Theory & Algorithms for Reinforcement Learning

Application in Public Transportation Planning

Evelyn Chee Yi Lyn
Supervisor: Dr. Li Qianxiao

Department of Mathematics
Faculty of Science
National University of Singapore

2017/2018

Summary

In recent years, there is tremendous progress in the field of artificial intelligence. The three main contributors to this improvement are higher computing power, greater amount of data and new developments in algorithms. AlphaGo, which is a Go-playing software, was one of the greatest advances ever in artificial intelligence as it became the first computer program to beat a human professional Go player. A key algorithm enabling this breakthrough is reinforcement learning.

Reinforcement learning has gradually become one of the active research areas in this field of study. It was inspired by behaviorist psychology with the idea of building a learning system that wants something and can adapt its behavior to maximize a signal it receives from the environment. The work on the optimal control of dynamical systems in the 1950s by Bellman is the underlying mathematical principle in reinforcement learning.

In this project, the goal is to understand the key ideas and algorithms in reinforcement learning. The content from Chapter 1 to 5 in this report, such as definitions, theorems and proofs, are mainly based on [2] and [5]. Pseudocodes of the algorithms introduced are included in the Appendix. One other objective of this project is to explore the applications of reinforcement learning to interesting problems. In Chapter 6, I have applied some of the algorithms to the public transportation planning problem. Computer programmes have been written to simulate the problem. I have also implemented the reinforcement learning algorithms into the programme and it will be shown that the results generated are useful and applicable in the planning of public transportation.

Contents

Chapter 1

Introduction

There are three branches in machine learning: supervised learning, unsupervised learning and reinforcement learning. Unlike supervised learning, there is no supervisor in reinforcement learning, which means there is no guidance on what is the best action to take. Instead, only a reward signal is provided. The two most important features distinguishing reinforcement learning from other machine learning methods are trial-and-error search and delayed reward.

It is often that examples of correct and representative behavior of all the available situation are not obtainable in interactive problems, rendering it impractical to implement supervised learning. Instead of being told which actions to take in each situation, the learner in reinforcement learning discovers the actions through experience, which leads to the trial-and-error search characteristic. Moreover, most times, the focus of the problem is not the immediate reward but the best return in the long run. A decision might be made now but its value is only known at a delayed time. Therefore, sometimes sacrificing immediate reward is a better choice in order to gain more long-term reward.

An example of reinforcement learning problem would be making a humanoid robot walk, in which the robot will be rewarded for each forward motion and penalized for falling over. The robot will interact with the environment and learn which action is best for each situation. Sometimes, instead of moving forward, it is better to stop and recharge in order to prevent itself from falling over in hours time. Stopping yields no immediate positive reward but in the long run, it will have a higher return because the robot is able to move for a longer period of time.

1.1 The Agent-Environment Interface

In reinforcement learning, there is always an interaction between the agent and the environment. The agent is the learner and the decision-maker while everything else outside the agent is the environment. The agent has no control over the environment except when the environment responds to the actions taken by the agent.

At each time step t, the agent is provided with the information $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the state space consisting all the possible state in the environment. An action, $A_t \in \mathcal{A}$ will then be chosen by the agent, where $\mathcal{A}$ is the action space consisting all the actions that the agent can take. The environment will react by providing the agent with a single numerical value, which is the reward, $R_{t+1} \in \mathbb{R}$ and a new state, S_{t+1}. The process then repeats itself. In this report, we only consider processes with finite state space and finite action space, i.e. $|\mathcal{S}| < \infty$ and $|\mathcal{A}| < \infty$.

Reinforcement learning specifies how the agent learns from the interaction with the environment based on the goal of maximizing the total reward over time.

1.1.1 Elements in Reinforcement Learning

Below are four major components of a reinforcement learning system:

1. **Policy**: A policy defines the agent's way of behaving at a given situation. There are two types of policy. A deterministic policy $\pi : \mathcal{S} \to \mathcal{A}$ describes the action taken in a state whereas a stochastic policy $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$ indicates the probability of taking an action in a given state.

2. **Reward signal**: Reward is a measure of the intrinsic desirability of that state or state-action pair. Rewards are immediate but our goal is to maximize the total reward it receives in the long run, known as the return. Generally, rewards and returns are random variables. Here, we define the function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ as

$$r(s, a, s') = \mathbb{E}[R_{t+1} | S_t = s, A_t = a, S_{t+1} = s'],$$

which is the expected immediate reward for a transition from state s to s' by taking action a.

3. **Value function**: A value of a state indicates the expected total reward in the long run, starting from that state. A state with high reward does not necessarily imply a high value because it might be followed by other states with low rewards and vice versa. There two different value functions: $v_\pi(s)$ which is the value of state s under policy π and $q_\pi(s, a)$ which is the value of taking action a in state s under policy π. These two functions will be formally defined in the later section.

4. **Model**: A model describes the behavior of the environment and predicts what the environment will do next, which are useful for planning. A model is represented by the function $p : \mathcal{S} \times \mathbb{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$, where

$$p(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a].$$

In other words, $p(s', r|a, s)$ denotes the probability of transitioning to the next state s' and obtaining reward r by taking action a at state s.

1.2 Markov Decision Process

Here, the environment of the reinforcement learning problem is discussed. It is described by the Markov decision process with components from the previous section.

1.2.1 Markov Property

An important property that the environment satisfies is the Markov property.

Definition 1.2.1. *A state is said to satisfy the Markov property if and only if*

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_0, S_1, ..., S_t]. \tag{1.1}$$

With this property, the next state could be predicted without any information on the complete history up to the current time. In other words, the current state captures all relevant historical information and is sufficient to predict the future.

1.2.2 Markov Decision Process

A Markov chain is a discrete time sequence of random states with the Markov property. It is a memoryless stochastic process as only the previous state is needed to decide the next state.

Definition 1.2.2. *A Markov process is defined as a 2-tuple,* $(\mathcal{S}, P)$ *where*

- $\mathcal{S}$ *is the state space,*
- P *is the state transition probability matrix which consists of* $\mathbb{P}[S_{t+1} = s' | S_t = s]$*, the transition probability from state* s *to state* s'*.*

Having actions and rewards in a Markov process leads to a Markov decision process. Actions allow choices to be made and rewards give motivation. At each time step, the process is at some state s. After taking an action a, the process responses with a new state s' and a reward r with a probability of $p(s', r|s, a)$. The Markov property is satisfied as the next state is independent of all previous states and actions. Almost all reinforcement learning problem can be formalized as Markov decision processes, as long as the task can be defined such that it satisfies the Markov property.

Definition 1.2.3. *A Markov decision process is a 5-tuple,* $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ *where*

- $\mathcal{S}$ *is the state space,*
- $\mathcal{A}$ *is the action space,*
- P *is the state transition probability matrix which consists of* $\mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$*, the transition probability from state* s *to* s' *under action* a*,*
- R *is a reward function containing information on* $\mathbb{E}[R_{t+1} | S_t = s, A_t = a]$*, the expected reward for taking action* a *at state* s*,*
- γ *is a discount factor,* $\gamma \in [0, 1]$*.*

Note that by using the function p which describes the behavior of the environment, we are able to determine elements P and R. As for the discount factor γ, it is used in the value functions and will be discussed in the next section.

1.3 Bellman Equations

In this section, value functions would be introduced as they are useful in determining the best performing policy. We also outline how to obtain conditions for optimality by considering recursive conditions that must be satisfied by the value functions.

1.3.1 Returns

As mentioned, the agent's goal is to maximize the cumulative reward. More precisely, it is to maximize the expected return, with the return defined as follows.

Definition 1.3.1. *The discounted return G_t is the total discounted reward from time-step t:*

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{1.2}$$

where γ is the discount rate with $0 \leq \gamma \leq 1$ and R_t is the reward received at time t.

There are two types of tasks: episodic and continuing. If there is a terminal state for each episode, followed by a reset to a starting state, then they are episodic tasks. An example would be plays of a Go game. Continuing tasks have no terminal state and the interaction goes on continually. An example would be the routing and planning in robots with a long lifespan.

For episodic tasks which terminate at time T, Equation (1.2) can be rewritten such that the rewards after time T to ∞ are zero:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}. \tag{1.3}$$

Next, note that $0 \leq \gamma \leq 1$. For a finite T, $\gamma = 1$ yields a finite value for G_t. But if $T = \infty$, the expected return value would be infinite with a probability of one if the immediate reward values do not decay with time, e.g. there exist ϵ such that $0 < \epsilon < R_t$, making the goal of maximizing the value G_t mathematically ill-defined. For instance, when $T = \infty$, having $\gamma < 1$ will give the sum a finite value when the reward sequence $\{R_t\}$ is bounded.

1.3.2 Value Functions

Value function gives the long-term value of a state or a state-action pair. It is used to evaluate their goodness or badness since rewards that are likely to follow thereafter when following a particular policy is taken into account. There two types of value functions: the state-value function and the action-value function.

Definition 1.3.2. *The state-value function, $v_\pi(s)$ is the value of a state s under a policy π. This is the expected return when starting from s and following π thereafter:*

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]. \tag{1.4}$$

The action-value function, $q_\pi(s,a)$ is the value of taking action a in state s under policy π. This is the expected return when starting from s, taking action a, and following π thereafter:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]. \tag{1.5}$$

Note that the state-value function (1.4) can be decomposed in a recursive manner:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]. \end{aligned} \tag{1.6}$$

Similarly, for the action-value function, we have

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]. \tag{1.7}$$

1.3.3 Bellman Equations

Using their definitions, state-value functions can be defined in terms of action-value functions and vice versa. The backup diagrams shown in Figure 1.3.1 and Figure 1.3.2 illustrates the relationship between the value of a state and a state-action pair.

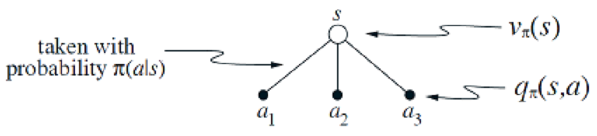


Figure 1.3.1: Backup diagram for v_π. (Figure from [5].)

Figure 1.3.1 shows that $v_\pi(s)$ depends on the values of the actions possible in that state and how likely each action is to be taken under the current policy π. The expected return is then an average over the values of the actions available at that state:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a). \tag{1.8}$$

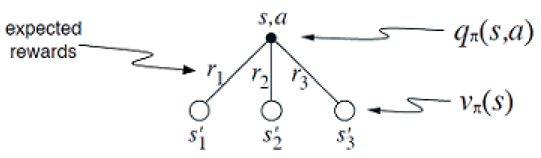


Figure 1.3.2: Backup diagram for q_π. (Figure from [5].)

As for $q_\pi(s,a)$, it depends on the expected next reward and the expected cumulative remaining rewards starting from the next state, as illustrated using Figure 1.3.2. Corresponding to this figure, the action-value function can be obtained by averaging over each possible situation after taking that action as follows:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]. \tag{1.9}$$

Substituting Equation (1.9) into (1.8) and (1.8) into (1.9) lead to recursive equations of v_π and q_π which are called the Bellman equations.

Definition 1.3.3. *The Bellman equation for v_π is*

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \tag{1.10}$$

and the Bellman equation for q_π is

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')]. \tag{1.11}$$

Next, the existence and uniqueness of a solution for the Bellman equations will be proven using the definitions and the contraction mapping theorem mentioned below.

Definition 1.3.4. *For any vector* $\mathbf{x} = [x_1 \cdots x_n]^T \in \mathbb{R}^n$*, the uniform norm is defined as* $||\mathbf{x}||_\infty = \max_i |x_i|$*. The distance between two vectors* $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ *is then given by* $||\mathbf{x} - \mathbf{y}||_\infty = \max_i |x_i - y_i|$.

Definition 1.3.5. *An operator* T *on a normed vector space* $\mathcal{X}$ *is a* γ*-contraction, for* $0 < \gamma < 1$*, provided that for all* $x, y \in \mathcal{X}$, $||T(x) - T(y)|| \leq \gamma ||x - y||$.

Theorem 1.3.1. ***(Contraction Mapping Theorem)*** *If* $T : \mathcal{X} \to \mathcal{X}$ *is a* γ*-contraction on a complete normed vector space* $\mathcal{X}$*, then* T *converges to a unique fixed point* $x \in \mathcal{X}$ *(i.e.* $T(x) = x$*) at a linear convergence rate* γ.

Proof. See [1, p. 121-122] □

Proposition 1.3.2. *There exists a unique solution for Equation (1.10) for all policies.*

Proof. The equations of $v_\pi(s)$ for all $s \in \mathcal{S}$ can be expressed concisely in matrix form:

$$\mathbf{v}_\pi = \mathbf{r}^\pi + \gamma P^\pi \mathbf{v}_\pi \tag{1.12}$$

where $\mathbf{v}_\pi \in \mathbb{R}^n, n = |\mathcal{S}|$ is a column vector with one entry per state and

$$\mathbf{v}_\pi = \begin{bmatrix} v_\pi(1) \\ \vdots \\ v_\pi(n) \end{bmatrix}, \mathbf{r}^\pi = \begin{bmatrix} r_1^\pi \\ \vdots \\ r_n^\pi \end{bmatrix}, P^\pi = \begin{bmatrix} P_{11}^\pi & \cdots & P_{1n}^\pi \\ \vdots & \ddots & \vdots \\ P_{n1}^\pi & \cdots & P_{nn}^\pi \end{bmatrix},$$

$$r_s^\pi = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r, \quad P_{ss'}^\pi = \sum_a \pi(a|s) \sum_r p(s', r|s, a).$$

First, we define the function $T^\pi : \mathbb{R}^n \to \mathbb{R}^n$ as $T^\pi(\mathbf{v}) = \mathbf{r}^\pi + \gamma P^\pi \mathbf{v}$. Then, for any $\mathbf{u} = [u(1) \cdots u(n)]^T$ and $\mathbf{v} = [v(1) \cdots v(n)]^T$,

$$\begin{aligned} ||T^\pi(\mathbf{u}) - T^\pi(\mathbf{v})||_\infty &= ||(\mathbf{r}^\pi + \gamma P^\pi \mathbf{u}) - (\mathbf{r}^\pi + \gamma P^\pi \mathbf{v})||_\infty \\ &= \gamma \max_s |\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(u(s') - v(s'))| \\ &\leq \gamma \max_s |\max_{s'}(u(s') - v(s'))| \\ &\leq \gamma \max_{s'} |u(s') - v(s')| = \gamma ||\mathbf{u} - \mathbf{v}||_\infty \end{aligned}$$

By Theorem 1.3.1, there exists a unique $\mathbf{v}_\pi$ that satisfies $T^\pi(\mathbf{v}_\pi) = \mathbf{v}_\pi$. The corresponding function v_π is then a unique solution satisfying Bellman equation (1.10). □

The existence and uniqueness of q_π can be proven similarly.

1.3.4 Bellman Optimality Equation

Now, the conditions for optimality for the Markov decision process will be derived. If $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$, it is said that policy π is better than or equal to policy π'. For any Markov decision process, there exists an optimal policy π_* which is better than or equal to all other policies. There may be one or more optimal policies which will all be denoted as π_* but they all have the same optimal state-value function $v_*(s)$ and optimal action-value function $q_*(s,a)$.

Definition 1.3.6. *The optimal state-value function is the maximum state-value function over all policies:*

$$v_*(s) = \max_\pi v_\pi(s), \qquad \forall s \in \mathcal{S}. \tag{1.13}$$

The optimal action-value function is the maximum action-value function over all policies:

$$q_*(s,a) = \max_\pi q_\pi(s,a), \qquad \forall s \in \mathcal{S}, a \in \mathcal{A}. \tag{1.14}$$

There is always a deterministic optimal policy for any Markov decision process. Suppose that q_* is known. For each state s, let $a_*(s) = \arg\max_{a \in \mathcal{A}} q_*(s,a)$ with ties broken arbitrarily. By choosing $a_*(s)$ at each state, we obtain an optimal policy:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = a_*(s), \\ 0 & \text{otherwise.} \end{cases} \tag{1.15}$$

The value of a state under policy π_* is the expected return for the best action of that state. By substituting the policy into Equation (1.8), the state-value function under policy π_* would be $v_{\pi_*}(s) = \max_{a \in \mathcal{A}} q_*(s,a)$. As will be proven in the proposition later, this state-value function satisfies the definition of optimal state-value function (1.13) and hence π_* is an optimal policy. In other words, for all state s,

$$v_*(s) = \max_a q_*(s,a). \tag{1.16}$$

Proposition 1.3.3. *$v_{\pi_*}(s) = \max_a q_*(s,a)$ for all states $s \in S$ satisfies the definition of optimal state-value function in (1.13).*

Proof. By the construction of π_* in (1.15), it lies in the space consisting all policies. Hence, the value each state under π_* is definitely smaller than or equal to the maximum value of that state over all policies, i.e. $v_{\pi_*}(s) \leq \max_\pi v_\pi(s) = v_*(s), \forall s \in \mathcal{S}$. Next, for all s, we also have

$$\begin{aligned} v_*(s) = \max_\pi v_\pi(s) &= \max_\pi \sum_a \pi(a|s) q_\pi(s,a) && \text{(by 1.8)} \\ &\leq \max_\pi \max_a q_\pi(s,a) \\ &= \max_a \max_\pi q_\pi(s,a) \\ &= \max_a q_*(s,a) = v_{\pi_*}(s). && \text{(by 1.14)} \end{aligned}$$

Since for all $s \in \mathcal{S}$, we have $v_{\pi_*}(s) \leq v_*(s)$ and $v_*(s) \leq v_{\pi_*}(s)$, then $v_*(s) = v_{\pi_*}(s)$. □

As for $q_*(s,a)$, it is the expected return for taking action a in state s and thereafter following an optimal policy. So, v_π in Equation (1.9) is replaced by v_*, which leads to

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')]. \tag{1.17}$$

Combining Equations (1.16) and (1.17) gives the Bellman optimality equations.

Definition 1.3.7. *The Bellman optimality equation for v_* is*

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')] \tag{1.18}$$

and the Bellman optimality equation for q_* is

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]. \tag{1.19}$$

It will be proven that there exists a unique solution which satisfy the Bellman optimality equation using the following proposition and the contraction mapping theorem.

Proposition 1.3.4. *Let $f, g : \mathcal{X} \to \mathbb{R}$ be two bounded functions with $\mathcal{X} \subseteq \mathbb{R}$. Then, $|\max_{x\in\mathcal{X}} f(x) - \max_{x\in\mathcal{X}} g(x)| \leq \max_{x\in\mathcal{X}} |f(x) - g(x)|$.*

Proof. Let $x_0 = \arg\max_x |f(x) - g(x)|$, $x_1 = \arg\max_x f(x)$ and $x_2 = \arg\max_x g(x)$, with any ties broken arbitrarily. Without loss of generality, assume that $f(x_1) \geq g(x_2)$. Since $|f(x_0) - g(x_0)| = \max_x |f(x) - g(x)|$, then $|f(x_1) - g(x_1)| \leq |f(x_0 - g(x_0))|$. Also, $g(x_1) \leq g(x_2)$ because $g(x_2) = \max_x g(x)$. Thus, $|f(x_1) - g(x_2)| \leq |f(x_0) - g(x_0)|$. □

Proposition 1.3.5. *There exists a unique solution v_* satisfying the Equation (1.18).*

Proof. First, we define that $\mathbf{x} \geq \mathbf{y}$ for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ if all elements of $\mathbf{x} - \mathbf{y}$ are non-negative. Next, Equation (1.18) can be written concisely in matrix form as

$$\mathbf{v}_* = \max_{\mathbf{a}}[\mathbf{r}(\mathbf{a}) + \gamma P(\mathbf{a})\mathbf{v}_*] \tag{1.20}$$

where $\mathbf{v}_* \in \mathbb{R}^n, n = |\mathcal{S}|$ is a column vector with one entry per state and

$$\mathbf{v}_* = \begin{bmatrix} v_*(1) \\ \vdots \\ v_*(n) \end{bmatrix}, \mathbf{a} = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}, \mathbf{r}(\mathbf{a}) = \begin{bmatrix} r_1(a_1) \\ \vdots \\ r_n(a_n) \end{bmatrix}, P(\mathbf{a}) = \begin{bmatrix} P_{11}(a_1) & \cdots & P_{1n}(a_1) \\ \vdots & \ddots & \vdots \\ P_{n1}(a_n) & \cdots & P_{nn}(a_n) \end{bmatrix},$$

$$r_s(a_s) = \sum_{s',r} p(s', r|s, a_s)r, \quad P_{ss'}(a_s) = \sum_{r} p(s', r|s, a_s), \quad a_s \in \mathcal{A}, s \in \mathcal{S}.$$

Let $T^* : \mathbb{R}^n \to \mathbb{R}^n$ be an operator such that $T^*(\mathbf{v}) = \max_{\mathbf{a}}[\mathbf{r}(\mathbf{a}) + \gamma P(\mathbf{a})\mathbf{v}]$. Then, for any $\mathbf{u} = [u(1) \cdots u(n)]^T$ and $\mathbf{v} = [v(1) \cdots v(n)]^T$,

$$\begin{aligned}
||T^*(\mathbf{u}) - T^*(\mathbf{v})||_\infty &= ||\max_{\mathbf{a}}[\mathbf{r}(\mathbf{a}) + \gamma P(\mathbf{a})\mathbf{u}] - \max_{\mathbf{a}}[\mathbf{r}(\mathbf{a}) + \gamma P(\mathbf{a})\mathbf{v}]||_\infty \\
&= \max_s |\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma u(s')] - \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]| \\
&\leq \max_s \max_a |\sum_{s',r} p(s', r|s, a)[r + \gamma u(s')] - \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]| \\
&\qquad \text{(by Proposition 1.3.4)} \\
&= \gamma \max_s \max_a |\sum_{s',r} p(s', r|s, a)(u(s') - v(s'))| \\
&\leq \gamma \max_s \max_a |\max_{s'}(u(s') - v(s'))| \\
&\leq \gamma \max_s |u(s) - v(s)| = \gamma||\mathbf{u} - \mathbf{v}||_\infty
\end{aligned}$$

Hence, T^* is a γ-contraction and there exists a unique v_* which satisfies the Bellman optimality equation (1.18). □

Similarly, it could be proven that there exists a unique solution q_* satisfying the Equation (1.19). With the existence and uniqueness of v_* and q_* proven, we could now proceed to discuss how to solve this system of equations to find v_* and q_*.

1.4 Summary

In this chapter, we introduced the reinforcement learning problem, which is for the agent to achieve the goal of maximizing the total reward by learning from interacting with the environment. The environment setup can be formulated as a Markov decision process. More specifically, our focus is on finite Markov decision processes.

Next, the value functions are presented. Given a policy, it is important to know the value of a state or state-action pair. With the value functions properly defined, the optimal value functions could be established. Among all the policies available, this would be the largest expected return achievable. The Bellman optimality equations are conditions that the optimal value functions need to satisfy. By solving the system of equations, the optimal value functions are found. From the solution, an optimal policy can then be determined.

In the next few chapters, methods on how to find the optimal state-value and action-value functions will be discussed.

Part I
Tabular Solution Methods

In this part of the report, some of the core ideas in reinforcement learning algorithms will be described. The focus would be on problems with state and action spaces that are small enough for value functions to be represented in a table. For small problems, these algorithms may be able to obtain the exact optimal value function and hence the optimal policy.

The reinforcement learning agents discussed in this part are classified into two groups: model-based and model-free. For model-based methods, a complete and accurate model of the environment is required whereas model-free methods do not. Each class of methods has its strengths and weakness which will be discussed further in the following chapters.

Chapter 2

Model-Based Methods

At the end of the first chapter, the optimal value functions, v_* and q_*, were constructed and it was proven that there exists a unique solution which satisfies the Bellman optimality equations. After obtaining the solution, the optimal policies can then be easily identified.

As mentioned earlier, the environment is assumed to be a finite Markov decision process, i.e. the state space $\mathcal{S}$ and action space $\mathcal{A}$ are finite and the dynamics are described by a set of probabilities $p(s', r|s, a)$ for all $s, s' \in \mathcal{S}$, $a \in \mathcal{A}$ and $r \in \mathbb{R}$. In this chapter, when showing how to solve the Bellman optimality equation, it will be assumed that the full information on the set of probabilities is provided. Methods which solve the equation with full use of this knowledge are classified as model-based methods. The focus of this chapter will be on dynamic programming which refers to a group of algorithms which solve complex problems by breaking them down into simpler subproblems in a recursive manner, solving them and combining their solutions. As introduced earlier, the Bellman equations give recursive decomposition, and hence dynamic programming methods are applicable in solving the equations.

2.1 Policy Iteration

One way to find an optimal policy of a reinforcement learning problem is to use policy iteration. Each iteration consists of two steps: policy evaluation and policy improvement:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_*,$$

where $\xrightarrow{E}$ and $\xrightarrow{I}$ represents policy evaluation and policy improvement respectively.

In policy evaluation, we want to find the state-value function v_π of a given policy π. Using the estimated v_π, a better performing policy π' is generated under the policy improvement step. By repeating these two steps, we will be able to obtain monotonically improving policies and value functions. It will be proven later that each policy is strictly better than the previous one if it is not an optimal policy and the sequence will eventually converge to an optimal policy.

2.1.1 Policy Evaluation

We first discuss on how to obtain the state-value function $v_\pi(s)$ for all $s \in \mathcal{S}$ given a policy π. Recall the definition of Bellman state-value function where we have

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]. \tag{from 1.10}$$

Observe that this is a system of $|\mathcal{S}|$ simultaneous linear equations with $|\mathcal{S}|$ unknowns. Technically, the solution can be computed in a straightforward method since a unique solution exists. In fact, the solution can be written concisely in matrix form:

$$\mathbf{v}_\pi = (I - \gamma P^\pi)^{-1}\mathbf{r}^\pi \tag{2.1}$$

where all the elements are the same as defined in (1.12). However, this would be a tedious computation especially when the state space is big. Instead, we will use iterative solution methods to solve the equations. Suppose $V_0(s)$ for all $s \in \mathcal{S}$ is an arbitrarily chosen value. By having Bellman equation (1.10) as an update rule, the successive approximation of the value functions V_k for $k = 1, 2, ...$ can be obtained:

$$V_k(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V_{k-1}(s')], \quad \forall s \in \mathcal{S}. \tag{2.2}$$

This algorithm is called iterative policy evaluation. As proven in Proposition 1.3.2 of the previous chapter, the sequence $\{V_k\}$ will converge to v_π as $k \to \infty$ by the contraction mapping theorem. Using the following theorem, it can be proven that once the change in the approximation is sufficiently small, the new approximated value must be close to the actual solution.

Theorem 2.1.1. *Given a sequence $V_0, V_1, ...$ such that each successive V_k is obtained using the update rule (2.2), let $\mathbf{V}_k \in \mathbb{R}^n$, $n = |\mathcal{S}|$ be a column vector with one entry per state such that $\mathbf{V}_k = [V_k(1) \cdots V_k(n)]^T$ for all k. Suppose that $||\mathbf{V}_{k+1} - \mathbf{V}_k||_\infty < \epsilon$ for some $\epsilon \in \mathbb{R}$. Then $||\mathbf{V}_{k+1} - \mathbf{v}_\pi||_\infty < \frac{\epsilon\gamma}{1-\gamma}$, where $\mathbf{v}_\pi = [v_\pi(1) \cdots v_\pi(n)]^T$.*

Proof. Let T be an operator such that $T^\pi(\mathbf{v}) = \mathbf{r}^\pi + \gamma P^\pi \mathbf{v}$ as defined in (1.12) and $\mathbf{v}_\pi$ to be a fixed point solution to the equation. Then, by the update rule,

$$\begin{aligned} ||\mathbf{V}_{k+1} - \mathbf{v}_\pi||_\infty &= ||T^\pi(\mathbf{V}_k) - T^\pi(\mathbf{v}_\pi)||_\infty \\ &\leq \gamma ||\mathbf{V}_k - \mathbf{v}_\pi||_\infty \qquad \text{(by proof of Proposition 1.3.2)} \\ &\leq \gamma(||\mathbf{V}_k - \mathbf{V}_{k+1}||_\infty + ||\mathbf{V}_{k+1} - \mathbf{v}_\pi||_\infty). \end{aligned}$$

Consequently, we have

$$||\mathbf{V}_{k+1} - \mathbf{v}_\pi||_\infty \leq \frac{\gamma}{1-\gamma} ||\mathbf{V}_{k+1} - \mathbf{V}_k||_\infty \leq \frac{\epsilon\gamma}{1-\gamma}.$$

□

With this theorem, we are able to terminate the iterative process once the update is sufficiently small and use the latest update as an approximate of v_π.

An observation of this method is that it uses synchronous backups, that is the old values will be updated simultaneously after all the new values are determined. An alternative way is to implement asynchronous updates. In other words, the old value will be overwritten immediately once the new value of a state is computed. This reduces the memory space required since only one copy of value function needs to be stored instead of two. This in-place update method also converges to v_π.

2.1.2 Policy Improvement

The goal of evaluating the value function of a policy π is to generate a better performing policy π'. With the information on the state value function of some deterministic policy π, we now know how good it is to follow this policy at a state s. In this section, we are want to find out if it would be better to change the policy such that for some state s

we choose to take action $a \neq \pi(s)$. The action-value function $q_\pi(s,a)$ would be useful as it evaluates the value of taking action a at state s and follow the policy π thereafter. Generally, when $q_\pi(s,a) > q_\pi(s,\pi(s)) = v_\pi(s)$, rather than following the policy π all the time, the performance would improve if action a is taken at state s and policy π is followed thereafter. This is proven in the policy improvement theorem.

Theorem 2.1.2. ***(Policy Improvement Theorem)*** *If there is a deterministic policy π' such that for all $s \in \mathcal{S}$,*

$$q_\pi(s,\pi'(s)) \geq v_\pi(s), \tag{2.3}$$

then $v_{\pi'}(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$.

Proof.

$$\begin{aligned} v_\pi(s) \leq q_\pi(s,\pi'(s)) &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2})) | S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... | S_t = s] \\ &= v_{\pi'}(s) \end{aligned}$$

□

Moreover, if the strict inequality of (2.3) holds for some state s, then $v_{\pi'}(s) > v_\pi(s)$ holds for at least one state, i.e. policy π' is strictly better than π. Hence, policy improvement could be easily achieved by acting greedily such that for all states $s \in \mathcal{S}$,

$$\pi'(s) = \arg\max_a q_\pi(s,a) = \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \tag{2.4}$$

where ties for any argmax are broken arbitrarily. Suppose that the improvement stops, i.e. $q_\pi(s,\pi'(s)) = \max_a q_\pi(s,a) = q_\pi(s,\pi(s)) = v_\pi(s)$ for all s. Then, Equation (1.16) is satisfied and hence the Bellman optimality equations hold. Thus, v_π must be v_* and both π and π' must be optimal policies. In short, when a policy is not optimal, the new policy generated must be strictly better and by alternating the policy evaluation and policy improvement steps, we will eventually converge to an optimal policy π_*.

2.2 Generalized Policy Iteration

In the policy iteration method described earlier, each iteration involves policy evaluation, which may require multiple sweeps through the state space before the sequence of approximated value function converges to the actual value function of the current policy. Previously, the stopping condition of the policy evaluation step is when the difference between the last two approximated value of all states are sufficiently small. Actually, the process could be stopped earlier before this is achieved. For instance, we could simply stop after a fixed number of iterations of the iterative policy evaluation. Convergence to the optimal value function and optimal policy will still be the achieved at the end of the policy iteration process as long as all states are updated consistently.

So, using the idea of policy iteration, a generalized version is introduced. Generalized policy iteration refers to having policy evaluation and policy improvement steps interacting with each other without concerning the details of the two processes. In any of the versions, once both the processes no longer produce any changes, the value function and policy must be optimal. This is because in policy evaluation, only when the value function is consistent with the current policy did the value function stabilize whereas in policy improvement, the policy will not stabilize as long as it is still not greedy with respect to the current value function. Hence, the stabilization of both processes occurs only when a policy that has been found is greedy with respect to its own evaluated value function. In this case, the Bellman optimality equation is satisfied, and thus we have an optimal policy and the optimal value function.

A special case of generalized policy iteration is the value iteration, where policy evaluation is stopped after just one update of each state. The two-step iteration can be written concisely using the following update rule:

$$V_k(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V_{k-1}(s')] \tag{2.5}$$

for all $s \in \mathcal{S}$. In fact, this update rule follows the Bellman optimality equation (1.18). For arbitrary V_0, the sequence $\{V_k\}$ will converge to v_* as shown in Proposition 1.3.5.

2.3 Summary

As a summary, we have introduced algorithms of dynamic programming to solve finite Markov decision processes. More specifically, they are viewed as generalized policy iteration, in which the policy evaluation and policy improvement processes interact with each other. The updates of the value of each state are closely based on the Bellman equations introduced in Chapter 1 and the end result of these algorithms are the optimal value functions and an optimal policy.

An important point to note is that the complete knowledge of the Markov decision process is known throughout the computation process of these algorithms. This would be impractical in real-life problems as we do not know the model in most situations. Hence, model-free methods will be introduced in the next chapter.

Chapter 3

Model-Free Methods

In the previous chapter, it was assumed that we have complete knowledge of the environment. From this chapter onwards, this assumption will no longer hold true and the focus will only be on model-free methods. These methods learn directly from episodes of experience, which are sample sequences of states, actions and rewards resulting from interaction with the environment. On the other hand, the idea of generalized policy iteration developed previously will still be adapted, that is at each iteration, we first estimate the value functions and later implement the policy improvement step.

To be able to suggest a better policy, the current value of each action is required. However, since the model of the environment is not available for us, state values are no longer sufficient. So, in this chapter, q_π will be estimated instead of v_π in the policy evaluation step and information on q_π will be used to generate a new policy in the policy improvement step. Different methods of learning the value functions without a model of the environment's dynamics will be introduced here.

3.1 Monte Carlo

The first method considered is Monte Carlo, which solves reinforcement learning problems based on averaging sample returns. Recall that in Chapter 1, we define return as the total discounted reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \qquad \text{(from 1.2)}$$

To calculate the actual return of an episode, the episode must terminate. In other words, Monte Carlo methods are only applicable to episodic Markov decision processes.

3.1.1 Policy Evaluation

The first question to ask would be: with episodes of experience under a policy, how do we learn the action-value function? Recall that action-value function is defined as:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]. \tag{from 1.5}$$

Intuitively, the expected value of a random variable is the long-run average value of repetitions of the experiment it represents. Hence, an estimate of the expected return starting from state s and taking action a could be obtained by using the empirical mean of the returns observed after visits to the state-action pair. With the set of episodes obtained by following a policy π, the return following a visit to a state-action pair can be easily calculated. Two ways to obtain the average of these returns are the first-visit and every-visit Monte Carlo methods. The former estimates $q_\pi(s,a)$ by averaging the returns following the first time that action a was selected at state s while the latter averages the returns following all visits to this pair.

Here, the focus will be on the first-visit method. For a policy π, the estimated value will converge to $q_\pi(s,a)$ as the number of first visits to the state-action pair goes to infinity. This is because each return is an independent, identically distributed estimate of $q_\pi(s,a)$ and the average would converge to the expected value by the law of large numbers. However, a common problem is that many state-action pairs might not be visited. With no returns to average, we are unable to estimate the value of all the actions from each state, causing the alternative actions to be incomparable. Hence, for the policy evaluation to work for action values, we must ensure continual exploration, which will be discussed further in the later section. For now, we retain the assumption that all state-action pairs are encountered infinitely often with a probability of one.

Next, we look at how to compute these averages in a computationally efficient manner. The usual way is to maintain a record of all the returns for each state-action pair and compute the summation each time a new return value is acquired. But, as the number of visits increases, the memory and computational requirements would grow. A better way would be to update the averages using the incremental formula below.

Proposition 3.1.1. *Suppose we have a sequence of values $x_1, x_2, ..., x_n$ such that $x_i \in \mathbb{R}$. Let μ_k denote the average of $x_1, x_2, ..., x_k$ for $k = 1, 2, ..., n$ and $\mu_0 = 0$. Then,*

$$\mu_k = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}), \qquad k = 1, 2, ...n. \tag{3.1}$$

Proof. With $\mu_0 = 0$, we have for any $k = 1, 2, ..., n$,

$$\begin{aligned}\mu_k &= \frac{1}{k}(x_k + \sum_{j=1}^{k-1} x_j) \\ &= \frac{1}{k}(x_k + (k-1)\mu_{k-1}) \\ &= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})\end{aligned}$$

□

The average of the returns can now be updated incrementally after each episode. For each pair of s,a appearing in an episode, the update rule can be written as

$$Q_k(s,a) = Q_{k-1}(s,a) + \frac{1}{k}(\tilde{G}_k - Q_{k-1}(s,a)). \tag{3.2}$$

where $Q_k(s,a)$ denotes the current estimated value of the state-action pair s and a after k updates with $Q_0(s,a) = 0$ and $\tilde{G}_k$ is the return value following the first occurrence of s,a in this episode. This could be generalized into

$$Q_k(s,a) = Q_{k-1}(s,a) + \alpha_k(\tilde{G}_k - Q_{k-1}(s,a)). \tag{3.3}$$

To assure convergence with the probability of one, the following need to be satisfied:

$$\sum_{k=1}^{\infty} \alpha_k = \infty \qquad \text{and} \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty \tag{3.4}$$

Suppose that a constant step-size, $\alpha_k = \alpha$ for all k, is used. This results in the action value being a weighted average. More weight will be given to recent returns than to that from long-past episodes as shown below:

$$\begin{aligned}Q_k &= Q_{k-1} + \alpha[\tilde{G}_k - Q_{k-1}] \\ &= \alpha\tilde{G}_k + (1-\alpha)Q_{k-1} \\ &= \alpha\tilde{G}_k + (1-\alpha)\alpha\tilde{G}_{k-1} + (1-\alpha)^2\alpha\tilde{G}_{k-2} + ... + (1-\alpha)^{k-1}\alpha\tilde{G}_1 + (1-\alpha)^k Q_0 \\ &= (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i}\tilde{G}_i.\end{aligned}$$

The second condition in (3.4) is not met here, indicating that the estimates are not guaranteed to converge. But, the weighted average is desirable as problems in reinforcement learning are commonly non-stationary, that is the true values of the actions will change over time. This is because the agent's policy changes as learning proceeds.

3.1.2 Policy Improvement

Now, let's consider how the estimated value function can be used in policy improvement. The overall idea is similar as introduced in the preceding chapter, that is policy improvement is done by making the policy greedy with respect to the current action-value function. Information regarding the model is no longer needed to construct the greedy policy since the action-value function is available. For each state s, action with maximal action-value is chosen such that

$$\pi'(s) = \arg\max_a q_\pi(s,a) \tag{3.5}$$

with ties broken arbitrarily. Similarly, by the policy improvement theorem, the way that policy π' is constructed assures that it performs better than or as good as the previous policy π and the overall process will converge to an optimal policy.

Here, we address the issue of ensuring continual exploration. If the policy improvement step is implemented each time by constructing a greedy policy as defined above, it is not ensured that all state-action pairs are visited infinitely often. A new policy constructed such that the agent continues to select all actions available with non-zero probability would help solve this problem. In other words, $\pi(a|s) > 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. In particular, we use the ϵ-greedy policies, where the new policy is defined as follows:

$$\pi'(a|s) = \begin{cases} 1-\epsilon+\frac{\epsilon}{|\mathcal{A}|} & \text{if } a = \arg\max_{a'\in\mathcal{A}} q_\pi(s,a') \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise,} \end{cases} \tag{3.6}$$

where ties for the argmax function are broken arbitrarily. We will show that π' performs better than or equal to the previous ϵ-greedy policy π.

Proposition 3.1.2. *For any ϵ-greedy policy π, suppose that ϵ-greedy policy π' is defined as shown in (3.6). Then, we have $\pi' \geq \pi$.*

Proof. For any state $s \in \mathcal{S}$,

$$\begin{aligned}
q_\pi(s, \pi'(s)) &= \sum_a \pi'(a|s) q_\pi(s,a) \\
&= \frac{\epsilon}{|\mathcal{A}|} \sum_a q_\pi(s,a) + (1-\epsilon) \max_a q_\pi(s,a) \\
&\geq \frac{\epsilon}{|\mathcal{A}|} \sum_a q_\pi(s,a) + (1-\epsilon) \sum_a \frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}|}}{1-\epsilon} q_\pi(s,a) \\
&\qquad \left(\because \textstyle\sum_a \frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}|}}{1-\epsilon} = 1 \text{ and } \frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}|}}{1-\epsilon} \geq 0, \forall a \in \mathcal{A}\right) \\
&= \frac{\epsilon}{|\mathcal{A}|} \sum_a q_\pi(s,a) - \frac{\epsilon}{|\mathcal{A}|} \sum_a q_\pi(s,a) + \sum_a \pi(a|s) q_\pi(s,a) \\
&= v_\pi(s)
\end{aligned}$$

Since $q_\pi(s, \pi'(s)) \geq v_\pi(s)$, by the policy improvement theorem, we have $v_{\pi'}(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$. Hence, policy π' is better than or equal to π. □

By this proposition, policy iteration still works for ϵ-greedy policies such that it will eventually converge to the best policy among the ϵ-greedy policies.

3.1.3 Generalized Policy Iteration

We now proceed according to the generalized policy iteration introduced in the previous chapter, which is to alternate between policy evaluation and policy improvement.

Previously, to solve the exploration issue, ϵ-greedy policy is introduced. However, the iteration is only guaranteed to converge to the best policy among the ϵ-greedy policies instead of an optimal policy. It is extremely unlikely for policies that include random behavior to be optimal. So, how do we eventually get to a best-performing policy which does not explore at all?

One idea is to use Greedy in the Limit with Infinite Exploration, where the following two conditions need to be met. First is that all state-action pairs are explored infinitely many times – a condition that ϵ-greedy policies satisfy. Secondly, the policy needs to converge on a greedy policy. This is to satisfy the Bellman optimality equation which requires the maximum over all the actions. One way to achieve this is to remain using the ϵ-greedy policy but with ϵ decaying to zero. For instance, at the kth iteration, we let $\epsilon = \frac{1}{k}$. In this way, the iteration does converge to the optimal action-value function and an optimal policy.

3.2 State-Action-Reward-State-Action (SARSA)

Similar to Monte Carlo methods, SARSA uses experience to solve the reinforcement learning problem. Given some experience following a policy π, it updates the estimate Q of q_π for the states occurring in that experience. In this section, the focus will be on the policy evaluation step because the remaining steps can proceed in the same manner as described in the Monte Carlo section.

Recall that the update rule for the first-visit Monte Carlo method is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_t - Q(S_t, A_t)], \tag{3.7}$$

where S_t, A_t is the state-action pair appearing in an episode with its first occurrence at time t, G_t is the actual return following time t, and α is a constant step-size parameter. In Monte Carlo methods, the episode must terminate first to be able to determine the increment to $Q(S_t, A_t)$. On the contrary, SARSA needs to wait only until the next time step. At time $t+1$, they immediately make a useful update using the observed reward R_{t+1} and the estimate $Q(S_{t+1}, A_{t+1})$. The update is made as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \tag{3.8}$$

We could view these update such that in Monte Carlo, $Q(S_t, A_t)$ is updated towards the actual return G_t while in SARSA, the value is updated towards an estimated return $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$. This is consistent with the idea that the action-value function can be defined in two ways as follows:

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

As stated in [5], this method has been proven to converge to q_π for any fixed policy π, in the mean for a constant α if it is sufficiently small, and with probability one if the step-size parameter satisfies the conditions in (3.4). With an estimate of q_π, we can continue to implement the policy improvement step. Using the same Greedy in the Limit with Infinite Exploration idea from the previous section, the generalized policy iteration will converge to an optimal policy and action-value function.

3.3 Q-Learning

Another model-free method commonly used in reinforcement learning is Q-learning, defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]. \tag{3.9}$$

Similar to SARSA, the update can be done at every time step but with $Q(S_t, A_t)$ updated towards $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$. In this method, the learned action-value function Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. An improved policy is still generated in each iteration but it is only used to determine which state-action pairs are visited and updated. It is still important that the policy is constructed such that all pairs continue to be updated, and hence ϵ-greedy policies can still be used. Again, it is indicated in [5] that this algorithm is also shown to converge to the optimal action-value function q_*.

3.4 Comparison of the Methods

These three methods have their respective advantages and disadvantages. One obvious advantage SARSA and Q-learning has over Monte Carlo methods is that they are naturally implemented in an online fashion. With Monte Carlo methods, we must wait until the end of an episode because only then is the return known. On the other hand, with SARSA and Q-learning, we only need to wait only one time step. This is important because some application have very long episodes and delaying all learning until the end of the episode would be too slow. There are also applications which are continuing tasks and have no episodes at all.

Another difference in these methods is that Monte Carlo methods have high variance and zero bias while SARSA and Q-learning have low variance but some bias. Both the return G_t and $R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})$ are unbiased estimates of $q_\pi(S_t, A_t)$. This is because the former is just a sample of the expectation while the latter is followed from the Bellman equation. However, $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ is a biased estimate of $q_\pi(S_t, A_t)$ as it depends on the current Q. Hence, Monte Carlo which uses returns to update the value function has zero bias while SARSA and Q-learning would have some bias. However, the way return G_t is calculated causes it to be dependent on many random actions, transitions and rewards while $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ depends on only one random action, transition and reward. Therefore, the return would have a much higher variance, causing Monte Carlo methods to have a larger variance.

3.5 Summary

In this chapter, we introduced a few model-free methods. Complete knowledge of the Markov decision process is no longer required as these algorithms learn directly through the agent's experience. On the other hand, these methods still use the idea of generalized policy iteration that was introduced in Chapter 2 to find the optimal values and an optimal policy. In general, the first step is to estimate the value function of the current policy from episodes of experience and followed by improving the policy with respect to the current value function. An additional idea is to use ϵ-greedy policy instead of greedy policy during policy improvement in order to maintain sufficient exploration. These two steps are then repeated until convergence.

These methods have their own pros and cons. However, it is important to consider whether they are still feasible when the state or action space is very huge. Would it still be possible to find the value of each state or state-action pair using these methods? In the next part, we will be dealing with this issue.

Part II
Approximate Solution Methods

In the previous part, we can see that the value function is represented by a lookup table. In other words, for a policy π, every state s has an entry $v_\pi(s)$ or every state-action pair s and a has an entry $q_\pi(s, a)$. However, it would be impractical to use such methods when solving problems which have enormous or continuous state space. The problem with such large problems is that it takes up a lot of memory space and a huge amount of time to learn the value of each state or state-action pair individually.

This part aims to solve this issue which is commonly encountered in real-life problems. One way would be to estimate the value functions with some good approximate solution using the limited computational resources available. In many cases, we are unable to encounter all the existing states. Hence, it is necessary to generalize from seen states to unseen states. This kind of generalization is known as function approximation. On the other hand, we could directly approximate the optimal policy instead of the value functions as what has been introduced in the previous chapters. Algorithms with this approach are called policy gradient methods.

Chapter 4

Function Approximation Methods

In this chapter, the goal is to find a representation of the approximate value function in the form of a parameterized functional form with weight vector $\boldsymbol{w} \in \mathbb{R}^d$ instead of a lookup table. Given weight vector $\boldsymbol{w}$, the estimated value of state s is $\hat{v}(s, \boldsymbol{w})$. Similarly, the approximate value of a state-action pair is given by $\hat{q}(s, a, \boldsymbol{w})$. We no longer need to update the values of each state or state-action pair individually. Instead, only the parameter $\boldsymbol{w}$ which best approximates the value function needs to be learned. This is useful especially for tasks with huge or continuous state space as it is impossible to explore all existing states and store all the value functions in memory.

There are several types of function approximators. An example would be $\hat{v}$ are linear combinations of features of the state with the weight vector. In general, $\hat{v}$ could be a function computed by a multi-layer neural network with $\boldsymbol{w}$ as the weights in all the layers. As a result, a change in the weight parameter will affect the estimated value of several states. The methods introduced in this chapter are used to learn the parameter $\boldsymbol{w}$ so that $\hat{v}$ and $\hat{q}$ best estimates the value functions.

4.1 Stochastic Gradient & Semi-gradient Methods

We first develop one class of learning methods for function approximation in policy evaluation, which is based on stochastic gradient descent. Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient of the function at the current point. Let $J(\boldsymbol{w})$ be a differentiable function of real valued parameter vector $\boldsymbol{w} = [w_1\, w_2\, \cdots\, w_d]^T$. We define the gradient of $J(\boldsymbol{w})$ to be

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial J(\boldsymbol{w})}{\partial w_1} \\ \vdots \\ \frac{\partial J(\boldsymbol{w})}{\partial w_d} \end{bmatrix}. \tag{4.1}$$

To find a local minimum of $J(\boldsymbol{w})$, we adjust the weight vector by an amount in the direction of negative gradient:

$$\Delta \boldsymbol{w} = -\frac{1}{2}\alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w}) \tag{4.2}$$

where α is a step-size parameter.

Our goal is to find parameter vector $\boldsymbol{w}$ which minimizes the mean-squared error between approximate value function $\hat{v}(s, \boldsymbol{w})$ and the true value function $v_\pi(s)$. In the previous chapters, the learned values at each state were decoupled, which means that an update at one state affects no other. But with approximation methods, an update at one state affects many others and it is not possible to get the values of all states exactly correct. Hence, we need to specify which states we care most about. In other words, we need a distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$, representing how much we care about the error in each state s.

We look deeper into the distribution μ. Often, $\mu(s)$ is chosen to be the fraction of time spent in s. In continuing tasks, this is the stationary distribution under a policy π. On the other hand, in episodic tasks, the distribution depends on the initial state of episodes. Let $h(s)$ be the probability that an episode starts with state s and $\eta(s)$ be the number of time steps spent, on average, in state s in a single episode. For any state s, time is spent in s if episodes start in s or if transitions are made into s. Hence, for all states s, $\eta(s)$ and $\mu(s)$ can be written as:

$$\eta(s) = h(s) + \gamma \sum_{s'} \eta(s') \sum_a \pi(a|s') \sum_r p(s, r|s', a), \tag{4.3}$$

$$\mu(s) = \frac{\eta(s)}{\sum_s \eta(s)}. \tag{4.4}$$

where $0 \leq \gamma \leq 1$ is the discount factor.

By weighting the square difference between the approximate value and true value of each state by μ, we obtain an objective function:

$$\begin{aligned} J(\boldsymbol{w}) &= \mathbb{E}_\pi[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w})]^2 \\ &= \sum_s \mu(s)[v_\pi(s) - \hat{v}(s, \boldsymbol{w})]^2. \end{aligned} \tag{4.5}$$

Gradient descent then finds the local minimum by updating the vector $\boldsymbol{w}$ such that

$$\Delta \boldsymbol{w} = \alpha \mathbb{E}_\pi[(v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{v}(S_t, \boldsymbol{w})]$$

Gradient descent methods are called stochastic when the update is done, on only a single example instead on a large dataset. We denote $\boldsymbol{w}_t$ as the weight vector at each time t. Stochastic gradient descent then adjusts the weight vector after each example:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)]\nabla_{\boldsymbol{w}} \hat{v}(S_t, \boldsymbol{w}_t) \tag{4.6}$$

Over many samples, the overall effect minimizes the average performance measure (4.5).

Up til now, we have assumed that we are given the true value function $v_\pi(s)$. Unfortunately, there is no supervisor in reinforcement learning which provides this information. Hence, in practice, we substitute it with a target output $U_t \in \mathbb{R}$ of the t-th training example which is an approximation to $v_\pi(S_t)$. This yields the following stochastic gradient descent method for state-value approximation:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[U_t - \hat{v}(S_t, \boldsymbol{w}_t)]\nabla_{\boldsymbol{w}} \hat{v}(S_t, \boldsymbol{w}_t) \tag{4.7}$$

If we let U_t be the return G_t, which is an unbiased estimate of $v_\pi(S_t)$, then this method converges to a locally optimal approximation to $v_\pi(S_t)$. We call this algorithm Monte-Carlo with value function approximation. Similarly for SARSA, we could also have a gradient descent version of it by letting U_t to be $R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w}_t)$. This target depends on the current value of the weight vector $\boldsymbol{w}_t$ and hence, will be biased. Also, it is no longer a true gradient method but a semi-gradient method. However, this method typically enables significantly faster learning. Also, it enables learning to be continual and online, which in turns enables it to be used for continuing problems and provides computational advantages.

4.2 Function Approximation

Now, we are ready to introduce some of the methods for function approximation. Here, we will focus on two types of methods, which are linear methods and non-linear methods using neural networks. Both use the idea of gradient descent updates.

4.2.1 Linear Function Approximation

One way to approximate the value function $\hat{v}$ is by a linear combination of features:

$$\hat{v}(s, \boldsymbol{w}) = \boldsymbol{x}(s)^T \boldsymbol{w} = \sum_{j=1}^{d} x_j(s) w_j \tag{4.8}$$

where $\boldsymbol{w}$ is the weight vector and $\boldsymbol{x}(s)$ is the real-valued vector representing state s, which is known as feature vector. For every state s, we could have a vector of features $\boldsymbol{x}(s) = [x_1(s)\ x_2(s)\ \cdots\ x_d(s)]^T$ where $x_i : \mathcal{S} \to \mathbb{R}$ for all i corresponds to a specific aspect of the state. For example, the vector representing the state of a walking humanoid robot could consist of features for its location, its remaining battery power, the nearest obstacle and so on. Feature vectors are important as they can provide prior domain knowledge regarding the task at hand to the system. Furthermore, through feature construction, the dimensionality of the feature space could be significantly smaller than that of the state space. For $\hat{v}$ to be a good approximation of the actual value function using a linear combination of features, we need to have a good feature representation to provide sufficient information in determining the value functions.

The gradient of the approximate value function with respect to $\boldsymbol{w}$ in this case is

$$\nabla_{\boldsymbol{w}} \hat{v}(s, \boldsymbol{w}) = \boldsymbol{x}(s).$$

Thus, in the linear case, the stochastic gradient descent update (4.7) reduces to:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[U_t - \hat{v}(S_t, \boldsymbol{w}_t)]\boldsymbol{x}(S_t).$$

There is only one optimum or one set of equally good optima in the linear case, and thus it is guaranteed to converge to or near the global optimum as long as the method used is guaranteed to converge to or near a local optimum of the objective function.

4.2.2 Non-linear Function Approximation: Neural Network

Neural networks are widely used for non-linear function approximation. For a policy π, network defines a mapping $\hat{v}(S_t, \boldsymbol{w})$ and learns the value of the parameters $\boldsymbol{w}$ that result in the best function approximation for $v_\pi(S_t)$. It is called a network because they are typically represented by composing together many different functions such as $\hat{v}(S_t) = f^{(m)}(...(f^{(2)}(f^{(1)}(S_t))))$ where $f^{(i)}$ is a function known as the i-th layer of the network. By having non-linear functions $f^{(i)}$ for some i, $\hat{v}$ is now a non-linear function of the states. This is useful because most of the times, v_π is not linear with respect to its variable. Using hidden layers in the neural network is also a way to automatically create features appropriate for a given problem instead of relying exclusively on hand-crafted features as shown in the linear methods.

To train a neural network, it requires input-output pairs. In our case, the input would be the state S_t and the output would be the target output U_t since we do not have the actual value $v_\pi(S_t)$ as mentioned in the previous section. Similar to gradient descent method, the error $U_t - \hat{v}(S_t, \boldsymbol{w})$ for each sample is used as a guide to adjust the weights in each layer in order to minimize the error. A common way to determine the update of each connection weight based is to use the back-propagation algorithm. For more information regarding this algorithm, please refer to Section 6.5 of [3].

4.3 Function Approximation of Action Values

The ideas described in the previous sections can be easily extended to from state values to action values. We have $\hat{q}(s, a, \boldsymbol{w}) \approx q_\pi(s, a)$, which approximates the value of a state-action pair. The way to learn the parameter $\boldsymbol{w}$ remains the same. The difference is that the objective function now becomes

$$J(\boldsymbol{w}) = \mathbb{E}_\pi[q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, \boldsymbol{w})]^2. \tag{4.9}$$

Estimating the value of each action would be more useful since the model is not known to us and we need to compare which action is better to take at a state. With the estimated action-value function, we can easily conduct the generalized policy iteration as shown in the previous two chapters in order to find a close to optimal policy with its estimated value function. Again, we use ϵ-greedy policies for the action selection after each approximation of q_π with the conditions of Greedy in the Limit with Infinite Exploration being satisfied.

Note that the resulting policy is close to optimal instead of optimal. This is because the space consisting all the possible functions $\hat{q}$, which depends on the configuration of the parameter $\boldsymbol{w}$, might not include the actual optimal action-value function q_*. The resulting action value function from the iteration can only be a close approximate of q_*, and hence may not be able to lead us to an optimal policy.

4.4 Summary

To summarize, we introduced function approximation methods which enable the reinforcement learning system to generalize. This is because for huge problems, most of the state-action pair are not visited in the episodes of experience and it is unreasonable to store the value of each pair individually. Instead, we approximate v_π and q_π by $\hat{v}$ and $\hat{q}$ respectively. These functions are parameterized by a weight vector $\boldsymbol{w}$. To find a weight vector which is able to best approximate the actual value function, we use variations of stochastic gradient descent method. The update targets used in the learning algorithm are adopted from the methods in Chapter 4 such as Monte Carlo and SARSA. Lastly, we introduced linear and non-linear function approximation. The former approximates the value function by a linear combination of features while the latter uses neural network to represent the function.

In short, we learned to evaluate the value function of a policy for tasks with large state or action space so that the generalized policy iteration idea can be used to find the optimal value function. In the next chapter, we will introduce a new approach to solve a reinforcement learning problem, that is to search for an optimal policy directly.

Chapter 5

Policy Gradient Methods

In the previous chapters, all methods described are to learn the values of every state-action pair. The actions are then selected solely based on the estimated action-values using ϵ-greedy methods. However, in some cases, it is not necessary to learn the value functions of each state and action as this information is not useful for us. Instead, a parameterized policy could be learned directly such that actions are selected without referring to the value functions. The policy is now expressed as $\pi_{\boldsymbol{\theta}}(s,a) = \pi(a|s,\boldsymbol{\theta})$, where $\boldsymbol{\theta} \in \mathbb{R}^d$ is the policy's parameter vector. The policy can be parametrized in any way as long as $\nabla_{\boldsymbol{\theta}}\pi(s,a|\boldsymbol{\theta})$ exist, i.e. $\pi_{\boldsymbol{\theta}}$ is differentiable with respect to its parameter. The aim of the policy gradient methods in this chapter is to learn the best $\boldsymbol{\theta}$ which gives an optimal policy. First, the objective function is set and $\boldsymbol{\theta}$ will be learned using the stochastic gradient method.

5.1 Policy Objective Functions

As mentioned, our goal is to find the best $\boldsymbol{\theta}$. However, to find such $\boldsymbol{\theta}$, we first need to have a way to measure the performance of a policy $\pi_{\boldsymbol{\theta}}$. There are several ways to define the performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter.

For episodic cases, the value of the start state s_0 of the episode could be used:

$$J_1(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0). \tag{5.1}$$

As for continuing environments, $J(\boldsymbol{\theta})$ needs to be defined differently since there is no episodic boundaries. The performance could be defined in terms of the average rate of reward per time step instead. The average reward used is the expected reward achieved when the system reaches steady-state:

$$\begin{aligned} J_{avR}(\boldsymbol{\theta}) = r(\pi_{\boldsymbol{\theta}}) &= \lim_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} \mathbb{E}[R_t | A_{0:t-1} \sim \pi_{\boldsymbol{\theta}}] \\ &= \lim_{t\to\infty} \mathbb{E}[R_t | A_{0:t-1} \sim \pi_{\boldsymbol{\theta}}] \\ &= \sum_s \mu_{\pi_{\boldsymbol{\theta}}}(s) \sum_a \pi_{\boldsymbol{\theta}}(a|s) \sum_{s',r} p(s', r|s, a) r, \end{aligned} \tag{5.2}$$

where $\mu_{\pi_{\boldsymbol{\theta}}} = \lim_{t\to\infty} \mathbb{P}(S_t = s | A_{0:t} \sim \pi_{\boldsymbol{\theta}})$ is the steady-state distribution under $\pi_{\boldsymbol{\theta}}$. We assume that μ exists and is independent of the starting state. In other words, we assume that in the long run, the expectation of being in a state is dependent only on the policy and the Markov decision process transition probabilities. Note that if you select actions according to π, you remain in the same distribution:

$$\sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_r p(s', r|s, a) = \mu_\pi(s'). \tag{5.3}$$

With these objective functions, we are able to find $\boldsymbol{\theta}$ which maximizes $J(\boldsymbol{\theta})$. Since the aim is to maximize, we use stochastic gradient ascent in J instead of gradient descent to learn $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t), \tag{5.4}$$

where $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ is the gradient of J with respect to $\boldsymbol{\theta}$ and α is a step-size parameter.

5.2 Policy Gradient Theorem

In any tasks, the performance of the agent depends on both the action selections and the distribution of states in which those selections are made. Both are affected by the policy parameter. At a state, the effect of the policy parameter on the actions can be computed directly through the rewards obtained. However, the effect of the policy on the state distribution is unknown most of the times since it is a function of the environment. It becomes challenging to ensure that the policy parameter change in the direction which improves the performance.

In this section, we introduce the policy gradient theorem which provides us an analytic expression for the gradient of performance with regards to the policy parameter that does not involve the derivative of the state distribution. This policy gradient theorem is applicable to all the objective functions mentioned in the previous section:

Theorem 5.2.1. *For any differentiable policy $\pi_{\boldsymbol{\theta}}(s, a)$, we have*

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla_{\boldsymbol{\theta}} \pi(a|s, \boldsymbol{\theta}). \tag{5.5}$$

We will prove that the objective functions for the episodic case $J_1(\boldsymbol{\theta})$ and the continuing case $J_{avR}(\boldsymbol{\theta})$ do satisfy the policy gradient theorem.

5.2.1 Episodic Case

Proposition 5.2.2. *Suppose that the performance is defined as the value of the start state of the episode (i.e. $J(\boldsymbol{\theta}) = J_1(\boldsymbol{\theta})$). Then, the policy gradient theorem holds.*

Proof. It is implicit that π is a function of $\boldsymbol{\theta}$ and all gradients are with respect to $\boldsymbol{\theta}$. Note that the gradient of the state-value function can be written as

$$\begin{aligned}
\nabla v_\pi(s) &= \nabla[\sum_a \pi(a|s) q_\pi(s, a)] && \text{(by 1.8)} \\
&= \sum_a [\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \nabla q_\pi(s, a)] && \text{(by product rule)} \\
&= \sum_a [\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \nabla \sum_{s', r} p(s', r|s, a)(r + \gamma v_\pi(s'))] && \text{(by 1.9)} \\
&= \sum_a [\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \sum_{s', r} \gamma p(s', r|s, a) \nabla v_\pi(s')] \\
&= \sum_a [\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \sum_{s', r} \gamma p(s', r|s, a) \\
&\qquad \sum_{a'} [\nabla \pi(a'|s') q_\pi(s', a') + \pi(a'|s') \sum_{s'', r} \gamma p(s'', r|s', a') \nabla v_\pi(s'')]] \\
&= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \gamma^k \mathbb{P}(s \to x, k, \pi) \sum_a \nabla \pi(a|x) q_\pi(x, a)
\end{aligned}$$

where $\mathbb{P}(s \to x, k, \pi)$ is the probability of transitioning from state s to state x in k

steps under policy π. Then,

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \nabla v_\pi(s_0) \\
&= \sum_s \sum_{k=0}^{\infty} \gamma^k \mathbb{P}(s_0 \to s, k, \pi) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(by def. of } \eta \text{ in 4.3)} \\
&= (\sum_s \eta(s)) \sum_s \frac{\eta(s)}{\sum_s \eta(s)} \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a). && \text{(by 4.4)}
\end{aligned}$$

□

5.2.2 Continuing Case

As defined in (5.2), $r(\pi)$ denotes the average rate of reward while following policy π. Previously, the discounted setting was used for formulating the goal in Markov decision problems where the returns are defined as the total discounted reward. Here, we introduce a new setting, which is the average reward setting. In this setting, returns are defined as follows.

Definition 5.2.1. *The differential return G_t is the total differences between the rewards and the average reward after time t.*

$$G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + ... \tag{5.6}$$

We then define the corresponding Bellman equations for the differential value functions, which is still denoted as v_π and q_π.

Definition 5.2.2. *Differential state-value function v_π and action-value function q_π is defined as the solutions to*

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s,a)[r - r(\pi) + v_\pi(s')] \tag{5.7}$$

and

$$q_\pi(s,a) = \sum_{r,s'} p(s', r|s,a)[r - r(\pi) + \sum_a \pi(a'|s') q_\pi(s',a')]. \tag{5.8}$$

With these alternate definitions, the policy gradient theorem remains true for the continuing case.

Proposition 5.2.3. *Suppose that the performance is defined as the average rate of reward per time step (i.e. $J(\boldsymbol{\theta}) = J_{avR}(\boldsymbol{\theta})$). Then, the policy gradient theorem holds.*

Proof. It is again implicit that π is a function of $\boldsymbol{\theta}$ and all gradients are with respect to $\boldsymbol{\theta}$. The gradient of the state-value function can be written, for any $s \in \mathcal{S}$, as

$$\begin{aligned}
\nabla v_\pi(s) &= \nabla[\sum_a \pi(a|s) q_\pi(s,a)] \\
&= \sum_a [\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a)] \\
&= \sum_a [\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s',r|s,a)(r - r(\boldsymbol{\theta}) + v_\pi(s'))] \quad \text{(by 5.8)} \\
&= \sum_a [\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s)[-\nabla r(\boldsymbol{\theta}) + \sum_{s',r} p(s',r|s,a) \nabla v_\pi(s')]].
\end{aligned}$$

After re-arranging, we have

$$\nabla r(\boldsymbol{\theta}) = \sum_a [\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s',r} p(s',r|s,a) \nabla v_\pi(s')] - \nabla v_\pi(s).$$

We have $J(\boldsymbol{\theta}) = r(\pi)$ and hence, the left-hand side can be written as $\nabla J(\boldsymbol{\theta})$. This is independent of s. So, on the right-hand side, we can sum it over all s, weighted by $\mu(s)$ which sums up to one. Thus,

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \sum_s \mu_\pi(s)[\sum_a [\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s',r} p(s',r|s,a) \nabla v_\pi(s')] - \nabla v_\pi(s)] \\
&= \sum_s \mu_\pi(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&\qquad + \sum_{s'} \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_r p(s',r|s,a) \nabla v_\pi(s') - \sum_s \mu_\pi(s) \sum_a \nabla v_\pi(s) \\
&= \sum_s \mu_\pi(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) + \sum_{s'} \mu_\pi(s') \nabla v_\pi(s') - \sum_s \mu_\pi(s) \sum_a \nabla v_\pi(s) \\
&\hfill \text{(by 5.3)} \\
&= \sum_s \mu_\pi(s) \sum_a \nabla \pi(a|s) q_\pi(s,a).
\end{aligned}$$

□

5.3 Methods

Having an expression for the gradient of performance with respect to the policy parameter, we now proceed to look into methods to find the parameter for the best performing policy. The policy can be parameterized in any way as long as $\nabla_{\boldsymbol{\theta}}\pi(a|s,\boldsymbol{\theta})$ exists and is always finite. Also, to ensure exploration, we require that $\pi(a|s,\boldsymbol{\theta}) \in (0,1)$. One way is to form parameterized numerical preferences $h(s,a,\boldsymbol{\theta}) \in \mathbb{R}$ for each state-action pair, which can be computed using the linear or neural network methods described in the previous chapter. For example, the preferences could be linear in features $\boldsymbol{x}$, i.e. $h(s,a,\boldsymbol{\theta}) = \boldsymbol{\theta}^T\boldsymbol{x}(s,a)$. Actions with the highest preferences in each state will then be given the highest probabilities of being selected and the action probabilities in each state should sum to one. The following exponential softmax distribution is able to achieve these conditions:

$$\pi(a|s,\boldsymbol{\theta}) = \frac{exp(h(s,a,\boldsymbol{\theta}))}{\sum_b exp(h(s,b,\boldsymbol{\theta}))}. \tag{5.9}$$

We can now proceed with the policy gradient learning algorithms using Equation (5.4).

5.3.1 REINFORCE

From the policy gradient theorem, we have:

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a)\nabla_{\boldsymbol{\theta}}\pi(a|s,\boldsymbol{\theta}) \\
&= \mathbb{E}_\pi[\sum_a q_\pi(S_t,a)\nabla_{\boldsymbol{\theta}}\pi(a|S_t,\boldsymbol{\theta})] \\
&= \mathbb{E}_\pi[\sum_a \pi(a|S_t,\boldsymbol{\theta})q_\pi(S_t,a)\frac{\nabla_{\boldsymbol{\theta}}\pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})}] \\
&= \mathbb{E}_\pi[q_\pi(S_t,A_t)\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}] \\
&= \mathbb{E}_\pi[G_t\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}]. && \text{(by 1.5)}
\end{aligned}$$

With this, we are able to instantiate the stochastic gradient ascent algorithm as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}. \tag{5.10}$$

As introduced in [5], this algorithm is called REINFORCE.

From the update rule, we could see that each increment is proportional to the product of a return G_t and a vector $\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}$. The vector is the direction in parameter space that most increases the probability of repeating the action A_t on future visits to state S_t. The update increases the parameter vector in this direction proportional to the return and inversely proportional to the action probability. This makes sense because the parameter will move most in the directions that favor actions that yield the highest return but actions that are selected frequently will not be at an advantage.

One drawback of this method is that it may have high variance due to the return G_t and thus produce slow learning.

5.3.2 REINFORCE with Baseline

The policy gradient theorem (5.5) can be generalized to include a comparison of the action value to an arbitrary baseline $b(s)$:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a (q_\pi(s,a) - b(s)) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}). \tag{5.11}$$

The baseline can be any function as long as it does not vary with a because

$$\sum_a b(s)\nabla_{\boldsymbol{\theta}}\pi(a|s,\boldsymbol{\theta}) = b(s)\nabla_{\boldsymbol{\theta}}\sum_a \pi(a|s,\boldsymbol{\theta}) = b(s)\nabla_{\boldsymbol{\theta}} 1 = 0.$$

Using similar steps in the previous section, we end up with a new version of REINFORCE which includes baseline:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(G_t - b(S_t))\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}. \tag{5.12}$$

In general, the baseline leaves the expected value of the update unchanged, but it can have a large effect on its variance. The baseline should vary with state. In some states, actions have high values and we need a high baseline to differentiate higher valued actions from the less highly valued ones while in other states a low baseline is appropriate since actions will have low values. One natural choice is an estimate of the state value, $\hat{v}(S_t, \boldsymbol{w})$, where $\boldsymbol{w}$ is a weight vector learned by one of the methods in the previous chapter.

5.4 Summary

In this chapter, we no longer focus on methods which learn action values and then use them to determine action selections. On the other hand, we considered methods which learn a policy directly, and hence enables actions to be taken without the information on the value of each action. Using stochastic gradient ascent, the policy parameter is updated in the direction of an estimate of the gradient of the performance measure. The policy gradient theorem gives an exact formula for how performance is affected by the policy parameter which does not involve derivatives of the state distribution. This theorem leads to the introduction of some policy gradient methods.

With all the methods introduced up to this chapter, we would be interested to compare their performance. Hence, in the last section, experiments are conducted using various methods to have a better idea of their strengths and weaknesses, and how useful they would be in real-life applications.

Part III
Application

In this last section, we look at a reinforcement learning application. More specifically, it is a public transportation planning problem. We will see later how this problem can be constructed in a way such that it becomes a reinforcement learning problem and the ideas in Chapter 1 would fit in nicely. A few of the methods described in the previous sections will then be applied and compared through some experiments on this problem.

Chapter 6

Public Transportation Planning

The aim of this experiment is to examine if reinforcement learning could be implemented in public transportation planning. Using reinforcement learning, we hope to provide better insights on the current system and hence improve the planning process. In this experiment, we will be using bus services as an example. A simulated environment will be set up to replicate the actual bus system closely. To simplify the problem, the buses will only be picking up passengers and not taking into account which bus station each passenger intend to get off.

6.1 Problem Statement

First, this problem is formulated as a reinforcement learning problem. The agent will be the bus operator which controls the deployment of buses. As for the environment, it involves anything outside the control of the agent. This includes the buses available, the stations in the bus route, the passengers arriving at each station and so on. There are a few variations of the problem.

1. The buses travel in one direction, i.e. in a loop. There is only one bus route, and hence all the buses available serve the same course.
2. The buses still travel in a loop but peak hours are introduced. In other words, at certain periods of time, passengers arrive at a faster rate in some of the bus stations. There is still only one bus route.
3. The buses travel back and forth between two terminals. Only one route is available but it is accessible in two directions.

In the first two cases, there are only two actions for the agent to choose from: to deploy a bus or not. As for the third situation, there would be four actions available: to deploy a bus from either terminal or from both terminals or none at all.

Nevertheless, in the simulated environment of any version of the problem, there are a few parameters that can be tuned:

- length of bus route,
- position of bus stations,
- number of buses available,
- mean time interval in between the arrival of each passenger at each bus station,
- maximum number of passengers possible at each bus station.

At each bus stop, the passengers will arrive randomly at a rate depending upon the mean time interval in between the arrival of each passenger assigned. By having different mean time interval, the passenger density at each bus stop can vary.

With all the variables above, we will now define the state space. Note that not all the information regarding these variables will be known to the agent. In other words, the agent has no complete knowledge of the environment. At every time step, the only information that the agent will be given are the current number of passengers at each bus stations and the current position of the bus nearest to the terminal. An additional information available for the agent in the second type of environment would be the current time since this could be important in deciding what action to take and hence the policy. Without sufficient information, the space consisting all policies that can be derived given the state and action space will not include the actual optimal policy in real life, causing the performance to be incomparable.

With the number of passengers possible at each bus station capped at a specific amount, the state space would be finite. This satisfies our assumption established earlier. Furthermore, the way the problem is formalized satisfies the Markov property because it is obvious that the current state is sufficient to predict the next state.

Another important element for the reinforcement learning problem is the reward system. The reward system consists of the following:

- cost for each bus departure,
- cost for a bus overtaking another bus,
- penalty for taking the action of deploying a bus when there is none,
- cost for one waiting passenger per unit time,
- reward for each passenger picked up.

These rewards and costs will be adjusted according to the objective of the experiment. For instance, if the aim is to improve the customer satisfaction, the cost for a waiting passenger per unit time should be high. On the other hand, to solve bus bunching problem, the cost of having a bus overtaking another should be have a high value. Lastly, in the situation where there is a limited number of buses, an optimal policy could be to always take the action of deploying a bus if there is no penalty for taking such action when no bus is available for deployment. This is because only when there is a bus ready at the terminal that the action would take effect and a cost would be incurred. Hence, the third penalty in the list above is introduced to avoid such case.

6.2 Methods Used

Based on the information available to the agent at each time step, the state space for this problem could be huge. For example, if there are six bus stops in a route and the number of passengers at each stop is capped at ten, then the size of the state space would be larger than a million. Hence, it is impractical to use tabular solution methods. Instead, only approximate solution methods introduced in Part III will be used. In particular, we will implement a non-linear function approximation method using Q-learning and the REINFORCE with baseline method as a policy gradient technique. In both methods, neural networks are built and trained to approximate the value function and the policy function respectively. The framework used to build the networks here is Tensorflow [4].

To evaluate how these the reinforcement learning methods are performing, we will use some hand-crafted policies. To compare the performance of these policies, we use the fact that policies with higher average reward per time step are doing better. Hence, the goal is to check if the resulting policies from the function approximation methods are able to achieve an average reward per time step that is as good as or better than the hand-crafted policies. However, it is important to note that in order to be comparable with a resulting policy, each of the hand-crafted policy must be a function from the state space to the action space of the current environment. In other words, only information that is available to the agent will be used by the hand-crafted policies in determining the actions to be taken. For example, if the state provided to the reinforcement learning agent at each time step does not contain any information on the current time, then the hand-crafted policies should not in any way depend on this information to choose the actions. Hence, depending on how the environment is set up, the hand-crafted policies need to be defined accordingly.

6.3 Observations & Inferences

Through the several experiments conducted on the three different types of environment setup indicated above, there are a few observations and inferences to be made. We first make some general comments on the performance of the reinforcement learning algorithms on this problem.

1. When the function approximation method and policy gradient method are both applied to the same environment setup, the former takes a much longer time to converge to a near optimal performance as the problem set gets larger. This is because when using function approximation method, we construct a new ϵ-greedy policy at each policy improvement step. ϵ is decayed after each iteration in order to converge to a deterministic policy, but it is a challenging task to determine the

decay rate. Since the state space is huge, a fast decaying ϵ would cause exploration issue. But by lowering the decay rate, we arrive at the current situation, which is slow learning. On the other hand, the goal of these experiments is for the agent to find a policy which performs near optimal and not the values of each state-action pair. Hence, policy gradient is much more suitable for this problem.

2. Using the same policy gradient method on environments with different parameter setting, the reinforcement learning agent is shown to be performing consistently well. More specifically, in each environment, parameters such as the mean time interval in between the arrival of each passenger at each bus station, the cost for one waiting passenger per unit time and the reward for each passenger picked up, are assigned differently. Also, a few hand-crafted policies are defined and will be used as baselines in all the experiments to compare the performance of the policy gradient method. It is observed that these hand-crafted policies perform inconsistently. In other words, Policy A might have a better average reward per time step compared to Policy B in one experiment but the situation is reversed in another experiment. However, in each experiment, the performance of the policy gradient method always converge at least to the performance of the better policy among the hand-crafted policies as shown in Figure 6.3.1. In short, the policy gradient method is reliable for this problem and preferable over searching for near optimal policies by trial-and-error.

Now that we have ensured the performance of the REINFORCE with baseline algorithm on this problem, we will use this approach on different environments in order show that we are able to gain insights into some specific issues which are commonly encountered in public transportation planning using reinforcement learning.

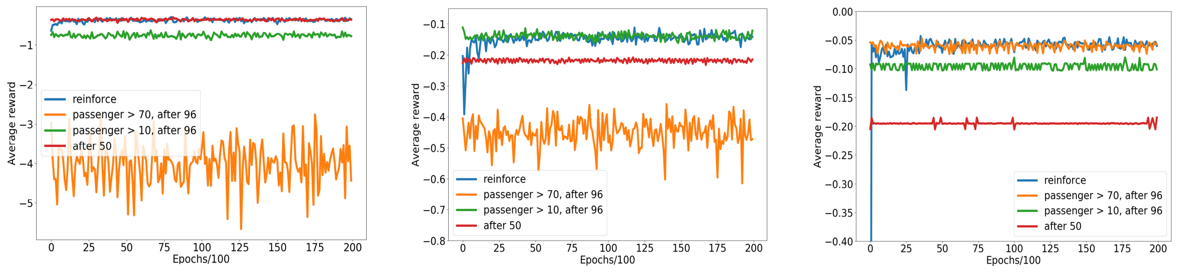
1. Bus bunching refers to two or more buses traveling along the same route, which were scheduled to be evenly spaced, were running in the same place at the same time. One reason causing this problem is when there is a large number of people boarding at each bus stops, causing it to delay. At the same time, the next bus

will need to spend less time at stops and hence could overtake its predecessor. It is hard to determine when is the best time to deploy a bus because deploying buses too frequently would be a waste of resources and even worsen the problem while delaying the deployment of buses would increase the passengers' waiting time and cause customer dissatisfaction.

To solve this scheduling problem by reinforcement learning, we set the reward system such that the cost of bus overtaking to be high. Of course, the agent should also be penalized for each waiting passenger. By having a high passenger arrival rate at the bus stations, we can now simulate the bus bunching problem. As shown in Figure 6.3.2, the result obtained from implementing the policy gradient method shows that the resulting performance is better or as good as policies which deploy buses every time the bus nearest to the terminal reaches a targeted location. Thus, the algorithm applied here is able to find a close to optimal solution for this multi-objective optimization problem which reduces bus bunching and at the same time, ensures that passengers need not wait too long to be picked up by a bus.

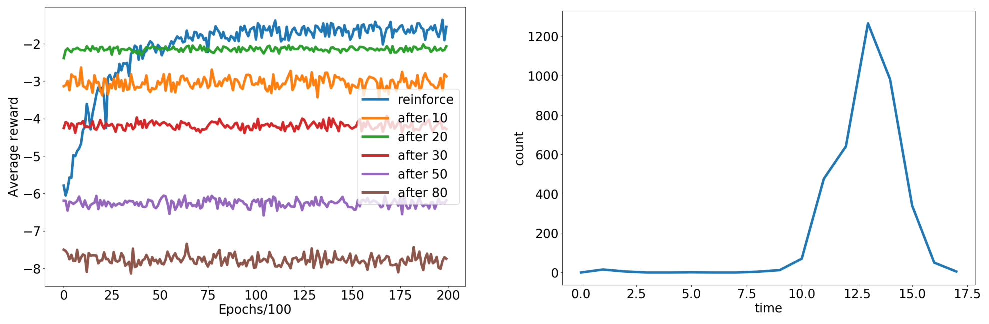
2. A rush hour is a part of the day during which crowding on public transport is the highest, which usually happens twice every weekday. During these periods of time, the arrival rate of passengers at some of the bus stops could be higher. Intuitively, buses need to be deployed at a higher frequency to avoid overcrowding and customer dissatisfaction.

   We are interested to know if the reinforcement learning agent is able to learn about the change in the passenger arrival rate during peak hour from experience. As mentioned previously, the agent will now have an extra information which is the current time of the day. As for the environment, each bus station has two passenger arrival rate. If these two values are different, passengers will arrive at this station at a faster rate during certain periods of time to simulate the peak hour scenario. The results obtained from several experiments show that the agent is able to perform as good as or better than some hand-crafted policies after interacting



(a) (b) (c)

Figure 6.3.1: The results of three experiments with the same type of environment but different parameter setup. In each experiment, the blue line shows the average rewards of policies during training using REINFORCE with baseline method while the orange, green and red line represents three different hand-crafted policies respectively which are fixed throughout (a), (b) and (c). The results show that the three hand-crafted policies are ranked differently in each experiment.



(a) (b)

Figure 6.3.2: The result of an experiment on the bus bunching problem. (a) The performance of REINFORCE with baseline method (represented by the blue line) shows that the resulting policy outperforms the hand-crafted policies. Each hand-crafted policy is such that buses are deployed after the previous bus is at a certain distance from the terminal. (b) Whenever a bus is deployed, the time since the deployment of the previous bus can be determined. The frequency graph shows that shows that the resulting policy deploys buses at approximately every 11-15 time steps.

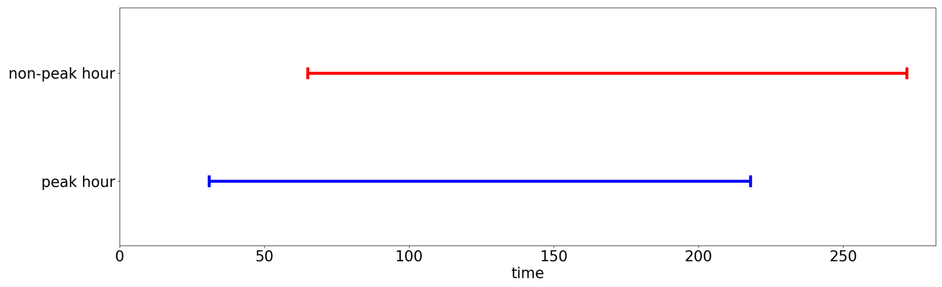


Figure 6.3.3: The result of an experiment on the peak hour problem. When testing the resulting policy, the time since the deployment of the previous bus is determined during each deployment of a bus. The graph shows that the range of the time interval between each deployment of buses is earlier during peak hours compared to non-peak hours.

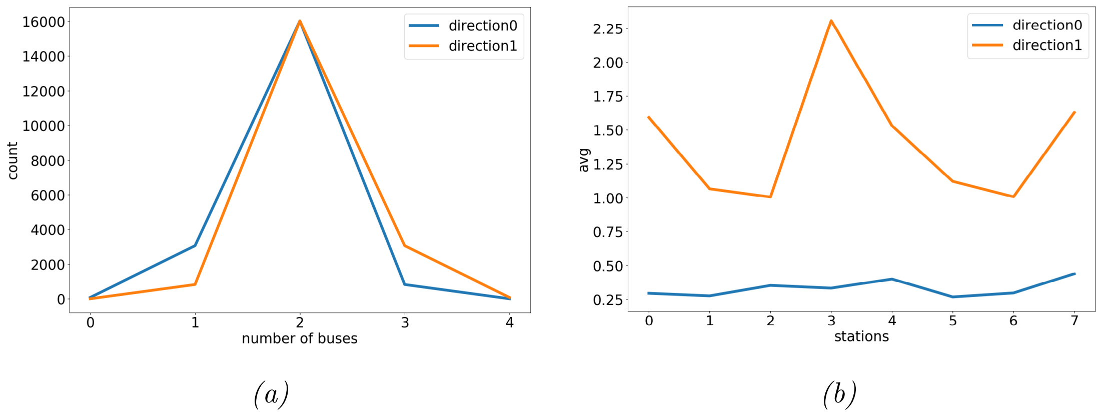


Figure 6.3.4: The result of an experiment on the environment where buses travel in both ways and the arrival rate of passengers traveling in 'direction1' being higher than 'direction0'. The total number of buses available is four. (a) At each time step, the number of buses traveling in each direction is determined. The frequency graph shows that the number of buses is balanced on both sides since it is maintained at two at most times. (b) On average, the number of passengers at each bus stations during each bus deployment for 'direction1' is higher than the other, indicating that the decision on whether to deploy a bus on one side does not depend only on the current number of passengers on that side of the route.

with the environment. This indicates that the agent is in fact learning and taking the peak hour scenario into account by having a different schedule during these hours as shown in Figure 6.3.3.

3. For buses traveling in two directions of the same route, it is important to keep the number of buses at both sides balanced as a deployment of a bus in one direction will affect the availability of buses in the other direction. If buses are deployed at a higher frequency from one terminal than the other, the buses will be concentrated on the other terminal and hence cause the passengers traveling in one of the direction to have longer waiting time. This issue could arise if the passenger density at the bus stations which are traveling towards a terminal is much higher than that of the other direction. Thus, we cannot treat the two directions separately as two different system since the future states of one direction is dependent on the action that was taken on the other direction.

   This problem can be easily simulated using the third type of environment setup by choosing one of the direction and setting the arrival rate for passengers traveling in this direction to be higher compared to the other. Also, it is important to set a penalty for taking the action of deploying a bus when there is none. This is because it is common that there is no bus available at the terminal and we want to avoid the case where the resulting policy takes the action of deploying buses at every time step even when there is no bus. The experiment shows a result which is consistent with the previous experiments on different cases. This indicates that the resulting policy does take the situation on both sides into account and the number of buses on both sides remains balanced throughout as illustrated in Figure 6.3.4.

An important point to note is that the hand-crafted policies used in each experiment might or might not be an optimal policy. In reality, there are many ways to schedule the bus deployment and we are not able to determine which would lead to the best performance. Therefore, we could only provide a few policies which are likely to be effective under the circumstance.

6.4 Conclusion & Further Work

Overall, the experiments show promising results to solve the public transportation planning by formulating the task as a reinforcement learning problem. The REINFORCE with baseline algorithm, which is a policy gradient method, manages to find near optimal solutions for these finite Markov decision processes. This algorithm fits into different environment setup and hence, would be more effective than finding a better performing policy by trial-and-error every time there is a new setup. In real life, it would be difficult to manually plan out a bus schedule solely by observing the data available. By using reinforcement learning, we could at least gain some insights from the huge amount of data through the resulting policy. From there, it would make planning easier as we have some important understanding of the problem to start with.

Finally, there are some further work which are worthy to be explored. So far, all the experiments are conducted on a single bus route, be it a loop or a two direction. A more complex but much closer to the real-life system would be to have multiple bus routes and buses that travel in different paths. The passengers at each bus stations would now have a specific bus that they want to board. Secondly, we have been using simulated environment in these experiments. It would interesting to see how would the performance be if actual data is used instead. If reinforcement learning continues to work well after these two improvements to the system, it would be possible to use it in real-time instead of for schedule planning purpose only. Lastly, the focus up to this point is on one specific type of public transport, which is on buses. This idea could be expanded into other common modes of public transport such as the train. By making some changes on how the environment is set up, we should be able to simulate the planning problem for these transport modes into a reinforcement learning task.

Appendix A - Pseudocodes

Algorithm 1: Policy Iteration

```
Initialize, for all $s \in S$:
    $V(s) \leftarrow$ arbitrary
    $\pi(s) \leftarrow$ an arbitrary deterministic policy
repeat
    $\pi' \leftarrow \pi$
    repeat
        $\Delta \leftarrow 0$
        for $s \in S$ do
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
            $\Delta \leftarrow \max(\Delta, |v - V(s))|)$
        end
    until $\Delta < \theta$ (a small positive number);
    for $s \in S$ do
        $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    end
until $\pi = \pi'$;
```

Algorithm 2: Value Iteration

```
Initialize, for all $s \in S$:
    $V(s) \leftarrow$ arbitrary
repeat
    $\Delta \leftarrow 0$
    for $s \in S$ do
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s))|)$
    end
until $\Delta < \theta$ (a small positive number);
for $s \in S$ do
    $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
end
```

Algorithm 3: First-Visit Monte-Carlo for estimating $Q \approx q_*$

```
Input: a decaying function $\epsilon(i) \in [0,1]$
Initialize $Q(s,a)$ arbitrarily for all $s \in S$, $a \in A$
for $i = 1, 2, ...$ do
    $\epsilon \leftarrow \epsilon(i)$
    Derive an $\epsilon$-greedy policy $\pi$ from $Q$
    Generate an episode using $\pi$
    for each pair s,a appearing in the episode do
        $G \leftarrow$ the return that follows the first occurrence of $s$,$a$
        $Q(s,a) \leftarrow Q(s,a) + \alpha(G - Q(s,a))$
    end
end
```

Algorithm 4: SARSA for estimating $Q \approx q_*$

```
Input: a decaying function $\epsilon(i) \in [0,1]$
Initialize $Q(s,a)$ arbitrarily for all $s \in S$, $a \in A$
for $i = 1, 2, ...$ do
    $\epsilon \leftarrow \epsilon(i)$
    Initialize $s_0$
    Choose $a_0$ from $s_0$ using $\epsilon$-greedy policy derived from $Q$
    repeat
        Take action $a_0$, observe $r$ and $s_1$
        Choose $a_1$ from $s_1$ using $\epsilon$-greedy policy derived from $Q$
        $Q(s_0,a_0) \leftarrow Q(s_0,a_0) + \alpha(r + \gamma Q(s_1,a_1) - Q(s_0,a_0))$
        $s_0 \leftarrow s_1$ ; $a_0 \leftarrow a_1$
    until $s_0$ is terminal;
end
```

Algorithm 5: Q-learning for estimating $Q \approx q_*$

```
Input: a decaying function $\epsilon(i) \in [0,1]$
Initialize $Q(s,a)$ arbitrarily for all $s \in S$, $a \in A$
for $i = 1, 2, ...$ do
    $\epsilon \leftarrow \epsilon(i)$
    Initialize $s_0$
    repeat
        Choose $a_0$ from $s_0$ using $\epsilon$-greedy policy derived from $Q$
        Take action $a_0$, observe $r$ and $s_1$
        $Q(s_0,a_0) \leftarrow Q(s_0,a_0) + \alpha(r + \gamma \max_a Q(s_1,a) - Q(s_0,a_0))$
        $s_0 \leftarrow s_1$
    until $s_0$ is terminal;
end
```

Algorithm 6: Monte Carlo Function Approximation for estimating $\hat{q} \approx q_*$

```
Input: a differentiable action-value parameterization $\hat{q}(s, a, \boldsymbol{w})$
       a decaying function $\epsilon(i) \in [0, 1]$
Initialize action-value weights $\boldsymbol{w}$
for $i = 1, 2, ...$ do
    $\epsilon \leftarrow \epsilon(i)$
    Derive an $\epsilon$-greedy policy $\pi$ from $\hat{q}$
    Generate an episode using $\pi$
    for each pair s,a appearing in the episode do
        $G \leftarrow$ the return that follows the first occurence of $s$,$a$
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[G - \hat{q}(s, a, \boldsymbol{w})]\nabla\hat{q}(s, a, \boldsymbol{w})$
    end
end
```

Algorithm 7: SARSA Function Approximation for estimating $\hat{q} \approx q_*$

```
Input: a differentiable action-value parameterization $\hat{q}(s, a, \boldsymbol{w})$
       a decaying function $\epsilon(i) \in [0, 1]$
Initialize action-value weights $\boldsymbol{w}$
for $i = 1, 2, ...$ do
    $\epsilon \leftarrow \epsilon(i)$
    Initialize $s_0$
    Choose $a_0$ from $s_0$ using $\epsilon$-greedy policy derived from $\hat{q}$
    repeat
        Take action $a_0$, observe $r$ and $s_1$
        Choose $a_1$ from $s_1$ using $\epsilon$-greedy policy derived from $\hat{q}$
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[r + \gamma\hat{q}(s_1, a_1, \boldsymbol{w}) - \hat{q}(s_0, a_0, \boldsymbol{w})]\nabla\hat{q}(s_0, a_0, \boldsymbol{w})$
        $s_0 \leftarrow s_1$ ; $a_0 \leftarrow a_1$
    until $s_0$ is terminal;
end
```

Algorithm 8: Q-learning Function Approximation for estimating $\hat{q} \approx q_*$

```
Input: a differentiable action-value parameterization $\hat{q}(s, a, \boldsymbol{w})$
       a decaying function $\epsilon(i) \in [0, 1]$
Initialize action-value weights $\boldsymbol{w}$
for $i = 1, 2, ...$ do
    $\epsilon \leftarrow \epsilon(i)$
    Initialize $s_0$
    repeat
        Choose $a_0$ from $s_0$ using $\epsilon$-greedy policy derived from $\hat{q}$
        Take action $a_0$, observe $r$ and $s_1$
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[r + \gamma \max_a \hat{q}(s_1, a, \boldsymbol{w}) - \hat{q}(s_0, a_0, \boldsymbol{w})]\nabla\hat{q}(s_0, a_0, \boldsymbol{w})$
        $s_0 \leftarrow s_1$
    until $s_0$ is terminal;
end
```

```
Algorithm 9: REINFORCE for estimating $\pi \approx \pi_*$
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta}$
for $i = 1, 2, ...$ do
    Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$ following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    for $t = 0, 1, ...T - 1$ do
        $G \leftarrow$ return from step $t$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha G \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})}$
    end
end
```

```
Algorithm 10: REINFORCE with Baseline for estimating $\pi \approx \pi_*$
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
       a differentiable state-value parameterization $\hat{v}(s, \boldsymbol{w})$
Initialize policy parameter $\boldsymbol{\theta}$ and state-value weight $\boldsymbol{w}$
for $i = 1, 2, ...$ do
    Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$ following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    for $t = 0, 1, ...T - 1$ do
        $G \leftarrow$ return from step $t$
        $\delta \leftarrow G - \hat{v}(S_t, \boldsymbol{w})$
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla_{\boldsymbol{w}} \hat{v}(S_t, \boldsymbol{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})}$
    end
end
```

References

[1] C. Chicone. (2006). *Ordinary Differential Equations with Applications.* New York: Springer.

[2] D. Silver. (2015). *COMPM050/COMPGI13: Advanced Topics in Machine Learning - Reinforcement Learning* [Lecture Slides]. Retrieved from `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`

[3] I. Goodfellow, Y. Bengio, & A. Courville. (2016). *Deep Learning.* Cambridge, MA: MIT Press. Retrieved from `http://www.deeplearningbook.org/`

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Man, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Vigas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, & X. Zheng. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems.* Software available from `tensorflow.org`.

[5] R. S. Sutton, & A. G. Barto. (2017). *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press. Retrieved from `http://incompleteideas.net/book/the-book-2nd.html`