag{2.43}$$

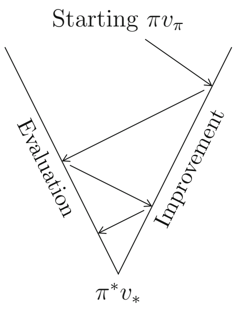

Fig. 2.11 Generalized policy iteration

Then for arbitrary value functions V and V', we have the following contraction proof for $\mathcal{T}^\pi$:

$$\begin{aligned}|\mathcal{T}^\pi V(s) - \mathcal{T}^\pi V'(s)| &= |\sum_{r,s'}(r + \gamma V(s'))P(r, s'|s, \pi(s)) \\ &\quad - \sum_{r,s'}(r + \gamma V'(s'))P(r, s'|s, \pi(s))| \\ &= |\sum_{r,s'}\gamma(V(s') - V'(s'))P(r, s'|s, \pi(s))| \\ &\leq \sum_{r,s'}\gamma|V(s') - V'(s')|P(r, s'|s, \pi(s)) \\ &\leq \sum_{r,s'}\gamma\|V - V'\|_\infty P(r, s'|s, \pi(s)) \\ &= \gamma\|V - V'\|_\infty,\end{aligned} \tag{2.44}$$

where $\|V - V'\|_\infty$ is the ∞-norm. By contraction mapping theorem (also known as the Banach fixed-point theorem), iterative policy evaluation will converge on the unique fixed point of $\mathcal{T}^\pi$. Since $\mathcal{T}^\pi v_\pi = v_\pi$ is a fixed point, so that iterative policy evaluation converges on v_π. Note that the policy improvement is monotonic, and there is only a finite number of greedy policies with respect to value functions in finite MDP. The policy improvement will stop after a finite number of steps, i.e., the policy iteration will converge on v_*.

2.4.2 Value Iteration

The theoretical basis of **value iteration** is the **principle of optimality** which tells us that π is the optimal policy on one state if and only if π achieves the optimal value for any reachable successor state. So if we know the solution to sub-problems $v_*(s')$, we can find the solution of any initial state s by one-step full backups:

$$v_*(s) = \max_{a\in\mathcal{A}} R(s, a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s, a)v_*(s'). \tag{2.45}$$

The procedure of value iteration is to apply the updates above from the final state and backward successively. Similar to the convergence proof in policy iteration, the **Bellman optimality operator** $\mathcal{T}^*$:

$$(\mathcal{T}^* V)(s) = (\max_{a\in\mathcal{A}} \mathcal{R}^a + \gamma\mathcal{P}^a V)(s) = \max_{a\in\mathcal{A}} R(s, a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s, a)V(s') \tag{2.46}$$

Algorithm 4 Policy iteration

```
Initialize $V$ and $\pi$ for all states
repeat
    // Do policy evaluation
    repeat
        $\delta \leftarrow 0$
        for $s \in \mathcal{S}$ do
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_{r,s'}(r + \gamma V(s'))P(r, s'|s, \pi(s))$
            $\delta \leftarrow \max(\delta, |v - V(s)|)$
        end for
    until $\delta$ is smaller than a positive threshold
    // Do policy improvement
    $stable \leftarrow true$
    for $s \in \mathcal{S}$ do
        $a \leftarrow \pi(s)$
        $\pi(s) \leftarrow \arg\max_a \sum_{r,s'}(r + \gamma V(s'))P(r, s'|s, a)$
        if $a \neq \pi(s)$ then
            $stable \leftarrow false$
        end if
    end for
until $stable = true$
return policy $\pi$
```

is also a contraction mapping for arbitrary value functions V and V'

$$\begin{aligned}
|\mathcal{T}^* V(s) - \mathcal{T}^* V'(s)| &= |\max_{a\in\mathcal{A}}\left[R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V(s')\right] \\
&\quad - \max_{a\in\mathcal{A}}\left[R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V'(s')\right]| \\
&\le \max_{a\in\mathcal{A}} |R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V(s') - R(s,a) \\
&\quad - \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V'(s')| \\
&= \max_{a\in\mathcal{A}} |\gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)(V(s') - V'(s'))| \\
&\le \max_{a\in\mathcal{A}} \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)|V(s') - V'(s')|
\end{aligned} \tag{2.47}$$

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

Since v_* is a fixed point of $\mathcal{T}^*$, the value iteration converges on the optimal value v_*. Note that in value iteration, only the actual value of successor states are known. In other words, the values are not complete so we use value function V instead of value v in the proof above.

It is not obvious when to stop the value iteration algorithm. Williams and Baird III (1993) gives a sufficient stopping criterion in theory that if the maximum difference between two successive value functions is less than ϵ, then the value of the greedy policy differs from the value function of the optimal policy by no more than $\frac{2\epsilon\gamma}{1-\gamma}$ at any state.

Algorithm 5 Value iteration

```
Initialize $V$ for all states
**repeat**
    $\delta \leftarrow 0$
    **for** $s \in \mathcal{S}$ **do**
        $u \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{r,s'} P(r,s'|s,a)(r + \gamma V(s'))$
        $\delta \leftarrow \max(\delta, |u - V(s)|)$
    **end for**
**until** $\delta$ is smaller than a positive threshold
Output greedily policy $\pi(s) = \arg\max_a \sum_{r,s'} P(r,s'|s,a)(r + \gamma V(s'))$
```

2.4.3 Other DPs: Asynchronous DP, Approximate DP, Real-Time DP

DP methods described so far use synchronous backups, i.e., the value of each state is backed up on the basis of systematic sweeps. One of the efficient variants is asynchronous updates, which is a trade-off between speed and accuracy. Asynchronous DP is also available for reinforcement learning settings and is guaranteed to converge if all states continue to be selected. There are three simple ideas behind asynchronous DP:

1. In-Place Update
   Synchronous value iteration stores two copies of value function $V_{t+1}(\cdot)$ and $V_t(\cdot)$:

$$V_{t+1}(s) \leftarrow \max_{a\in\mathcal{A}} R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V_t(s'). \tag{2.48}$$

   In-place value iteration only stores one copy of value function:

$$V(s) \leftarrow \max_{a\in\mathcal{A}} R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V(s'). \tag{2.49}$$

2. Prioritized Sweeping
   In asynchronous DP, one more thing that needs to be considered is the update order. Given a transition (s, a, s'), prioritized sweeping views the absolute value of its Bellman error as its magnitude:

$$|V(s) - \max_{a\in\mathcal{A}}(R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)V(s'))|. \tag{2.50}$$

   It can be implemented efficiently by maintaining a priority queue where the Bellman error of each state is stored or updated after each backup.
3. Real-Time Update
   After each time step t, no matter which action is taken, real-time update will only back up the current state S_t by:

$$V(S_t) \leftarrow \max_{a\in\mathcal{A}} R(S_t,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|S_t,a)V(s'). \tag{2.51}$$

   It can be viewed as selecting the states to update by the guide of the agent's experience.

Both synchronous and asynchronous DP back up over the full state set to estimate the expected return of the next state. Under the perspective of probability, a biased but efficient choice is using sampled data. We will discuss this topic extensively in the next section.

2.5 Monte Carlo

Unlike DP, Monte Carlo (MC) methods do not require perfect knowledge of the environment. MC only needs experience for learning. MC is also a class of sampling-based methods. MC can obtain good performance by learning from experience with little prior knowledge about the environment. "Monte Carlo" refers to the class of algorithms that have a large component of randomness. Indeed so, when using MC in reinforcement learning, we will average the rewards for each state-action pair from different episodes. One example can be with the contextual bandit problem that we talked about earlier in this chapter. If there is an LED light on different slot machines, the player can gradually learn from the association between the lighting information with the relevant reward. We will consider a particular arrangement of the lights for our state, and the corresponding possible reward is the value for this state. Initially, we might not have a good estimate for the state-value, but gradually as we play more, the average state-value pairs should be closer to the real ones. In this section, we will investigate how to do this estimation properly and then how to make the best use of this information. Also, we assume that the problem is episodic and an episode will always terminate regardless of the actions taken.

2.5.1 Monte Carlo Prediction

State-Value Prediction To start with, we will look at the case when using MC methods to estimate the state-value function for a given policy π. The most intuitive way to do this is to estimate the state-value function from experience by simply averaging the return from a particular policy. More specifically, let the function $v_\pi(s)$ be the state-value function under policy π. We then collect a pool of episodes that pass through s. We call each appearance of state s in an episode a visit to state s. There are two types of estimations, **first-visit MC** and **every-visit MC**. The first-visit MC only considers the return of the first visit to state s in the whole episode, however, every-visit MC considers every visit to state s in the episode. These two methods share lots of similarities but have a few theoretical differences. In Algorithm 6, we are showing exactly how $v_\pi(s)$ is computed with first-visit MC estimation. It is simple to convert the first-visit MC prediction to the every-visit MC prediction by removing the check for a state being the first state. Note that both types of methods will converge to $v_\pi(s)$ if we take the number of visits to state s to infinity.

MC methods can estimate different states independently from each other. Unlike DP, MC does not use bootstrapping, estimating the value of the current step with the estimation from other steps (e.g. the next step). This unique feature will enable one to estimate the state-value directly from the true sampled returns, which can be of less bias but higher variances.

The state-value function will be handy if a model is given as we can easily select the optimal action for an arbitrary state by looking at the combined average of the state-value for a specific action, as in DP. When a model is not known, we will have to estimate the state-action value instead. Each state-action pair has to be estimated separately. Now, one learning objective has become the estimation of $q_\pi(s, a)$, the expected return at state s by performing action a, under policy π. This is essentially the same as the estimation of the state-value function as we can just take the average return at state s and when action a is taken. The only issue is that there might exist

Algorithm 6 First-visit MC prediction

```
Input: Initialize policy $\pi$
Initialize $V(s)$ for all states
Initialize a list of returns: $Returns(s)$ for all states
**repeat**
    Generate an episode under $\pi$: $S_0, A_0, R_0, S_1, \cdots, S_{T-1}, A_{T-1}, R_t$
    $G \leftarrow 0$
    $t \leftarrow T - 1$
    **for** $t >= 0$ **do**
        $G \leftarrow \gamma G + R_{t+1}$
        **if** $S_0, S_1, \cdots, S_{t-1}$ does not have $S_t$ **then**
            $Returns(S_t)$.append($G$)
            $V(S_t) \leftarrow$ mean($Returns(S_t)$)
        **end if**
        $t \leftarrow t - 1$
    **end for**
**until** convergence
```

states that could never be visited, and thus have zero return. To choose the optimal strategy, we must fully explore all states. A naive solution to this issue is to directly specify the starting state-action pair for each episode and each state-action pair has a non-zero probability of getting selected. In this way, we can ensure that all state-action pairs can be visited if we have enough episodes. We refer this assumption as exploring starts.

2.5.2 Monte Carlo Control

Now, we shall adapt GPI to MC to see how it is used in control. Recall that GPI consists of two stages: policy evaluation and policy improvement. Policy evaluation is the same as that of DP as introduced in the previous section and therefore, we will discuss more about policy improvement. We will make use of a greedy policy for the action-value, we do not need to have a model in this case. The greedy policy will always choose the action that has maximal value for a given state:

$$\pi(s) = \arg\max_a q(s, a) \tag{2.52}$$

We will go from policy evaluation to policy improvement. For each policy improvement, we will need to construct π_{t+1} based on q_{π_t}. We will show how the policy improvement theorem is applicable here:

$$q_{\pi_t}(s, \pi_{t+1}(s)) = q_{\pi_t}(s, \arg\max_a q_{\pi_t}(s, a)) \tag{2.53}$$

$$= \max_a q_{\pi_t}(s, a) \tag{2.54}$$

$$\geq q_{\pi_t}(s, \pi_t(s)) \tag{2.55}$$

$$\geq v_{\pi_t}(s) \tag{2.56}$$

The above proves that π_{t+1} will be no worse than π_t, and thus eventually the optimal policy can be found. This means that we can use MC for control without much knowledge about the environment but only the sampled episodes. Here, we have two assumptions that we need to resolve. The first is the exploring starts and the second is that we have an infinite number of episodes. We will keep the exploring starts for now but focus on the second assumption. An easy way to relax this assumption is to avoid the infinite number of episodes needed for policy evaluation by directly alternating between evaluation and improvement for single states.

Algorithm 7 MC exploring starts

```
Initialize $\pi(s)$ for all states
Initialize $Q(s, a)$ and $Returns(s, a)$ for all state-action pairs
**repeat**
    Randomly select $S_0$ and $A_0$ s.t. all state-action pairs' probabilities are nonzero.
    Generate an episode from $S_0, A_0$ under $\pi$: $S_0, A_0, R_0, S_1, \cdots, S_{T-1}, A_{T-1}, R_t$
    $G \leftarrow 0$
    $t \leftarrow T - 1$
    **for** $t >= 0$ **do**
        $G \leftarrow \gamma G + R_{t+1}$
        **if** $S_0, A_0, S_1, A_1 \cdots, S_{t-1}, A_{t-1}$ does not have $S_t, A_t$ **then**
            $Returns(S_t, A_t)$.append($G$)
            $Q(S_t, A_t) \leftarrow$ mean($Returns(S_t, A_t)$)
            $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$
        **end if**
        $t \leftarrow t - 1$
    **end for**
**until** convergence
```

2.5.3 Incremental Monte Carlo

As we have seen in both Algorithm 6 and Algorithm 7, we have to take the averages of the lists of observed rewards, the state values and the state-action values respectively. There exists a more efficient computational method that allows us to get rid of the lists of observed returns and simplify the mean calculation step. We will thus do the update in an episode by episode way. We let the $Q(S_t, A_t)$ be the estimation of the state-action value after it has been selected for $t - 1$ times, which can be then rewritten as:

$$Q(S_t, A_t) = \frac{G_1 + G_2 + \cdots + G_{t-1}}{t - 1} \tag{2.57}$$

The naive implementation of this is to keep a record of all the returned G values, and then divide their sum by the visit times. However, we can also compute the same value by the following:

$$Q_{t+1} = \frac{1}{t}\sum_{i=1}^{t} G_i \tag{2.58}$$

$$= \frac{1}{t}\left(G_t + \sum_{i=1}^{t-1} G_i\right) \tag{2.59}$$

$$= \frac{1}{t}\left(G_t + (t-1)\frac{1}{t-1}\sum_{i=1}^{t-1} Q_i\right) \tag{2.60}$$

$$= \frac{1}{t}(G_t + (t-1)Q_t) \tag{2.61}$$

$$= Q_t + \frac{1}{t}(G_t - Q_t) \tag{2.62}$$

The formulation will give us a much easier time when it comes to the return computation. This can also appear in a more general form as:

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize} \cdot (\text{Target} - \text{OldEstimate}) \tag{2.63}$$

The “StepSize” is a parameter that controls how fast the estimate is being updated.

2.6 Temporal Difference Learning

Temporal difference (TD) learning describes another class of algorithm that is at the core of reinforcement learning by combining the ideas both from DP and MC. Similar to DP, TD uses bootstrapping in the estimation, however, like MC, it does not require full knowledge of the environment in the learning process, but applies a sampling-based optimization approach. In this chapter, we will first introduce how TD can be used in policy evaluation and then elaborate on the differences and commonalities between MC, TD, and DP. Lastly, we will end this chapter with Q-learning, an extremely useful and powerful learning algorithm in both classical reinforcement learning and deep reinforcement learning.

2.6.1 TD Prediction

As its name suggests, TD utilizes the error, the difference between the target value and the estimated value, at different time steps to learn. That reason why it is also using bootstrapping is that TD forms the target from the observed return and an estimated state value for the next state. More precisely, the most basic TD method makes the update using:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.64}$$

This method is also called TD(0) or one-step TD for looking one-step ahead. N-step TD can also be developed easily by extending the target value with discounted rewards in the N-step future and the estimated state value at the N-th step. If we observe carefully, the target value during update for MC is G_t which is known only after one episode, whereas for TD the target value is $R_{t+1} + \gamma V(S_{t+1})$ which can be computed step by step. In Algorithm 8, we are showing how TD(0) can be used to do policy evaluation.

Algorithm 8 TD(0) for state-value estimation

```
Input policy $\pi$
Initialize $V(s)$ and step size $\alpha \in (0, 1]$
for each episode do
    Initialize $S_0$
    for Each step $S_t$ in the current episode do
        $A_t \leftarrow \pi(S_t)$
        $R_{t+1}, S_{t+1} \leftarrow Env(S_t, A_t)$
        $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$
    end for
end for
```

Before we move on, it is worthwhile spending time to take a closer look at what DP, MC, and TD have in common and how they differ from one another. These three types of algorithms sit at the heart of reinforcement learning and often their usage is combined together in modern reinforcement learning application. Even though all of them can be used for policy evaluation and policy improvement, their subtle differences can contribute to major performance variations in deep reinforcement learning.

Some forms of GPI are being used by these three methods. The main difference lies in their policy evaluation schemes. The most obvious difference is that both DP and TD use bootstrapping but MC does not, and DP requires full knowledge of the model but MC and TD do not. Furthermore, let us dive deeper into how the learning objectives differ among these three.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.65}$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \tag{2.66}$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \tag{2.67}$$

Equation (2.65) stands for the state value estimation for MC methods and Eq. (2.67) represents the same for DP methods. Both of them are only estimation but not the true values. TD combines both the MC sampling and DP bootstrapping. We will now explain briefly how TD can be better than either DP or MC.

First of all, TD does not need a model to learn which DP requires. When TD is being compared with MC, TD is using an online learning approach meaning it can learn at every step, however, in order for MC methods to learn, it will have to wait until one episode is finished which can be tricky to deal with if the task has very long episodes. There are also problems that are continuous and cannot be learned in an episodic fashion. Moreover, TD can be faster because it can learn from transitions disregarding the actions being taken. MC cannot do this. Both TD and MC methods will eventually converge to $v_\pi(s)$ asymptotically, nonetheless, we do not have a proof to show which converges faster but that TD methods converge faster empirically.

Before we move on, it is also worth discussing the **variance and bias trade-off** between TD and MC methods. We know that in a supervised setting a large bias means that the model is underfitting for the data distribution and a large variance means that the model is overfitting the data. The bias of an estimator is the difference between the estimation and the true value. In the case of state value estimation, bias can be defined as $\mathbb{E}[V(S_t)] - V(S_t)$. The variance of an estimator describes how noisy the estimator is. Again for state-value estimation, variance can be defined as $\mathbb{E}[(\mathbb{E}[V(S_t)] - V(S_t))^2]$. In prediction, regardless of whether it is for the state or state-action approximation, both TD and MC are doing the update of the form:

$$V(S_t) \leftarrow V(S_t) + \alpha[\text{Target Value} - V(S_t)]$$

Essentially, we are doing a weighted average across different episodes. TD and MC differ in their ways of handling the target value. MC methods directly estimate the accumulative rewards until the end of an episode, which is exactly how the state value is defined. They will have no bias, however TD has a greater bias because its target is estimated with bootstrapping method, $R_{t+1} + \gamma v_\pi(S_{t+1})$. Now, let us see why MC tends to have a larger variance. The accumulated reward it has is computed until the end of each episode, which can vary a lot as different episodes can have vastly different outcomes. TD resolves this issue by just looking at the target value locally depending on the current reward and the estimated reward for the next state/action. Naturally, TD should have less variance.

TD(λ)

DP and MC have lots of similarities and perhaps there is a mid-ground between these two paradigms which could be more efficient in solving the problem at hand. Indeed, TD (λ) is the mid-ground between DP and MC, but we will need to introduce the concept of **eligibility traces** and λ return first.

To put it in a simple way, an eligibility trace can provide us with various computational advantages. To see this, we need to talk about semi-gradient methods quickly and then explain how eligibility traces can be used. For a detailed treatment of the policy-gradient method, please refer to Sect. 2.7. Here, we are simply using some concepts from the gradient-based methods to explain what eligibility traces can do. Imagine if our state-value function is not in a tabular form, but in a functional form parameterized by a weight vector $\boldsymbol{w} \in \mathbb{R}^n$. $\boldsymbol{w}$ can be, for instance, be the weight for a neural network. We aim to have $v(s, \boldsymbol{w}) \approx v_\pi(s)$. To do this, we can use stochastic gradient update to reduce the quadratic loss between our approximation and the true value function. The update rule w.r.t. the weight vector can be written as the following:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{1}{2}\alpha \nabla_{\boldsymbol{w}_t}[v_\pi(S_t) - V(S_t, \boldsymbol{w}_t)]^2 \tag{2.68}$$

$$= \boldsymbol{w}_t + \alpha[v_\pi(S_t) - V(S_t, \boldsymbol{w}_t)]\nabla_{\boldsymbol{w}_t} V(S_t, \boldsymbol{w}_t) \tag{2.69}$$

where α is a positive step size.

An eligibility trace is a vector $z_t \in \mathbb{R}^n$, which is used in such a way that, during learning, whenever a component of w_t is used for estimation, the corresponding component value in z_t also increases and then starts to fade away. The learning will take place if there is a TD error happening before the value in the trace falls back to zero. We first initialize all values using zero and then increase the trace using the gradient. The decay rate is $\gamma\lambda$:

$$z_{-1} = 0 \tag{2.70}$$

$$z_t = \gamma\lambda z_{t-1} + \nabla_{\boldsymbol{w}_t} V(S_t, \boldsymbol{w}_t) \tag{2.71}$$

It becomes easy to see that when $\lambda = 1$, the former sum becomes zero and the return is the same as that of an MC method. When $\lambda = 0$, it essentially becomes a one-step TD method. This is because the trace will always only contain the gradient of the one-step TD error. An eligibility trace is thus a great way to combine MC and TD methods.

Moving along, a λ-return is an estimated return value over the next n steps. λ-returns are a combination of n discounted returns with an existing estimate at the last step. Formally, it can be written as:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n v(S_{t+n}, \boldsymbol{w}_{t+n-1}) \tag{2.72}$$

t here is a nonzero scalar and is also less than or equal to $T - n$. We can make use of a weighted return in estimation as long as their weights sum up to one. TD(λ) makes use of this weighted averaging in its update with $\lambda \in [0, 1]$:

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_{t:t+n} \tag{2.73}$$

Intuitively, what this means is that the very next step return has the largest weight $1 - \lambda$, the two-step return has a weight of $(1 - \lambda)\lambda$, and the weight decays at each step with a rate of λ. To have a clear picture, let us have a terminal state at time T, then the above can be rewritten as

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n} + \lambda^{T-t-1}G_t \tag{2.74}$$

The TD error δ_t can be defined as:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}, \boldsymbol{w}_t) - V(S_t, \boldsymbol{w}_t) \tag{2.75}$$

The update rule is based on the proportion of the TD error and trace. See Algorithm 9 for details.

Algorithm 9 Semi-gradient TD(λ) for state-value

```
Input: policy $\pi$
Initialize a differentiable state function v, step size $\alpha$ and value function weight **w**
**for** each episode **do**
    Initialize $S_0$
    $z \leftarrow 0$
    **for** Each step $S_t$ in the current episode **do**
        Select $A_t$ using policy that is based on $\pi$
        $R_{t+1}, S_{t+1} \leftarrow Env(S_t, A_t)$
        $z \leftarrow \gamma\lambda z + \nabla V(S_t, \boldsymbol{w}_t)$
        $\delta \leftarrow R_{t+1} + \gamma V(S_{t+1}, \boldsymbol{w}_t) - V(S_t, \boldsymbol{w}_t)$
        $\boldsymbol{w} \leftarrow w + \alpha\delta z$
    **end for**
**end for**
```

2.6.2 *Sarsa: On-Policy TD Control*

For TD control, the methodology is similar to that of the prediction task except that we will transition from the state-to-state alternation to state-action pair alternation.

The update rule can, therefore, be framed as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (2.76)$$

When S_t is the terminal state, the Q value for the next state-action pair will be zero. It is referred to by the acronym Sarsa because we have this chain of being in a state, taking an action, receiving a reward and being in a new state to take another action. The chain allows us to do a simple update step. The state value gets updated for each transition and the updated state value is influencing the policy being used to determine the behavior, so it is also an on-policy method. On-policy methods generally describe the class of methods that have an update policy which is the same as its behavior policy, whereas, for the off-policy methods, these two are different. An example of an off-policy method is Q-learning which we will talk about later. It assumes a greedy approach while doing the update of its q-value function, whereas, in fact, for its behavior it is using other policies such as ϵ-greedy. We now entail the steps for Sarsa in Algorithm 10. We will have the convergence for both the optimal policy and action-state values as long as each state-action pair is visited an infinite number of times.

Algorithm 10 Sarsa (on-policy TD control)

```
Initialize $Q(s, a)$ for all state-action pairs.
for each episode do
    Initialize $S_0$
    Select $A_0$ using policy that is based on $Q$
    for Each step $S_t$ in the current episode do
        Select $A_t$ from $S_t$ using policy that is based on $Q$
        $R_{t+1}, S_{t+1} \leftarrow Env(S_t, A_t)$
        Select $A_{t+1}$ from $S_{t+1}$ using policy that is based on $Q$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$
    end for
end for
```

What we have shown only has one step time horizon meaning that the approximation only involves the state-action value of the next step. We can call this 1-step Sarsa or Sarsa(0), although we can easily extend the bootstrapped target value to include future steps down the road to, for instance, reduce our bias. As shown by the backup tree in Fig. 2.12, we see an array of Sarsa variants' state-action spectrum starting from the most basic 1-step Sarsa all the way up to the infinite step Sarsa which is an equivalent of the MC because its target value accounts for the accumulated rewards until the terminal state. To incorporate this change, we rewrite the discounted returns as the following:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) \quad (2.77)$$

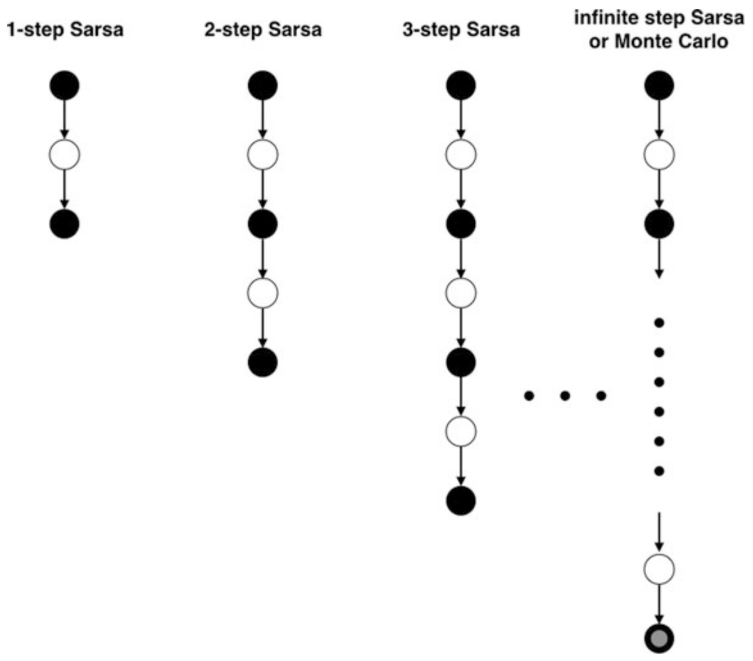

Fig. 2.12 The backup tree for the coverage of the n-step Sarsa methods. Each black circle represents a state and each white circle represents an action. The last state of the infinite step Sarsa is a terminal state

The n-step Sarsa is described in Algorithm 11. The major difference it has from the one-step version is that it has to go back in time to do the update, whereas the one-step version can do the update as it goes.

Convergence of Sarsa

Now we will discuss the convergence theory of Sarsa algorithm for finite action space (discrete cases), which requires some additional conditions described below.

Definition 2.1 A learning policy is defined as **Greedy in the Limit with Infinite Exploration (GLIE)** if it satisfies the following two properties:

1. If a state is visited infinitely often, then each possible action in that state is chosen infinitely often, i.e., $\lim_{k\to\infty} N_k(s,a) = \infty, \forall a$, if $\lim_{k\to\infty} N_k(s) = \infty$.
2. The policy converges on a greedy policy with respect to the learned Q-function in the limit (as $t \to \infty$), i.e., $\lim_{k\to\infty} \pi_k(s,a) = \mathbb{1}(a == \arg\max_{a'\in\mathcal{A}} Q_k(s,a'))$, where the "==" is a comparison operator and $\mathbb{1}(a == b)$ is 1 if true and 0 otherwise.

Algorithm 11 n-step Sarsa

```
Initialize $Q(s, a)$ for all state-action pairs.
Initialize step-size $\alpha \in (0, 1]$.
Determine a fixed policy $\pi$ or use $\epsilon$-greedy.
for each episode do
  Initialize $S_0$
  Select $A_0$ using $\pi(S_0, A)$
  $T \leftarrow$ INTMAX (the length of an episode)
  $\gamma \leftarrow 0$
  for $t \leftarrow 0, 1, 2, \ldots$ until $\gamma - T - 1$ do
    if $t < T$ then
      $R_{t+1}, S_{t+1} \leftarrow Env(S_t, A_t)$
      if $S_{t+1}$ is terminal then
        $T \leftarrow t + 1$
      else
        Select $A_{t+1}$ using $\pi(S_t, A)$
      end if
    end if
    $\tau \leftarrow t - n + 1$ (the time step to update. This is an n-step Sarsa, so we will only update the
    estimate that is $n + 1$ steps ago and we will continue to do so until all the eligible states
    have been updated.
    if $\tau \geq 0$ then
      $G \leftarrow \sum_{i=\tau+1}^{min(r+n,T)} \gamma^{i-\gamma-1} R_i$
      if $\gamma + n < T$ then
        $G \leftarrow G + \gamma^n Q(S_{t+n}, A_{\gamma+n})$
      end if
      $Q(S_\gamma, A_\gamma) \leftarrow Q(S_\gamma, A_\gamma) + \alpha[G - Q(S_\gamma, A_\gamma)]$
    end if
  end for
end for
```

The GLIE is a condition for the convergence of the learning policies, for any reinforcement learning algorithm that converges to the optimal value function and whose estimates are always bounded. For example, we can derive a GLIE policy with ϵ-greedy strategy as follows:

Lemma 2.1 *The ϵ-greedy policy is GLIE if ϵ reduces to zero with $\epsilon_k = \frac{1}{k}$.*

We can therefore have the convergence theorem of Sarsa algorithm.

Theorem 2.1 *For a finite state-action MDP and a GLIE learning policy, with the action-value function Q estimated with Sarsa (1-step) by Q_t for time step t. Then Q_t converges to Q^* and the learning policy π_t converges to an optimal policy π^*, if the following conditions are satisfied:*

1. *The Q values are stored in a lookup table;*
2. *The learning rate $\alpha_t(s, a)$ associated with the state-action pair (s, a) at time t satisfies $0 \leq \alpha_t(s, a) \leq 1$, $\sum_t \alpha_t(s, a) = \infty$ and $\sum_t \alpha_t^2(s, a) < \infty$ and $\alpha_t(s, a) = 0$ unless $(s, a) = (S_t, A_t)$;*
3. *$Var[R(s, a)] < \infty$.*

A typical sequence of the learning rate as the second condition required is $\alpha_t(S_t, A_t) = \frac{1}{t}$. The proofs of above theorems are not introduced here, but interested readers can refer to the paper by Singh et al. (2000).

2.6.3 Q-Learning: Off-Policy TD Control

Q-learning is an off-policy TD method that is very similar to Sarsa and plays an important role in deep reinforcement learning application such as the deep Q-network, which we will discuss in the next chapter. As shown in Eq. (2.78), the main difference that Q-learning has from Sarsa is that the target value now is no longer dependent on the policy being used but only on the state-action function.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (2.78)$$

Algorithm 12 Q-learning (off-policy TD control)

```
Initialize $Q(s, a)$ for all state-action pairs and step size $\alpha \in (0, 1]$
for each episode do
    Initialize $S_0$
    for Each step $S_t$ in the current episode do
        Select $A_t$ using policy that is based on $Q$
        $R_{t+1}, S_{t+1} \leftarrow Env(S_t, A_t)$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$
    end for
end for
```

In Algorithm 12, we have shown how Q-learning can be used for TD control. It is easy to convert Q-learning to Sarsa by first choosing the action using the state and return, and second changing the target value in the update step to be the estimated action value for the next step instead. This is also a one-step version. We can adapt the Q-learning into a n-step version by adapting the target value in Eq. (2.78) to include the discounted returns for the future steps.

Convergence of Q-Learning

The convergence of Q-learning follows similar conditions as the Sarsa algorithm. Apart from the GLIE condition for the policy, the convergence of Q function in Q-learning also requires the same requirements on its learning rate and the bounded reward values, which will not be duplicated here. Details and proofs are available in the papers (Szepesvári 1998; Watkins and Dayan 1992).

2.7 Policy Optimization

2.7.1 Overview

In reinforcement learning, the ultimate goal of the agent is to improve its policy to acquire better rewards. Policy improvement in the optimization domain is called policy optimization (Fig. 2.13). For deep reinforcement learning, the policy and value functions are usually parameterized by variables in deep neural networks, and therefore enable the gradient-based optimization methods to be applied. For example, Fig. 2.14 shows the graphical model of MDP with the policy parameterized by variables θ, on a discrete finite time horizon $t = 0, \ldots, N - 1$. The reward function follows $R_t = R(S_t, A_t)$ and action $A_t \sim \pi(\cdot|S_t; \theta)$. The dependencies among variables in the graphical models can help us to understand the underlying relationships of the MDP for estimation, and it can be useful when we take derivatives on the final objective to optimize variables on the dependency graphs, so we will display all those graphical models in this chapter to help understand the deduction process, especially for differential process. Recently, Levine (2018), Fu et al. (2018) proposed the method of **control as inference**, which uses a graphical model with additional variables indicating optimality on the MDP to incorporate the probabilistic/variational inference framework into maximum entropy reinforcement learning with the same objective. This method enables the inference tools to be applied in the reinforcement learning policy optimization process. But the details of those methods are beyond the scope of the book here.

Apart from some linear methods, the parameterization of value functions with deep neural networks is one way of achieving **value function approximation**, and it's the most popular way in the modern deep reinforcement learning domain. Value function approximation is useful because we cannot always acquire the true value function easily, and actually we cannot get the true function for most cases in practice. Figure 2.15 shows the model of MDP with both parameterized policy

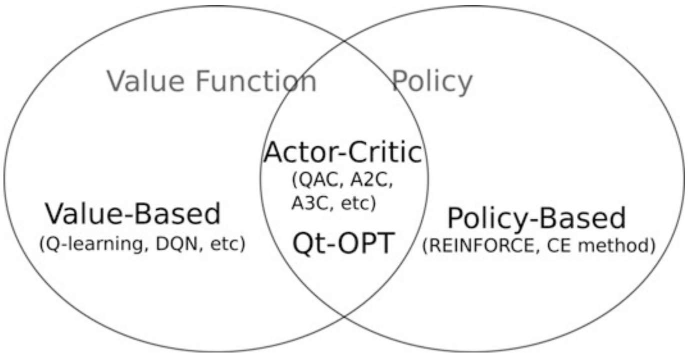

Fig. 2.13 Overview of policy optimization in reinforcement learning

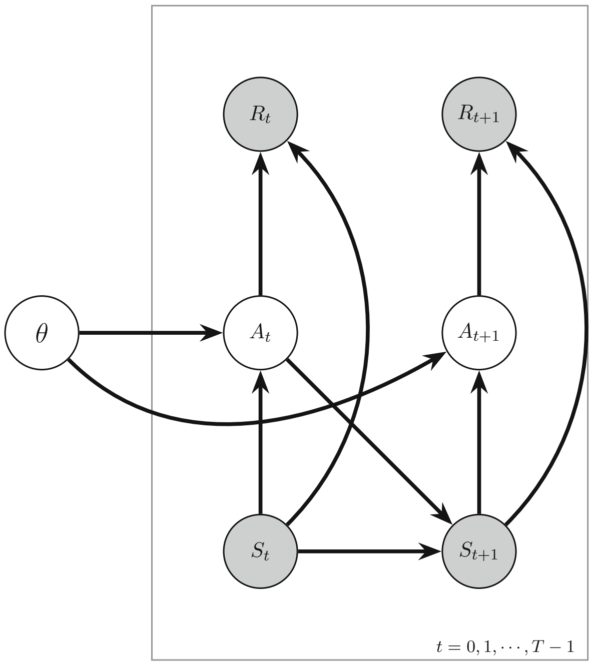

Fig. 2.14 Graphical model of MDP with parameterized policy

π_θ and parameterized value function $V_w^\pi(S_t)$, via parameters θ and w respectively. Figure 2.16 shows the model with parameterized policy π_θ and Q-value functions $Q_w^\pi(S_t, A_t)$. The gradient-based optimization methods can be used for improving parameterized policies, usually through the method called **policy gradient** in reinforcement learning terminology. However, there are also non-gradient-based methods for optimizing less complicated policies, like the cross-entropy (CE) method and so on.

As shown in Fig. 2.13, the methods in policy optimization fall into two main categories: (1) **value-based optimization** methods like Q-learning, DQN, etc., which optimize the action-value function to obtain the preferences for the action choice, and (2) **policy-based optimization** methods like REINFORCE, the cross-entropy method, etc., which directly optimize the policy according to the sampled reward values. A combination of these two categories was found to be a more effective approach by people (Sutton et al. 2000; Peters and Schaal 2008; Kalashnikov et al. 2018), which forms one of the most widely used architecture in model-free reinforcement learning called **actor-critic**. Actor-critic methods employ the optimization of value function as the guidance of policy improvement. The typical algorithms in the combined category include actor-critic-based algorithms and other algorithms built upon that, which will be described in detail in later this chapter and the following chapters.

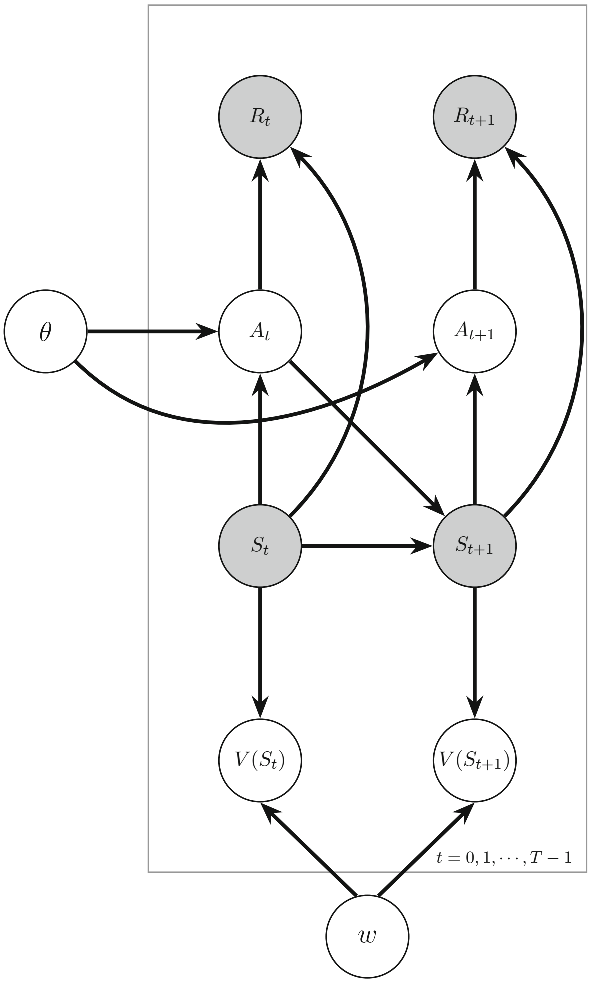

Fig. 2.15 Graphical model of MDP with parameterized policy and parameterized value functions

Recap of RL Skeleton

The **On-policy Value Function**, $v_\pi(s)$, which gives the expected return if you start in state s and always act according to policy π:

$$v_\pi(s) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s] \tag{2.79}$$

Recall that the reinforcement learning optimization problem can be expressed as:

$$\pi_* = \arg\max_\pi J(\pi) \tag{2.80}$$

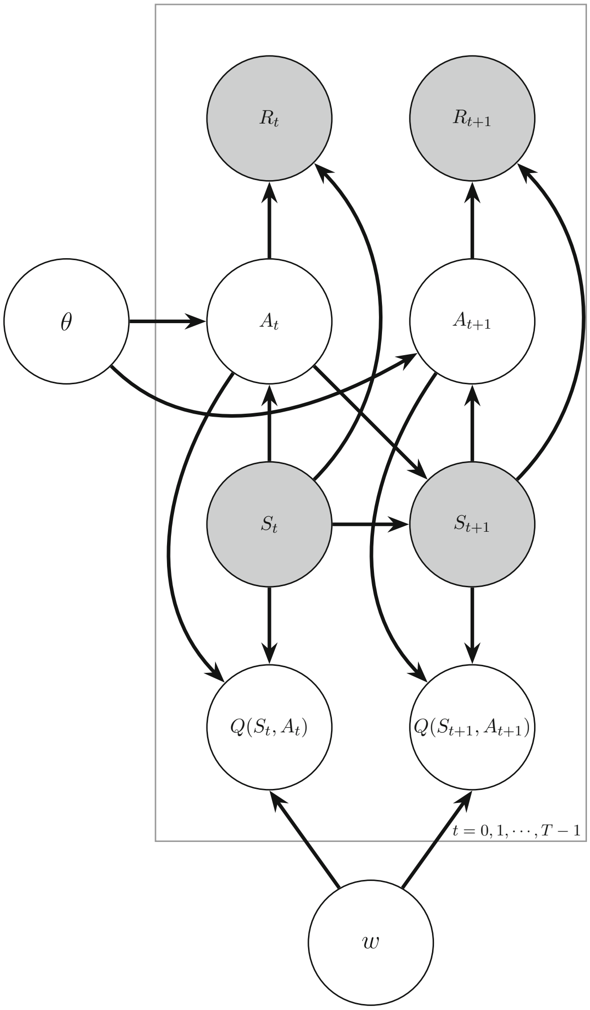

Fig. 2.16 Graphical model of MDP with parameterized policy and parameterized Q-value functions

The **Optimal Value Function**, $V^*(s)$, which gives the expected return if we start in state s and always act according to the optimal policy in the environment:

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.81}$$

$$v_*(s) = \max_\pi \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s] \tag{2.82}$$

The **On-Policy Action-Value Function**, $q_\pi(s, a)$, which gives the expected return if we start in state s, take an arbitrary action a (which may not come from the policy), and then forever after act according to policy π:

$$q_\pi(s, a) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s, A_0 = a] \tag{2.83}$$

The **Optimal Action-Value Function**, $q_*(s, a)$, which gives the expected return if you start in state s, take an arbitrary action a, and then forever after act according to the optimal policy in the environment:

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{2.84}$$

$$q_*(s, a) = \max_\pi \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s, A_0 = a] \tag{2.85}$$

Value Function and Action-Value Function

$$v_\pi(s) = \mathbb{E}_{a\sim\pi}[q_\pi(s, a)] \tag{2.86}$$

$$v_*(s) = \max_a q_*(s, a) \tag{2.87}$$

Optimal Action

$$a_*(s) = \arg\max_a q_*(s, a) \tag{2.88}$$

Bellman Equations
Bellman equations for state value and action value are:

$$v_\pi(s) = \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}[R(s, a) + \gamma v_\pi(s')] \tag{2.89}$$

$$q_\pi(s, a) = \mathbb{E}_{s'\sim p(\cdot|s,a)}[R(s, a) + \gamma \mathbb{E}_{a'\sim\pi(\cdot|s')}[q_\pi(s', a')]] \tag{2.90}$$

Bellman Optimality Equations
Bellman optimality equations for state value and action value are:

$$v_*(s) = \max_a \mathbb{E}_{s'\sim p(\cdot|s,a)}[R(s, a) + \gamma v_*(s')] \tag{2.91}$$

$$q_*(s, a) = \mathbb{E}_{s'\sim p(\cdot|s,a)}[R(s, a) + \gamma \max_{a'} q_*(s', a')] \tag{2.92}$$

2.7.2 Value-Based Optimization

A **value-based optimization** method always needs to alternate between value function estimation under the current policy and policy improvement with the estimated value function. However, the estimation of a complex value function may not be a trivial problem (Fig. 2.17).

From the previous sections, we see that the Q-learning can be used for solving some simple tasks in reinforcement learning. However, the real-world applications

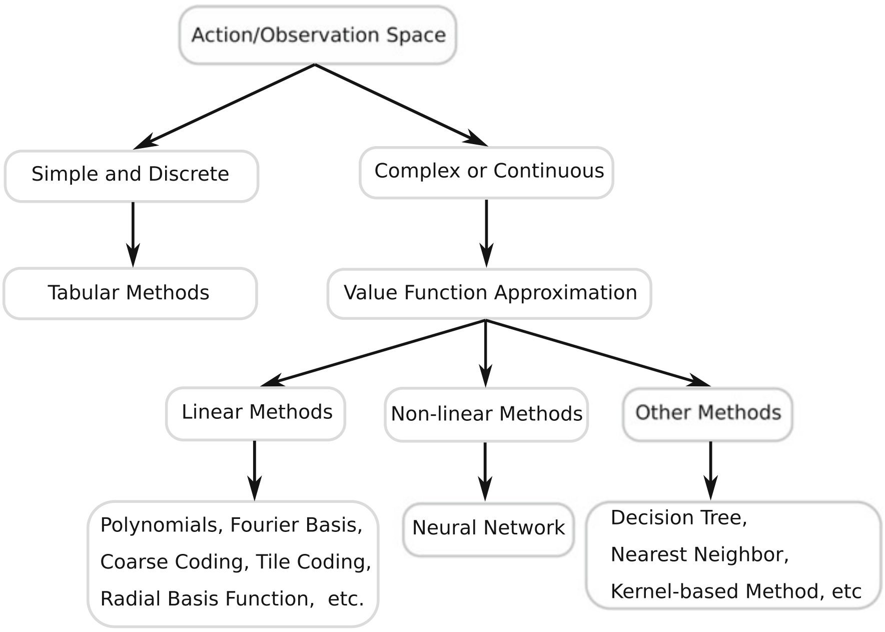
Action/Observation Space
Simple and Discrete
Complex or Continuous
Tabular Methods
Value Function Approximation
Linear Methods
Non-linear Methods
Other Methods
Polynomials, Fourier Basis, Coarse Coding, Tile Coding, Radial Basis Function, etc.
Neural Network
Decision Tree, Nearest Neighbor, Kernel-based Method, etc

Fig. 2.17 An overview of methods for solving the value function

or even the quasi-real-world applications may have much larger and complicated state and action spaces, and the action is usually continuous in practice. For example, the Go game has 10^{170} states. In these cases, the traditional lookup table method in Q-learning cannot work well with the limitation of its scalability, because each state will have an entry $V(s)$ and each state-action pair will need an entry $Q(s, a)$. The values in the table are updated one-by-one in practice. Therefore the requirement of the memory and computational resources will be huge with tabular-based Q-learning. Moreover, state representations usually need to be manually specified with aligned data structures in practice.

Value Function Approximation

In order to apply the value-based reinforcement learning in relatively large-scale tasks, function approximators are applied to handle the above limitations (Fig. 2.17). Different types of value function approximation are summarized as follows and shown in Fig. 2.18:

- **Linear methods**: the approximated function is a linear combination of weights $\boldsymbol{\theta}$ and real-valued vector of features $\boldsymbol{\phi}(s) = (\phi_1(s), \phi_2(s)), \ldots, \phi_n(s))^T$, where s is the state. It is denoted as $v(s, \boldsymbol{\theta}) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s)$. The TD($\lambda$) method is proven to be convergent with linear function approximators under certain conditions

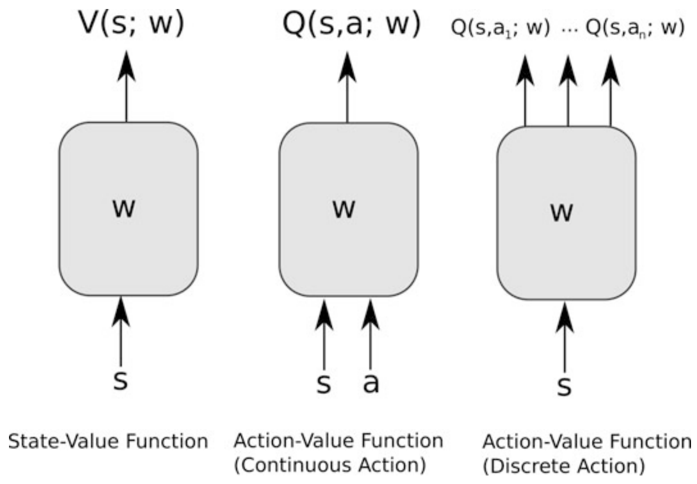
V(s; w)
Q(s,a; w)
Q(s,a1; w) ··· Q(s,an; w)
w
w
w
s
s a
s
State-Value Function
Action-Value Function (Continuous Action)
Action-Value Function (Discrete Action)

Fig. 2.18 Different value function approximation frameworks. The gray boxes with parameters w are the function approximators

as shown in Tsitsiklis and Roy (1997). Although the convergence guarantee of linear methods are attractive, the feature selection or feature representation $\boldsymbol{\phi}(s)$ can be critical in practice when applying linear representations. Different ways of constructing the features for linear methods are as follows:

- **Polynomials**: basic polynomial families can be used as feature vectors for function approximation. Assuming that every state $\boldsymbol{s} = (S_1, S_2, \ldots, S_d)^T$ is a d-dimensional vector, then we have a d-dimensional polynomial basis as $\phi_i(\boldsymbol{s}) = \prod_{j=1}^{d} S_j^{c_{i,j}}$, where each $c_{i,j}$ is an integer in set $\{0, 1, \ldots, N\}$. This forms order N polynomial basis, with $(N+1)^d$ different functions.
- **Fourier basis**: the Fourier transformation is usually used to represent sequential signals in the time/frequency domain. The one-dimensional order-N Fourier cosine basis with $N+1$ functions is: $\phi_i(s) = \cos(i\pi s)$ for $s \in [0, 1]$ and $i = 0, \ldots, N$.
- **Coarse coding**: the state space can be reduced from high-dimensional to low-dimensional, like binary representation through a region covering the determination process, which is called coarse coding.
- **Tile coding**: in the category of coarse coding, tile coding is an efficient approach for feature representation on multi-dimensional continuous spaces. The receptive field of features in tile coding are grouped into partitions of the input space. Each such partition is called a tilling, and each element of the partition is called a tile. Multiple tillings are usually applied in combination with overlapping receptive fields to give the feature vectors in practice.
- **Radial basis functions**: the radial basis functions (RBF) naturally generalize the coarse coding, which is binary-valued, to be continuous-valued features in

[0, 1]. The typical RBF is in Gaussian format $\phi_i(s) = \exp(-\frac{||s-c_i||^2}{2\sigma_i^2})$, where s is the state, c_i is the feature's prototypical or center state, and σ_i is the feature width.

- **Non-linear methods:**
  - **Artificial neural networks**: different from the above function approximation methods, artificial neural networks are widely used as non-linear function approximators, which are proven to have universal approximation ability under certain conditions (Leshno et al. 1993). Based on deep learning techniques, artificial neural networks form the main body of modern DRL methods with function approximation. Details of deep learning are introduced in Chap. 1. A typical example of it is the DQN algorithm, deploying an artificial neural network for Q-value approximation.
- **Other methods:**
  - **Decision trees** (Pyeatt et al. 2001): the decision trees can be used to represent the state space by dividing it with decision nodes, which forms a considerable method for state feature representation.
  - **Nearest neighbor method**: it measures the difference of current state and previous state in memory, and applies the value of the most similar state in memory to approximate the value of the current state.

The benefits of using value function approximation include not only the scalability to large-scale tasks, but also the ease to generalize to unseen states from the seen states given continuous state spaces. Moreover, ANN-based function approximation also reduces or eliminates the need for manually designing features to represent the states. For model-free methods, the parameters w of the approximators can be updated with Monte Carlo (MC) or TD learning. The updating of parameters can be conducted with a batch of samples instead of updating each value in a tabular-based method one-by-one. This makes it computational efficient when handling large-scale problems. For model-based methods, the parameters can be updated with dynamic programming. Details about MC, TD, and DP are introduced in previous sections.

Potential function approximators include a linear combination of features, neural networks, decision trees, the nearest neighbor method, etc. The most practical approximation method for present DRL algorithms is using the neural network, for its great scalability and generalization for various specific functions. A neural network is a differential method with gradient-based optimization, which has a guarantee of convergence to optimum within convex cases and can achieve near-optimal solutions for some non-convex functions approximation. However, it may require a large amount of data for training in practice and may cause other difficulties.

Extending deep learning problems to those of reinforcement learning comes with additional challenges including non-independently and identically distributed data (i.e. non-i.i.d.). Most supervised learning methods are constructed with the assumption that training data is from an i.i.d. and stationary distribution (Schmidhuber 2015). However, the training data in reinforcement learning usually consists of highly correlated samples from sequential agent–environment interactions, which violates the independence condition in supervised learning. Even worse, the training data in reinforcement learning is usually non-stationary as the value function is estimated with current policy, or at least the state-visit-frequency determined by current policy, and the policy is updated all the time during training. The agent learns through exploring different partitions of the state space. All these cases violate the condition of sampled data being identically distributed.

There are some practical requirements for the representations when using value function approximation in reinforcement learning, which may lead to divergence if not considered properly (Achiam et al. 2019). Specifically, the danger of instability and divergence arises whenever the three conditions are combined: (1) training on a distribution of transitions other than those naturally generated by the process whose expectation is being estimated (e.g., off-policy learning); (2) scalable function approximations (e.g., linear semi-gradient); (3) bootstrapping (e.g., DP, TD learning). These three main properties can lead to learning divergence only when they are combined, which is known as **the deadly triad** (Van Hasselt et al. 2018). Value-based methods using function approximation can also have an over-/under-estimation problem, if the way of leveraging function approximation is not fair enough. For example, original DQN has the problem of overestimating the Q-value (Van Hasselt et al. 2016), which decreases the learning performances in practice, and the double/dueling DQN techniques are proposed to alleviate the problem. Generally, policy-based methods with policy gradients have stronger convergence guarantee compared with value-based methods.

Gradient-Based Value Function Approximation

Considering the value function is parameterized as $V^\pi(s) = V^\pi(s; w)$ or $Q^\pi(s, a) = Q^\pi(s, a; w)$, we can derive the udpate rules with different methods of estimation. The optimization objective is set to be the mean-squared error (MSE) between the approximate function $V^\pi(s; w)$ (or $Q^\pi(s, a; w)$) and the true value function $v_\pi(s)$ (or $q_\pi(s, a)$):

$$J(w) = \mathbb{E}_\pi[(V^\pi(s; w) - v_\pi(s))^2] \tag{2.93}$$

or,

$$J(w) = \mathbb{E}_\pi[(Q^\pi(s, a; w) - q_\pi(s, a))^2] \tag{2.94}$$

Therefore the gradients with stochastic gradient descent are:

$$\Delta w = \alpha(V^\pi(s; w) - v_\pi(s))\nabla_w V^\pi(s; w) \tag{2.95}$$

or,

$$\Delta w = \alpha(Q^{\pi}(s, a; w) - q_{\pi}(s, a))\nabla_w Q^{\pi}(s, a; w) \tag{2.96}$$

where the gradients are estimated with each sample in the batch and the weights are updated in a stochastic manner. The target (true) value functions v_{π} or q_{π} in above equations are usually estimated, sometimes with a target network (DQN) or a max operator (Q-learning), etc. We show some basic estimations of the value functions here.

For **MC** estimation, the target value is estimated with the sampled return G_t. Therefore, the update gradients of value-function parameters are:

$$\Delta w_t = \alpha(V^{\pi}(S_t; w_t) - G_t)\nabla_{w_t} V^{\pi}(S_t; w_t) \tag{2.97}$$

or,

$$\Delta w_t = \alpha(Q^{\pi}(S_t, A_t; w_t) - G_{t+1})\nabla_{w_t} Q^{\pi}(S_t, A_t; w_t) \tag{2.98}$$

For **TD(0)**, the target is the TD target $R_t + \gamma V_{\pi}(S_{t+1}; w_t)$ according to the Bellman Optimality Equation as Eq. (2.92), therefore:

$$\Delta w_t = \alpha(V^{\pi}(S_t; w_t) - (R_t + \gamma V_{\pi}(S_{t+1}; w_t)))\nabla_{w_t} V^{\pi}(S_t; w_t) \tag{2.99}$$

or,

$$\Delta w_t = \alpha(Q^{\pi}(S_t, A_t; w_t) - (R_{t+1} + \gamma Q_{\pi}(S_{t+1}, A_{t+1}; w_t))\nabla_{w_t} Q^{\pi}(S_t, A_t; w_t)) \tag{2.100}$$

For **TD(λ)**, the target is the λ-return G_t^{λ}, so the update rule is:

$$\Delta w_t = \alpha(V^{\pi}(S_t; w_t) - G_t^{\lambda})\nabla_{w_t} V^{\pi}(S_t; w_t) \tag{2.101}$$

or,

$$\Delta w_t = \alpha(Q^{\pi}(S_t, A_t; w_t) - G_t^{\lambda})\nabla_{w_t} Q^{\pi}(S_t, A_t; w_t) \tag{2.102}$$

Different estimations have different preferences in bias and variances, which has already been discussed in previous sections about different estimation methods like MC and TD.

Example: Deep *Q*-Network

Deep Q-Network (DQN) is one of the most typical examples for value-based optimization. It uses a deep neural network for Q-value function approximation in

Q-Learning, and maintains an experience replay buffer to store transition samples during the agent–environment interactions. DQN also applies a target network Q^T, which is parameterized by a copy of the original network Q parameter and updated in a delayed manner, to stabilize the learning process, i.e. to alleviate the non-stationary data distribution problem in deep learning. It uses the MSE loss following the above Eq. (2.96), with the true value function q_π replaced by the approximation function $r + \gamma \max_{a'} Q^T(s', a')$ in a greedy manner.

The experience replay buffer provides stability for learning as random batches are sampled from the buffer to help to alleviate the problems of non-i.i.d. data. It makes the policy update to be an off-policy manner due to the mismatch between buffer content from the earlier policy and from the current policy. More details about the DQN algorithm are introduced in Chap. 4.

2.7.3 Policy-Based Optimization

Before we talk about policy-based optimization, we first introduce common policies in reinforcement learning. As introduced in previous sections, policies in reinforcement learning can be divided into deterministic and stochastic policies. In deep reinforcement learning, we use neural networks to represent the policies of both categories, which are called **parameterized policies**. Specifically, the parameterization here indicates the abstract policy is parameterized with the neural network (including single layer perceptrons), rather than other parametric representations. With the network parameters θ, the deterministic and stochastic policy can be written as $A_t = \mu_\theta(S_t)$ and $A_t \sim \pi_\theta(\cdot|S_t)$, respectively.

In deep reinforcement learning domain, there are several commonly seen specific distributions for representing the action distribution of a stochastic policy: the Bernoulli distribution, categorical distribution, and diagonal Gaussian distribution. The Bernoulli and categorical distributions work for the discrete action spaces, either binary or multi-category, while the diagonal Gaussian distributions work for the continuous action spaces.

The Bernoulli distribution of a single variable $x \in 0, 1$ with parameter θ is: $P(s; \theta) = \theta^x(1-\theta)^{(1-x)}$. Therefore it can be used to represent the actions with binary value, for either single or multiple dimensions (with a vector of variables), which works for the so-called **binary-action policies**.

A **categorical policy** with categorical distribution as its output can be used in discrete and finite action spaces, it considers the policy as a classifier, which outputs the probabilities of each action in the finite action space conditioned on a state e.g., $\pi(a|s) = \boldsymbol{P}[A_t = a|S_t = s]$. The sum of all probabilities is equal to one, therefore the softmax activation function is usually applied in the last output layer when the categorical policy is parameterized. Instead of using probability function $p(\cdot|\cdot)$, here we use $\boldsymbol{P}[\cdot|\cdot]$ specifically for representing the cases with finite action space in a matrix. The agent can choose one action by sampling according to the categorical distribution. In practice, the action in this case is usually encoded as a one-hot vector with the same dimension as the action space as $a_i = (0, 0, \ldots, 1, \ldots, 0)$, so that

$a_i \odot \boldsymbol{p}(\cdot|s)$ gives $p(a_i|s)$, where $\odot$ is the element-wise product operator and $\boldsymbol{p}(\cdot|s)$ is the vector of matrix with fixed state s, usually also as the normalized output layer of the categorical policy. ***Gumbel-Softmax* trick** can be applied in practice to keep the sampling process of categorical distribution differentiable if the categorical policy is parameterized. Without specific tricks applied, the stochastic node with a sampling process and operations like arg max are usually non-differentiable, which is problematic when employed in parameterized policies depending on gradient-based optimization (introduced in later sections).

Gumbel-Softmax trick (Jang et al. 2016): first, the *Gumbel-Max* trick allows us to draw samples from categorical distribution π:

$$z = \text{one\_hot}[\arg\max_i(z_i + \log \pi_i)] \tag{2.103}$$

where "one_hot" is an operation transferring a scalar into a one-hot vector. However, as mentioned above, the arg max operation is generally non-differentiable. Therefore, in *Gumbel-Softmax* trick, a *Softmax* operation is applied to approximate the arg max continuously in *Gumbel-Max* trick:

$$a_i = \frac{\exp((\log \pi_i + g_i)/\tau)}{\sum_j \exp((\log \pi_j + g_j)/\tau}, \forall i = 0, \ldots, k \tag{2.104}$$

where k is the dimension of the desired variable $\boldsymbol{a}$ (the action for reinforcement learning policy) and g_i is the Gumbel variable sampled from the Gumbel distribution. The Gumbel (0,1) distribution can be sampled using inverse transform sampling by drawing $u \sim \text{Uniform}(0, 1)$ and computing $g = \log(\log(u))$ in practice.

The **diagonal Gaussian policy** outputting the means and variances of a diagonal Gaussian distribution can be used in continuous action spaces. A normal multivariate Gaussian distribution contains a mean vector $\boldsymbol{\mu}$ and a covariance matrix $\boldsymbol{\Sigma}$, while the diagonal Gaussian distribution is a special case where only the diagonal of covariance matrix is non-zero, so we can use a vector $\boldsymbol{\sigma}$ to represent it. When applying the diagonal Gaussian distribution to represent the probabilistic actions, it removes the covariance relationships among different dimensions of the actions. When the policy is parameterized, the **reparametrization trick** as below (similar as in variational autoencoder by Kingma and Welling (2014)) can be applied to sample actions from the mean and variance vectors, as well as keeping the operations differentiable.

Reparameterization trick: sampling the action a from a diagonal Gaussian distribution $a \sim \mathcal{N}(\boldsymbol{\mu}_\theta, \boldsymbol{\sigma}_\theta)$ with the mean and variance vectors $\boldsymbol{\mu}_\theta$ and $\boldsymbol{\sigma}_\theta$ (parameterized) can be alternatively achieved with sampling a hidden vector $\boldsymbol{z}$ from a normal Gaussian $\boldsymbol{z} \sim \mathcal{N}(0, \mathbf{1})$ and derive the action as:

$$a = \boldsymbol{\mu}_\theta + \boldsymbol{\sigma}_\theta \odot \boldsymbol{z} \tag{2.105}$$

where $\odot$ is the elementwise product for two vectors of the same shape.

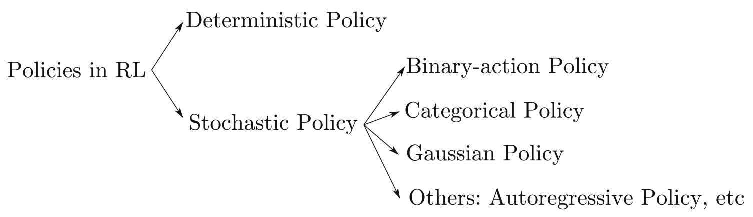

Fig. 2.19 Different policies in deep reinforcement learning

An overview of common policies in deep reinforcement learning is displayed in Fig. 2.19, for providing the readers a better understanding.

Policy-Based Optimization methods directly optimize the policy of the agent in reinforcement learning scenarios without estimating or learning an action-value function. The sampled reward values are usually used in the optimization process for improving action preferences. Either gradient-based or gradient-free methods are applied in the optimization process. Gradient-based methods always apply the policy gradient, which perhaps represents the most popular class of algorithms used in continuous-action reinforcement learning, benefiting from scalability to high-dimensional cases. The typical methods in gradient-based optimization include REINFORCE, etc. On the other hand, gradient-free algorithms usually have a faster learning process for relatively simple cases in policy searching, free from the computationally expensive process of calculating derivatives. The typical methods in gradient-free category include cross-entropy (CE) method and so on.

Recall that the goal of the agent in reinforcement learning is to maximize the cumulative discounted reward from the start state, in an expected or estimated view, which can be denoted as:

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}[R(\tau)] \tag{2.106}$$

where $R(\tau) = \sum_{t=0}^{T} \gamma^t R_t$ as a discounted expected reward with finite steps (fits most scenarios), and τ are sampled trajectories.

The policy-based optimization will optimize the policy π with respect to the above goal $J(\pi)$, through gradient-based or gradient-free methods. We will first introduce **gradient-based** methods and give an example of REINFORCE algorithm, then introduce a **gradient-free** (non-gradient-based) algorithms and show the example CE method.

Gradient-Based Optimization

Gradient-based optimization uses an estimator for the gradients on the expected return (total reward) obtained from sample trajectories to improve the policy with gradient descent/ascent, and the gradient with respect to the policy parameter is called the **policy gradient** as follows:

$$\Delta\theta = \alpha \nabla_\theta J(\pi_\theta) \tag{2.107}$$

where θ indicates the policy parameters and α is the learning rate. Methods based on these gradients of policy parameters are called the policy gradient method. The **policy gradient theorem** proposed by Sutton et al. (2000) and Silver et al. (2014) is shown as follows and will be proved in the following sections.

Note: the representation θ of parameters in Eq. (2.107) is actually improper, which is supposed to be $\boldsymbol{\theta}$ for representing the vector as a default format of the book (see the chapter of math notation). However, here we apply the vanilla format θ as an interchangeable way of $\boldsymbol{\theta}$ whenever representing the model parameters. This follows the common format in literature and is also simple. One way to consider the rationality of this representation is: the gradients of parameters can be taken for each parameter individually, which is denoted by θ, while the equations are the same for all parameters. Therefore it also works for applying θ to represent all parameters. The rest of the book follows the above statements.

Theorem 2.2 (Policy Gradient Theorem)

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta(\log \pi_\theta(A_t|S_t)) Q^{\pi_\theta}(S_t, A_t)\right] \tag{2.108}$$

$$= \mathbb{E}_{S_t\sim\rho^\pi, A_t\sim\pi_\theta}[\nabla_\theta(\log \pi_\theta(A_t|S_t)) Q^{\pi_\theta}(S_t, A_t)] \tag{2.109}$$

where the second form is derived through defining the discounted state distribution as in Silver et al. (2014) by $\rho^\pi(s') := \int_{\mathcal{S}} \sum_{t=0}^{T} \gamma^{t-1}\rho_0(s)p(s'|s,t,\pi)ds$ *and* $p(s'|s,t,\pi)$ *is the transition probability of* s *to* s' *under policy* π *at time step* t.

The policy gradient theorem works for both stochastic policies and deterministic policies. It was originally proposed by Sutton et al. (2000) for stochastic policies, but extended to deterministic policies by Silver et al. (2014). For the deterministic cases, although the deterministic policy gradient theorem (introduced later) does not look like the above policy gradient theorem, it is proved that the deterministic policy gradient (DPG) is just a special (limiting) case of the stochastic policy gradient (SPG), if we parameterize the stochastic policy $\pi_{\mu_\theta,\sigma}$ by a deterministic policy $\mu_\theta : \mathcal{S} \to \mathcal{A}$ and a variance parameter σ, such that for $\sigma = 0$ the stochastic policy is equivalent to the deterministic policy, $\pi_{\mu_\theta,0} \equiv \mu$. A detailed proof will be provided in the section of the deterministic policy gradient.

1. Stochastic Policy Gradient

Now we first prove the policy gradient theorem for the stochastic policy, which is called the stochastic policy gradient method. For simplicity, we assume an episodic setting in finite MDP with the length of each trajectory fixed as $T + 1$ in this section. Considering a parameterized stochastic policy $\pi_\theta(a|s)$, we then have the probability of trajectory $p(\tau|\pi) = \rho_0(S_0)\prod_{t=0}^{T} p(S_{t+1}|S_t, A_t)\pi(A_t|S_t)$ for MDP process with $\rho_0(S_0)$ as initial state distribution, we can get the logarithm of the probability of trajectory with parameterized policy π_θ as:

$$\log p(\tau|\theta) = \log \rho_0(S_0) + \sum_{t=0}^{T}\bigg(\log p(S_{t+1}|S_t, A_t) + \log \pi_\theta(A_t|S_t)\bigg). \tag{2.110}$$

We also need the **Log-Derivative Trick**: $\nabla_\theta p(\tau|\theta) = p(\tau|\theta)\nabla_\theta \log p(\tau|\theta)$

Therefore we can get the derivative of the log-probability of a trajectory as:

$$\nabla_\theta \log p(\tau|\theta) = \nabla_\theta \log \rho_0(S_0) + \sum_{t=0}^{T}\bigg(\nabla_\theta \log p(S_{t+1}|S_t, A_t) + \nabla_\theta \log \pi_\theta(A_t|S_t)\bigg) \tag{2.111}$$

$$= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t). \tag{2.112}$$

where the terms containing $\rho_0(S_0)$ and $p(S_{t+1}|S_t, A_t)$ are removed because they do not depend on parameters θ, although unknown.

Recall that the learning objective is to maximize the expected cumulative reward:

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} R_t\right] = \sum_{t=0}^{T}\mathbb{E}_{\tau\sim\pi_\theta}[R_t], \tag{2.113}$$

where $\tau = (S_0, A_0, R_0, \ldots, S_T, A_T, R_T, S_{T+1})$ and $R(\tau) = \sum_{t=0}^{T} R_t$. We can directly perform gradient ascent on the parameters of the policy θ to gradually improve the performance of the policy π_θ.

Note that R_t only depends on τ_t, where $\tau_t = (S_0, A_0, R_0, \ldots, S_t, A_t, R_t, S_{t+1})$.

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R_t] = \nabla_\theta \int_{\tau_t} R_t\, p(\tau_t|\theta)d\tau_t \qquad \text{Expand expectation} \tag{2.114}$$

$$= \int_{\tau_t} R_t \nabla_\theta p(\tau_t|\theta)d\tau_t \qquad \text{Exchange gradient and integral} \tag{2.115}$$

$$= \int_{\tau_t} R_t\, p(\tau_t|\theta)\nabla_\theta \log p(\tau_t|\theta)d\tau_t \qquad \text{Log-derivative trick} \tag{2.116}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[R_t \nabla_\theta \log p(\tau_t|\theta)\right] \qquad \text{Return to expectation form} \tag{2.117}$$

The third equality above is due to the log-derivative trick introduced before.

Plug the above formula back to $J(\pi_\theta)$,

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} R_t \nabla_\theta \log p(\tau_t|\theta)\right].$$

Now we need to compute $\nabla_\theta \log p_\theta(\tau_t)$, where $p_\theta(\tau_t)$ depends on both the policy π_θ and the ground truth of the model $p(R_t, S_{t+1}|S_t, A_t)$ which is not available to the agent. Luckily, to apply the policy gradient method, we only need the gradient of $\log p_\theta(\tau_t)$ instead of its original value, which can be derived easily by replacing the $\tau = \tau_{0:T}$ in Eq. (2.112) to be $\tau_t = \tau_{0:t}$, which gives:

$$\nabla_\theta \log p(\tau_t|\theta) = \sum_{t'=0}^{t} \nabla_\theta \log \pi_\theta(A_{t'}|S_{t'}). \tag{2.118}$$

Therefore,

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} R_t \nabla_\theta \sum_{t'=0}^{t} \log \pi_\theta(A_{t'}|\, S_{t'})\right]\\ &= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t'=0}^{T} \nabla_\theta \log \pi_\theta(A_{t'}|\, S_{t'}) \sum_{t=t'}^{T} R_t\right].\end{aligned} \tag{2.119}$$

Here the last equality is simply by rearranging the summation.

Notice that in the above derivation process we use both the exchanging between sum and expectation and the exchanging between expectation and sum and derivative (both valid):

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} R_t\right] = \sum_{t=0}^{T} \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R_t] \tag{2.120}$$

which ends up to take the integral in Eq. (2.114) over the partial trajectory τ_t of length $t + 1$. However, there is also other way of taking the expectation of the cumulative reward along the whole trajectory:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta} R(\tau) \tag{2.121}$$

$$= \nabla_\theta \int_\tau p(\tau|\theta) R(\tau) \quad \text{Expand expectation} \tag{2.122}$$

$$= \int_{\tau} \nabla_{\theta} p(\tau|\theta) R(\tau) \quad \text{Exchange gradient and integral} \tag{2.123}$$

$$= \int_{\tau} p(\tau|\theta) \nabla_{\theta} \log p(\tau|\theta) R(\tau) \quad \text{Log-derivative trick} \tag{2.124}$$

$$= \mathbb{E}_{\tau \sim \pi_{\theta}} [\nabla_{\theta} \log p(\tau|\theta) R(\tau)] \quad \text{Return to expectation form} \tag{2.125}$$

$$\Rightarrow \nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^{T} \nabla_{\theta} \log \pi_{\theta}(A_t|S_t) R(\tau) \right] \tag{2.126}$$

$$= \mathbb{E}_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^{T} \nabla_{\theta} \log \pi_{\theta}(A_t|S_t) \sum_{t'=0}^{T} R_{t'} \right] \tag{2.127}$$

A careful reader may notice that the second result in Eq. (2.127) is slightly different from the first result as in Eq. (2.119). Specifically, the time scales of the cumulative reward are different. The first result uses only the cumulative future rewards $\sum_{t=t'}^{T} R_t$ after action A_t to evaluate the action, while the second result uses the cumulative rewards on the whole trajectory $\sum_{t=0}^{T} R_t$ to evaluate each action A_t on that trajectory, including the rewards before choosing that action. Intuitively, the action should not be evaluated by the rewards happened before that action is conducted, which is also reinforced by mathematical proof that the rewards obtained before the action have zero effects on the final expected gradients. Those past rewards can, therefore, be simply dropped in the derived policy gradient to have Eq. (2.119), which is called the "reward-to-go" policy gradient. A strict proof of the equivalence of the two policy gradient formulas is not provided here but can be referred to here.[1] The two derivations here can also be regarded as a proof of the equivalence of two results.

The ∇ in the above formulas called "nabla" is a specific computational operator with three basic meanings (gradient, divergence, and curl) in the physics and mathematics domains, depending on its operational objectives. But in the computer science domain, the "nabla" operator ∇ is usually used as partial derivative, which derives the derivative on the following objective explicitly containing the variable in the footnote position. As the $R(\tau)$ in above formulas does not explicitly contain θ, the ∇_{θ} does not operate on $R(\tau)$, although the τ implicitly depends on θ (according to the graphical model of MDP). We also notice that the expectation in Eq. (2.127) can be estimated with the sample mean. If we collect a set of trajectories $\mathcal{D} = \{\tau_i\}_{i=1,\dots,N}$ where each trajectory is obtained by letting the agent act in the

[1]Proof of equivalence of two versions of stochastic policy gradient: https://spinningup.openai.com/en/latest/spinningup/extra_pg_proof1.html.

environment using the policy π_θ, the policy gradient can be estimated with

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta (A_t | S_t) R(\tau), \tag{2.128}$$

The **Expected Grad-Log-Prob (EGLP)** lemma[2] is commonly used in policy gradient optimization, so we introduce it here.

Lemma 2.2 (EGLP Lemma) *Suppose that p_θ is a parameterized probability distribution over a random variable, x. Then:*

$$\mathbb{E}_{x \sim p_\theta} [\nabla_\theta \log P_\theta(x)] = 0. \tag{2.129}$$

Proof Recall that all probability distributions are normalized:

$$\int_x p_\theta(x) = 1. \tag{2.130}$$

Take the gradient of both sides of the normalization condition:

$$\nabla_\theta \int_x p_\theta(x) = \nabla_\theta 1 = 0. \tag{2.131}$$

Use the log derivative trick to get:

$$\begin{aligned} 0 = &\quad \nabla_\theta \textstyle\int_x p_\theta(x) && (2.132)\\ = &\quad \textstyle\int_x \nabla_\theta p_\theta(x) && (2.133)\\ = &\textstyle\int_x p_\theta(x) \nabla_\theta \log p_\theta(x) && (2.134)\\ \therefore 0 = &\, \mathbb{E}_{x \sim p_\theta} [\nabla_\theta \log p_\theta(x)]. && (2.135) \end{aligned}$$

From the EGLP lemma we can directly derive that:

$$\mathbb{E}_{A_t \sim \pi_\theta} [\nabla_\theta \log \pi_\theta(A_t | S_t) b(S_t)] = 0. \tag{2.136}$$

where $b(S_t)$ is called a baseline and is independent of the future trajectory the expectation is taken over. The baseline is any function dependent only on the currents state, without affecting the overall expected value in the optimization formula.

[2]Referred to OpenAI Spinning Up: https://spinningup.openai.com/en/latest/.

In the above formulas the optimization goal is finally:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t)R(\tau)\right] \tag{2.137}$$

We can also modify the reward for total trajectory $R(\tau)$ to be reward-to-go G_t following time step t:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t)G_t\right] \tag{2.138}$$

With the above EGLP lemma, the expected return can be generalized to be:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t)\Phi_t\right] \tag{2.139}$$

where $\Phi_t = \sum_{t'=t}^{T}(R(S_{t'}, a_{t'}, S_{t'}+1) - b(S_t))$.

Actually Φ_t could be the following formats for more practical usage:

$$\Phi_t = Q^{\pi_\theta}(S_t, A_t) \tag{2.140}$$

or,

$$\Phi_t = A^{\pi_\theta}(S_t, A_t) = Q^{\pi_\theta}(S_t, A_t) - V^{\pi_\theta}(S_t) \tag{2.141}$$

which are both proven to be identical to the original format in the expected value, just with different variances in practice. The proof of these requires law of iterated expectations: $\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X|Y]]$ for two random variables (discrete or continuous). And this is easy to prove. The rest of the proof is given below:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t)R(\tau)\right] \tag{2.142}$$

$$= \sum_{t=0}^{T} \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(A_t|S_t)R(\tau)] \tag{2.143}$$

$$= \sum_{t=0}^{T} \mathbb{E}_{\tau_{:t}\sim\pi_\theta}[\mathbb{E}_{\tau_{t:}\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(A_t|S_t)R(\tau)|\tau_{:t}]] \tag{2.144}$$

$$= \sum_{t=0}^{T} \mathbb{E}_{\tau_{:t}\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(A_t|S_t)\mathbb{E}_{\tau_{t:}\sim\pi_\theta}[R(\tau)|\tau_{:t}]] \tag{2.145}$$

$$= \sum_{t=0}^{T} \mathbb{E}_{\tau_{:t} \sim \pi_\theta} [\nabla_\theta \log \pi_\theta (A_t|S_t) \mathbb{E}_{\tau_{t:} \sim \pi_\theta} [R(\tau)|S_t, A_t]] \tag{2.146}$$

$$= \sum_{t=0}^{T} \mathbb{E}_{\tau_{:t} \sim \pi_\theta} [\nabla_\theta (\log \pi_\theta (A_t|S_t)) Q^{\pi_\theta} (S_t, A_t)] \tag{2.147}$$

in which $\mathbb{E}_\tau[\cdot] = \mathbb{E}_{\tau_{:t}}[\mathbb{E}_{\tau_{t:}}[\cdot|\tau_{:t}]]$ and $\tau_{:t} = (S_0, A_0, \ldots, S_t, A_t)$, and $Q^{\pi_\theta}(S_t, A_t) = \mathbb{E}_{\tau_{t:} \sim \pi_\theta}[R(\tau)|S_t, A_t]$.

Therefore, it's common to see

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta (\log \pi_\theta (A_t|S_t)) Q^{\pi_\theta} (S_t, A_t) \right] \tag{2.148}$$

or,

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta (\log \pi_\theta (A_t|S_t)) A^{\pi_\theta} (S_t, A_t) \right] \tag{2.149}$$

in the literature. In other words, it is equivalent to changing the optimization objective to be $J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi}[Q^{\pi_\theta}(S_t, A_t)]$ or $J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi}[A^{\pi_\theta}(S_t, A_t)]$ instead of original $\mathbb{E}_{\tau \sim \pi}[R(\tau)]$, in the sense of optimal policy. The $A^{\pi_\theta}(S_t, A_t)$ are usually estimated with TD-error in practice.

According to whether the environment model is used or not, reinforcement learning algorithms can be classified into model-free and model-based categories. For model-free reinforcement learning, pure gradient-based optimization algorithms are originated from the REINFORCE algorithm, or called the policy gradient method. For the model-based reinforcement learning category, there are also policy-based algorithms, like the method applying backpropagation through time (BPTT) for updating the policy using sampled rewards within episodes. No more details about model-based methods will be discussed here, and we instead direct the readers to Chap. 9.

Example: REINFORCE Algorithm

REINFORCE is an algorithm using stochastic policy gradient method as in Eq. (2.139), where $\Phi_t = Q^\pi(S_t, A_t)$ and it is estimated with sampled rewards along the trajectory $G_t = \sum_{t'=t}^{\infty} R_{t'}$ (or discounted version $G_t = \sum_{t'=t}^{\infty} \gamma^{t'-t} R_{t'}$) in REINFORCE. The gradients for updating the policy are:

$$g = \mathbb{E} \left[\sum_{t=0}^{\infty} \sum_{t'=t}^{\infty} R_{t'} \nabla_\theta \log \pi_\theta (A_t|S_t) \right] \tag{2.150}$$

Details of REINFORCE algorithm are introduced in Chap. 5.

2. Deterministic Policy Gradient
What has been described above belongs to stochastic policy gradient (SPG), and it works for optimizing the stochastic policy $\pi(a|s)$, which represents the action as a probabilistic distribution based on the current state. The contrary case to the stochastic policy is the deterministic policy, where $a = \pi(s)$ is a deterministic action instead of probability. We can derive the deterministic policy gradient (DPG) similarly as in SPG, and it also follows the policy gradient theorem numerically (as a limit case), although they have different explicit expressions.

Note: in the following part of this section, we use $\mu(s)$ instead of $\pi(s)$ as previously defined to represent the deterministic policy, for removing ambiguity in the distinction with stochastic policy $\pi(a|s)$.

For a more rigorous and general definition of DPG we refer to the deterministic policy gradient theorem proposed by Silver et al. (2014) in Eq. (2.159). We will introduce the deterministic policy gradient theorem and prove it, in an on-policy manner first and off-policy later, as well as discussing the relationship of DPG with SPG in detail.

First of all, we define the performance objective for the deterministic policy following the same expected discounted reward definition in stochastic policy gradient:

$$J(\mu) = \mathbb{E}_{S_t\sim\rho^\mu, A_t=\mu(S_t)}[\sum_{t=1}^{\infty}\gamma^{t-1}R(S_t, A_t)] \tag{2.151}$$

$$= \int_{\mathcal{S}}\int_{\mathcal{S}}\sum_{t=1}^{\infty}\gamma^{t-1}\rho_0(s)p(s'|s, t, \mu)R(s', \mu(s'))]\mathrm{d}s\mathrm{d}s' \tag{2.152}$$

$$= \int_{\mathcal{S}}\rho^\mu(s)R(s, \mu(s))\mathrm{d}s \tag{2.153}$$

where $p(s'|s, t, \mu) = p(S_{t+1}|S_t, A_t)p^\mu(A_t|S_t)$, the first probability is the transition probability and the second is the probability of the action choice. Since it is deterministic policy, we have $p^\mu(A_t|S_t) = 1$ and therefore $p(s'|s, t, \mu) = p(S_{t+1}|S_t, \mu(S_t))$. Also, the state distribution in above formula is $\rho^\mu(s') := \int_{\mathcal{S}}\sum_{t=1}^{\infty}\gamma^{t-1}\rho_0(s)p(s'|s, t, \mu)\mathrm{d}s$.

As $V^\mu(s) = \mathbb{E}[\sum_{t=1}^{\infty}\gamma^{t-1}R(S_t, A_t)|S_1 = s; \mu] = \int_{\mathcal{S}}\sum_{t=1}^{\infty}\gamma^{t-1}p(s'|s, t, \mu)$ $R(s', \mu(s'))]\mathrm{d}s'$ following the same definition in stochastic policy gradient except for applying the deterministic policy, we can also derive that

$$J(\mu) = \int_{\mathcal{S}}\rho_0(s)V^\mu(s)\mathrm{d}s \tag{2.154}$$

$$= \int_{\mathcal{S}}\int_{\mathcal{S}}\sum_{t=1}^{\infty}\gamma^{t-1}\rho_0(s)p(s'|s, t, \mu)R(s', \mu(s'))]\mathrm{d}s\mathrm{d}s' \tag{2.155}$$

which is the same as the above representation directly using discounted rewards. The relationships here also hold for stochastic policy gradient, just with the deterministic policy $\mu(s)$ replaced by the stochastic policy $\pi(a|s)$. For deterministic policy, we have $V^{\mu}(s) = Q^{\mu}(s, \mu(s))$ as the Q-value is an expectation over the action distribution for stochastic policy, but there is no action distribution but a single value for the deterministic policy. Therefore we also have the following representation for deterministic policy,

$$J(\mu) = \int_{\mathcal{S}} \rho_0(s) V^{\mu}(s) \mathrm{d}s \tag{2.156}$$

$$= \int_{\mathcal{S}} \rho_0(s) Q^{\mu}(s, \mu(s)) \mathrm{d}s \tag{2.157}$$

The different formats of performance objective will be used in the proof of DPG theorem, as well as several conditions. We list the conditions here without a detailed derivation process, which can be checked in the original paper by Silver et al. (2014):

- ***C.1 The Existence of Continuous Derivatives:*** $p(s'|s, a)$, $\nabla_a p(s'|s, a)$, $\mu_\theta(s)$, $\nabla_\theta \mu_\theta(s)$, $R(s, a)$, $\nabla_a R(s, a)$, $\rho_0(s)$ *are continuous in all parameters and variables* s, a, s', and x.
- ***C.2 The Boundedness Condition:*** *there exist* a, b, and L *such that* $\sup_s \rho_0(s) < b, \sup_{a,s,s'} p(s'|s, a) < b, \sup_{a,s} R(s, a) < b, \sup_{a,s,s'} ||\nabla_a p(s'|s, a)|| < L, \sup_{a,s} ||\nabla_a R(s, a)|| < L$.

Theorem 2.3 (Deterministic Policy Gradient Theorem) *suppose that the MDP satisfies conditions C.1 for the existence of* $\nabla_\theta \mu_\theta(s)$, $\nabla_a Q^{\mu}(s, a)$ *and the deterministic policy gradient, then,*

$$\nabla_\theta J(\mu_\theta) = \int_{\mathcal{S}} \rho^{\mu}(s) \nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu}(s, a)|_{a=\mu_\theta(s)} ds \tag{2.158}$$

$$= \mathbb{E}_{s \sim \rho^{\mu}} [\nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu}(s, a)|_{a=\mu_\theta(s)}] \tag{2.159}$$

Proof The proof of deterministic policy gradient theorem generally follows the same lines of the standard stochastic policy gradient theorem by Sutton et al. (2000). First of all, in order to exchange derivatives and integrals, and the order of integration whenever needed in the following proof, we need to use two lemmas, which are basic mathematical rules in calculus as follows:

- **Lemma 2.3 (Leibniz Integral Rule)** *let* $f(x, t)$ *be a function such that both* $f(x, t)$ *and its partial derivative* $f'_x(x, t)$ *are continuous in* t *and* x *in some region of the* (x, t)*-plane, including* $a(x) \leq t \leq b(x)$, $x_0 \leq x \leq x_1$. *Also suppose that the functions* $a(x)$ *and* $b(x)$ *are both continuous and both have continuous* *derivatives for $x_0 \leq x \leq x_1$. Then, for $x_0 \leq x \leq x_1$,*

$$\frac{d}{dx}\int_{a(x)}^{b(x)} f(x,t)dt = f(x,b(x)) \cdot \frac{d}{dx}b(x) - f(x,a(x)) \cdot \frac{d}{dx}a(x) + \int_{a(x)}^{b(x)} \frac{\partial}{\partial x} f(x,t)dt \tag{2.160}$$

- **Lemma 2.4 (Fubini's Theorem)** *Suppose $\mathcal{X}$ and $\mathcal{Y}$ are σ-finite measure spaces, and suppose that $\mathcal{X} \times \mathcal{Y}$ is given the product measure (which is unique as $\mathcal{X}$ and $\mathcal{Y}$ are σ-finite). Fubini's theorem states that if f is $\mathcal{X} \times \mathcal{Y}$ integrable, meaning that f is a measurable function and*

$$\int_{\mathcal{X}\times\mathcal{Y}} |f(x,y)|d(x,y) < \infty \tag{2.161}$$

then,

$$\int_{\mathcal{X}} \left(\int_{\mathcal{Y}} f(x,y)dy\right) dx = \int_{\mathcal{Y}} \left(\int_{\mathcal{X}} f(x,y)dx\right) dy = \int_{\mathcal{X}\times\mathcal{Y}} f(x,y)d(x,y) \tag{2.162}$$

To satisfy these two lemmas, we require the necessary conditions provided in C.1 as the Leibniz integral rule requires, which imply that $V^{\mu_\theta}(s)$ and $\nabla_\theta V^{\mu_\theta}(s)$ are continuous functions of θ and s. We also follow the assumption of the compactness of the state space $\mathcal{S}$, which is in C.2 required by Fubini's theorem and implies that for any θ, $||\nabla_\theta V^{\mu_\theta}(s)||$, $||\nabla_a Q^{\mu_\theta}(s,a)|_{a=\mu_\theta(s)}||$ and $||\nabla_\theta \mu_\theta(s)||$ are bounded functions of s. With above conditions, we have the following derivations:

$$\nabla_\theta V^{\mu_\theta}(s) = \nabla_\theta Q^{\mu_\theta}(s, \mu_\theta(s)) \tag{2.163}$$

$$= \nabla_\theta (R(s, \mu_\theta(s)) + \int_{\mathcal{S}} \gamma p(s'|s, \mu_\theta(s)) V^{\mu_\theta}(s')\mathrm{d}s') \tag{2.164}$$

$$= \nabla_\theta \mu_\theta(s) \nabla_a R(s,a)|_{a=\mu_\theta(s)} + \nabla_\theta \int_{\mathcal{S}} \gamma p(s'|s, \mu_\theta(s)) V^{\mu_\theta}(s')\mathrm{d}s' \tag{2.165}$$

$$= \nabla_\theta \mu_\theta(s) \nabla_a R(s,a)|_{a=\mu_\theta(s)} + \int_{\mathcal{S}} \gamma (p(s'|s, \mu_\theta(s)) \nabla_\theta V^{\mu_\theta}(s') + \nabla_\theta \mu_\theta(s) \nabla_a p(s'|s,a) V^{\mu_\theta}(s'))\mathrm{d}s' \tag{2.166}$$

$$= \nabla_\theta \mu_\theta(s) \nabla_a (R(s,a) + \int_{\mathcal{S}} \gamma p(s'|s,a) V^{\mu_\theta}(s') \mathrm{d}s')|_{a=\mu_\theta(s)})$$

$$+ \int_{\mathcal{S}} \gamma p(s'|s, \mu_\theta(s)) \nabla_\theta V^{\mu_\theta}(s') \mathrm{d}s' \tag{2.167}$$

$$= \nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu_\theta}(s,a)|_{a=\mu_\theta(s)} + \int_{\mathcal{S}} \gamma p(s'|s, \mu_\theta(s)) \nabla_\theta V^{\mu_\theta}(s') \mathrm{d}s' \tag{2.168}$$

In the above derivations, the Leibniz integral rule is used to exchange the order of derivative and integration, requiring the continuity conditions of $p(s'|s,a)$, $\mu_\theta(s)$, $V^{\mu_\theta}(s)$ and their derivatives with respect to θ. Now we iterate the above formula with $\nabla_\theta V^{\mu_\theta}(s)$ to have:

$$\nabla_\theta V^{\mu_\theta}(s) = \nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu_\theta}(s,a)|_{a=\mu_\theta(s)} \tag{2.169}$$

$$+ \int_{\mathcal{S}} \gamma p(s'|s, \mu_\theta(s)) \nabla_\theta \mu_\theta(s') \nabla_a Q^{\mu_\theta}(s',a)|_{a=\mu_\theta(s')} \mathrm{d}s' \tag{2.170}$$

$$+ \int_{\mathcal{S}} \gamma p(s'|s, \mu_\theta(s)) \int_{\mathcal{S}} \gamma p(s''|s', \mu_\theta(s')) \nabla_\theta V^{\mu_\theta}(s'') \mathrm{d}s'' \mathrm{d}s' \tag{2.171}$$

$$= \nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu_\theta}(s,a)|_{a=\mu_\theta(s)} \tag{2.172}$$

$$+ \int_{\mathcal{S}} \gamma p(s \to s', 1, \mu_\theta(s)) \nabla_\theta \mu_\theta(s') \nabla_a Q^{\mu_\theta}(s',a)|_{a=\mu_\theta(s')} \mathrm{d}s' \tag{2.173}$$

$$+ \int_{\mathcal{S}} \gamma^2 p(s \to s', 2, \mu_\theta(s)) \nabla_\theta \mu_\theta(s') \nabla_a Q^{\mu_\theta}(s',a)|_{a=\mu_\theta(s')} \mathrm{d}s' \tag{2.174}$$

$$+ \ldots \tag{2.175}$$

$$= \int_{\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t p(s \to s', t, \mu_\theta(s)) \nabla_\theta \mu_\theta(s') \nabla_a Q^{\mu_\theta}(s',a)|_{a=\mu_\theta(s')} \mathrm{d}s' \tag{2.176}$$

where we use Fubini's theorem for changing the order of integration, which requires the condition that $||\nabla_\theta V^{\mu_\theta}(s)||$ is bound. The above integration contains a special case with $p(s \to s', 0, \mu_\theta(s)) = 1$ for $s' = s$ and is 0 for other s'. Now we take derivative on the modified performance objective, which is the expected value function,

$$\begin{aligned}\nabla_\theta J(\mu_\theta) &= \nabla_\theta \int_{\mathcal{S}} \rho_0(s) V^{\mu_\theta}(s)\mathrm{d}s && (2.177)\\ &= \int_{\mathcal{S}} \rho_0(s) \nabla_\theta V^{\mu_\theta}(s)\mathrm{d}s && (2.178)\\ &= \int_{\mathcal{S}} \int_{\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t \rho_0(s) p(s \to s', t, \mu_\theta(s)) \nabla_\theta \mu_\theta(s') \\ &\quad \times \nabla_a Q^{\mu_\theta}(s', a)|_{a=\mu_\theta(s')}\mathrm{d}s'\mathrm{d}s && (2.179)\\ &= \int_{\mathcal{S}} \rho^{\mu_\theta}(s) \nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu_\theta}(s, a)|_{a=\mu_\theta(s)}\mathrm{d}s && (2.180)\end{aligned}$$

where we use the Leibniz integral rule for exchanging the derivative and integral, requiring the conditions that $\rho_0(s)$ and $V^{\mu_\theta}(s)$ and their derivatives with respect to θ are continuous, and also the Fubini's theorem to exchange the order of integration with the boundedness conditions of integrand. Proof is completed.

Off-Policy Deterministic Policy Gradient

Apart from the on-policy version of DPG derived above, we can also derive the deterministic policy gradient in an off-policy manner, using the DPG theorem above and γ-discounted state distribution $\rho^\mu(s') := \int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} p(s) p(s'|s, t, \mu)\mathrm{d}s$. Off-policy deterministic policy gradient estimates current policy with samples from the behavior policy (e.g. previous policies if using replay buffer), which is different from current policy. In the off-policy settings, the gradients are estimated using trajectories sampled from a distinct behavior policy $\beta(s) \neq \mu_\theta(s)$, and the corresponding state distribution is $\rho^\beta(s)$, which is not dependent on the policy parameter θ. And in off-policy case, the performance objective is modified to be the value function of target policy averaged over the state distribution of the behavior policy $J_\beta(\mu_\theta) = \int_{\mathcal{S}} \rho^\beta(s) V^\mu(s)\mathrm{d}s = \int_{\mathcal{S}} \rho^\beta(s) Q^\mu(s, \mu_\theta(s))\mathrm{d}s$, while original objective follows Eq. (2.157) as $J(\mu_\theta) = \int_{\mathcal{S}} \rho_0(s) V^\mu(s)\mathrm{d}s$. Note that it is the first approximation we take in deriving the off-policy deterministic policy gradient, as $J(\mu_\theta) \approx J_\beta(u_\theta)$, and we will have another approximation in the following. We can directly apply the differential operator on the modified objective as follows:

$$\begin{aligned}\nabla_\theta J_\beta(\mu_\theta) &= \int_{\mathcal{S}} \rho^\beta(s)(\nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu_\theta}(s, a) + \nabla_\theta Q^{\mu_\theta}(s, a))|_{a=\mu(s)}\mathrm{d}s && (2.181)\\ &\approx \int_{\mathcal{S}} \rho^\beta(s) \nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu_\theta}(s, a)\mathrm{d}s && (2.182)\\ &= \mathbb{E}_{s\sim\rho^\beta}[\nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu_\theta}(s, a)|_{a=\mu(s)}] && (2.183)\end{aligned}$$

The approximately equivalent symbol in above formulas indicates the difference between the on-policy DPG and off-policy DPG. The dependency relationships in above formula need to be carefully considered. The derivative of θ goes into the integration because $\rho^\beta(s)$ is independent on θ, therefore no term with derivative on $\rho^{\beta}(s)$. As the $Q^{\mu_\theta}(s, \mu_\theta(a))$ actually depends on θ in two ways (two μ_θ in the expression): (1) it depends on the action a determined by the deterministic policy μ_θ with current state s, and (2) the on-policy estimation of Q value also depends on the policy μ_θ for choosing actions for future states, as in $Q^{\mu_\theta}(s, a) = R(s, a) + \int_{\mathcal{S}} \gamma p(s'|s, a) V^{\mu_\theta}(s') \mathrm{d}s'$. So the derivative needs to be conducted separately. However, the second term $\nabla_\theta Q^{\mu_\theta}(s, a)|_{a=\mu(s)}$ in the first formula is dropped in the approximation due to the difficulty in estimation in practice, which has a similar corresponding operation in off-policy stochastic policy gradients (Degris et al. 2012).[3,4]

Relationship of Stochastic Policy Gradient and Deterministic Policy Gradient As shown in Eq. (2.148), the stochastic policy gradient has the same format as in policy gradient theorem in the previous paragraph, while the deterministic policy gradient in Eq. (2.159) seems to have an inconsistent format at first look. However, it can be proved that for a wide range of stochastic policies, the DPG is a special (limit) case of the SPG. In this sense, the DPG also satisfies the policy gradient theorem under certain conditions. In order to achieve that, we parameterize the stochastic policy $\pi_{\mu_\theta,\sigma}$ by a deterministic policy $\mu_\theta : \mathcal{S} \to \mathcal{A}$ and a variance parameter σ, such that for $\sigma = 0$ the stochastic policy is equivalent to the deterministic policy, $\pi_{\mu_\theta,0} \equiv \mu$. An additional condition is needed for defining the relationship between SPG and DPG, which is a composite condition to define the regular delta-approximation:

- ***C.3 Regular Delta-Approximation:*** *functions v_σ parameterized by σ are said to be a regular delta-approximation on $\mathcal{R} \in \mathcal{A}$ if they satisfy the conditions: (1) the distribution v_σ converges to a delta distribution $\lim_{\sigma\downarrow 0} \int_{\mathcal{A}} v_\sigma(a', a) f(a) \mathrm{d}a = f(a')$ for $a' \subseteq \mathcal{R}$ and suitably smooth f; (2) $v_\sigma(a', \cdot)$ is supported on compact $\mathcal{C}'_a \subseteq \mathcal{A}$ with Lipschitz boundary, vanishes on the boundary and is continuously differentiable on $\mathcal{C}_{a'}$; (3) the gradient $\nabla_{a'} v_\sigma(a', a)$ always exists; (4) translation invariance: $v(a', a) = v(a'+\delta, a+\delta)$ for any $a \in \mathcal{A}$, $a' \in \mathcal{R}$, $a+\delta \in \mathcal{A}$, $a'+\delta \in \mathcal{A}$.*

Theorem 2.4 (Deterministic Policy Gradient as Limit of Stochastic Policy Gradient) *Consider a stochastic policy $\pi_{\mu_\theta,\sigma}$ such that $\pi_{\mu_\theta,\sigma}(a|s) = v_\sigma(\mu_\theta(s), a)$, where σ is a parameter controlling the variance and $v_\sigma(\mu_\theta(s), a)$ satisfy C.3 and the MDP satisfies C.1 and C.2, then,*

$$\lim_{\sigma\downarrow 0} \nabla_\theta J(\pi_{\mu_\theta,\sigma}) = \nabla_\theta J(\mu_\theta) \tag{2.184}$$

[3]Details and arguments for this operation can be referred to the original paper.

[4]The paper of *Silver D, Lever G, Heess N, et al. Deterministic policy gradient algorithms[C]. 2014.* drops the ∇_a on the Q term after approximation in Eq. (15) of the paper, and here we modified this typo.

which indicates the gradient of the DPG (r.h.s) is the limit case of standard SPG (l.h.s).

The proof of the above relationship is an outline of this book and we will not discuss it here. Details refer to the original paper (Silver et al. 2014).

Applications and Variants of Deterministic Policy Gradient
One of the most famous algorithms of DPG is the deep deterministic policy gradient (DDPG), which is a deep variant of DPG. DDPG combines DQN and actor-critic algorithms to use deterministic policy gradients for updating the policy, via a deep-learning approach. It has the target networks for both the actor and the critic, and provides for sample-efficient learning but is notoriously challenging to use due to its extreme brittleness and hyperparameter sensitivity in practice (Duan et al. 2016). The details and implementation of DDPG will be introduced in later chapters.

From the above we can see that the policy gradient can be estimated in at least two approaches: SPG and DPG, depending on the type of policy. Actually, they use two kinds of different estimators, the score function estimator for SPG and the pathwise derivative estimators for DPG, in the terminology of variational inference (VI).

A reparameterization trick makes it possible to apply policy gradients derived from the value function for stochastic policy, which is called the **stochastic value gradients (SVG)** (Heess et al. 2015). In SVG algorithms, a value λ is usually used as SVG(λ) to indicate how many steps are expanded in Bellman recursion. For example, the SVG(0) and SVG(1) indicate the Bellman recursion expanded with 0 and 1 step, respectively, and SVG(∞) indicates the Bellman recursion is expanded along the whole episodic trajectory in a finite horizon. SVG(0) is a model-free method with the action-value estimated with current policy, and therefore the value gradients are back-propagated to the policy; while SVG(1) is a model-based method using a learned transition model to evaluate the value of next state, as in original paper (Heess et al. 2015).

A very simple but useful example of reparameterization trick is to write a conditional Gaussian density $p(y|x) = \mathcal{N}(\mu(x), \sigma^2(x))$ as the function $y(x) = \mu(x) + \sigma(x)\epsilon, \epsilon \sim \mathcal{N}(0, 1)$. So one can produce samples procedurally by first sampling ϵ then deterministically constructing y, which makes the sampling process from the stochastic policy trackable for gradients. And the same procedure for back-propagating the gradients from the action-value function to the policy is made feasible in practice. In order to get the gradients for the stochastic policy through the value function as in DPG, SVG applies the reparameterization trick and takes extra expectation on stochastic noise. Soft actor-critic (SAC) and the original SVG (Heess et al. 2015) algorithm both follow this routine to use stochastic policy for continuous control.

For example, in SAC, the stochastic policy is reparameterized with a mean and a variance, together with a noise term sampled from a normal distribution. The optimization objective in SAC with an additional entropy term is:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t (R(S_t, A_t, S_{t+1}) + \alpha H(\pi(\cdot|S_t)))\right] \tag{2.185}$$

and therefore the relationship of value function and Q-value function becomes:

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}[Q^{\pi}(s,a)] + \alpha H(\pi(\cdot|s)) \tag{2.186}$$

$$= \mathbb{E}_{a\sim\pi}[Q^{\pi}(s,a) - \alpha \log \pi(a|s)] \tag{2.187}$$

The policy used in SAC is a normalized Gaussian distribution, which is different from traditional settings. The action in SAC can be represented as below via reparameterization trick:

$$a_\theta(s,\epsilon) = \tanh(\mu_\theta(s) + \sigma_\theta(s)\cdot\epsilon), \epsilon \sim \mathcal{N}(0, I) \tag{2.188}$$

Due to the stochasticity of the policy in SAC, the policy gradients are derived with the reparametrization trick through maximizing the expected value function, which is:

$$\max_\theta \mathbb{E}_{a\sim\pi_\theta}[Q^{\pi_\theta}(s,a) - \alpha \log \pi_\theta(a|s)] \tag{2.189}$$

$$= \max_\theta \mathbb{E}_{\epsilon\sim\mathcal{N}}[Q^{\pi_\theta}(s, a(s,\epsilon)) - \alpha \log \pi_\theta(a(s,\epsilon)|s)] \tag{2.190}$$

and the gradients can therefore go back through the Q-networks to the policy network, similar as in DPG, which are:

$$\nabla_\theta \frac{1}{|\mathcal{B}|}\sum_{S_t\in\mathcal{B}} (Q^{\pi_\theta}(S_t, a(S_t,\epsilon)) - \alpha \log \pi_\theta(a(S_t,\epsilon)|S_t)) \tag{2.191}$$

with a sampled batch $\mathcal{B}$ for updating the policy and $a(S_t, \epsilon)$ sampled from the stochastic policy with reparameterization trick. In this sense, the reparameterization trick makes it possible for the stochastic policy to be updated in a similar manner as DPG, and the resulting SVG is kind of an intermediate between DPG and SPG. DPG can also be regarded as a deterministic limit of SVG(0).

Gradient-Free Optimization

Apart from gradient-based optimization for policy-based learning, there are also non-gradient-based (also called gradient-free) optimization methods, which include cross-entropy (CE) method, covariance matrix adaptation (CMA) (Hansen and Ostermeier 1996), hill climbing, simplex/amoeba/Nelder-Mead algorithm (Nelder and Mead 1965), etc.

Example: Cross-Entropy (CE) Method

Instead of using gradient-based optimization for policies, CE method is usually faster for policy search in reinforcement learning as a non-gradient-based optimization method. In a CE method, the policy is updated iteratively and the optimization objective for parameters θ of the parameterized policy π_θ is:

$$\theta^* = \arg\max S(\theta) \tag{2.192}$$

where the $S(\theta)$ is the general objective function, and for our reinforcement learning cases, it could be the discounted expected return: $S(\theta) = R(\tau) = \sum_{t=0}^{T} \gamma^t R_t$.

The policy in the CE method can be parameterized as a multi-variate linear independent Gaussian distribution, and the distribution of the parameter vector at iteration t is $f_t \sim N(\mu_t, \sigma_t^2)$. After drawing n sample vectors $\theta_1, \ldots, \theta_n$ and evaluating their value $S(\theta_1), \ldots, S(\theta_i)$, we sort them and select the best $\lfloor \rho \cdot n \rfloor$ samples, where $0 < \rho < 1$ is the selection ratio. Denoting the set of indices of the selected samples by $I \in 1, 2, \ldots, n$, the mean of the distribution is updated using:

$$\mu_{t+1} =: \frac{\sum_{i \in I} \theta_i}{|I|} \tag{2.193}$$

and the deviation update is:

$$\sigma_{t+1}^2 := \frac{\sum_{i \in I} (\theta_i - \mu_{t+1})^T (\theta_i - \mu_{t+1})}{|I|} \tag{2.194}$$

The cross-entropy method is an efficient and general optimization algorithm. However, preliminary investigations showed that the applicability of CE to reinforcement learning problems is restricted severely by the phenomenon that the distribution concentrates to a single point too fast. Therefore, its applicability in reinforcement learning seems to be limited though fast, because it often converges to suboptimal policies. A standard technique for preventing early convergence is to introduce noise. General methods include adding a constant or an adaptive value on the standard deviation term during the iterative process for the Gaussian distribution like:

$$\sigma_{t+1}^2 := \frac{\sum_{i \in I} (\theta_i - \mu_{t+1})^T (\theta_i - \mu_{t+1})}{|I|} + Z_{t+1} \tag{2.195}$$

where $Z_t = \max(5 - \frac{t}{10}, 0)$ in work of Szita, et al.

2.7.4 Combination of Policy-Based and Value-Based Methods

With the above vanilla policy gradient method, some simple reinforcement learning tasks can be solved. However, there is usually a large variance in the evaluated updates if we choose to use Monte Carlo or TD(λ) estimation. We can use a critic as described in the value-based optimization to estimate the action-value function. So there will be two sets of parameters if we use parameterized value function as an approximation for policy optimization: the actor parameters and the critic parameters. This actually forms a very important algorithm architecture called actor-critic (AC), and typical algorithms include Q-value actor-critic, deep deterministic policy gradient (DDPG), etc.

Recall the policy gradient theorem introduced in previous sections, the derivatives of performance objective J with respect to the policy parameters θ are:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t) Q^\pi(S_t, A_t) \tag{2.196}$$

in which the $Q^\pi(S_t, A_t)$ are the true action-value function, and the simplest way of estimating $Q^\pi(S_t, A_t)$ is to use a sampled cumulative return $G_t = \sum_{t=0}^{\infty} \gamma^{t-1} R(S_t, A_t)$. In AC, we apply a critic to estimate the action-value function: $Q^w(S_t, A_t) \approx Q^\pi(S_t, A_t)$. And the update rule of the policy in AC is therefore:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t) Q^w(S_t, A_t) \tag{2.197}$$

where w are parameters of the critic for value function approximation. The critic can be evaluated with an appropriate policy evaluation algorithms such as temporal difference (TD) learning, like $\Delta w = \alpha(Q^\pi(S_t, A_t; w) - R_{t+1} + \gamma v_\pi(S_{t+1}, w))\nabla_w Q^\pi(S_t, A_t; w)$ for TD(0) estimation as in Eq. (2.100). More details about AC algorithm and implementation will be discussed in Chaps. 5 and 6.

Although the AC framework helps alleviate the variances in policy updates, it can introduce bias and potential instability due to replacing the true action-value function with the estimated one, which requires **compatible function approximation** to ensure its unbiased estimation as proposed by Sutton et al. (2000).

Compatible Function Approximation

The compatible function approximation conditions hold for both SPG and DPG. We will show them individually. The "compatible" here indicates that the approximate action-value function $Q^w(s, a)$ is compatible with the corresponding policy.

For SPG Specifically, the compatible function approximation proposes two conditions to ensure the unbiased estimation using the approximated action-value function $Q^{\pi}(s, a)$: (1) $Q^{w}(s, a) = \nabla_{\theta} \log \pi_{\theta}(a|s)^{T} w$ and (2) the parameters w are chosen to minimize the mean-squared error $\text{MSE}(w) = \mathbb{E}_{s\sim\rho^{\pi}, a\sim\pi_{\theta}}[(Q^{w}(s, a) - Q^{\pi}(s, a))^{2}]$. More intuitively, condition (1) says that compatible function approximators are linear in "features" of the stochastic policy, $\nabla_{\theta} \log \pi_{\theta}(a|s)$, and condition (2) requires that the parameters w are the solution to the linear regression problem that estimates $Q^{\pi}(s, a)$ from these features. In practice, condition (2) is usually relaxed in favor of policy evaluation algorithms that estimate the value function more efficiently by temporal difference learning.

If both conditions are satisfied, then the overall algorithm of AC is equivalent to not using the critic for approximation, like in the REINFORCE algorithm. This can be proved simply by setting the MSE in the condition (2) equivalent to 0 and taking gradients, then substituting the condition (1) into it:

$$\nabla_{w}\text{MSE}(w) = \mathbb{E}[2(Q^{w}(s, a) - Q^{\pi}(s, a))\nabla_{w} Q^{w}(s, a)] \tag{2.198}$$

$$= \mathbb{E}[2(Q^{w}(s, a) - Q^{\pi}(s, a))\nabla_{\theta} \log \pi_{\theta}(a|s)] \tag{2.199}$$

$$= 0 \tag{2.200}$$

$$\Rightarrow \mathbb{E}[Q^{w}(s, a)\nabla_{\theta} \log \pi_{\theta}(a|s)] = \mathbb{E}[Q^{\pi}(s, a)\nabla_{\theta} \log \pi_{\theta}(a|s)] \tag{2.201}$$

For DPG The two conditions in compatible function approximation are modified accordingly with respect to the deterministic policy $\mu_{\theta}(s)$: (1) $\nabla_{a} Q^{w}(s, a)|_{a=\mu_{\theta}(s)} = \nabla_{\theta}\mu_{\theta}(s)^{T} w$ and (2) w minimizes the mean-squared error, $\text{MSE}(\theta, w) = \mathbb{E}[\epsilon(s; \theta, w)^{T}\epsilon(s; \theta, w)]$ where $\epsilon(s; \theta, w) = \nabla_{a} Q^{w}(s, a)|_{a=\mu_{\theta}(s)} - \nabla_{a} Q^{w}(s, a)|_{a=\mu_{\theta}(s)}$. It can also be proved that these conditions ensure the unbiased estimation through transferring the approximation back to no-critic cases:

$$\nabla_{w}\text{MSE}(\theta, w) = 0 \tag{2.202}$$

$$\Rightarrow \mathbb{E}[\nabla_{\theta}\mu_{\theta}(s)\epsilon(s; \theta, w)] = 0 \tag{2.203}$$

$$\Rightarrow \mathbb{E}[\nabla_{\theta}\mu_{\theta}(s)\nabla_{a} Q^{w}(s, a)|_{a=\mu_{\theta}(s)}] = \mathbb{E}[\nabla_{\theta}\mu_{\theta}(s)\nabla_{a} Q^{\mu}(s, a)|_{a=\mu_{\theta}(s)}] \tag{2.204}$$

And it applies to both on-policy $\mathbb{E}_{s\sim\rho^{\mu}}[\cdot]$ and off-policy $\mathbb{E}_{s\sim\rho^{\beta}}[\cdot]$ cases.

Other Methods

If we replace the action-value function $Q^{\pi}(s, a)$ with advantage function $A^{\pi}(s, a)$ in Eq. (2.196) (as subtracting the baseline does not affect the gradients):

$$A^{\pi_{\theta}}(s, a) = Q^{\pi_{\theta}}(s, a) - V^{\pi_{\theta}}(s) \tag{2.205}$$

Then we actually get a more advanced algorithm called advantage actor-critic (A2C), which can use the TD-error to estimate the advantage function. This will not affect above theorems but changes the variances of gradient estimators.

Recently, people have proposed actor-free methods, like the QT-Opt algorithm (Kalashnikov et al. 2018) and the Q2-Opt algorithm (Bodnar et al. 2019) based on that. These methods are combinations of policy-based and value-based optimization as well, gradient-free CE method and DQN specifically. Instead of using the sampled discounted return as an estimation of actions sampled from Gaussian distributions, they apply action value approximation for learning $Q^{\pi_\theta}(s, a)$ instead, which are proved to be efficient and effective for robot learning in reality especially when there are demonstration datasets.

References

Achiam J, Knight E, Abbeel P (2019) Towards characterizing divergence in deep Q-learning. Preprint. arXiv:190308894

Auer P, Cesa-Bianchi N, Freund Y, Schapire RE (1995) Gambling in a rigged casino: the adversarial multi-armed bandit problem. In: Proceedings of IEEE 36th annual foundations of computer science. IEEE, Piscataway, pp 322–331

Bellman R et al (1954) The theory of dynamic programming. Bull Am Math Soc 60(6):503–515

Bodnar C, Li A, Hausman K, Pastor P, Kalakrishnan M (2019) Quantile QT-Opt for risk-aware vision-based robotic grasping. Preprint. arXiv:191002787

Bubeck S, Cesa-Bianchi N et al (2012) Regret analysis of stochastic and nonstochastic multi-armed bandit problems. Found Trends® Mach Learn 5(1):1–122

Degris T, White M, Sutton RS (2012) Linear off-policy actor-critic. In: In international conference on machine learning. Citeseer

Duan Y, Chen X, Houthooft R, Schulman J, Abbeel P (2016) Benchmarking deep reinforcement learning for continuous control. In: International conference on machine learning, pp 1329–1338

Fu J, Singh A, Ghosh D, Yang L, Levine S (2018) Variational inverse control with events: a general framework for data-driven reward definition. In: Advances in neural information processing systems, pp 8538–8547

Hansen N, Ostermeier A (1996) Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In: Proceedings of IEEE international conference on evolutionary computation. IEEE, Piscataway, pp 312–317

Heess N, Wayne G, Silver D, Lillicrap T, Erez T, Tassa Y (2015) Learning continuous control policies by stochastic value gradients. In: Advances in neural information processing systems, pp 2944–2952

Jang E, Gu S, Poole B (2016) Categorical reparameterization with gumbel-softmax. Preprint. arXiv:161101144

Kalashnikov D, Irpan A, Pastor P, Ibarz J, Herzog A, Jang E, Quillen D, Holly E, Kalakrishnan M, Vanhoucke V et al (2018) Qt-opt: scalable deep reinforcement learning for vision-based robotic manipulation. Preprint. arXiv:180610293

Kingma DP, Welling M (2014) Auto-encoding variational Bayes. In: Proceedings of the international conference on learning representations (ICLR)

Leshno M, Lin VY, Pinkus A, Schocken S (1993) Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural Netw 6(6):861–867

Levine S (2018) Reinforcement learning and control as probabilistic inference: tutorial and review. Preprint. arXiv:180500909

Nelder JA, Mead R (1965) A simplex method for function minimization. Comput J 7(4):308–313

Peters J, Schaal S (2008) Natural actor-critic. Neurocomputing 71(7–9):1180–1190

Pyeatt LD, Howe AE et al (2001) Decision tree function approximation in reinforcement learning. In: Proceedings of the third international symposium on adaptive systems: evolutionary computation and probabilistic graphical models, Cuba, vol 2, pp 70–77

Schmidhuber J (2015) Deep learning in neural networks: an overview. Neural Netw 61:85–117

Silver D, Lever G, Heess N, Degris T, Wierstra D, Riedmiller M (2014) Deterministic policy gradient algorithms. In: Proceedings of the 31 st international conference on machine learning, Beijing

Singh S, Jaakkola T, Littman ML, Szepesvári C (2000) Convergence results for single-step on-policy reinforcement-learning algorithms. Mach Learn 38(3):287–308

Sutton RS, McAllester DA, Singh SP, Mansour Y (2000) Policy gradient methods for reinforcement learning with function approximation. In: Advances in neural information processing systems, pp 1057–1063

Szepesvári C (1998) The asymptotic convergence-rate of Q-learning. In: Advances in neural information processing systems, pp 1064–1070

Tsitsiklis JN, Roy BV (1997) An analysis of temporal-difference learning with function approximation. Technical Report. IEEE transactions on automatic control

Van Hasselt H, Guez A, Silver D (2016) Deep reinforcement learning with double Q-learning. In: Thirtieth AAAI conference on artificial intelligence

Van Hasselt H, Doron Y, Strub F, Hessel M, Sonnerat N, Modayil J (2018) Deep reinforcement learning and the deadly triad. Preprint. arXiv:181202648

Watkins CJ, Dayan P (1992) Q-learning. Mach Learn 8(3–4):279–292

Williams RJ, Baird III LC (1993) Analysis of some incremental variants of policy iteration: first steps toward understanding actor-critic learning systems. Technical Report. NU-CCS-93-11. Northeastern University, College of Computer Science