A Mathematical Introduction to Reinforcement Learning

Xintian Han

1 Introduction

Reinforcement learning (RL) is a general approach to solving reward-based problems. RL tries to mimic the way that human learns new things not from a teacher but from interaction with the environment. For example, when a baby learns to wave hands, cry and laugh, it learns from the feedback from parents. When we drive a car, we learn to turn left and right to avoid the crash of the car on the road. RL is the way that machines learn to achieve the goal from the interactions with the environment. Mathematically, RL is also a sequential decision making and control problem. For instance, when driving a car, we need to choose turning left or right every time after we make the previous decision.

Reinforcement learning is one type of machine learning. The most famous type of machine learning is supervised learning. In supervised learning, algorithms are developed to make outputs mimic the labels given in the training set. Unlike supervised learning, it is difficult to provide a supervisor in the problem of RL, because usually we have no idea what the right decision is. For example, if we want to drive a car, we cannot give a label to every picture the camera takes.

Besides driving a car, RL is widely used in engineering, neuroscience, psychology, mathematics and economics. It's power has also been demonstrated in practice by showing that RL-based systems can:

- Make a humanoid robot walk
- Fly stunt maneuvers in a helicopter.
- Defeat the world champion at Backgammon and Go.
- Manage an investment portfolio.
- Control a power station.
- Play many different Atari games better than humans.

Though complicated systems control all of the challenging examples above, there is still simple mathematics behind the systems. In this paper, we will introduce the mathematics of RL, mainly based on David Silver's lecture *Reinforcement Learning* [3] and Richard Sutton's book *Reinforcement Learning: An Introduction* [5].

We proceed as follows. In Section 2, we model the environment formally by Markov decision process. We also define return, policy and value function mathematically, and introduce the Bellman equation. Section 3 introduces dynamic programming and shows how to solve the MDP by dynamic programming, including policy evaluation, policy iteration, and value iteration. Finally, in Section 4, we briefly summarizes the difficulty of reinforcement learning in real life which leads to the development of deep reinforcement learning.

2 Markov Decision Process (MDP)

We will describe the key ingredient in the reinforcement learning, Markov Decision Process, using a simple example. We only talk about finite states MDP here. For infinite state and continuous state MDP, you can refer to Sutton's book [5]. We also introduce other important elements of reinforcement learning, such as return, policy and value function, in this section. Then we derive the Bellman Equation. Finally, we discuss optimal policy, optimal value function and Bellman optimality equation.

2.1 Markov Process and Markov Decision Process

We start with the definition of a Markov process. A sequence of states is Markov if and only if the probability of moving to the next state S_{t+1} depends only on the present state S_t and not on the previous states $S_1, S_2, \cdots, S_{t-1}$. That is, for all t,

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, S_2, \cdots, S_t].$$

We always talk about time-homogeneous Markov chain in RL, in which the probability of the transition is independent of t:

$$\mathbb{P}[S_{t+1} = s'|S_t = s] = \mathbb{P}[S_t = s'|S_{t-1} = s].$$

Formally,

Definition 1 (Markov Process)**.** a Markov Process (or Markov Chain) is a tuple $(\mathcal{S}, \mathcal{P})$, where

- $\mathcal{S}$ is a (finite) set of states
- $\mathcal{P}$ is a state transition probability matrix. $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s]$.

The dynamics of the Markov process proceeds as follows: We start in some state s_0, and moves to some successor state s_1, drawn from $\mathcal{P}_{s_0 s_1}$. We then moves to s_2 drawn from $\mathcal{P}_{s_1 s_2}$ and so on. We represent this dynamic as follows:

$$s_0 \to s_1 \to s_2 \to s_3 \to \dots$$

If we introduce reward, action and discount into a Markov process, we get a Markov decision process.

Definition 2 (Markov Decision Process)**.** A Markov decision process is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \gamma, \mathcal{R})$, where:

- $\mathcal{S}$ is a finite set of *states*.
- $\mathcal{A}$ is a finite set of *actions*.
- $\mathcal{P}$ is the *state transition probability matrix*, $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]$.
- $\gamma \in [0, 1]$ is called the *discount factor*.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a *reward function*.

The MDP is used to model the environment in reinforcement learning. In the MDP, the transition to the next state S_{t+1} depends not only on the current state S_t, but also depends on the action A_t you make at the current state. Also, each state-action pair is attached with a reward function. In the previous autonomous car driving example, states are pictures that cameras take; actions can be turning left or right; transition probability can be the probability that you are going right after you make the action turning right; reward can be positive if the car does not crash and negative if the car crashes. We will talk about the discount factor in the next subsection.

The dynamic of MDP proceeds as follows: We start in some state s_0, and choose some actions $a_0 \in \mathcal{A}$ to take in the MDP. As a result of our choice, the state of the MDP randomly transits to some successor state s_1, drawn from $\mathcal{P}^{a_0}_{s_0 s_1}$. Then, from state s_1, we pick another action a_1. Again, we come to some state s_2, drawn from $\mathcal{P}^{a_1}_{s_1 s_2}$. We then pick a_2, and so on. We can represent this sequential decision making process as follows:

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

2.2 Return, Policy and Value function

Our goal in RL is to choose actions over time so as to maximize the expected value of the *return*, i.e. choose the optimal *policy*. We define *return* and *policy* as follows. The *return* G_t is the total discounted reward from time-step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

The discounted future rewards can be interpreted as the current value of future rewards. The discount factor values the immediate reward above delayed reward: γ close to 0 leads to "myopic" evaluation; γ close to 1 leads to "far-sighted" evaluation. Several reasons why we introduce this discount factor in RL are:

- Discount factor avoids infinite returns in cyclic Markov process (due to the property of geometric series);
- There is uncertainty about the future reward;
- If the reward is financial, immediate rewards may earn more interest than delayed rewards;
- Animal/human behavior shows preference for immediate reward.

It is sometimes possible to use undiscounted Markov decision processes (i.e. $\gamma = 1$). We will show an example later.

A *policy* π is a distribution over actions given states,

$$\pi(a|s) = \mathbb{P}[A_t = a | S_t = s].$$

It is also time independent. A policy guides the choice of action at a given state.

We then introduce two value functions, *state-value function* and *action-value function*. They help us to find the optimal policy. The *state-value function* v_π of an MDP is the expected return starting from state s, and then following policy π

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

The state-value function $v_\pi(s)$ gives the long-term value of state s when following policy π. We can decompose the state-value function into two parts: the immediate reward R_{t+1} and discounted value of successor state $\gamma v_\pi(S_{t+1})$.

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots |S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots)|S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \\
&= \underbrace{\mathbb{E}_\pi[R_{t+1}|S_t = s]}_{\text{immediate reward}} + \underbrace{\mathbb{E}_\pi[\gamma v_\pi(S_{t+1})|S_t = s]}_{\text{discounted value of successor state}} .
\end{aligned}$$

The *action-value function* $q_\pi(s, a)$ is the expected return starting from state s, taking action a, and then following policy π

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

Similarly, the action-value function can be decomposed as follows:

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

To simplify notations, we define $\mathcal{R}_s^a = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$. We can also see the relationship between $v_\pi(s)$ and $q_\pi(s, a)$:

$$\begin{aligned}
v_\pi(s) &= \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s, a), \\
q_\pi(s, a) &= \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s').
\end{aligned}$$

Expressing $q_\pi(s, a)$ in terms of $v_\pi(s)$ in the expression of $v_\pi(s)$, we get a *Bellman equation* for v_π,

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right). \tag{1}$$

The Bellman equation relates the state-value function of one state with that of other states. Similarly, we also have a Bellman equation for $q_\pi(s, a)$,

$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a'\in\mathcal{A}} \pi(a'|s') q_\pi(s', a').$$

One use for the Bellman equation is to compute the value function for a given policy. If we combine all the Bellman equations in an MDP with n states, we get n linear equations for the n unknown value functions. We can get the value functions by solving linear equations. However, this step may take $O(n^3)$ time complexity. We will show how to solve these equations by dynamic programming in the next section.

2.3 Optimal Value Function and Optimal Policy

What we care about for an MDP is the optimal value function and the optimal policy. The *optimal state-value function* $v_*(s)$ is the maximum value function over all policies:

$$v_*(s) = \max_\pi v_\pi(s).$$

The optimal value function specifies the best possible performance in the MDP. We say an MDP is "solved" when we know the optimal value function.

The *optimal action-value function* $q_*(s, a)$ is the maximum action-value function over all policies:

$$q_*(s, a) = \max_\pi q_\pi(s, a).$$

Before we talk about optimal policies, we define a partial ordering over policies:

$$\pi \geq \pi' \quad \text{if } v_\pi(s) \geq v_{\pi'}(s), \forall s. \tag{2}$$

The *optimal policy* π_* is better than or equal to all other policies, $\pi_* \geq \pi$, for all π, where $\geq$ is the partial ordering in (2). The following Theorem 1 guarantees that such optimal policy exists.

Theorem 1. *For any Markov Decision Process,*

- *There exists an optimal policy π_* that is better than or equal to all other policies, $\pi_* \geq \pi$, for all π.*
- *All optimal policies achieve the optimal value function, $v_{\pi_*}(s) = v_*(s)$.*
- *All optimal policies achieve the optimal action-value function, $q_{\pi_*}(s, a) = q_*(s, a)$.*

From the theorem, we can find an optimal policy immediately by maximizing $q_*(s, a)$ over all actions,

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} q_*(s, a), \\ 0 & \text{otherwise.} \end{cases}$$

The remaining question is how to get the optimal value function. To answer this, we introduce the Bellman optimality equation. We can find the relationship between the optimal state-value function and the optimal action-value function,

$$v_*(s) = \max_a q_*(s, a),$$
$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s').$$

Expressing $q_*(s, a)$ in terms of $v_*(s)$ in the expression of $v_*(s)$, we get a Bellman optimality equation for v_* and q_*.

$$v_*(s) = \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right) \tag{3}$$

We also have a Bellman equation for q_*.

$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s', a').$$

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

Table 1: Small Gridworld: A simple example for MDP. In this Small Gridworld, each square with number on it is a non-terminal state. The two shaded squares are the terminal state. The goal is to make a robot move to the terminal state from any non-terminal state.

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Table 2: State-Value Function for a random policy in Small Gridworld. We obtain the value functions for this Small Gridworld by solving linear systems of Bellman equations.

The Bellman optimality equations are non-linear and there is no closed form solution in general. We will show how to solve the system of Bellman equations for all the states by dynamic programming in Section 3.

2.4 Example

We use the simple Gridworld example (see Table 1) to illustrate what an MDP is. To make things simpler, we construct an undiscounted episodic MDP (i.e. the discount factor $\gamma = 1$). We describe it as follows:

- *States.* There are 15 states. 14 non-terminal states are labeled 1,2,...,14. There is one terminal state which is at the northwest and southeast corner and is shown as shaded squares.
- *Actions.* There are four actions {N,S,W,E}, i.e. moving north, south, west and east. Actions leading out of the grid will leave the state unchanged.
- *State transition probability matrix.* There is no randomness after we make the action at the given state so each element of the state transition matrix is either 1 or 0. For example, $\mathbb{P}[S_{t+1} = 1|S_t = 5, A_t = \text{N}] = 1$ and $\mathbb{P}[S_{t+1} = 4|S_t = 5, A_t = \text{N}] = 0$.
- *Discount factor.* We use $\gamma = 1$ in this example.
- *Reward function.* Our goal is to make the robot achieve the terminal state from an initial state s_0 so we design the reward signal to be that the robot get -1 reward for each transition until the terminal state is reached. We want the robot to get to the terminal as soon as possible. And the robot will stop at the terminal state with no transition any more.

In this example, we can get the state-value function of each state for a given policy by solving the linear equations. For example, for a random policy $\pi(\text{N}|s) = \pi(\text{S}|s) = \pi(\text{W}|s) = \pi(\text{E}|s) = 0.25, \forall s$, the value functions are shown in Table 2.

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Table 3: Iterative policy evaluation for Small Gridworld. We show the value functions we get at iterations $k = 0, 1, 2, 3, 10$ of policy evaluation, and the true value functions denoted by $k = \infty$. We could see that the value function are converging to the true value functions as k becomes larger.

3 Dynamic Programming

In the previous section, we introduce MDP to formalize the reinforcement learning problem. In this section, we describe how to solve an MDP. Basically, it is done by dynamic programming. Generally speaking, dynamic programming solves a complex problem by breaking it down into simpler recursive subproblems, solving each of those subproblems just once, and storing and reusing their solutions. In RL, Bellman equation gives recursive decompositions; value function stores and reuses solutions. We will introduce three paradigms of dynamic programming in reinforcement learning: policy evaluation, policy iteration and value iteration. Policy evaluation is used to find the value function of a certain policy. Policy iteration and value iteration are used to find the optimal value function and optimal policy.

3.1 Policy Evaluation

In section 2, we evaluate a given policy by solving a couple of linear equations. Now we use dynamic programming to solve it. The basic idea is that we believe the Bellman equation is true. We start from an initial guess v_1 and then apply Bellman equation iteratively to it: $v_1 \to v_2 \to \dots v_\pi$. We use synchronous updates, that is, we update the value functions of the present iteration at the same time based on the that of the previous iteration. At each iteration $k + 1$, for all states $s \in \mathcal{S}$, we update v_{k+1} from $v_k(s')$ according to Bellman equations, where s' is a successor state of s:

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right).$$

The iterative process stops when the maximum difference between value function at the current step and that at the previous step is smaller than some small positive constant. See Table 3 for the convergence of the policy evaluation in the Small Gridworld example.

3.2 Policy Iteration

Our ultimate goal is to find the optimal policy. We could use policy iteration to find it. The policy iteration algorithm proceeds as follows

	W	W	W,S
N	E,N	E,S	S
N	N,E	S,E	S
N,E	E	E	

Table 4: Optimal Policy in Small Gridworld. We illustrate the optimal policy obtained by policy iteration in Small Gridworld. The robot will choose the greedy direction towards the terminal state.

1. Initialize π randomly
2. Repeat until the previous policy and the current policy are the same.
   (a) Evaluate v_π by policy evaluation.
   (b) Using synchronous updates, for each state s, let

$$\pi(s) := \arg\max_{a \in \mathcal{A}} q(s,a) = \arg\max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right).$$

For our Small Gridworld example, the update in the inner-loop is just choosing the action towards the largest previous state-value function.

We can prove that this policy iteration algorithm will improve the policy for each step. Suppose we have a deterministic policy π and then after one step we get π'. The iteration improves the value from any state s over one step,

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s,a) \geq q_\pi(s, \pi(s)) = v_\pi(s).$$

The iteration also improves the value function, $v_{\pi'}(s) \geq v_\pi(s)$ because:

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

If improvements stop,

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s,a) = q_\pi(s, \pi(s)) = v_\pi(s),$$

which means the Bellman optimality equation has been satisfied,

$$v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s,a).$$

Therefore, $v_\pi(s) = v_*(s)$ for all $s \in \mathcal{S}$ and π is an optimal policy. If we evaluate the optimal policy, we get the optimal value function. For our Gridworld example, actually one step of policy iteration can help us get the optimal policy. The optimal policy is shown in 4.

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.0	-2.0	-2.0
-1.0	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.0
-2.0	-2.0	-1.0	0.0

$k = 3$

0.0	-1.0	-2.0	-3.0
-1.0	-2.0	-3.0	-2.0
-2.0	-3.0	-2.0	-1.0
-3.0	-2.0	-1.0	0.0

Table 5: Value Iteration for Small Gridworld. We show iterations $k = 0, 1, 2, 3$ in value iteration. Three steps of value iterations give the optimal value function.

3.3 Value Iteration

We will introduce an alternative way to find the optimal policy and value function, which is value iteration. The value iteration proceeds as follows:

1. For each state s, initialize $v(s) = 0$
2. Repeat until the maximum difference between value function at the current step and that at the previous step is smaller than some small positive constant: Using synchronous updates, for each state s, let $v(s) := \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right)$.

Then we can get the optimal policy from the optimal function. We show how value iteration works for our Small Gridworld example in Table 5.

3.4 Convergence

We will now show the convergence guarantee of policy evaluation, and value iteration by Contraction Mapping Theorem. Consider the vector space $\mathcal{V}$ over value functions. The dimension is $|\mathcal{S}|$. Each point in this space specifies a state-value function. We measure the distance between two state-value functions u and v by the ∞-norm, i.e. the largest difference between state values:

$$\|u - v\|_\infty = \max_{s \in \mathcal{S}} |u(s) - v(s)|.$$

We can rewrite the Bellman equation (1) in the matrix form.

$$v_\pi = \mathcal{R}_\pi + \gamma \mathcal{P}_\pi v_\pi,$$

where v_π is a column vector with one entry per state.

$$\begin{bmatrix} v_\pi(1) \\ \vdots \\ v_\pi(n) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1^\pi \\ \vdots \\ \mathcal{R}_n^\pi \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11}^\pi & \cdots & \mathcal{P}_{1n}^\pi \\ \vdots & \vdots & \vdots \\ \mathcal{P}_{n1}^\pi & \cdots & \mathcal{P}_{nn}^\pi \end{bmatrix} \begin{bmatrix} v_\pi(1) \\ \vdots \\ v_\pi(n) \end{bmatrix}.$$

Define the Bellman equation backup operator

$$T_\pi(v) = R_\pi + \gamma \mathcal{P}_\pi v.$$

It turns out that this operator is a γ-contraction, i.e. it makes functions closer by at least $\gamma < 1$:

$$\begin{aligned}\|T_\pi(u) - T_\pi(v)\|_\infty &= \|(\mathcal{R}_\pi + \gamma\mathcal{P}_\pi u) - (\mathcal{R}_\pi + \mathcal{R}\mathcal{P}_\pi v)\|_\infty \\ &= \|\gamma\mathcal{P}_\pi(u - v)\|_\infty \\ &\leq \|\gamma\mathcal{P}_\pi\|u - v\|_\infty\|_\infty \\ &\leq \gamma\|u - v\|_\infty.\end{aligned}$$

By the following Contraction Mapping Theorem, we can see iterative policy evaluation will converge to the true state-value function of the policy. Since there is finite number of states and actions, policy iteration will also converge.

Theorem 2 (Contraction Mapping Theorem)**.** *For any metric space $\mathcal{V}$ that is complete under an operator $T(v)$, where T is a γ-contraction, T converges to a unique fixed point at a linear convergence rate of γ.*

Similarly, we can write the Bellman optimality equation (3) into matrix form:

$$v_* = \max_{a\in\mathcal{A}} \mathcal{R}_a + \gamma\mathcal{P}_a v_*.$$

And we have the Bellman optimality backup operator T^*:

$$T_*(v) = \max_{a\in\mathcal{A}} \mathcal{R}_a + \gamma\mathcal{P}_a v.$$

This operator is also a γ-contraction map, so value iteration will converge.

4 Conclusion

From playing games to automatic driving, reinforcement learning plays an increasingly important role in daily life. We have shown that the building of RL is based on mathematical foundations. The MDP provides a formulation of the environment while the policy iteration and value iteration help us to find the optimal policy. There is no universal argument about choosing value iteration or policy iteration. Generally speaking, for small MDPs, policy iteration is often very fast and converges with very few iterations. For large MDPs, evaluation of a policy would take very large time complexity and we may use value iteration. For this reason, in practice value iteration seems to be used more often than policy iteration. The choice of policy iteration or value iteration still needs to be explored in specific problems.

In addition, in real life applications, the state space and action space are much more complicated. It is extremely hard to solve the reinforcement learning problem with simple policy and value iterations. We may need the help of deep learning to model the environment, the policy and the value functions. And a new tool, policy gradient, is used to find the optimal value function. Recently, deep reinforcement learning enjoys the current state-of-art performance, which utilizes the popular neural networks. But the theory about why deep reinforcement learning work is still unknown. We just know the neural network can approximate a large family of functions.

Besides deep reinforcement learning, we also do not mention model free reinforcement learning. Basically, in model-free reinforcement learning we do not have the model of the environment, or it is hard for us to get the model of the environment. Like playing poker, it is hard for us to calculate

the distribution of the next state since we do not know the decker of our opponents. But we could get a sample from the model. In this case, we just learn the value function and the policy from the sample state and sample reward we get.

If you are interested in exploring more in reinforcement learning, you can refer to [2, 4, 1, 3] for further readings.

References

[1] Andrew ng's lecture. https://www.youtube.com/watch?v=LKdFTsM3hl4&index=17&list=PLA89DCFA6ADACE599. 4

[2] Berkeley's lecture. http://rll.berkeley.edu/deeprlcourse/. 4

[3] David silver's lecture. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html. 1, 4

[4] Feifei li's lecture. https://www.youtube.com/watch?v=lvoHnicueoE&t=260. 4

[5] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998. 1, 2