university of groningen / faculty of science and engineering

Artificial Intelligence

Master's Thesis

Investigating Overestimation Bias in Reinforcement Learning

Author
Andreas Pentaliotis
s3667537

Internal Supervisor
Dr. Marco Wiering
Artificial Intelligence
University of Groningen

External Supervisor
Matthia Sabatelli
Electrical Engineering and Computer Science
University of Liège

June 20, 2020

Contents

Abstract

Overestimation bias is an inherent property of reinforcement learning algorithms that approximate maximum expected values by maximizing uncertain estimates. Since overestimation bias was identified in the literature, it has been generally considered to have a negative effect on reinforcement learning algorithms. In this thesis we investigate overestimation bias by examining Q-learning and conclude that overestimation bias may have either a negative or positive effect on reinforcement learning algorithms depending on the reinforcement learning problem. Based on this conclusion, we propose a new variant of Q-learning, called Variation-resistant Q-learning, to control and utilize estimation bias for better performance. We present the tabular version of Variation-resistant Q-learning, prove a convergence theorem for the algorithm in the tabular case, and extend the algorithm to a function approximation solution method. Additionally, we present empirical results from three different experiments, in which we compared the performance of Variation-resistant Q-learning, Q-learning, and Double Q-learning. The empirical results verify that Variation-resistant Q-learning can control and utilize estimation bias for better performance in the experimental tasks.

Keywords— Overestimation Bias; Reinforcement Learning; Machine Learning

Acknowledgements

Firstly, I would like to thank my internal supervisor, Marco, and my external supervisor, Matthia, for always having an insightful answer to my questions and for guiding me throughout this research project. I want to especially thank Marco, because through his courses he helped me discover my interest in machine learning.

Secondly, I would like to thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster.

Thirdly, I would like to thank my parents, Pampos and Mary, for always supporting me in every possible way in my long journey until this moment.

Fourthly, I am very grateful to my cousin, Panayiotis, for introducing me to the field of mathematics and for all the invaluable information.

Fifthly, I am very grateful to my friend, Nicolas, for guiding me in the field of computer science and for the many insightful conversations that enhanced my critical thinking.

Finally, special thanks to my brother, Panayiotis, and my extended family and friends, for all the experiences that shaped my personality in a meaningful way.

Note on Notation

In this thesis we mainly follow the notation used in the book by Sutton and Barto [36]. We use uppercase letters for sets, matrices, random vectors, random variables, and approximate value functions, whereas we use lowercase letters for vectors, values of random variables, and value functions. Additionally, matrices, random vectors, and vectors are denoted by bold letters. Scalar functions other than the value functions are denoted either by uppercase or lowercase letters.

Chapter 1

Introduction

Reinforcement learning is the subfield of machine learning that studies intelligent agents that learn how to act optimally by interacting with an environment without explicit instructions. Ideas from reinforcement learning were used in computer programs that achieved human-level performance in the game of checkers [31] and grandmaster-level performance in the game of backgammon [38]. Although these computer programs are considered to be great achievements for reinforcement learning and artificial intelligence, even greater achievements were accomplished when reinforcement learning was combined with deep learning.

Deep learning is a set of machine learning methods that use artificial neural networks with many layers to perform automatic feature extraction from data. These methods can learn useful representations of their input data and perform tasks that are considered difficult for computers. Some application areas for deep learning are speech recognition, computer vision, and natural language processing.

Deep reinforcement learning is a term that describes algorithms that combine ideas from reinforcement learning and deep learning. Recently, deep reinforcement learning algorithms were used in computer programs that have been shown to perform successful control in tasks that were previously considered impossible for computers. Specifically, these programs achieved human-level control in video games [24] and managed to master the game of Go [33]. After these breakthroughs, there has been an increasing interest in deep reinforcement learning and many improvements were suggested for deep reinforcement learning algorithms [32, 44, 24, 41, 11, 4, 9, 30, 29]. Furthermore, many of these improvements have been shown to be independent of each other and to increase performance dramatically when combined together [17].

Reinforcement learning problems are mathematically formalized in a way that involves maximization. Moreover, many reinforcement learning algorithms try to estimate expected values from a number of observed samples. Therefore, there are reinforcement learning algorithms that approximate maximum expected values by maximizing uncertain estimates. Consequently, these algorithms tend to overestimate the maximum expected values. This overestimation bias is more extreme in the beginning of learning, when the amount of observed samples is still limited. Furthermore, artificial neural networks have been shown to cause instability and to increase overestimation bias when combined with such algorithms [15].

Overestimation bias is generally considered to have a negative effect on reinforcement learning algorithms [39, 40, 16, 41, 42, 12], and many methods were proposed

to reduce or remove overestimation bias [40, 16, 20, 8, 41, 3, 21]. In fact, the most well-tested and successful method to overcome the problems caused by overestimation bias replaces overestimation bias with underestimation bias [40, 16]. However, a recent study suggests that the effect of overestimation bias on reinforcement learning algorithms depends on the reinforcement learning problem, and that it may be positive under certain conditions [19].

In this thesis we investigate overestimation bias by examining Q-learning [46], which is a reinforcement learning algorithm that approximates maximum expected values by maximizing uncertain estimates. In the remainder of this chapter we first present the research questions that we aim to answer in this thesis and then discuss the significance of this study. Finally, we conclude this chapter by presenting the thesis layout.

1.1 Research Questions

In this thesis we aim to answer the following research questions:

1. Under which conditions is overestimation bias harmful for reinforcement learning algorithms?

2. How can overestimation bias be reduced or removed when it is harmful for reinforcement learning algorithms?

3. Are there any conditions under which overestimation bias is desirable for reinforcement learning algorithms?

4. Assuming that there are conditions under which overestimation bias is desirable for reinforcement learning algorithms, how can overestimation bias be controlled and utilized for better performance?

1.2 Significance of the Study

Firstly, this study may benefit the reinforcement learning community by improving the state-of-the-art methods. Since deep reinforcement learning has been shown to be very successful, many state-of-the-art methods use artificial neural networks. However, artificial neural networks have been shown to increase overestimation bias, and overestimation bias has been shown to have a negative effect on reinforcement learning algorithms. Therefore, understanding the effect of overestimation bias on reinforcement learning algorithms may increase the performance of many state-of-the-art methods or lead to the development of better methods.

Secondly, this study may bring the artificial intelligence community one step closer to achieving artificial general intelligence. One of the long-term goals of reinforcement learning is to deliver intelligent agents that can learn and act in real time. However, all the successful computer programs that implement reinforcement learning algorithms and that are in our knowledge have partly used offline techniques to perform well. The reason is that reinforcement learning algorithms that are applied to complex problems do not perform well in real time because of their relatively high computational requirements. Overestimation bias seems to be one

of the causes of this problem, and understanding its effect on these algorithms may reduce their computational requirements.

Finally, this study may benefit society as well by advancing automation. There are optimization problems in the industry that can be solved automatically when they are modeled as reinforcement learning problems. Some examples are maximizing the performance of an automated manufacturing system and minimizing the energy consumption of an energy system. However, existing reinforcement learning algorithms are not able to solve these problems efficiently. Understanding the effect of overestimation bias on reinforcement learning algorithms may increase their efficiency.

1.3 Thesis Layout

This thesis is structured as follows. In the first chapter we introduced the research topic of this study, presented the research questions that we aim to answer in this thesis, and discussed the significance of this study. In the second chapter we present the theoretical background that guided this study. In the third chapter we propose a new method that is derived from the main conclusion of the second chapter. In the fourth chapter we describe the experiments that we conducted in order to evaluate the new method. In the fifth chapter we present the results obtained from the experiments and discuss our findings. Finally, in the sixth chapter we conclude this thesis with a discussion and the answers to the research questions.

Chapter 2

Theoretical Background

In this chapter we present the theoretical background that guided this study. In the first section we give a brief overview of reinforcement learning. In the second section we describe finite Markov decision processes, which can be used to model reinforcement learning problems. In the third section we describe the two main categories of solution methods that are applied to reinforcement learning problems. In the fourth section we describe multilayer perceptrons, which can be used as a tool in some solution methods. In the fifth section we define overestimation bias and show how it can be replaced with underestimation bias. In the sixth section we present Q-learning, show that it has overestimation bias,[1] and examine a case where overestimation bias could have a negative effect on the algorithm. In the seventh section we present Double Q-learning, which is the most well-tested and successful method to overcome the negative effect of overestimation bias on Q-learning. We show that Double Q-learning has underestimation bias,[2] and examine a case where underestimation bias could have a negative effect on the algorithm. In the eighth section we give a summary of all the other Q-learning variants that were proposed to overcome the negative effect of overestimation bias on Q-learning and that are in our knowledge. Finally, in the last section we discuss the conclusions of this chapter.

2.1 Reinforcement Learning

Reinforcement learning is one of the three main subfields of machine learning along with supervised learning and unsupervised learning. Whereas in supervised learning we study learning algorithms that are applied on training examples that have targets, and in unsupervised learning our goal is to understand the structure of training examples that do not have targets, in reinforcement learning we focus on intelligent agents that interact with an environment and try to achieve a goal.

Fundamentally, reinforcement learning problems are problems of optimal control. The agent is a decision maker that interacts with an environment. At each point in time the environment is in a certain state that the agent observes. Every time the agent acts on the environment, the environment changes its state and provides a reward signal to the agent. The goal of the agent is to act optimally in

[1] We refer to overestimation bias as an inherent property of Q-learning. This does not imply that the algorithm shows overestimation in every reinforcement learning problem.

[2] We refer to underestimation bias as an inherent property of Double Q-learning. This does not imply that the algorithm shows underestimation in every reinforcement learning problem.

order to maximize its total reward. In most cases the agent's actions may influence not only its immediate rewards but also its future rewards and the environment's future states. Moreover, there is an inherent uncertainty in reinforcement learning problems, as the agent's actions and the environment's reactions to those actions can be highly stochastic. Therefore, the problems that are studied in reinforcement learning can become very complex.

One main challenge in reinforcement learning is the exploration–exploitation dilemma. On the one hand, the agent should exploit (i.e. repeatedly take actions that provide high rewards) in order to achieve the goal of maximizing its total reward. On the other hand, the agent should explore (i.e. take actions for which it has no prior experience) in order to discover actions that are more rewarding than the ones it already knows. For the agent to perform well and achieve its goal, a balance must be found between exploration and exploitation.

A widely used method to achieve this balance is the ϵ-greedy method. We did not yet discuss how actions can be evaluated, but imagine that the agent has an idea of which the optimal action is in a certain state. Perhaps it estimated the expected rewards for all the actions in that state and determined the action that maximizes the estimates with ties broken arbitrarily. Note that the agent could be wrong because it did not yet take all the actions in that state enough times. When using the ϵ-greedy method, the agent acts randomly and takes any action that is permitted in a certain state with probability ϵ. Otherwise, it takes the greedy (i.e. most highly valued) action. The amount of exploration can be adjusted by changing the value of ϵ.

We described the main ideas of reinforcement learning in a qualitative way. To quantify these ideas, we mathematically formalize reinforcement learning problems as finite Markov decision processes. We describe these processes in the next section.

2.2 Finite Markov Decision Processes

A finite Markov decision process is a discrete-time, stochastic, sequential, decision-making process that can be used to mathematically formalize reinforcement learning problems. This process involves a controlling entity, called the agent, that continuously interacts with an external entity, called the environment, by selecting and taking actions. The environment reacts to the agent's actions by changing its condition, called state, and providing the agent with numerical signals, called rewards, that the agent should learn to maximize over time.

2.2.1 Basic Structure

Formally, a finite Markov decision process is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma, t)$ where:

1. $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ is a finite set of states
2. $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ is a finite set of actions
3. $\mathcal{R} = \{r_1, r_2, \ldots, r_\kappa\}$ is a finite set of rewards
4. $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ is the dynamics function
5. $\gamma \in [0, 1]$ is the discount factor

6. $t = 0, 1, 2, 3, \ldots$ is the time counter

The time counter t encodes the passage of time in the form of discrete time steps. At each time step t the environment is in a certain state $S_t \in \mathcal{S}$. The agent receives an observation from the environment, which is a representation of S_t, and decides to take an action $A_t \in \mathcal{A}$.[3] The environment reacts to A_t by transitioning to a next state $S_{t+1} \in \mathcal{S}$ and providing a reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ to the agent. The probability of the next state and reward is determined by the dynamics function p as follows,

$$p(s', r \mid s, a) = \Pr\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\} \tag{2.1}$$

for all $s', s \in \mathcal{S}, r \in \mathcal{R}$, and $a \in \mathcal{A}$. In other words, p defines a probability distribution over state-reward pairs for each state-action pair and therefore satisfies,

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1, \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A} \tag{2.2}$$

We graphically show the agent–environment interaction in a finite Markov decision process in figure 2.1.

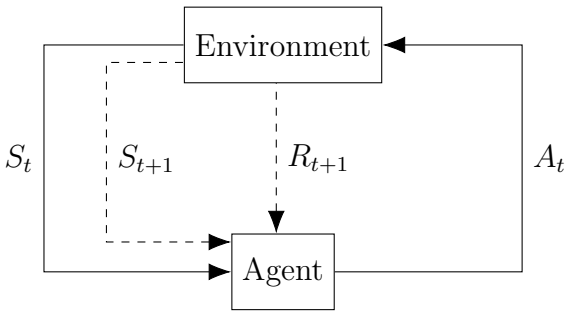


Figure 2.1: The agent–environment interaction in a finite Markov decision process.

One important element of a finite Markov decision process is that its dynamics follow the Markov property. This means that the next state and reward depend only on the current state and action and not on any past states and actions. In other words, each state contains sufficient information about the past interactions between the agent and the environment in order to determine the future.

2.2.2 Episodes, Rewards, and Returns

In this thesis we concentrate on episodic problems. In this family of problems, the agent begins an episode in a state $S_0 \in \mathcal{S}$, which is called the starting state. Additionally, there exists a state $S_T \in \mathcal{S}^+$, which is called the terminal state.[4] If the agent reaches S_T, the episode ends, the environment is reset to S_0, and a new episode begins. The random variable T is the final time step of the episode.

As we stated in the previous section, the agent's goal is to maximize the total reward it receives from the environment. However, the agent's actions can influence

[3] In this thesis we examine the case where all actions are available in each state. In general, there may be actions that are not available in S_t.

[4] In this thesis we assume that $S_T \notin \mathcal{S}$ and denote the set $\mathcal{S} \cup \{S_T\}$ by $\mathcal{S}^+$.

the environment's future reactions. This implies that maximizing only immediate rewards may result in a relatively low total reward in the long run. Therefore, when the agent selects actions, it should take future rewards into account as well.

Specifically, during an episode, the agent should learn to maximize the total expected discounted return. The discounted return at time step t is defined as,

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t-1} R_T \tag{2.3}$$

The discount factor γ controls how much weight the agent assigns to future rewards. As $\gamma \to 1$ the agent takes future rewards more into account, and as $\gamma \to 0$ the agent takes future rewards less into account. In the extreme cases, when $\gamma = 1$ the agent assigns the same weight to all rewards until the end of the episode, and when $\gamma = 0$ the agent only cares for R_{t+1}. Note that the definition of the discounted return in 2.3 can be extended to non-episodic problems by replacing T with ∞.

2.2.3 Policies and Value Functions

The essence of mathematically formalizing a reinforcement learning problem as a finite Markov decision process is for the agent to learn a behavioral rule, which is called a policy. A policy π is defined as,

$$\pi(a \,|\, s) = \Pr\{A_t = a \,|\, S_t = s\}, \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A} \tag{2.4}$$

In other words, π defines a probability distribution over all actions for each state and therefore satisfies,

$$\sum_{a \in \mathcal{A}} \pi(a \,|\, s) = 1, \text{ for all } s \in \mathcal{S} \tag{2.5}$$

Note that π can be deterministic, which means that it can map each state to only one action.

We can evaluate a policy with value functions. The state-value function for policy π is the expected discounted return for being in state $s \in \mathcal{S}$ at time step t and then following π, and it is defined as,

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t \,|\, S_t = s \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \,\middle|\, S_t = s \right], \text{ for all } s \in \mathcal{S} \end{aligned} \tag{2.6}$$

The action-value function for policy π is the expected discounted return for taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$ at time step t and then following π, and it is defined as,

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi \left[G_t \,|\, S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \,\middle|\, S_t = s, A_t = a \right], \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A} \end{aligned} \tag{2.7}$$

Note that both value functions are always zero in the case of the terminal state. From the definitions of the two value functions, it follows that,

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \,|\, s) q_\pi(s, a) \tag{2.8}$$

The two value functions satisfy recursive relationships, which are called the Bellman equations [5]. For the state-value function v_π the Bellman equation is,

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t \,|\, S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \,|\, S_t = s \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a \,|\, s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \,|\, s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} \,|\, S_{t+1} = s' \right] \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a \,|\, s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \,|\, s, a) \left[r + \gamma v_\pi(s') \right] \end{aligned} \tag{2.9}$$

and for the action-value function q_π the Bellman equation is,

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi \left[G_t \,|\, S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \,|\, S_t = s, A_t = a \right] \\ &= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \,|\, s, a) \left[r + \gamma \sum_{a' \in \mathcal{A}} \pi(a' \,|\, s') \mathbb{E}_\pi \left[G_{t+1} \,|\, S_{t+1} = s', A_{t+1} = a' \right] \right] \\ &= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \,|\, s, a) \left[r + \gamma \sum_{a' \in \mathcal{A}} \pi(a' \,|\, s') q_\pi(s', a') \right] \end{aligned} \tag{2.10}$$

The Bellman equations in 2.9 and 2.10 have as unique solutions v_π and q_π respectively, and they are used in many reinforcement learning algorithms in order to learn the two value functions for a policy π.

2.2.4 Optimal Policies and Optimal Value Functions

In reinforcement learning we are more interested in the optimal value functions because they are maximized by a policy π. Specifically, an optimal policy π_* maximizes the state-value function and defines the optimal state-value function as follows,

$$v_*(s) = v_{\pi_*}(s) = \max_\pi v_\pi(s), \text{ for all } s \in \mathcal{S} \tag{2.11}$$

Similarly, the optimal action-value function is defined as,

$$q_*(s, a) = q_{\pi_*}(s, a) = \max_\pi q_\pi(s, a), \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A} \tag{2.12}$$

Note that there may be more than one optimal policy, but v_* and q_* are unique. Since q_* is the expected discounted return for taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$ and then following π_*, it is possible to uncover π_* from q_* by choosing a in s that maximizes q_* with ties broken arbitrarily. Therefore, the following equality holds,

$$v_*(s) = \max_{a \in \mathcal{A}} q_*(s, a) \tag{2.13}$$

The Bellman equations for the optimal value functions are called Bellman optimality equations [5]. For the optimal state-value function v_* this equation is,

$$v_*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \,|\, s, a) \left[r + \gamma v_*(s') \right] \tag{2.14}$$

and for the optimal action-value function q_* this equation is,

$$q_*(s,a) = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma \max_{a' \in \mathcal{A}} q_*(s', a') \right] \tag{2.15}$$

The Bellman optimality equations in 2.14 and 2.15 have as unique solutions the optimal value functions. Knowing the optimal value functions is very powerful, because the agent can use this information to determine an action that ensures the optimal expected discounted return from each state. In the case of the optimal state-value function the agent must perform a one-step lookahead to determine an optimal action, whereas in the case of the optimal action-value function the agent can determine an optimal action without considering the possible next states. Nevertheless, both value functions are useful and are widely used in reinforcement learning.

The reinforcement learning algorithms that are used to learn the optimal value functions can be divided into two main categories. We describe these categories in the next section.

2.3 Solution Methods

In reinforcement learning problems we often seek to compute the optimal value functions in order to uncover an optimal policy. In certain problems we can also compute an optimal policy directly, although we do not discuss this approach in this thesis. Whatever our goal is, the reinforcement learning algorithms that are used to achieve this goal can be divided into two main categories. The first one is tabular solution methods, and the second one is function approximation solution methods.

2.3.1 Tabular Solution Methods

The key idea of tabular solution methods is that the reinforcement learning problems we would like to solve can be formally represented by state and action spaces that are relatively small. If this is the case, it is possible to represent the approximate value functions as lookup tables that store all the values. The values in such a table are then updated as the reinforcement learning algorithm progresses.

The main advantage of tabular solution methods is that they can usually converge to the optimal value functions and learn an optimal policy. In fact, most of the theoretical guarantees we have in reinforcement learning are for tabular solution methods [6, 36].

On the other hand, these methods are computationally infeasible for reinforcement learning problems that are formally represented by large state and/or action spaces. The reasons are that a large amount of memory is required to store the values in the lookup table and that a large amount of time is required for gathering enough samples to sufficiently update all the stored values [36]. Unfortunately, the most interesting problems are usually very complex and cannot be solved with tabular solution methods. Therefore, we have to resort to solution methods that use function approximation.

2.3.2 Function Approximation Solution Methods

We use function approximation solution methods to tackle reinforcement learning problems that are formally represented by large state and/or action spaces. In these problems it is not possible to represent the optimal value functions exactly. Therefore, our goal is to find approximate solutions that are good enough to allow the agent to act in an optimal, or close to optimal, way.

The key idea of function approximation solution methods is that there exist states that are similar to each other. Therefore, the agent could learn how to act well in such states without visiting all of them. In other words, in these methods we try to generalize over the state space from a limited amount of samples using function approximators.

Although function approximation solution methods can help us deal with reinforcement learning problems that are intractable with tabular solution methods, there are limited theoretical guarantees when function approximation is used. In fact, function approximators have been shown to be unstable and to not always converge to the optimal value functions [15].

In supervised learning, we study algorithms that learn from training examples for which the regression or classification targets are provided. Many methods that have been developed in supervised learning can be used in reinforcement learning, acting as function approximators that try to learn the optimal value functions. In the next section we describe one of them, the multilayer perceptron.

2.4 Multilayer Perceptrons

The perceptron was introduced by Rosenblatt in 1958 [26] but was shown to have severe limitations [23] and did not become popular until after some decades. The interest of the artificial intelligence community in the perceptron increased when a multilayer version of the perceptron was introduced [27]. The reason is that the multilayer perceptron overcomes the perceptron's limitations and can train efficiently. Multilayer perceptrons can be used in reinforcement learning as function approximators.

2.4.1 The Perceptron

The perceptron is the basic information unit that is used to construct multilayer perceptrons. It receives information from the input units $x_j \in \mathbb{R},\ j = 1, 2, \ldots, m$, and computes a weighted sum of those inputs to produce an output $\hat{y}$ as follows,

$$\hat{y} = \sum_{j=1}^{m} w_j x_j + w_0 \tag{2.16}$$

where w_0 is a bias weight associated with a bias unit $x_0 = +1$. The output $\hat{y}$ can be used to approximate a solution to a linear regression problem or to discriminate between two classes. Classification can be achieved by using a threshold function g that is defined as,

$$g(\hat{y}) = \begin{cases} 1, & \text{if } \hat{y} \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.17}$$

and subsequently mapping the output of g to the two different classes. We graphically show a perceptron with two input units and one bias unit in figure 2.2.

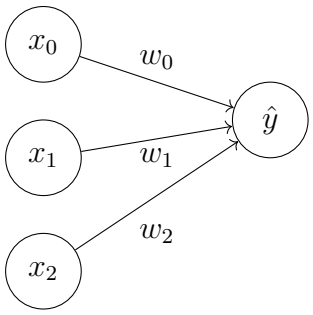


Figure 2.2: A perceptron with two input units and one bias unit.

Since our end goal is to approximate the optimal value functions, we assume a regression problem. To evaluate the quality of the perceptron on one training example, we define an objective function J as follows,

$$J(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2 \tag{2.18}$$

that quantifies how far the output $\hat{y}$ is from the target y. Note that there are many objective functions that can be used, but here we only discuss the one in equation 2.18. After defining an objective function, we initialize all the weights of the perceptron randomly and provide the perceptron with training examples along with their targets. To train the perceptron on one training example, we compute the partial derivative of J with respect to each weight w_j,

$$\frac{\partial J(y, \hat{y})}{\partial w_j} = -(y - \hat{y})x_j \tag{2.19}$$

and we adjust the weights to minimize the error. We update w_j as follows,

$$w_j \leftarrow w_j - \alpha \frac{\partial J(y, \hat{y})}{\partial w_j} \tag{2.20}$$

where α is the learning rate.

2.4.2 Multilayer Perceptrons and Rectified Linear Units

Perceptrons are limited because they cannot solve problems that are not linearly separable, such as the XOR problem [14, 2]. One way to overcome this limitation is to construct a multilayer perceptron, which is a network of perceptron layers (see figure 2.3 for an example). The first layer of the multilayer perceptron is called the input layer, the final layer is called the output layer, and all the layers between the input and output layers are called hidden layers. Note that the basic theory we discussed for the perceptron can be extended to the multilayer perceptron.

One important aspect of multilayer perceptrons is that the output of each unit in a hidden layer is given as an argument to a nonlinear activation function before it is provided as an input to the next layer. This is essential for the multilayer perceptron to function correctly and solve problems that are not linearly separable. If there

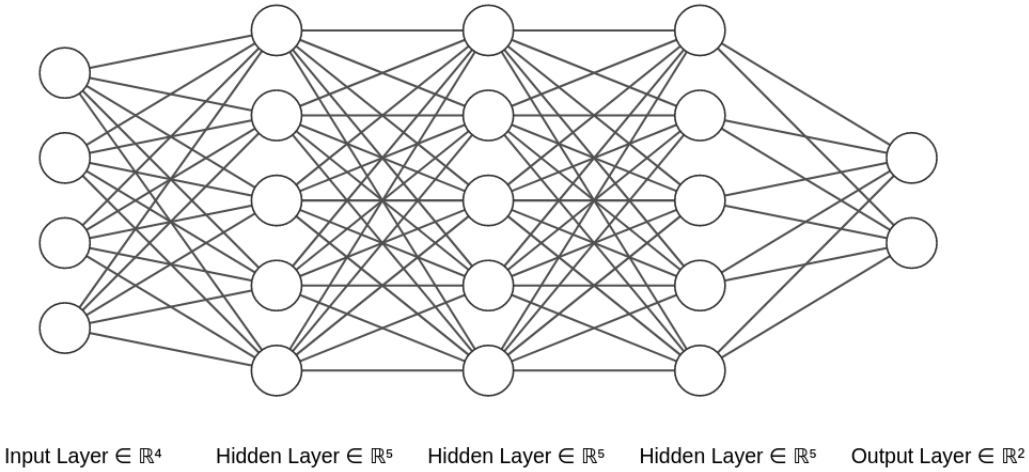


Figure 2.3: A multilayer perceptron with an input layer of four units, three hidden layers of five units, an output layer of two units, and no bias units. Constructed with the software described in [22].

were no nonlinear activation functions in the hidden layers, the multilayer perceptron would be equivalent to a perceptron, because the sum of linear combinations is again a linear combination [2].

In this thesis we describe the rectifier, which is a nonlinear activation function that is widely used in multilayer perceptrons and was also used in our experiments. The units in a multilayer perceptron that use the rectifier activation function are called rectified linear units. The rectifier activation function is defined as,

$$g(x) = \max(0, x) \tag{2.21}$$

This activation function behaves similarly to a linear function and therefore makes the optimization of models that use it easier [14]. The derivative of g is defined as,

$$\frac{dg(x)}{dx} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{2.22}$$

Although the derivative of g is undefined for $x = 0$, in computational problems we can overcome this issue by adopting a convention for the value of the derivative when $x = 0$. Furthermore, the fact that the derivative of g is equal to zero or one for every $x \neq 0$ is very convenient. The reason is that it facilitates the most widely used algorithm for training the multilayer perceptron, especially when the multilayer perceptron has many layers.

2.4.3 Backpropagation and Stochastic Gradient Descent

Multilayer perceptrons are usually trained using the backpropagation algorithm combined with stochastic gradient descent [27]. As in the case of the perceptron, we assume the objective function J defined in equation 2.18, initialize all the weights of the multilayer perceptron randomly, provide the multilayer perceptron with training examples and their targets, and the multilayer perceptron computes an output for each training example. This procedure is called forward propagation because

the information for each training example is transmitted through the multilayer perceptron in the forward direction.

The first part of the training algorithm of the multilayer perceptron is called backpropagation because the error information for each training example is transmitted through the multilayer perceptron in the backward direction. This is done to compute the gradient of J with respect to the weight vector $\boldsymbol{w} \in \mathbb{R}^m$, which is defined as,

$$\nabla_{\boldsymbol{w}} J(y, \hat{y}) = \begin{bmatrix} \frac{\partial J(y,\hat{y})}{\partial w_1} & \frac{\partial J(y,\hat{y})}{\partial w_2} & \dots & \frac{\partial J(y,\hat{y})}{\partial w_m} \end{bmatrix}^T \tag{2.23}$$

where m is the number of all the weights of the multilayer perceptron.

Backpropagation computes the partial derivatives in equation 2.23 using the chain rule of calculus. The algorithm operates in the backward direction to avoid computing intermediate terms in the chain rule more than once. Although this is an efficient way to compute the gradient, the partial derivatives that correspond to weights in the earlier layers require the chain rule to be applied many times. This can cause the gradient to vanish or explode due to repeated multiplication with numbers that are less than or greater than one in magnitude respectively [14]. The rectifier activation function partially overcomes this problem because its derivative is either zero or one, assuming that the critical point $x = 0$ is handled.

After computing the gradient in equation 2.23, we update each weight of the multilayer perceptron using the update rule in equation 2.20. This part of the training algorithm is called stochastic gradient descent. It is called stochastic because the gradient is computed on only one training example and is therefore an expectation of the true gradient that would be computed on all training examples. It is called gradient descent because we are moving in the direction opposite to the gradient of the objective function in order to minimize the error.

2.5 Overestimation Bias

Overestimation bias is an inherent property of reinforcement learning algorithms that approximate maximum expected values by maximizing uncertain estimates. Some examples of such algorithms are Q-Learning [46] and Sarsa [28].

The effect of overestimation bias on reinforcement learning algorithms is generally considered to be negative [39, 40, 16, 41, 42, 12], and there have been many attempts to reduce or remove overestimation bias [40, 16, 20, 8, 41, 3, 21]. Moreover, the combination of overestimation bias and function approximation has been shown to cause learning failure under certain conditions [39]. However, a recent study suggests that overestimation bias may have either a negative or positive effect on reinforcement learning algorithms depending on the reinforcement learning problem [19].

2.5.1 The Single Estimator Approach

As shown by Van Hasselt [40, 16], overestimation bias can be formally presented as follows. Let $X = \{X_1, X_2, \dots, X_n\}$ be a set of n random variables. Assume that we seek to estimate $\max_{i \in \{1,2,\dots,n\}} \mathbb{E}[X_i]$.

However, the underlying distributions of the variables $X_i \in X$ are unknown. Therefore, we gather a set of samples $D = \bigcup_{i=1}^{n} D_i$, where each D_i contains samples of X_i, and construct estimators for the expected values by computing the sample mean for each variable,

$$\mathbb{E}[X_i] = \mathbb{E}[\mu_i] \approx \mu_i := \frac{1}{|D_i|} \sum_{x_i \in D_i} x_i \tag{2.24}$$

where μ_i is the estimator corresponding to X_i. Assuming that the samples in each D_i are independent and identically distributed, μ_i is an unbiased estimator of $\mathbb{E}[X_i]$.

One way to approximate the maximum expected value is to use the max operator on μ_i as follows,

$$\max_{i \in \{1,2,\dots,n\}} \mathbb{E}[X_i] = \max_{i \in \{1,2,\dots,n\}} \mathbb{E}[\mu_i] \approx \max_{i \in \{1,2,\dots,n\}} \mu_i \tag{2.25}$$

This approach is called the single estimator approach because only one estimator is used for each variable. However, it does not provide an unbiased estimate of the maximum expected value. The reason is that $\max_{i \in \{1,2,\dots,n\}} \mu_i$ is an unbiased estimator of $\mathbb{E}\left[\max_{i \in \{1,2,\dots,n\}} \mu_i\right]$ but a biased estimator of $\max_{i \in \{1,2,\dots,n\}} \mathbb{E}[\mu_i]$ [35]. Van Hasselt showed that this bias is strictly positive when any random variable that corresponds to the maximum expected value has a non-zero probability of not corresponding to the maximum estimator [40, 16]. In general, it holds that,

$$\mathbb{E}\left[\max_{i \in \{1,2,\dots,n\}} \mu_i\right] \geq \max_{i \in \{1,2,\dots,n\}} \mathbb{E}[\mu_i] \tag{2.26}$$

which means that the maximum expected value can be overestimated.

2.5.2 The Double Estimator Approach

As we stated above, the single estimator approach can cause overestimation of the maximum expected value. There is another approach, called the double estimator approach, that can be used to replace overestimation bias with underestimation bias.

As shown by Van Hasselt [40, 16], in the double estimator approach we use two estimators for each random variable. We gather again a set of samples $D = \bigcup_{i=1}^{n} D_i$, where each D_i contains samples of the random variable X_i, and randomly split it into two sets, D_1 and D_2, such that $D = D_1 \cup D_2$ and $D_1 \cap D_2 = \emptyset$. We can now define two sets of estimators, $M_1 = \{\mu_{11}, \mu_{12}, \dots, \mu_{1n}\}$ and $M_2 = \{\mu_{21}, \mu_{22}, \dots, \mu_{2n}\}$. The two estimators for $\mathbb{E}[X_i]$ are defined as,

$$\mu_{1i} := \frac{1}{|D_{1i}|} \sum_{x_i \in D_{1i}} x_i \quad \text{and} \quad \mu_{2i} := \frac{1}{|D_{2i}|} \sum_{x_i \in D_{2i}} x_i \tag{2.27}$$

Since D is split randomly, both estimators in 2.27 are unbiased under the assumption that the samples in D_{1i} and D_{2i} are independent and identically distributed.

In this approach, we first consider the estimators in the set M_1 and determine $* = \arg\max_{i \in \{1,2,\dots,n\}} \mu_{1i}$ with ties broken arbitrarily, and we then approximate the maximum expected value as follows,

$$\max_{i \in \{1,2,\dots,n\}} \mathbb{E}[X_i] = \max_{i \in \{1,2,\dots,n\}} \mathbb{E}[\mu_{2i}] \approx \mu_{2*} \tag{2.28}$$

where μ_{2*} is the estimator for $\mathbb{E}[X_*]$ in the set M_2. Note that in the limit all the estimators converge to the expected values and the approximation in 2.28 becomes an equality. Note also that we can repeat the same procedure by swapping the roles of M_1 and M_2.

This approach removes overestimation bias but does not necessarily provide an unbiased estimate of the maximum expected value. Van Hasselt showed that this bias is strictly negative when μ_{1*} has a non-zero probability of not corresponding to any random variable that corresponds to the maximum expected value [40, 16]. In general, it holds that,

$$\mathbb{E}[\mu_{2*}] \leq \max_{i \in \{1,2,\ldots,n\}} \mathbb{E}[\mu_{2i}] \tag{2.29}$$

which means that the maximum expected value can be underestimated.

2.6 Q-learning

Q-learning was introduced by Watkins [46] and is one of the most widely used reinforcement learning algorithms. It was recently combined with deep neural networks and has been shown to perform successful control in the video game domain [24].

This algorithm tries to compute the optimal action-value function q_* by repeatedly sampling experience tuples of the form $(S_t, A_t, R_{t+1}, S_{t+1})$ and using them to perform updates on action-value estimates.

Q-learning is a model-free algorithm as it does not require a model of the environment. Furthermore, it is an off-policy algorithm because it learns an optimal policy π_* irrespective of the policy π that is followed at each time step t. It is also relatively easy to implement and has a relatively simple update rule.

2.6.1 Tabular Q-learning

In tabular Q-learning we first initialize an approximate action-value function Q arbitrarily and then use a policy based on Q to sample $(S_t, A_t, R_{t+1}, S_{t+1})$ tuples. At each time step t we perform the update,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(S_{t+1}, a') - Q(S_t, A_t)] \tag{2.30}$$

where α is the step size and γ is the discount factor. Note that this update depends on the value of $\max_{a' \in \mathcal{A}} Q(S_{t+1}, a')$ at time step t.

This version of Q-learning converges to the optimal action-value function q_* with probability one [45, 6]. There are two main convergence conditions. The first one is that each state-action pair is sampled infinitely many times in the limit, and the second one is that the step size sequence for each state-action pair satisfies,

$$\sum_{t=0}^{\infty} \alpha_t(s,a) = \infty \quad \text{and} \quad \sum_{t=0}^{\infty} \alpha_t^2(s,a) < \infty, \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A} \tag{2.31}$$

In algorithm 1 we show the pseudocode for tabular Q-learning. Note that the step size α is usually annealed in each step to satisfy the convergence condition in 2.31. Similarly, the exploration parameter ϵ is usually annealed in each step to gradually reduce exploration. This ensures that the policy used for the action selection asymptotically approaches the greedy policy.

Algorithm 1: Tabular Q-learning

Input: step size $\alpha \in (0, 1]$, exploration parameter $\epsilon > 0$
Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
Observe initial state s
while *Agent is interacting with the Environment* **do**
  Choose action a in s using policy based on Q (e.g. ϵ-greedy)
  Take action a, observe r and s'
  $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s, a)\right]$
  $s \leftarrow s'$
end

2.6.2 Q-learning with Function Approximation

Q-learning can be extended to a function approximation solution method. We present the case where a differentiable nonlinear function approximator is used, because we used multilayer perceptrons as function approximators in our experiments.

Assume a differentiable nonlinear function approximator with a weight vector $\boldsymbol{w} \in \mathbb{R}^m$. At each time step t the function approximator takes as input a representation of a state $S_t \in \mathcal{S}$ and outputs a set of action-value estimates for S_t. In other words, the approximate action-value function Q is now parametrized by $\boldsymbol{w}$ and is denoted by $Q(\cdot, \cdot; \boldsymbol{w})$.

We define the target Y_t at time step t as follows,

$$Y_t = R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(S_{t+1}, a'; \boldsymbol{w}) \tag{2.32}$$

where γ is the discount factor. The objective function J can now be defined as,

$$J(Y_t, Q(S_t, A_t; \boldsymbol{w})) = \frac{1}{2} \left[Y_t - Q(S_t, A_t; \boldsymbol{w})\right]^2 \tag{2.33}$$

Notice that the target Y_t depends on the weight vector $\boldsymbol{w}$. In fact, this is called bootstrapping in reinforcement learning and can cause problems when combined with gradient descent methods [36]. However, it is beyond the scope of this thesis and we do not discuss it further.

Having in mind that the next derivation is not a true gradient descent method, we assume that Y_t is independent of $\boldsymbol{w}$ and compute the gradient of J with respect to $\boldsymbol{w}$. We then perform the update,

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[Y_t - Q(S_t, A_t; \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w}) \tag{2.34}$$

where α is the learning rate.

The pseudocode for Q-learning with function approximation is shown in algorithm 2. Note that the exploration parameter ϵ is usually annealed in each step for the same reason as in tabular Q-learning. On the other hand, the learning rate α is usually set to a relatively low value before the algorithm is executed and remains fixed during learning.

2.6.3 The Overestimation Bias of Q-learning

As we mentioned before, Q-learning is model-free, off-policy, and relatively easy to implement. Furthermore, it has a relatively simple update rule and a convergence

Algorithm 2: Q-learning with Function Approximation

Input: learning rate $\alpha > 0$, exploration parameter $\epsilon > 0$
Initialize weight vector $\boldsymbol{w} \in \mathbb{R}^m$ arbitrarily
Observe initial state s
while *Agent is interacting with the Environment* **do**
  Choose action a in s using policy based on Q (e.g. ϵ-greedy)
  Take action a, observe r and s'
  $y \leftarrow r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \boldsymbol{w})$
  $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[y - Q(s, a; \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} Q(s, a; \boldsymbol{w})$
  $s \leftarrow s'$
end

proof for its tabular version. However, Q-learning has overestimation bias [39, 40, 16], and this has been repeatedly shown to influence the learning procedure [39, 40, 16, 41].

Q-learning was mathematically proven to overestimate the optimal action values under mild assumptions by Thrun and Schwartz [39]. A less limiting proof followed by Van Hasselt [40, 16], and it was also proven in actor-critic methods by Fujimoto et al. [12].

To understand why Q-learning can overestimate, assume that the tabular version of the algorithm updates $Q(S_t, A_t)$ at a certain time step t by estimating the optimal value of the next state $S_{t+1} \in \mathcal{S}$ as follows,

$$V(S_{t+1}) = \max_{a' \in \mathcal{A}} Q(S_{t+1}, a') \tag{2.35}$$

However, the action values of S_{t+1} are not the optimal action values because the algorithm is still in the process of learning. Assume that a noise term $e(S_{t+1}, \cdot)$ is associated with $Q(S_{t+1}, \cdot)$ as follows,

$$Q(S_{t+1}, a) = q_*(S_{t+1}, a) + e(S_{t+1}, a), \text{ for all } a \in \mathcal{A} \tag{2.36}$$

The noise term e can be there for many reasons, such as stochastic transitions, stochastic rewards, function approximation, non-stationary environment, or a combination of these. The core idea is that there is some source of random approximation error that can be either positive or negative. Since the algorithm uses the max operator over all the actions in state S_{t+1} as shown in equation 2.35, the optimal value of S_{t+1} can be overestimated due to positive noise.

Another way to analyze the overestimation bias of Q-learning is to think of the algorithm as using the single estimator approach to determine the maximum optimal action value of the next state. In other words, $\max_{a' \in \mathcal{A}} Q(S_{t+1}, a')$ tries to approximate $\max_{a' \in \mathcal{A}} \mathbb{E}\left[Q(S_{t+1}, a')\right]$. However, we know that $\max_{a' \in \mathcal{A}} Q(S_{t+1}, a')$ is an unbiased estimator of $\mathbb{E}\left[\max_{a' \in \mathcal{A}} Q(S_{t+1}, a')\right]$. From the analysis of the single estimator approach we know that,

$$\mathbb{E}\left[\max_{a' \in \mathcal{A}} Q(S_{t+1}, a')\right] \geq \max_{a' \in \mathcal{A}} \mathbb{E}\left[Q(S_{t+1}, a')\right] \tag{2.37}$$

In figure 2.4 we show an episodic finite Markov decision process that was inspired by [36] to examine the negative effect of overestimation bias on Q-learning. In

this process there are two non-terminal states, s_0 and s_1, and the terminal state is depicted by gray squares. The starting state is s_0 and there are two possible actions the agent can take in s_0. If the agent takes the action a_2, it causes a deterministic transition to the terminal state with a deterministic reward of zero and the episode ends. If the agent takes the action a_1, it causes a deterministic transition to s_1 with a deterministic reward of zero and the episode continues. In s_1 there are four possible actions that cause a deterministic transition to the terminal state. The rewards received for those actions are drawn from a Normal distribution with mean $\mu = -0.1$ and standard deviation $\sigma = 0.5$. If the discount factor γ is set to one, the expected return for any possible trajectory that begins with a_1 is -0.1, whereas the expected return for taking a_2 is zero. Therefore, the optimal policy π_* is to always choose a_2 in s_0 and end the episode in one time step. However, a Q-learning agent following an ϵ-greedy policy could choose a_1 many times in the beginning of learning, because it would potentially overestimate the optimal value of s_1.

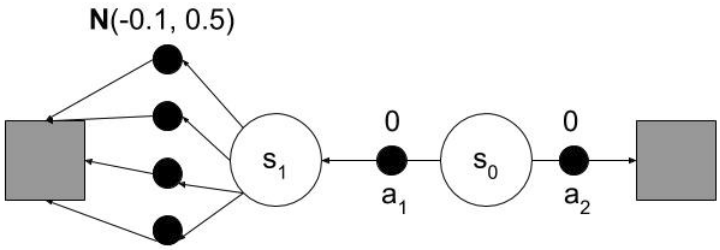


Figure 2.4: An episodic finite Markov decision process to highlight the problems caused by overestimation bias. The starting state is s_0 and the terminal state is depicted by gray squares. All the transitions are deterministic and the rewards are shown above the actions.

2.7 Double Q-learning

Double Q-learning was proposed by Van Hasselt [40, 16] to remove the overestimation bias of Q-learning. Similarly to Q-learning, it has been recently shown to perform successful control in the video game domain when combined with deep neural networks [41].

The main idea of this algorithm is to simultaneously update two approximate action-value functions on two disjoint sets of samples of the form $(S_t, A_t, R_{t+1}, S_{t+1})$. When one of the two action-value functions is updated, it is also used to determine the action the maximizes the action values of the next state. However, the maximizing action is evaluated by the other action-value function. This ensures that the optimal value of the next state is not overestimated, although it may be underestimated.

Double Q-learning shares all the advantages of Q-learning that we have already discussed. Furthermore, it replaces overestimation bias with underestimation bias by doubling the memory requirements. The reason is that two approximate action-value functions must be stored instead of one.

2.7.1 Tabular Double Q-learning

In tabular Double Q-learning, we initialize two approximate action-value functions, Q_1 and Q_2, arbitrarily and use a policy based on both of them to sample experience tuples of the form $(S_t, A_t, R_{t+1}, S_{t+1})$. At each time step t we choose to update one of the two action-value functions with equal probability. The update rule for Q_1 at time step t is defined as,

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_2(S_{t+1}, A_*) - Q_1(S_t, A_t)] \tag{2.38}$$

where α is the step size, γ is the discount factor, and $A_* = \arg\max_{a' \in \mathcal{A}} Q_1(S_{t+1}, a')$. Note that Q_2, which is not updated at time step t, is used to evaluate A_*. The update rule for Q_2 at time step t is similar to the one in 2.38 but with the two action-value functions swapped.

Tabular Double Q-learning converges to the optimal action-value function q_* with probability one [40, 16]. The main convergence conditions are similar to those of tabular Q-learning.

We show the pseudocode for tabular Double Q-learning in algorithm 3. Note that the step size α is usually annealed in each step to satisfy the convergence condition in 2.31, and the exploration parameter ϵ is usually annealed in each step to gradually reduce exploration. Note also that to choose an action in each state we use a policy based on the mean of the two approximate action-value functions. This ensures that all the information acquired in the past steps is used for the action selection [40, 16].

Algorithm 3: Tabular Double Q-learning

Input: step size $\alpha \in (0, 1]$, exploration parameter $\epsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$ arbitrarily for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
Observe initial state s
while *Agent is interacting with the Environment* **do**
  Choose action a in s using policy based on $\frac{Q_1 + Q_2}{2}$ (e.g. ϵ-greedy)
  Take action a, observe r and s'
  With probability 0.5:
    $a_* \leftarrow \arg\max_{a' \in \mathcal{A}} Q_1(s', a')$
    $Q_1(s, a) \leftarrow Q_1(s, a) + \alpha\,[r + \gamma Q_2(s', a_*) - Q_1(s, a)]$
  else:
    $a_* \leftarrow \arg\max_{a' \in \mathcal{A}} Q_2(s', a')$
    $Q_2(s, a) \leftarrow Q_2(s, a) + \alpha\,[r + \gamma Q_1(s', a_*) - Q_2(s, a)]$
  $s \leftarrow s'$
end

2.7.2 Double Q-learning with Function Approximation

Double Q-learning can be extended to a function approximation solution method in a similar way to Q-learning. As we mentioned before, we used multilayer perceptrons as function approximators in our experiments. Therefore, we again present the case where a differentiable nonlinear function approximator is used.

Assume two differentiable nonlinear function approximators with two different weight vectors that are denoted by $\boldsymbol{w_1} \in \mathbb{R}^m$ and $\boldsymbol{w_2} \in \mathbb{R}^m$. At each time step

t each function approximator can take as input a representation of a state $S_t \in \mathcal{S}$ and output a set of action-value estimates for S_t. Therefore, the two weight vectors are used to parametrize two different approximate action-value functions that are denoted by $Q_1(\cdot, \cdot; \boldsymbol{w_1})$ and $Q_2(\cdot, \cdot; \boldsymbol{w_2})$.

Similarly to tabular Double Q-learning, at each time step t we choose to update one of the two weight vectors with equal probability. The target Y_t for $\boldsymbol{w_1}$ at time step t is defined as,

$$Y_t = R_{t+1} + \gamma Q_2(S_{t+1}, A_*; \boldsymbol{w_2}) \tag{2.39}$$

where γ is the discount factor and $A_* = \arg\max_{a' \in \mathcal{A}} Q_1(S_{t+1}, a'; \boldsymbol{w_1})$.

Assuming the same objective function J as in equation 2.33, we update $\boldsymbol{w_1}$ as follows,

$$\boldsymbol{w_1} \leftarrow \boldsymbol{w_1} + \alpha \left[Y_t - Q_1(S_t, A_t; \boldsymbol{w_1})\right] \nabla_{\boldsymbol{w_1}} Q_1(S_t, A_t; \boldsymbol{w_1}) \tag{2.40}$$

where α is the learning rate. Note that the update rule in 2.40 is not a true gradient descent method because the target Y_t depends on the weight vector $\boldsymbol{w_1}$. The target and update rule for $\boldsymbol{w_2}$ at time step t are similar to the ones in 2.39 and 2.40 respectively but with the two weight vectors swapped.

In algorithm 4 we show the pseudocode for Double Q-learning with function approximation. Note that to choose an action in each state we use a policy based on one of the two function approximators with equal probability. The weight vector of the function approximator that is used to choose the action is updated in the same step. Note also that the learning rate α is usually set to a relatively low value before the algorithm is executed and does not change during learning, whereas the exploration parameter ϵ is usually annealed in each step to gradually reduce exploration.

Algorithm 4: Double Q-learning with Function Approximation

Input: learning rate $\alpha > 0$, exploration parameter $\epsilon > 0$
Initialize weight vectors $\boldsymbol{w_1} \in \mathbb{R}^m$ and $\boldsymbol{w_2} \in \mathbb{R}^m$ arbitrarily
Observe initial state s
while *Agent is interacting with the Environment* **do**
  With probability 0.5:
    Choose action a in s using policy based on Q_1 (e.g. ϵ-greedy)
    Take action a, observe r and s'
    $a_* \leftarrow \arg\max_{a' \in \mathcal{A}} Q_1(s', a'; \boldsymbol{w_1})$
    $y \leftarrow r + \gamma Q_2(s', a_*; \boldsymbol{w_2})$
    $\boldsymbol{w_1} \leftarrow \boldsymbol{w_1} + \alpha \left[y - Q_1(s, a; \boldsymbol{w_1})\right] \nabla_{\boldsymbol{w_1}} Q_1(s, a; \boldsymbol{w_1})$
  else:
    Choose action a in s using policy based on Q_2 (e.g. ϵ-greedy)
    Take action a, observe r and s'
    $a_* \leftarrow \arg\max_{a' \in \mathcal{A}} Q_2(s', a'; \boldsymbol{w_2})$
    $y \leftarrow r + \gamma Q_1(s', a_*; \boldsymbol{w_1})$
    $\boldsymbol{w_2} \leftarrow \boldsymbol{w_2} + \alpha \left[y - Q_2(s, a; \boldsymbol{w_2})\right] \nabla_{\boldsymbol{w_2}} Q_2(s, a; \boldsymbol{w_2})$
  $s \leftarrow s'$
end

2.7.3 The Underestimation Bias of Double Q-learning

Although Double Q-learning removes overestimation bias, it is not a complete solution to the problem of estimating the maximum optimal action values because it has underestimation bias [40, 16].

To understand why Double Q-learning can underestimate, assume that the tabular version of the algorithm updates the approximate action-value function Q_1 at time step t. Therefore, the optimal value of the next state $S_{t+1} \in \mathcal{S}$ is estimated as follows,

$$V(S_{t+1}) = Q_2(S_{t+1}, A_*) \tag{2.41}$$

where $A_* = \arg\max_{a' \in \mathcal{A}} Q_1(S_{t+1}, a')$. As in the case of Q-Learning, assume that there is some source of random approximation error, and therefore a positive or negative noise term $e_1(S_{t+1}, \cdot)$ is associated with $Q_1(S_{t+1}, \cdot)$ as follows,

$$Q_1(S_{t+1}, a) = q_*(S_{t+1}, a) + e_1(S_{t+1}, a), \text{ for all } a \in \mathcal{A} \tag{2.42}$$

Since the algorithm uses the argmax operator over all the actions in state S_{t+1}, it may be the case that A_* is not the action that maximizes the optimal action values of S_{t+1} due to positive noise. Therefore, the optimal value of S_{t+1} can be underestimated.

Another way to analyze the underestimation bias of Double Q-learning is to think of the algorithm as using the double estimator approach to determine the maximum optimal action value of S_{t+1}. From the analysis of the double estimator approach we know that $Q_2(S_{t+1}, A_*)$ is an unbiased estimator of $\mathbb{E}[Q_2(S_{t+1}, A_*)]$. However, we also know that,

$$\mathbb{E}[Q_2(S_{t+1}, A_*)] \leq \max_{a' \in \mathcal{A}} \mathbb{E}[Q_2(S_{t+1}, a')] \tag{2.43}$$

The episodic finite Markov decision process shown in figure 2.5 was inspired by [16] to examine a case where the underestimation bias of Double Q-learning could be harmful for the algorithm. This process is similar to the one shown in figure 2.4 and we only highlight their differences. Notice that there are now only two possible actions in state s_1. The reward received by the agent for taking the action a_3 is drawn from a Normal distribution with mean $\mu = +0.2$ and standard deviation $\sigma = 0.2$, whereas the reward received by the agent for taking the action a_4 is drawn from a Normal distribution with mean $\mu = -0.2$ and standard deviation $\sigma = 0.2$. If the discount factor γ is set to one, the expected return for taking a_1 and then a_3 is 0.2, which is greater than the expected return for taking the action a_2. Consequently, the optimal action in state s_0 is a_1. However, a Double Q-learning agent following an ϵ-greedy policy could choose a_2 many times in the beginning of learning, because it could underestimate the optimal value of s_1. The reason is that Double Q-learning could use one of the two approximate action-value functions to determine that the suboptimal action a_4 maximizes the optimal action values of s_1 and then evaluate a_4 with the other approximate action-value function. On the other hand, the overestimation bias of Q-learning could be beneficial, because it would potentially allow the agent to visit s_1 many times in the beginning of learning and learn the optimal policy fast.

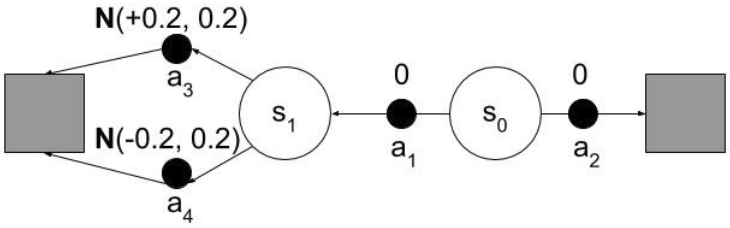


Figure 2.5: An episodic finite Markov decision process to highlight the problems caused by underestimation bias. The starting state is s_0 and the terminal state is depicted by gray squares. All the transitions are deterministic and the rewards are shown above the actions.

2.8 Other Q-learning Variants

After the negative effect of overestimation bias on Q-learning was identified, there have been a number of approaches in the literature to address this issue. The most well-tested and successful method is Double Q-learning, which we have already discussed.

One proposed method that is different from Double Q-learning is Bias-corrected Q-learning [20, 21], in which a bias correction term is subtracted from the update target in each step to remove overestimation bias. However, this bias correction term is computed by taking into account only stochastic transitions and stochastic rewards. There are other sources of approximation error, such as function approximation and non-stationary environment, that cannot be addressed by this method. Furthermore, overestimation bias can facilitate the learning algorithm under certain conditions (see figure 2.5 and relevant discussion), and therefore it would be preferable to control it than remove it.

Another proposed method is Weighted Q-learning [8], which computes the value of the next state in the update target in each step as a weighted average of all its action-value estimates. The weight for each action-value estimate approximates the probability that the corresponding action maximizes the optimal action values. However, this method did not outperform Double Q-learning in all the tasks it was tested on. Moreover, similarly to Bias-corrected Q-learning, this algorithm does not provide a way to control its estimation bias.

The next approach proposed two methods, Averaged Q-learning and Ensemble Q-learning, that were directly applied in a deep reinforcement learning setting [3]. In this approach, past action-value estimates of a model, or action-value estimates of different models, are averaged and used in the update target in each step as the value of the next state. Consequently, the overestimation bias and estimation variance of the algorithms are lower than those of Q-learning. However, overestimation bias is never reduced to zero because the average operator is applied to a finite number of approximate action-value functions. Moreover, similarly to the methods we discussed above, Averaged Q-learning and Ensemble Q-learning do not provide a way to control estimation bias.

One proposed method that attempts to find a balance between overestimation and underestimation is Weighted Double Q-learning [48], which uses a weighted version of Q-learning and Double Q-learning to compute the maximum action value of the next state in the update target in each step. This method was also extended

to a multi-agent deep reinforcement learning algorithm [49]. Although this method provides functionality to control its estimation bias, it cannot underestimate more than Double Q-learning or overestimate more than Q-learning.

A more recent proposed method is Maxmin Q-learning [19], in which an ensemble of agents is used to learn the optimal action values. Specifically, in this algorithm a number of past experience tuples of the form $(S_t, A_t, R_{t+1}, S_{t+1})$ are stored in a replay buffer. In each step a minibatch of past experiences is randomly sampled from the replay buffer and is used to update the action-value estimates of one or more agents. For each experience in the minibatch, all the agents compute an estimate for the maximum optimal action value of the next state, and the minimum of those estimates is used in the corresponding update target. The update procedure starts after all the update targets are determined for all experiences. The authors proposed this method because they identified that underestimation bias may be preferable to overestimation bias and vice versa depending on the reinforcement learning problem, and they showed that the estimation bias of the algorithm can be controlled by tweaking the number of agents. However, although the algorithm can underestimate more than Double Q-learning, there is a limit to its underestimation. Moreover, the algorithm cannot overestimate more than Q-learning. Furthermore, this method requires more computational resources than any other method we discussed above. The reasons are that a number of past experience tuples must be stored in memory, that more than one updates is performed in each step, and that the number of approximate action-value functions increases as the number of agents increases.

Although some of the methods we discussed above outperformed Double Q-learning in some tasks, this difference in performance seems to be task-specific. Therefore, we need more empirical results to conclude that they are useful alternatives to Double Q-learning.

2.9 Conclusion

In this chapter we presented the theoretical background that guided this study. First we described the relevant concepts and tools, and then we defined overestimation bias and showed how it can be replaced with underestimation bias. Subsequently, we presented Q-learning, showed that it has overestimation bias, and examined the negative effect that overestimation bias can have on this algorithm. After that, we presented Double Q-learning, showed how it replaces overestimation bias with underestimation bias, and examined the negative effect that underestimation bias can have on this algorithm. Finally, we presented a number of Q-learning variants other than Double Q-learning that were proposed to overcome the negative effect of overestimation bias on Q-learning.

The main conclusion of this chapter is that overestimation bias may have either a negative or positive effect on reinforcement learning algorithms depending on the reinforcement learning problem. We examined a case in which underestimation bias is preferable to overestimation bias and vice versa. Although most of the methods that were proposed to overcome the negative effect of overestimation bias on Q-learning reduce or remove overestimation bias, there are some methods that control estimation bias. However, the methods that control estimation bias still have disadvantages and are not the majority.

Chapter 3

Variation-resistant Q-learning

In this chapter we propose a new method to control and utilize estimation bias for better performance. The algorithm we propose is called Variation-resistant Q-learning. Similarly to Q-learning, this new algorithm tries to compute the optimal action-value function q_* by repeatedly sampling experience tuples of the form $(S_t, A_t, R_{t+1}, S_{t+1})$ and using them to update an approximate action-value function. However, in this algorithm a number of past action-value estimates are stored in memory. The capacity of this memory is a hyperparameter that is determined before the algorithm is executed. In each step the algorithm uses an update rule similar to the one of Q-learning, but the maximum action value of the next state in the update target is translated by a positive or negative quantity. This quantity is called the variation quantity and is proportional to the mean absolute deviation of the stored past estimates of the maximum action value of the next state. The constant of proportionality in the variation quantity is called the variation resistance parameter and is also a hyperparameter that is determined before the algorithm starts learning. The variation resistance parameter affects the magnitude and determines the sign of the variation quantity.

Variation-resistant Q-learning is based on the principle that applying the max operator on uncertain action-value estimates can cause overestimation. Specifically, the probability and amount of overestimation are expected to increase as the number of actions that correspond to uncertain action-value estimates in each state increases [16, 42]. When there exists any possibility of overestimation, the algorithm increases or decreases the values of uncertain action-value estimates in the update targets in order to introduce systematic overestimation or underestimation respectively. Therefore, the variation resistance parameter can control the estimation bias of the algorithm by affecting the magnitudes and determining the signs of the variation quantities.

Variation-resistant Q-learning controls estimation bias in a qualitatively different way than the other methods that control estimation bias (see section 2.8). The algorithm does not merely increase the probability of overestimation or underestimation, but ensures estimation bias of a certain magnitude and direction by translating the values of uncertain action-value estimates in the update targets. Consequently, Variation-resistant Q-learning influences the agent's exploration behavior in a more direct way than the other methods that control estimation bias. Specifically, since overestimation bias encourages exploration of overestimated actions and underesti-

mation bias discourages exploration of underestimated actions,[1] the magnitudes and signs of the variation quantities determine whether the agent is encouraged or discouraged from exploring states that correspond to uncertain action-value estimates and by how much. Therefore, the variation resistance parameter can influence the agent's exploration behavior.

In the remainder of this chapter we first present the tabular version of the algorithm and then present the function approximation version of the algorithm. Finally, we conclude this chapter with a discussion about the new method.

3.1 Tabular Variation-resistant Q-learning

In tabular Variation-resistant Q-learning we initialize an approximate action-value function Q arbitrarily, and we also initialize a memory with capacity $n > 1$ for each action value. We then use a policy based on Q to sample experience tuples of the form $(S_t, A_t, R_{t+1}, S_{t+1})$. At each time step t we first compute the value of the next state as follows,

$$V(S_{t+1}) = Q(S_{t+1}, A_*) + \lambda\sigma_\kappa(S_{t+1}, A_*) \tag{3.1}$$

where $A_* = \arg\max_{a' \in \mathcal{A}} Q(S_{t+1}, a')$, $\lambda \neq 0$ is the variation resistance parameter, $0 \leq \kappa \leq n$ is the number of past values of $Q(S_{t+1}, A_*)$ that are in memory at time step t, and $\sigma_\kappa(S_{t+1}, A_*)$ is the mean absolute deviation of those κ past values. The mean absolute deviation σ_κ is defined as,

$$\sigma_\kappa(s,a) = \begin{cases} 0, & \text{if } \kappa = 0 \\ \dfrac{\sum_{i=1}^{\kappa} \left| Q_i(s,a) - \overline{Q}_\kappa(s,a) \right|}{\kappa}, & \text{otherwise} \end{cases} \tag{3.2}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. The mean of the κ past values of $Q(s,a)$ is denoted by $\overline{Q}_\kappa(s,a)$ and is defined as,

$$\overline{Q}_\kappa(s,a) = \frac{\sum_{i=1}^{\kappa} Q_i(s,a)}{\kappa} \tag{3.3}$$

Having computed the value of the next state, we perform the update,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - Q(S_t, A_t)] \tag{3.4}$$

where α is the step size and γ is the discount factor. After performing the update, the new value of $Q(S_t, A_t)$ is stored in memory. If there are already n past values of $Q(S_t, A_t)$ stored in memory, the oldest of those values is discarded.

Note that the action-value memory capacity n should be set to an appropriate value in order to allow the algorithm to discard information about past action-value estimates that are outdated. The reason is that the variation quantities should start decreasing when the action-value estimates start becoming better. Note also

[1] A necessary condition for overestimation bias to encourage exploration of overestimated actions and underestimation bias to discourage exploration of underestimated actions is that the algorithm uses a partially greedy policy for action selection (e.g. ϵ-greedy). In this thesis we assume that this condition is satisfied.

that the variation resistance parameter λ can be set to a value greater than one in magnitude if required by the reinforcement learning problem.

In appendix A we present and prove a convergence theorem for tabular Variation-resistant Q-learning. The main convergence conditions are similar to those of tabular Q-learning.

We show the pseudocode for tabular Variation-resistant Q-learning in algorithm 5. In our experiments with this version of the algorithm the step size α was annealed in each step to satisfy the step size sequence convergence condition and the exploration parameter ϵ was annealed in each step to gradually reduce exploration.

Algorithm 5: Tabular Variation-resistant Q-learning

Input: step size $\alpha \in (0, 1]$, exploration parameter $\epsilon > 0$, action-value memory capacity $n > 1$, variation resistance parameter $\lambda \neq 0$

Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$

Initialize memory with capacity n for each $Q(s, a)$

Observe initial state s

while *Agent is interacting with the Environment* **do**

- Choose action a in s using policy based on Q (e.g. ϵ-greedy)
- Take action a, observe r and s'
- $a_* \leftarrow \arg\max_{a' \in \mathcal{A}} Q(s', a')$
- $V(s') \leftarrow Q(s', a_*) + \lambda \sigma_\kappa(s', a_*)$
- $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma V(s') - Q(s, a) \right]$
- Store $Q(s, a)$ in memory
- $s \leftarrow s'$

end

3.2 Variation-resistant Q-learning with Function Approximation

Variation-resistant Q-learning can be extended to a function approximation solution method. As we mentioned in the previous chapter, we used multilayer perceptrons as function approximators in our experiments. For this reason, we again present the case where a differentiable nonlinear function approximator is used.

Assume a differentiable nonlinear function approximator with a weight vector $\boldsymbol{w} \in \mathbb{R}^m$ that is used to parametrize the approximate action-value function $Q(\cdot, \cdot; \boldsymbol{w})$ and the absolute deviations of the action-value estimates $\sigma(\cdot, \cdot; \boldsymbol{w})$. In other words, at each time step t the function approximator takes as input a representation of a state $S_t \in \mathcal{S}$ and outputs a set of action-value estimates for S_t along with their absolute deviations.

Since the function appoximator is not only trained to predict the action-value estimates but is also trained to predict their absolute deviations, we define two targets at each time step t. The first target Y_t at time step t is defined as,

$$Y_t = R_{t+1} + \gamma \left[Q(S_{t+1}, A_*; \boldsymbol{w}) + \lambda \sigma(S_{t+1}, A_*; \boldsymbol{w}) \right] \tag{3.5}$$

where γ is the discount factor, $A_* = \arg\max_{a' \in \mathcal{A}} Q(S_{t+1}, a'; \boldsymbol{w})$, and $\lambda \neq 0$ is the variation resistance parameter. The second target Y_t' at time step t is defined as,

$$Y_t' = |Y_t - Q(S_t, A_t; \boldsymbol{w})| \tag{3.6}$$

We define the objective function J as follows,

$$J(\boldsymbol{Y}_t, \hat{\boldsymbol{Y}}_t) = \frac{1}{2}\left[Y_t - Q(S_t, A_t; \boldsymbol{w})\right]^2 + \frac{1}{2}\left[Y_t' - \sigma(S_t, A_t; \boldsymbol{w})\right]^2 \tag{3.7}$$

where the random vectors $\boldsymbol{Y}_t$ and $\hat{\boldsymbol{Y}}_t$ are defined as,

$$\boldsymbol{Y}_t = \begin{bmatrix} Y_t & Y_t' \end{bmatrix}^T \quad \text{and} \quad \hat{\boldsymbol{Y}}_t = \begin{bmatrix} Q(S_t, A_t; \boldsymbol{w}) & \sigma(S_t, A_t; \boldsymbol{w}) \end{bmatrix}^T \tag{3.8}$$

To perform an update at time step t, we assume that the target vector $\boldsymbol{Y}_t$ does not depend on the weight vector $\boldsymbol{w}$ and compute the gradient of J with respect to $\boldsymbol{w}$ as follows,

$$\begin{aligned} \nabla_{\boldsymbol{w}} J(\boldsymbol{Y}_t, \hat{\boldsymbol{Y}}_t) = & -\left[Y_t - Q(S_t, A_t; \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w}) \\ & -\left[Y_t' - \sigma(S_t, A_t; \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \sigma(S_t, A_t; \boldsymbol{w}) \end{aligned} \tag{3.9}$$

and we update $\boldsymbol{w}$ with the following update rule,

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{Y}_t, \hat{\boldsymbol{Y}}_t) \tag{3.10}$$

where α is the learning rate. Note that the update rule in 3.10 is not a true gradient descent method because the target vector $\boldsymbol{Y}_t$ depends on the weight vector $\boldsymbol{w}$.

Although this version of Variation-resistant Q-learning does not have an action-value memory, the algorithm can still start decreasing the variation quantities when the action-value estimates start becoming better. The reason is that the function approximator adjusts its predictions for the absolute deviations of the action-value estimates as new information is provided. From our experience, this version of the algorithm seems to be more sensitive to the value of the variation resistance parameter λ, and selecting $|\lambda| < 1$ may provide better empirical results.

In algorithm 6 we show Variation-resistant Q-learning with function approximation in pseudocode. In our experiments with this version of the algorithm, the learning rate α was set to a relatively low value before the algorithm was executed and remained fixed during learning, whereas the exploration parameter ϵ was annealed in each step to gradually reduce exploration.

Algorithm 6: Variation-resistant Q-learning with Function Approximation

Input: learning rate $\alpha > 0$, exploration parameter $\epsilon > 0$, variation resistance parameter $\lambda \neq 0$

Initialize weight vector $\boldsymbol{w} \in \mathbb{R}^m$ arbitrarily

Observe initial state s

while *Agent is interacting with the Environment* **do**

- Choose action a in s using policy based on Q (e.g. ϵ-greedy)
- Take action a, observe r and s'
- $a_* \leftarrow \arg\max_{a' \in \mathcal{A}} Q(s', a'; \boldsymbol{w})$
- $y \leftarrow r + \gamma\left[Q(s', a_*; \boldsymbol{w}) + \lambda\sigma(s', a_*; \boldsymbol{w})\right]$
- $y' \leftarrow |y - Q(s, a; \boldsymbol{w})|$
- $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{y}, \hat{\boldsymbol{y}})$
- $s \leftarrow s'$

end

3.3 Discussion

We first discuss why Variation-resistant Q-learning uses mean absolute deviation as a measure of statistical dispersion. Note that the algorithm can be seen as applying a translation operator on the maximum action value of the next state in the update target in each step. The variation quantity is used for the translation and depends on the measure of dispersion used by the algorithm. Therefore, variance would not be a suitable measure of dispersion, because its magnitude would result in unrealistically high variation quantities. Moreover, standard deviation would also not be a suitable measure of dispersion, because it would assign more weight to past action-value estimates that are statistical outliers. To understand why this could be a problem, think of a case where an action-value estimate is close to the optimal value in general, but an extremely rare transition causes an extreme change in its value. In this case, standard deviation would be affected by the outlier and the variation quantity would be higher than desired. On the other hand, mean absolute deviation is less sensitive to outliers and therefore makes the variation quantity more robust.

We now examine how the variation resistance parameter can control the estimation bias of Variation-resistant Q-learning. Assume that the tabular version of the algorithm is used to solve a reinforcement learning problem. Notice that at a certain time step t we have that,

$$V(S_{t+1}) = Q(S_{t+1}, A_*) + \lambda\sigma_\kappa(S_{t+1}, A_*) \tag{3.11}$$

where A_*, λ, κ, and σ_κ are defined in the same way as in equation 3.1. Notice that $\sigma_\kappa(S_{t+1}, A_*) \geq 0$ by definition. Assume now that $\kappa > 0$ in 3.11, which means that there are past values of $Q(S_{t+1}, A_*)$ in memory. If κ is sufficiently large and $\sigma_\kappa(S_{t+1}, A_*) \approx 0$, the algorithm should compute an estimate close to $q_*(S_{t+1}, A_*)$ and its estimation bias should be close to zero. On the other hand, if the κ past values of $Q(S_{t+1}, A_*)$ are noisy, the estimation bias of the algorithm depends on the value of λ. Specifically, if $\lambda < 0$ and $\sigma_\kappa(S_{t+1}, A_*) > 0$, the following inequality holds,

$$V(S_{t+1}) < Q(S_{t+1}, A_*) \tag{3.12}$$

which means that the algorithm should overestimate less than Q-learning. Symmetrically, if $\lambda > 0$ and $\sigma_\kappa(S_{t+1}, A_*) > 0$, the following inequality holds,

$$V(S_{t+1}) > Q(S_{t+1}, A_*) \tag{3.13}$$

which means that the algorithm should overestimate more than Q-learning. Notice that the magnitude and direction of the estimation bias of the algorithm depend on λ. Specifically, as $\lambda \to \infty$ Variation-resistant Q-learning can arbitrarily overestimate more than Q-learning, and as $\lambda \to -\infty$ Variation-resistant Q-learning can arbitrarily underestimate more than Double Q-learning. Note that this discussion also holds for the function approximation version of the algorithm.

The main advantage of Variation-resistant Q-learning is that it utilizes estimation bias to influence the agent's exploration behavior for better performance. Consider the following thought experiment that is mentioned in [19] to understand why. Assume an environment with highly stochastic states, which means that the actions in

those states provide highly stochastic rewards and/or cause highly stochastic transitions. If the highly stochastic states are of high value, overestimation bias would encourage the agent to explore those states and learn their optimal values faster. On the other hand, if the highly stochastic states are of low value, underestimation bias would encourage the agent to ignore those states and move faster to the highly valued states. Taking this thought experiment one step further, think of an environment where highly valued states are blocked by highly stochastic states. The only way for the agent to reach the highly valued states would be through the highly stochastic states, and overestimation bias could speed up this process. Symmetrically, if an environment has lowly valued states that are blocked by highly stochastic states, we would prefer the agent to ignore those regions of the state space and move on. In this case, underestimation bias would be preferable to overestimation bias.

We now consider what happens when there are other sources of approximation error in the reinforcement learning problem, such as function approximation and non-stationary environment. Variation-resistant Q-learning can deal with these sources of approximation error because it operates directly on the action-value estimates. However, it is more difficult to analyze how estimation bias affects the performance of the algorithm in these cases. For example, think of a problem where the function approximation version of the algorithm is used. Because function approximation is a technique that tries to generalize over the state space, updating the weight vector of the function approximator can change several action-value estimates simultaneously. This makes the variation quantity less reliable and may be the reason that the function approximation version of the algorithm seems to be more sensitive to the value of the variation resistance parameter. Another example would be a problem with a non-stationary environment that changes its rules after the action-value estimates become accurate. In this case, the tabular version of the algorithm would have in memory past action-value estimates that would become unreliable, whereas the function approximation version of the algorithm would predict the absolute deviations of the action-value estimates incorrectly. This would prevent the algorithm from adapting quickly to the change in the environment.

One disadvantage of Variation-resistant Q-learning is that its tabular version requires sufficient memory to store a number of past action-value estimates. Moreover, the function approximation version of the algorithm must allocate part of the capacity of its function approximator to predict the absolute deviations of the action-value estimates. Consequently, a function approximator with more capacity may be needed for the algorithm to perform well, and this requires more memory. Therefore, Variation-resistant Q-learning has relatively high memory requirements.

Another disadvantage of Variation-resistant Q-learning is that it cannot arbitrarily change its estimation bias during learning. Specifically, the algorithm starts learning with an estimation bias of a certain magnitude and direction, which depends on the variation resistance parameter. As the action-value estimates become better, the estimation bias of the algorithm is gradually reduced. To understand why this could be harmful for the algorithm, think of a problem where the environment is non-stationary and has many highly stochastic states of high value. In this case, overestimation bias could speed up learning. Now assume that at a certain time step t the rules of the environment change, and the highly stochastic states suddenly become lowly valued. We would want the algorithm to detect this change automatically and start underestimating the optimal values, but our method cannot

arbitrarily switch between underestimation and overestimation during learning.

Our main purpose in this chapter was to explain a new method that is based on the principle that estimation bias should be controlled and utilized for better performance. Variation-resistant Q-learning provides functionality to control and utilize its estimation bias in order to influence the agent's exploration behavior for better performance. Furthermore, the algorithm can arbitrarily overestimate more than Q-learning and can arbitrarily underestimate more than Double-Q-learning.

Chapter 4

Experiments

In this chapter we describe the experiments that we conducted in order to compare the performance of Q-learning, Double Q-learning, and Variation-resistant Q-learning. In each experiment we simulated the interaction of three different agents with an environment. Each agent used one of the three algorithms. In the first section we describe the first experiment, in which we used a grid world environment and the tabular versions of the algorithms. In the second section we describe the second experiment, in which we used a similar grid world environment to the one of the first experiment and the function approximation versions of the algorithms. In the third section we describe the third experiment, in which we used a more complex grid world environment than the ones of the other two experiments and the function approximation versions of the algorithms. Finally, in the fourth section we give the implementation details of the three experiments.

4.1 Grid World

4.1.1 World Structure

In figure 4.1 we show the environment used in our first experiment, which is a 3×3 grid world. In this world each cell is a different state, and therefore each state can be uniquely represented by the tuple (i, j), where $i, j \in \{1, 2, 3\}$ are the row number and column number of the corresponding cell respectively. In other words, the set of possible states is defined as,

$$\mathcal{S} = \{(i, j) \,|\, i, j \in \{1, 2, 3\}\} \tag{4.1}$$

The agent's starting cell is the bottom left cell and the goal cell is the top right cell. Moreover, the set of possible actions is defined as,

$$\mathcal{A} = \{\text{“left”}, \text{“up”}, \text{“right”}, \text{“down”}\} \tag{4.2}$$

and the four actions in $\mathcal{A}$ match the directions in which the agent can move. At each time step t the agent can choose to take one of the four actions in its current cell and cause a deterministic transition to a neighboring cell. Note that an attempt to move beyond the world's boundaries results in no movement. The agent must move to the goal cell and take any of the four actions in order to end the episode.

Inspired by [40, 16, 8, 48], we used four different reward functions in this experiment, which are described as follows:

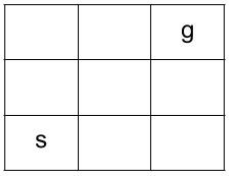


Figure 4.1: A 3×3 grid world. The agent's starting cell is the bottom left cell and the goal cell is the top right cell.

1. **Bernoulli:** The agent receives a reward of +50 or -40 with equal probability for taking an action in the goal cell, and a reward of -12 or +10 with equal probability for taking an action in any other cell.
2. **High-variance Gaussian:** The agent receives a reward of +5 for taking an action in the goal cell, and a reward drawn from a Normal distribution with mean $\mu = -1$ and standard deviation $\sigma = 5$ for taking an action in any other cell.
3. **Low-variance Gaussian:** The agent receives a reward of +5 for taking an action in the goal cell, and a reward drawn from a Normal distribution with mean $\mu = -1$ and standard deviation $\sigma = 1$ for taking an action in any other cell.
4. **Non-terminal Bernoulli:** The agent receives a reward of +5 for taking an action in the goal cell, and a reward of -12 or +10 with equal probability for taking an action in any other cell.

4.1.2 Optimal Values

For all the reward functions described above, the expected reward at time step t is,

$$\mathbb{E}\left[R_{t+1} \mid S_t = s\right] = \begin{cases} +5, & \text{if } s = (1,3) \\ -1, & \text{otherwise} \end{cases} \tag{4.3}$$

Since an optimal policy π_* ends the episode in five actions, the optimal expected reward per time step is,

$$\frac{\sum_{t=0}^{4} \mathbb{E}_{\pi_*}\left[R_{t+1}\right]}{5} = \frac{5 + 4(-1)}{5} = 0.2 \tag{4.4}$$

Assuming that the discount factor γ is set to 0.95, the maximum optimal action value of the starting state is,

$$\max_{a \in \mathcal{A}} q_*((3,1), a) = 5\gamma^4 - \sum_{k=0}^{3} \gamma^k \approx 0.36 \tag{4.5}$$

4.1.3 Normalized Entropy of State Visits

Given a set of state visit counts $N = \{n_1, n_2, \ldots, n_9\}$, where n_i is the state visit count for the state $s_i \in \mathcal{S}$, the empirical probability of s_i is defined as,

$$\hat{p}_i = \frac{n_i}{\sum_{j=1}^{9} n_j} \tag{4.6}$$

The normalized entropy of the state visits can then be defined as,

$$H(\hat{\boldsymbol{p}}) = -\sum_{i=1}^{9} \frac{\hat{p}_i \log_2(\hat{p}_i)}{\log_2(9)} \tag{4.7}$$

where the vector $\hat{\boldsymbol{p}}$ is defined as,

$$\hat{\boldsymbol{p}} = \begin{bmatrix} \hat{p}_1 & \hat{p}_2 & \dots & \hat{p}_9 \end{bmatrix}^T \tag{4.8}$$

Therefore, when $H(\hat{\boldsymbol{p}}) \approx 1$ the state visit counts are approximately equal to each other, and when $H(\hat{\boldsymbol{p}}) \approx 0$ some state visit counts are greater than others.

4.1.4 Hyperparameters

In this experiment we set the discount factor γ to 0.95 and used the tabular versions of the three algorithms. The step size α_t at time step t was defined as,

$$\alpha_t(s,a) = \frac{1}{n_t(s,a)^{0.8}}, \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A} \tag{4.9}$$

where $n_t(s,a)$ is the update count for the action-value estimate of the state-action pair (s,a) at time step t. For Double Q-learning we used one step size for each action-value table at time step t, and the two step sizes were defined in the same way as in equation 4.9. This definition of the step size α satisfies the step size sequence convergence condition for tabular Q-learning [45, 6, 47] and tabular Double Q-learning [40, 16, 47] and has been shown to provide better performance in theory and practice [10].

For the action selection we used an ϵ-greedy policy, in which the exploration parameter ϵ_t at time step t was defined as,

$$\epsilon_t(s) = \frac{1}{\sqrt{n_t(s)}}, \text{ for all } s \in \mathcal{S} \tag{4.10}$$

where $n_t(s)$ is the state visit count for state s at time step t. This definition of the exploration parameter ϵ satisfies the infinite exploration in the limit convergence condition for tabular Q-learning [45, 6] and tabular Double Q-learning [40, 16].

The hyperparameters described above were used in all three algorithms and their choice was guided by previous work [40, 16, 8, 48]. In table 4.1 we provide the algorithm-specific hyperparameters for Variation-resistant Q-learning, which were determined after a manual search in the hyperparameter space. Note that the variation resistance parameter was varied, whereas all other hyperparameters were fixed.

Hyperparameter	**Value(s)**
action-value memory capacity	150
variation resistance parameter	-3, -1.5, -1, -0.5, -0.2, +0.3

Table 4.1: Hyperparameters used in Variation-resistant Q-learning in the grid world experiment.

4.1.5 Evaluation

To evaluate the algorithms, we let each agent interact with the environment and update its action-value estimates for 10,000 time steps. We measured the reward per time step, the maximum action value of the starting state, and the normalized entropy of the state visits. The quantities were averaged over 10,000 simulations.

4.2 Grid World with Function Approximation

4.2.1 World Structure

In figure 4.2 we show the environment used in our second experiment, which is a similar grid world to the one used in our first experiment. The agent's starting cell is the bottom left cell, the goal cell is the top right cell, the transitions are deterministic, an attempt to move beyond the world's boundaries results in no movement, and the action space and reward functions remain the same. However, in this world each cell is not a different state (see section 4.2.2 for an explanation). Moreover, the size of the world is 10×10 instead of 3×3, which makes the task of the agent more difficult because more exploration is needed to discover the goal cell.

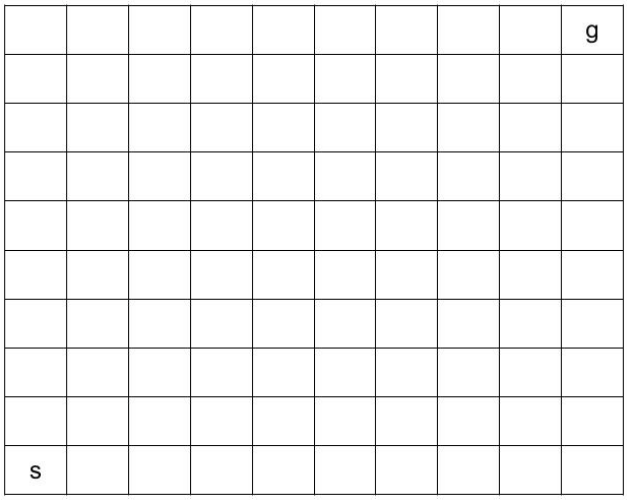


Figure 4.2: A 10×10 grid world. The agent's starting cell is the bottom left cell and the goal cell is the top right cell.

4.2.2 State Representation

In this experiment we used the function approximation versions of the three algorithms and implemented the function approximators as multilayer perceptrons. Therefore, we used a state representation that is a suitable input for multilayer perceptrons. Specifically, at each time step t the current state was represented by a 2×100 matrix $\boldsymbol{M}$. To achieve this, we first defined a mapping f as follows,

$$f(i, j) = i + 10(j - 1) \tag{4.11}$$

that maps a cell tuple (i, j) to a unique integer. The first row of $\boldsymbol{M}$ was then defined by the mapping,

$$g(c) = \begin{cases} 1, & \text{if } c = f(k, l) \\ 0, & \text{otherwise} \end{cases} \tag{4.12}$$

where $c \in \{1, 2, \ldots, 100\}$ is the column number of $\boldsymbol{M}$ and (k, l) is the agent's cell at time step t. Therefore, the first row of $\boldsymbol{M}$ represented the agent's position in the grid. The second row of $\boldsymbol{M}$ was defined by the mapping,

$$h(c) = \begin{cases} 1, & \text{if } c = f(1, 10) \\ 0, & \text{otherwise} \end{cases} \tag{4.13}$$

where $c \in \{1, 2, \ldots, 100\}$ is the column number of $\boldsymbol{M}$. Therefore the second row of $\boldsymbol{M}$ represented the position of the goal cell in the grid. Note that in principle representing the position of the goal cell is not necessary to increase performance, since it remains the same across episodes. However, this is a representation that is easily generalized to an arbitrary position of the goal cell.

4.2.3 Optimal Values

Note that equation 4.3 still holds, because the reward functions used in this experiment are identical to the ones used in the first experiment. Since an optimal policy π_* ends the episode in 19 actions, the optimal expected reward per time step is,

$$\frac{\sum_{t=0}^{18} \mathbb{E}_{\pi_*}[R_{t+1}]}{19} = \frac{5 + 18(-1)}{19} \approx -0.68 \tag{4.14}$$

Assuming that the discount factor γ is set to 0.99, the maximum optimal action value of all visited states per time step is,

$$\frac{\sum_{t=0}^{18} \max_{a \in \mathcal{A}} q_*(S_t, a)}{19} = \frac{\sum_{t=0}^{18} \mathbb{E}_{\pi_*}[G_t]}{19} \approx -3.94 \tag{4.15}$$

Under the same assumption, the maximum optimal action value of the starting state s_0 is,

$$\max_{a \in \mathcal{A}} q_*(s_0, a) = 5\gamma^{18} - \sum_{k=0}^{17} \gamma^k \approx -12.38 \tag{4.16}$$

4.2.4 Hyperparameters

In table 4.2 we report the hyperparameters used in this experiment, which were determined after a manual search in the hyperparameter space. Similarly to the first experiment, all hyperparameters other than the variation resistance parameter were fixed and used in all three algorithms, whereas the variation resistance parameter was varied. Note that in this experiment the policy used for the action selection was again ϵ-greedy, but ϵ was linearly annealed from the initial exploration value to the final exploration value based on the exploration decay steps value. Moreover, we used rectified linear units in the hidden layers of the multilayer perceptrons, and all the weights of the multilayer perceptrons were randomly initialized using the Glorot uniform initializer [13].

Hyperparameter	Value(s)
discount factor	0.99
initial exploration	0.5
final exploration	0.01
exploration decay steps	150,000
learning rate	0.0025
number of hidden layers	1
number of hidden layer nodes	512
variation resistance parameter	-0.5, -0.3, -0.1, -0.05, +0.05, +0.1

Table 4.2: Hyperparameters used in the grid world with function approximation experiment.

4.2.5 Evaluation

To evaluate the algorithms, we let each agent interact with the environment and update the weights of its function approximator(s) for 250,000 time steps. We measured the reward per time step, the maximum action value of all visited states per time step, and the maximum action value of the starting state per time step. We repeated this procedure five times with five different random seeds and computed the median of each quantity over the seeds. As a measure of uncertainty for each quantity, we computed the interval between the mean of the two greatest values and the mean of the two least values.

4.3 Package Grid World

4.3.1 World Structure

In figure 4.3 we show the environment used in our third experiment, which is a more complex grid world than the ones used in our other two experiments. As in the worlds used in the other two experiments, the agent's starting cell is the bottom left cell, the transitions are deterministic, and an attempt to move beyond the world's boundaries results in no movement. As in the world used in the second experiment, the size of the world is 10×10 and each cell is not a different state (see section 4.3.2 for an explanation). However, this world has no goal cell and has five cells that contain packages. Furthermore, the set of possible actions is now defined as,

$$\mathcal{A} = \{\text{“left”}, \text{“up”}, \text{“right”}, \text{“down”}, \text{“collect”}\} \tag{4.17}$$

In addition to the four actions that cause transitions to neighboring cells, at each time step t the agent can also choose to take the action “collect”. This action results in no movement, and if the agent's cell contains a package which is active (i.e. exists in the world), the package is collected (i.e. removed from the world). The agent must collect all five packages in order to end the episode. This can be seen as a variation of the travelling salesman problem, as the agent must determine the shortest path that begins from its starting cell and visits all the cells that contain packages.

We used a deterministic reward function in this experiment, in which the agent receives a reward of +100 for collecting all the packages, and a reward of -1 per time step otherwise.

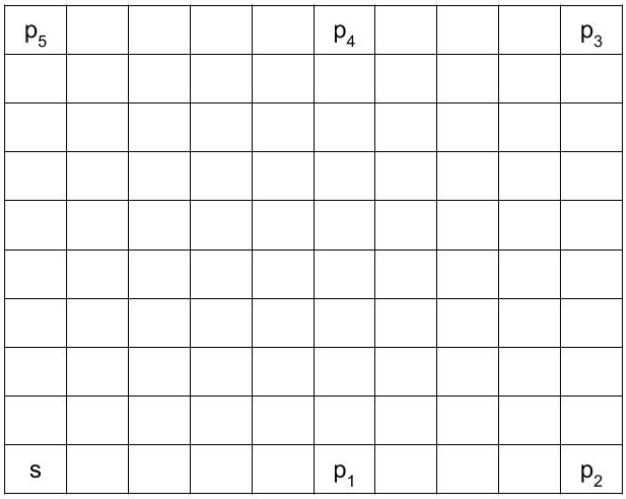


Figure 4.3: A 10×10 package grid world. The agent's starting cell is the bottom left cell and there are five cells that contain packages along the walls of the grid.

4.3.2 State Representation

In this experiment we again used the function approximation versions of the three algorithms and implemented the function approximators as multilayer perceptrons. Therefore, at each time step t the current state was again represented by a 2×100 matrix $\boldsymbol{M}$. As in the second experiment, the first row of $\boldsymbol{M}$ was defined by the same mapping g as in equation 4.12 and represented the agent's position in the grid. However, the second row of $\boldsymbol{M}$ was defined by the mapping,

$$h(c) = \begin{cases} 1, & \text{if } c \in P \\ 0, & \text{otherwise} \end{cases} \tag{4.18}$$

where $c \in \{1, 2, \ldots, 100\}$ is the column number of $\boldsymbol{M}$ and the set P is defined as,

$$P = \{f(i, j) \mid \text{cell } (i, j) \text{ contains an active package}\} \tag{4.19}$$

where the mapping f is defined in the same way as in equation 4.11. Therefore, the second row of $\boldsymbol{M}$ represented the position(s) of the active package(s) in the world. Note that this kind of representation is essential for the agent to perform well. Since the episode does not end when one package is collected, it is crucial for the agent to understand that a package is not active anymore in a certain cell after it has taken the action "collect" in that cell once. This allows the agent to visit other cells in order to collect the remaining packages and end the episode.

4.3.3 Optimal Values

Since the reward function is deterministic and the optimal policy π_* ends the episode in 32 actions, it follows that the optimal expected reward per time step is,

$$\frac{\sum_{t=0}^{31} \mathbb{E}_{\pi_*}[R_{t+1}]}{32} = \frac{100 + 31(-1)}{32} \approx 2.16 \tag{4.20}$$

Assuming that the discount factor γ is set to 0.95, the maximum optimal action value of all visited states per time step is,

$$\frac{\sum_{t=0}^{31} \max_{a \in \mathcal{A}} q_*(S_t, a)}{32} = \frac{\sum_{t=0}^{31} \mathbb{E}_{\pi_*}[G_t]}{32} \approx 40.47 \tag{4.21}$$

Under the same assumption, the maximum optimal action value of the starting state s_0 is,

$$\max_{a \in \mathcal{A}} q_*(s_0, a) = 100\gamma^{31} - \sum_{k=0}^{30} \gamma^k \approx 4.47 \tag{4.22}$$

4.3.4 Hyperparameters

In table 4.3 we show the hyperparameters that were used to run the experiment with this world. As in the second experiment, we determined the hyperparameters manually, all hyperparameters other than the variation resistance parameter were fixed and used in all three algorithms, and the variation resistance parameter was varied. Furthermore, the policy used for the action selection, the linear annealing procedure of the exploration parameter ϵ, the activation functions in the hidden layers of the multilayer perceptrons, and the initialization of the weights of the multilayer perceptrons were identical to the ones used in the second experiment.

Hyperparameter	**Value(s)**
discount factor	0.95
initial exploration	1
final exploration	0.05
exploration decay steps	750,000
learning rate	0.005
number of hidden layers	1
number of hidden layer nodes	256
variation resistance parameter	-0.5, -0.3, -0.1, +0.2, +0.4, +0.6

Table 4.3: Hyperparameters used in the package grid world experiment.

4.3.5 Evaluation

To evaluate the algorithms, we let each agent interact with the environment and update the weights of its function approximator(s) for 1,000,000 time steps. As in the second experiment, we measured the reward per time step, the maximum action value of all visited states per time step, and the maximum action value of the starting state per time step. We again repeated this procedure five times with five different random seeds and computed the median of each quantity over the seeds. We also computed uncertainty intervals for each quantity in the same way as in the second experiment.

4.4 Implementation Details

The simulation software for our experiments was implemented using Python 3 [43]. For numerical computing we used Numpy [25], and the multilayer perceptrons were

implemented using Keras [7] running on top of Tensorflow [1]. To carry out the experiments we used Peregrine, the high performance computing cluster of the University of Groningen, and ran all the simulations on CPU. To control the randomness in the simulations of the second and third experiments, we set a random seed in any part of the program that involved random operations and ran the tensorflow operations on a single CPU thread.

The source code of the simulation software can be found at https://github.com/anpenta/overestimation-bias-reinforcement-learning-simulation-code

Chapter 5

Results

In this chapter we present the results of our experiments and discuss our findings. In the first section we present and discuss the results of the grid world experiment. In the second section we present and discuss the results of the grid world with function approximation experiment. In the third and final section we present and discuss the results of the package grid world experiment. Note that in the figures of this chapter we use the abbreviations shown in table 5.1.

Algorithm	**Abbreviation**
Double Q-learning	DQL
Q-learning	QL
Variation-resistant Q-learning	VRQL

Table 5.1: Abbreviations used in the figures of this chapter.

5.1 Grid World

In figure 5.1 we show the results obtained when the variation resistance parameter was set to -3. The reward per time step is shown in the top row, the maximum action value of the starting state is shown in the middle row, and the normalized entropy of the state visits is shown in the bottom row. The plots in each column correspond to a different reward function, and the optimal value is marked with a black horizontal line in the plots of the top and middle rows. The quantities were averaged over 10,000 simulations.

When the Bernoulli reward function was used, Q-learning performed poorly, because the highly stochastic rewards received for all actions in the non-goal states caused the algorithm to often overestimate the optimal action values of the non-goal states that correspond to suboptimal actions. Double Q-learning did not perform much better than Q-learning, because the highly stochastic rewards received for all actions in the goal state caused the algorithm to often select suboptimal actions in the non-goal states. The reason is that there was no incentive for the algorithm to explore the goal state and learn its optimal action values. Consequently, both algorithms ignored the goal state and followed bad policies for many steps. On the other hand, Variation-resistant Q-learning performed well, because in the beginning of learning the action-value estimates of the algorithm for the non-goal states were updated with targets that contained uncertain action-value estimates.

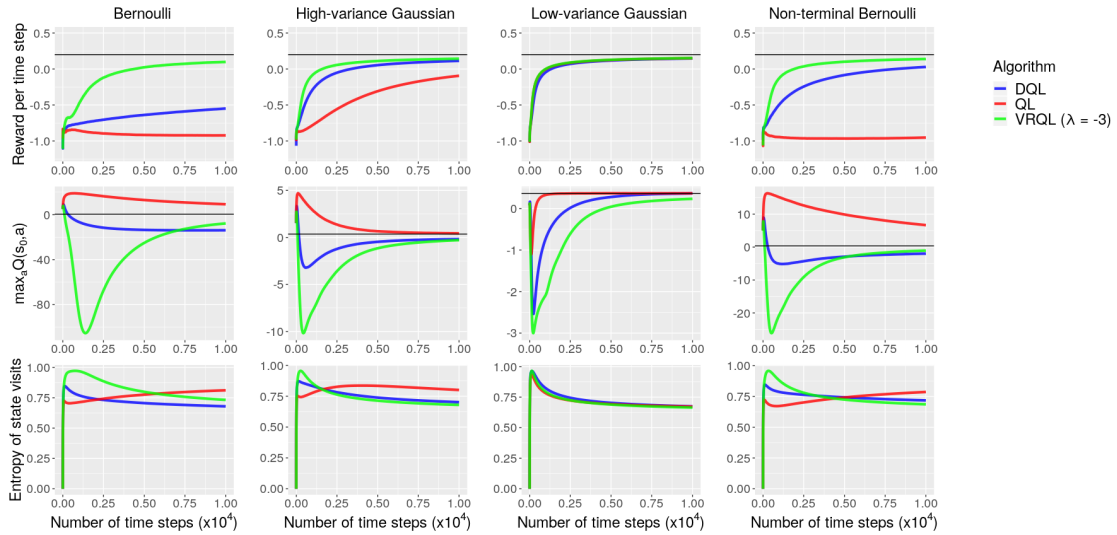


Figure 5.1: Results obtained from the grid world experiment when the variation resistance parameter was set to -3. The reward per time step is shown in the top row, the maximum action value of the starting state is shown in the middle row, and the normalized entropy of the state visits is shown in the bottom row. Each column corresponds to a different reward function. The optimal values are marked with black horizontal lines in the plots of the top and middle rows. The quantities were averaged over 10,000 simulations.

The algorithm translated the uncertain action-value estimates in the update targets using negative variation quantities that were relatively high in magnitude. Consequently, the algorithm visited the goal state many times, learned its optimal action values, and followed good policies for many steps. Notice that Variation-resistant Q-learning underestimated the maximum optimal action value of the starting state extremely in the beginning of learning but computed estimates close to the optimal value near the end of learning. Moreover, the normalized entropy of the state visits indicates that the algorithm did not overexplore any non-goal state in the beginning of learning.

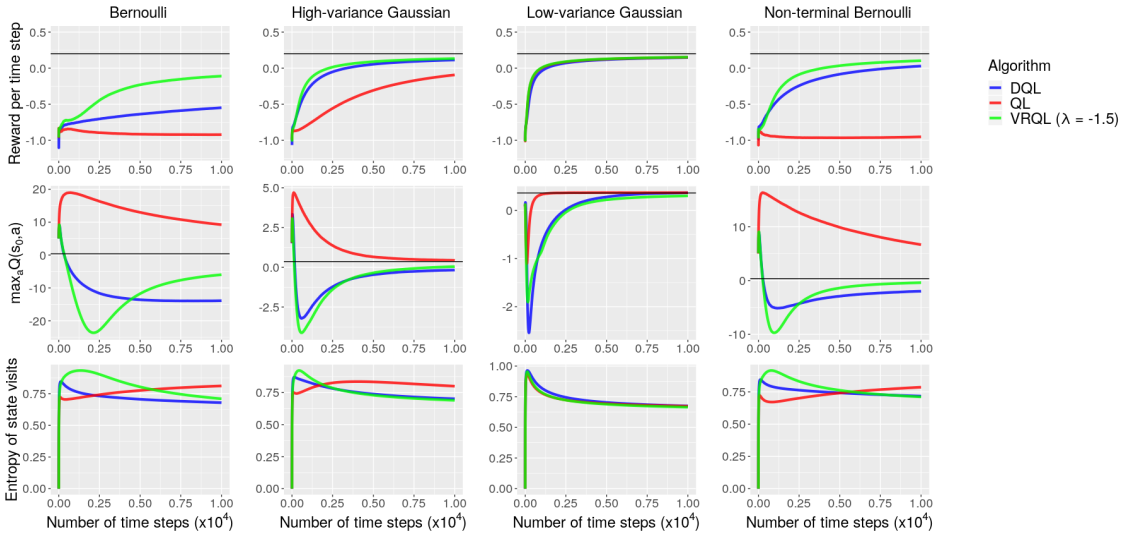
When the High-variance Gaussian reward function was used, Q-learning performed worse than the other two algorithms for the same reason as in the case of the Bernoulli reward function. On the other hand, Double Q-learning performed well because the rewards received for all actions in the goal state were deterministic. Since the algorithm could not overestimate the optimal action values of the non-goal states that correspond to suboptimal actions, it often selected the optimal actions in the non-goal states and followed good policies for many steps. Variation-resistant Q-learning performed slightly better than Double Q-learning for the same reason as in the case of the Bernoulli reward function.

The Low-variance Gaussian reward function was the most favorable for all three algorithms. Although the rewards received for all actions in the non-goal states were stochastic, their variance was not high enough to confuse the algorithms. Therefore, all three algorithms followed good policies for many steps and performed well.

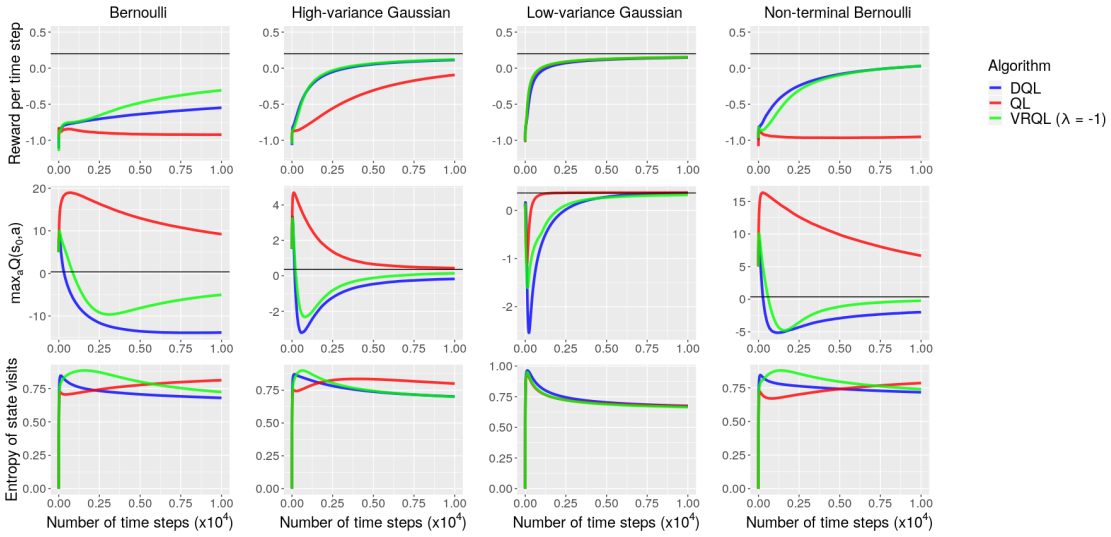
When the Non-terminal Bernoulli reward function was used, Q-learning performed poorly for the same reason as in the case of the Bernoulli reward function, whereas Double Q-learning performed well for the same reason as in the case of the

High-variance Gaussian reward function. Variation-resistant Q-learning performed moderately better than Double Q-learning for the same reason as in the case of the Bernoulli reward function.

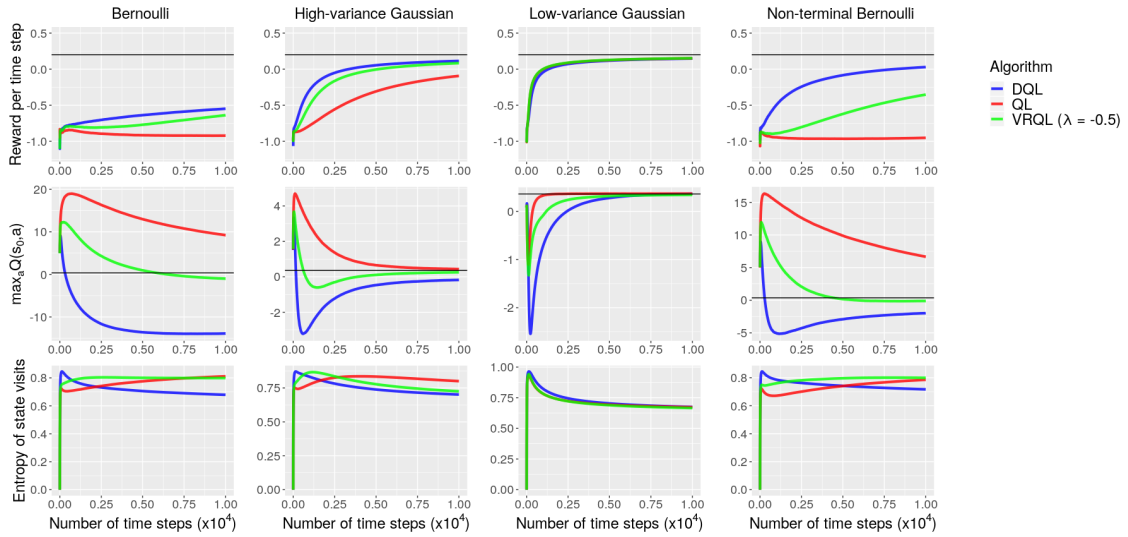
In the following figures we show the results obtained when the variation resistance parameter λ was set to the other five values mentioned in table 4.1. Notice that the estimates of Variation-resistant Q-learning for the maximum optimal action value of the starting state gradually move from underestimation to overestimation as λ increases. Notice also that the performance of the algorithm gradually becomes worse as λ increases. Additionally, notice that the normalized entropy of the state visits that corresponds to Variation-resistant Q-learning gradually decreases as λ increases. This indicates that the algorithm computed higher estimates for the optimal action values of the non-goal states and explored the non-goal states for more steps when λ was set to higher values. Therefore, the results of the first experiment support the claim that λ can control and utilize the estimation bias of the algorithm in order to influence the agent's exploration behavior for better performance.



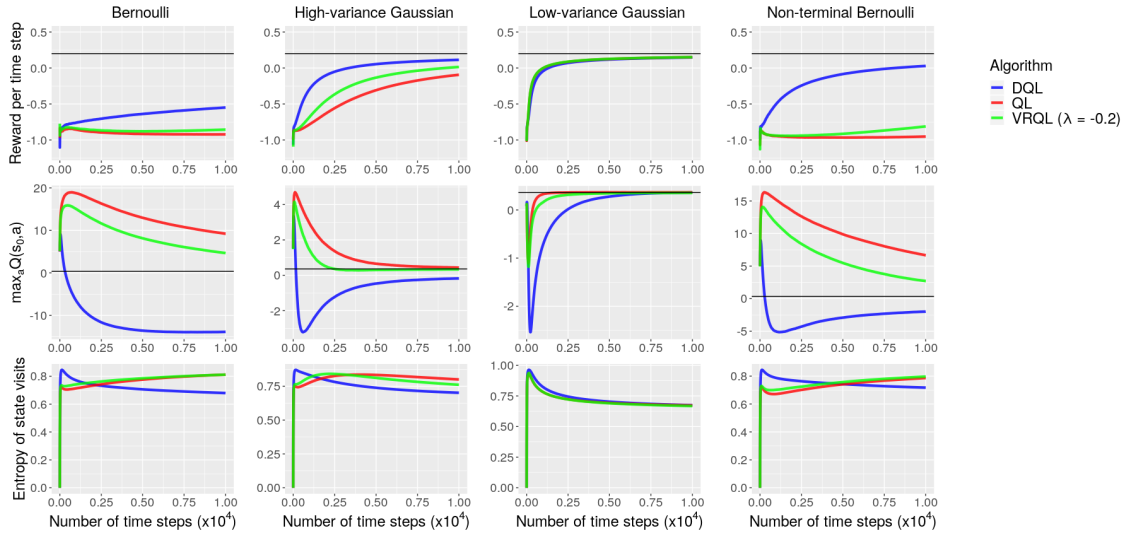
(a) λ = -1.5



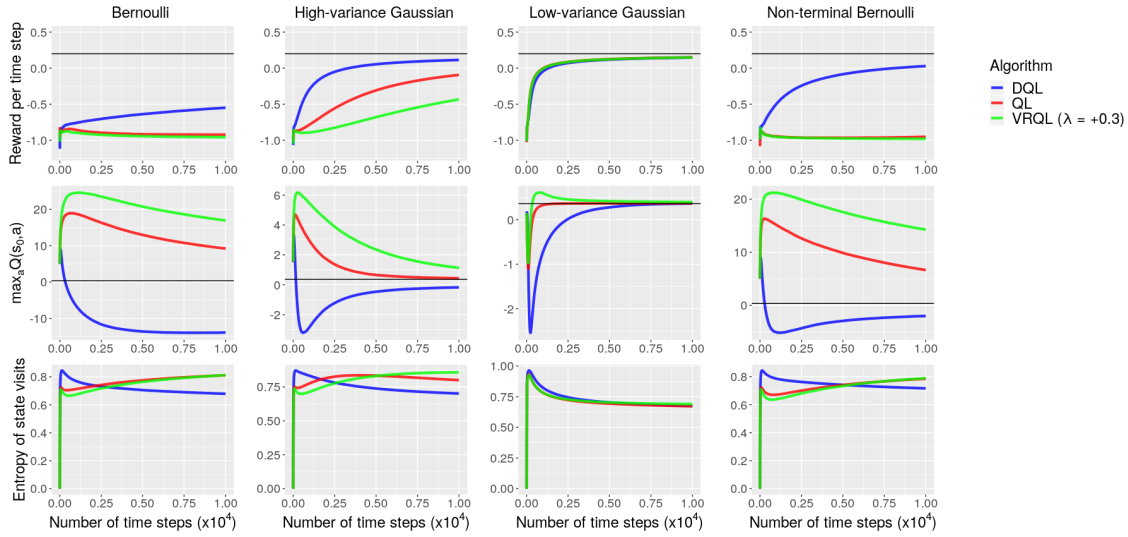
(b) λ = -1



(c) λ = -0.5



(d) λ = -0.2



(e) λ = +0.3

5.2 Grid World with Function Approximation

In figure 5.2 we show the results obtained when the variation resistance parameter was set to -0.5. The reward per time step is shown in the top row, the maximum action value of all visited states per time step is shown in the middle row, and the maximum action value of the starting state per time step is shown in the bottom row. The plots in each column correspond to a different reward function, and the optimal value is marked with a black horizontal line in each plot. The quantities are median values over five simulations with five different random seeds, and the uncertainty intervals for each quantity are represented by shaded areas.

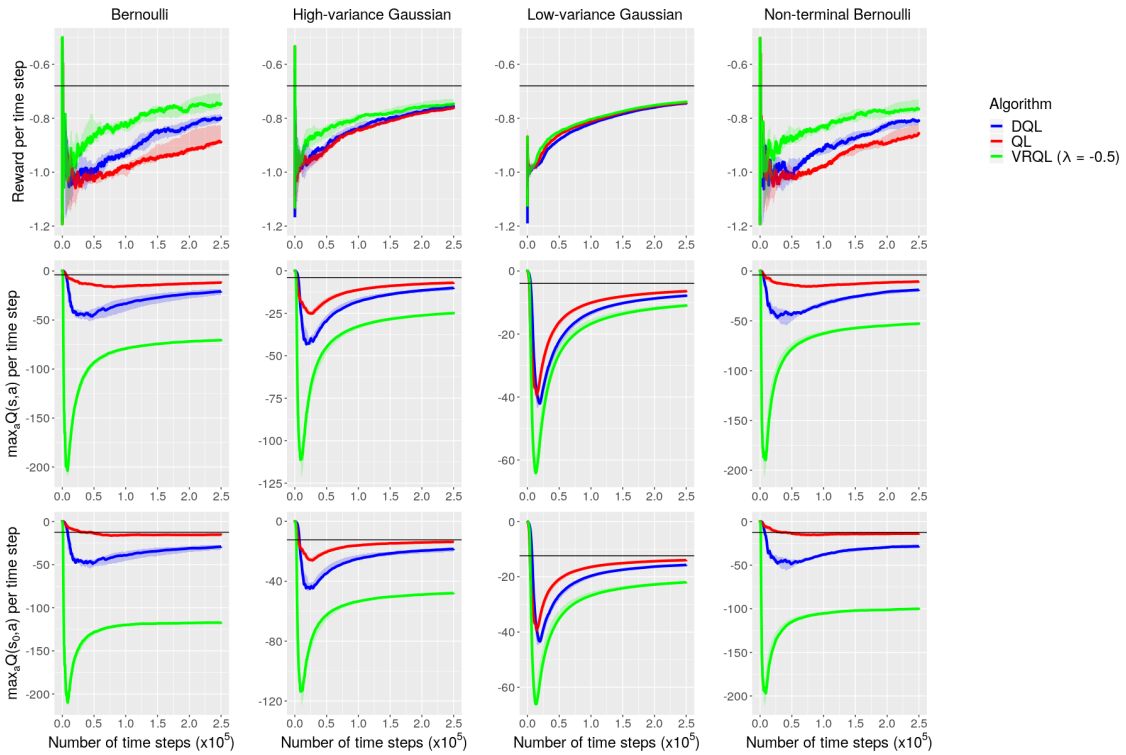


Figure 5.2: Results obtained from the grid world with function approximation experiment when the variation resistance parameter was set to -0.5. The reward per time step is shown in the top row, the maximum action value of all visited states per time step is shown in the middle row, and the maximum action value of the starting state per time step is shown in the bottom row. Each column corresponds to a different reward function. The optimal values are marked with black horizontal lines. The quantities are median values over five simulations with five different random seeds, and the shaded areas represent the uncertainty intervals.

Notice that the performance of all three algorithms fluctuated greatly in the beginning of learning. The reasons are that the reward per time step was computed with a limited amount of reward samples in the beginning of learning and that the median values were computed over only five simulations.

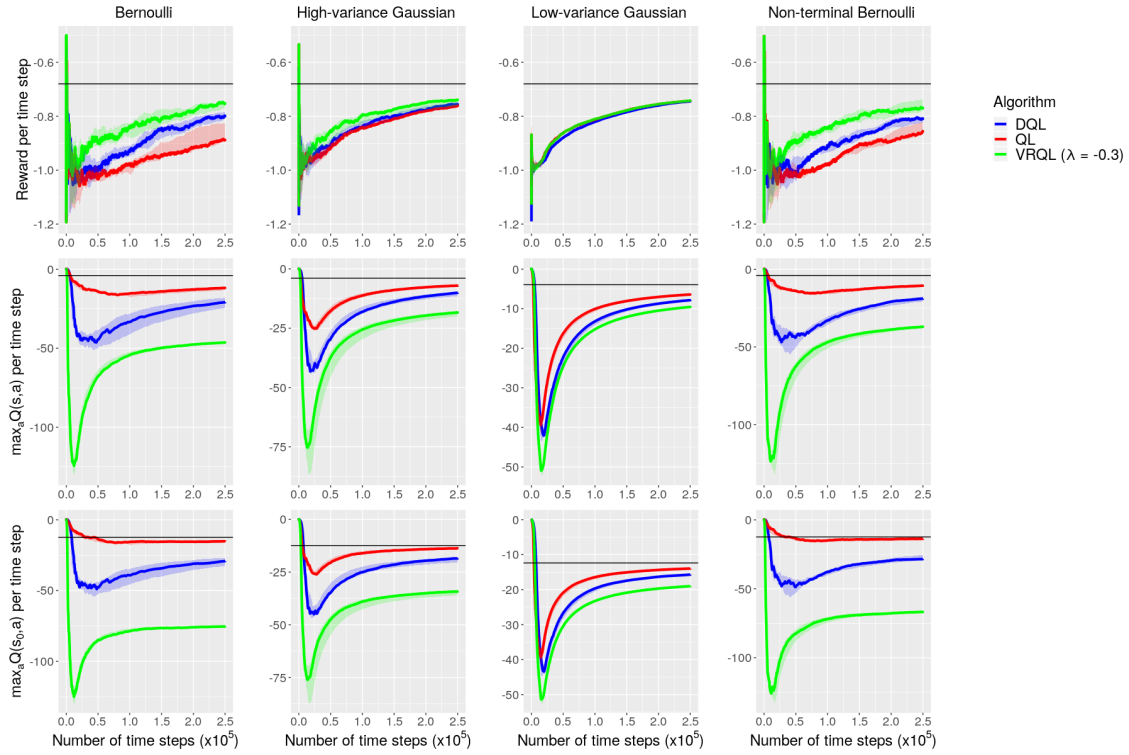
Note that in the second experiment Variation-resistant Q-learning and Double Q-learning were at a disadvantage compared to Q-learning. In the case of Variation-resistant Q-learning the reason is that the algorithm had to allocate part of the capacity of its multilayer perceptron to predict the absolute deviations of the action-value estimates, whereas in the case of Double Q-learning the reason is that the

algorithm updated the weight vector of only one of its two multilayer perceptrons in each step. Nevertheless, Variation-resistant Q-learning performed better than Double Q-learning, and Double Q-learning performed better than Q-learning. Note that this difference in performance was more extreme when the Bernoulli and Non-terminal Bernoulli reward functions were used. This happened for the same reasons we discussed in the previous section (see figure 5.1 and relevant discussion).

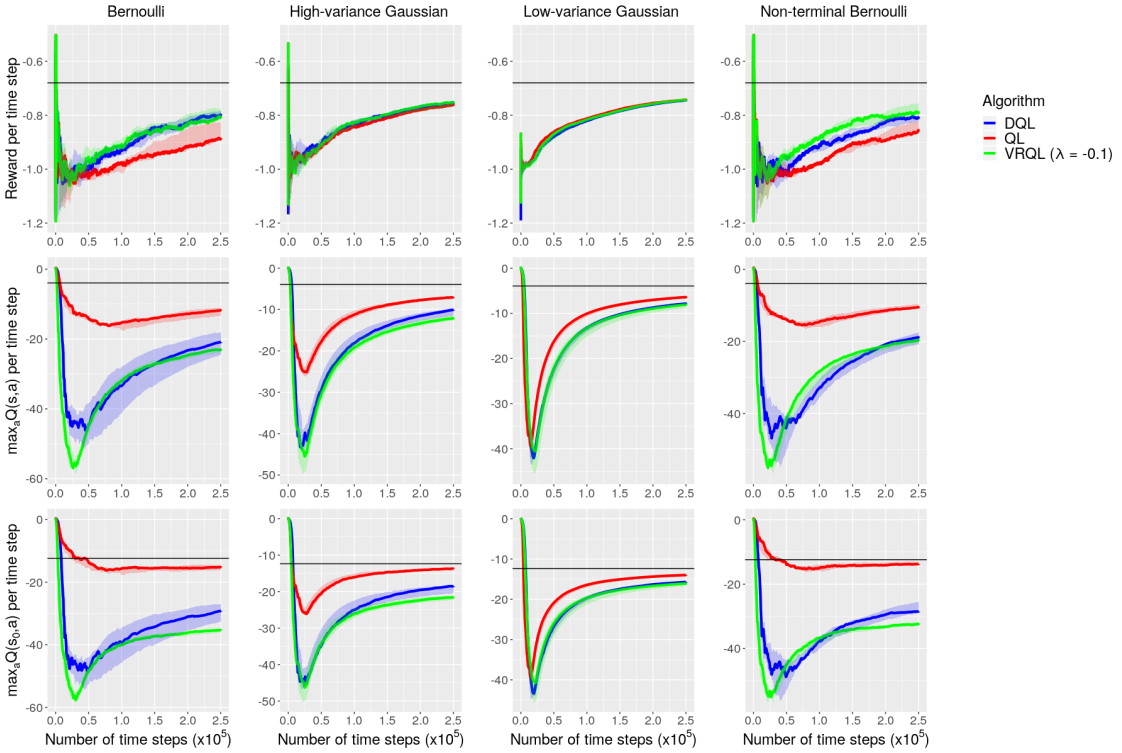
Notice that Double Q-learning performed better than in the first experiment when the Bernoulli reward function was used. Similarly, Q-learning performed better than in the first experiment when the Bernoulli, Non-terminal Bernoulli, and High-variance Gaussian reward functions were used. Note that this happened despite the fact that the task of the second experiment was more difficult than the task of the first experiment. The reason is that the multilayer perceptrons generalized over the state space and tried to learn the mean value of the update targets for each action irrespective of the state.

Because of the behavior of the multilayer perceptrons, all three algorithms underestimated the maximum optimal action values. Nevertheless, Variation-resistant Q-learning underestimated the optimal values more than Double Q-learning, and Double Q-learning underestimated the optimal values more than Q-learning. Notice that the estimates of Variation-resistant Q-learning did not get close to the optimal values within the 250,000 time steps. Underestimation was positively correlated with performance as expected.

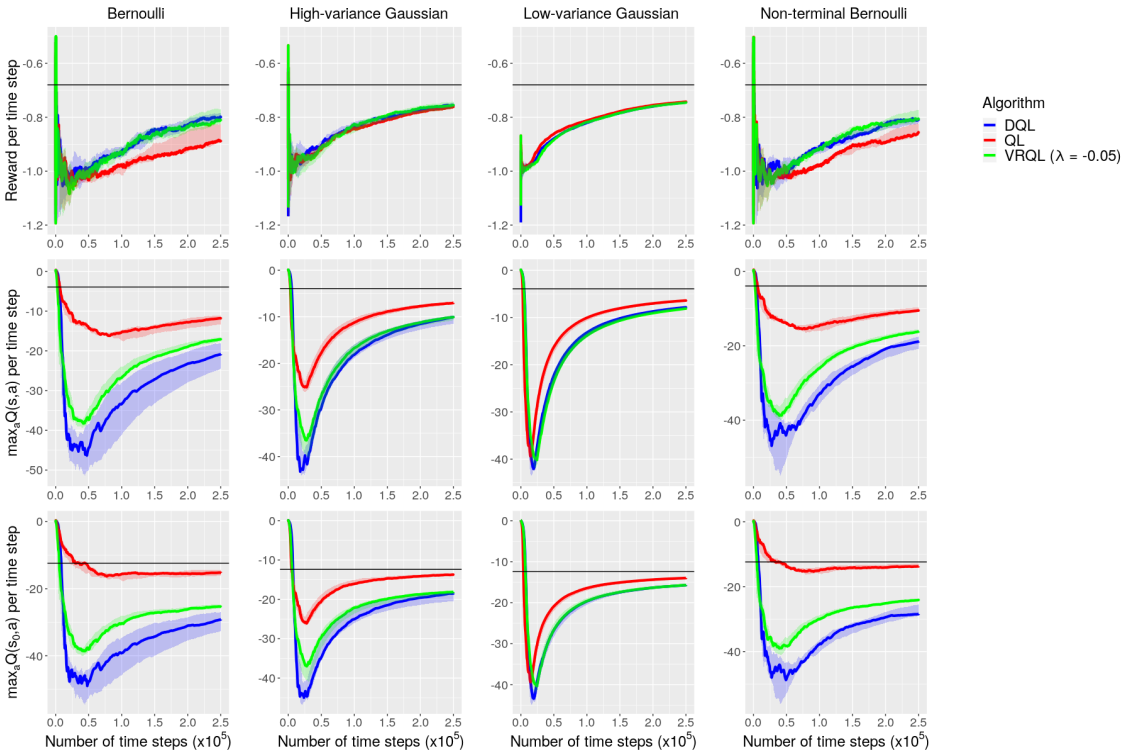
In the following figures we show the results obtained when the variation resistance parameter λ was set to the other five values mentioned in table 4.2. The results of the second experiment are similar to the results of the first experiment and support the claim that λ can control and utilize the estimation bias of Variation-resistant Q-learning for better performance.



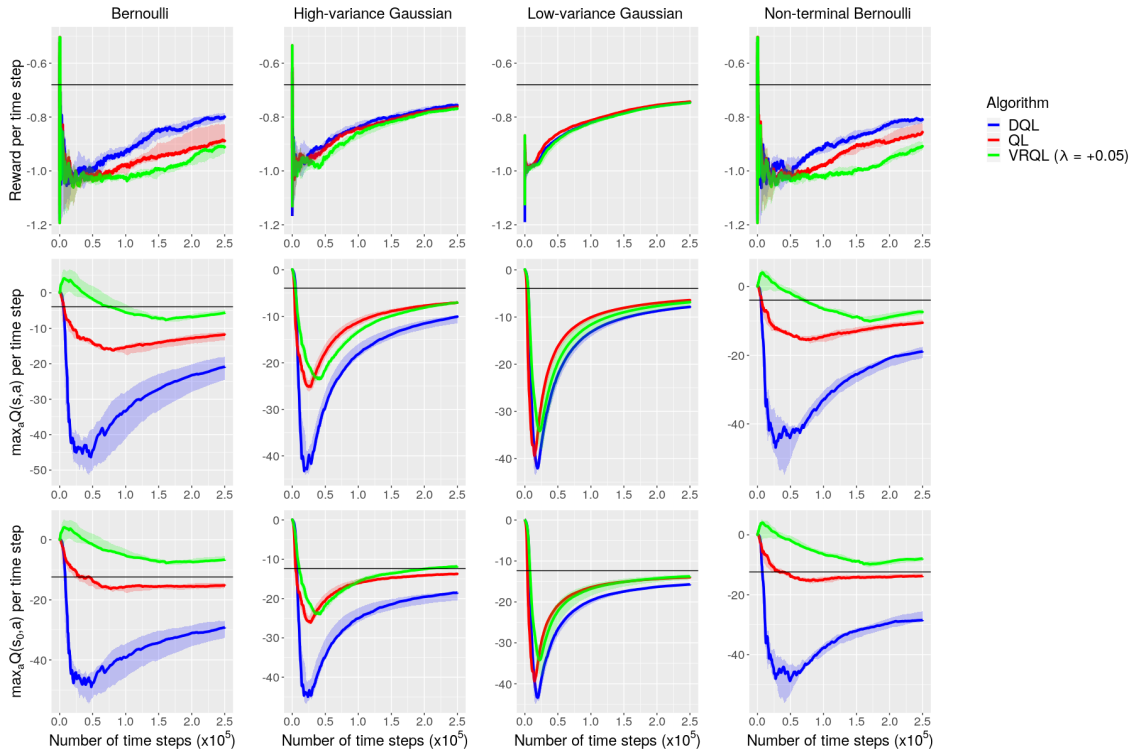
(a) λ = -0.3



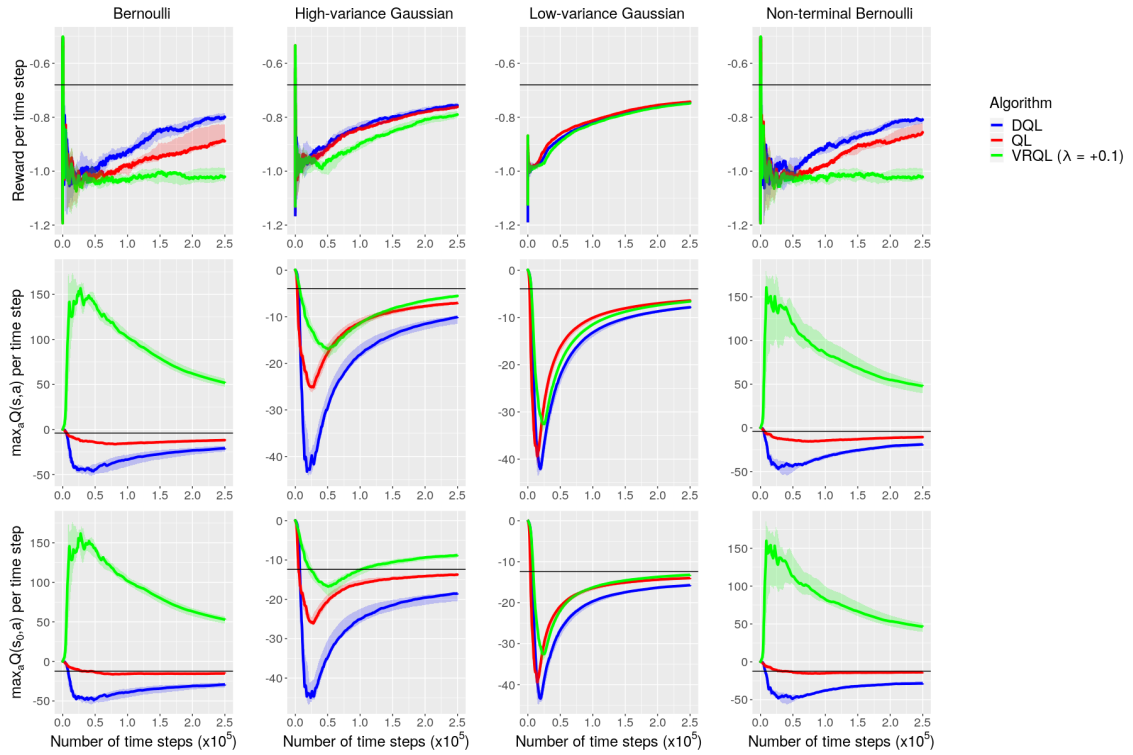
(b) λ = -0.1



(c) λ = -0.05



(d) $\lambda = +0.05$



(e) $\lambda = +0.1$

Note that in the second experiment the variation resistance parameter λ was set to lower values in magnitude than in the first experiment. The reason is that preliminary experiments showed that setting λ to a value greater than one in magnitude causes the function approximation version of Variation-resistant Q-learning to diverge. Notice that the estimates of the algorithm for the maximum optimal action

values change more extremely as a function of λ in the results of the second experiment than in the results of the first experiment. This suggests that the algorithm is more sensitive to the value of λ when function approximation is used.

5.3 Package Grid World

In figure 5.3 we show the results obtained when the variation resistance parameter was set to +0.6. The reward per time step is shown in the left plot, the maximum action value of all visited states per time step is shown in the center plot, and the maximum action value of the starting state per time step is shown in the right plot. The optimal value is marked with a black horizontal line in each plot. The quantities are median values over five simulations with five different random seeds, and the uncertainty intervals for each quantity are represented by shaded areas.

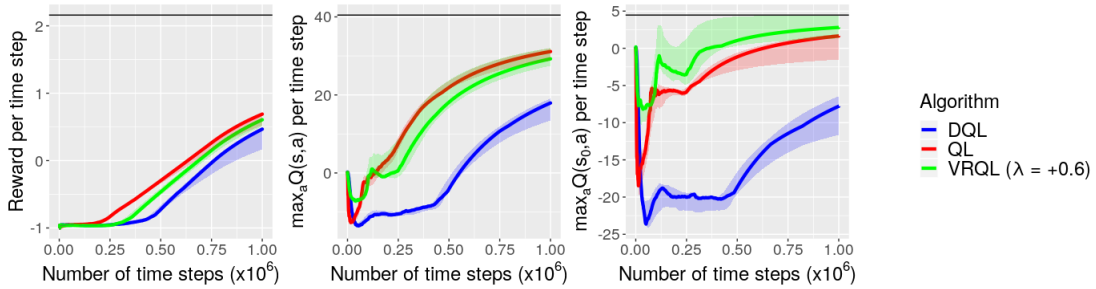


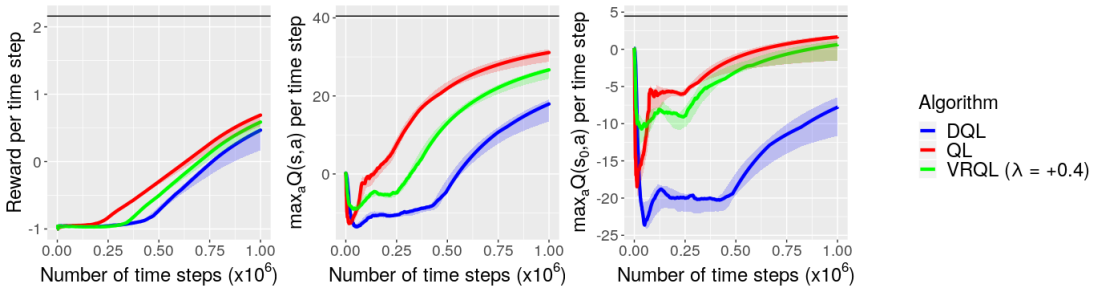
Figure 5.3: Results obtained from the package grid world experiment when the variation resistance parameter was set to +0.6. The reward per time step is shown in the left plot, the maximum action value of all visited states per time step is shown in the center plot, and the maximum action value of the starting state per time step is shown in the right plot. The optimal values are marked with black horizontal lines. The quantities are median values over five simulations with five different random seeds, and the shaded areas represent the uncertainty intervals.

Note that in the third experiment Variation-resistant Q-learning and Double Q-learning were at a disadvantage compared to Q-learning for the same reasons as in the second experiment. This allowed Q-learning to achieve superior performance in the task of the third experiment. The reason is that in this task it is relatively difficult to discover the terminal state, and therefore experiences with the terminal state are relatively difficult to sample. Q-learning utilized those experiences better and therefore followed good policies for more steps that the other two algorithms.

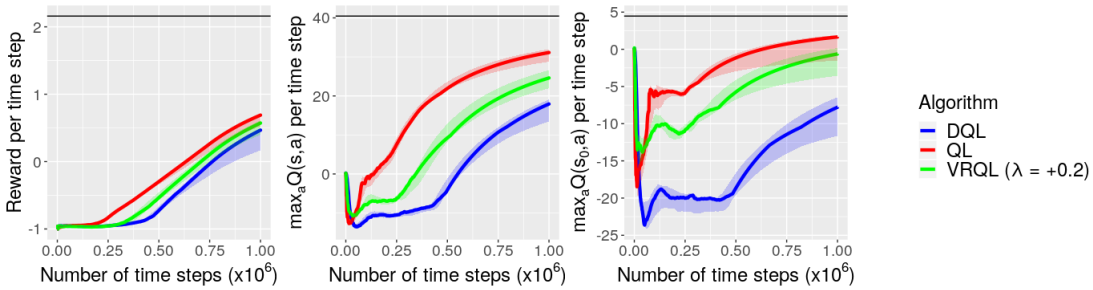
Notice that Double Q-learning performed worse than the other two algorithms. The reason is that the task of the third experiment does not favor underestimation. As we mentioned above, in this task it is relatively difficult to sample experiences with the terminal state. Furthermore, the initial states are relatively far from the terminal state, and therefore the optimal action values of the initial states that correspond to optimal and suboptimal actions do not differ extremely. Double Q-learning computed lower estimates for the optimal action values that correspond to optimal actions than the other two algorithms in the beginning of learning. This indicates that Double Q-learning explored suboptimal actions and followed bad policies for more steps than the other two algorithms, and that Double Q-learning needed more

experiences with the terminal state to determine the optimal actions than the other two algorithms.

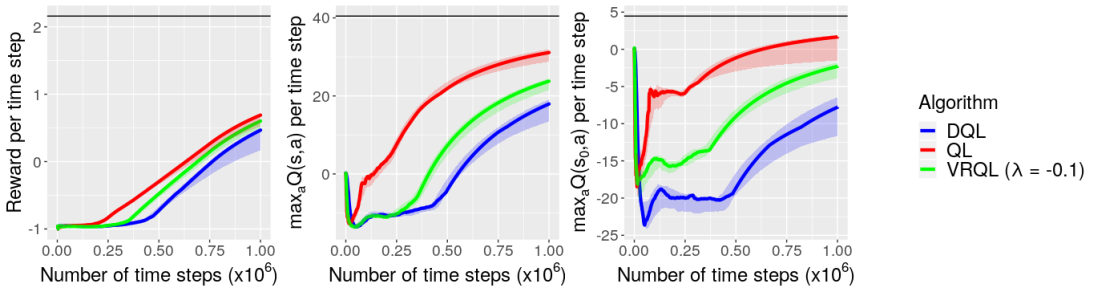
In the following figures we show the results obtained when the variation resistance parameter λ was set to the other five values mentioned in table 4.3. Notice that the estimates of Variation-resistant Q-learning for the maximum optimal action values gradually decrease as λ decreases. Notice also that the performance of the algorithm gradually becomes worse as λ decreases. This indicates that the algorithm behaved similarly to Double Q-Learning when λ was set to lower values. Therefore, the results of the third experiment support the claim that λ can control and utilize the estimation bias of Variation-resistant Q-learning for better performance.



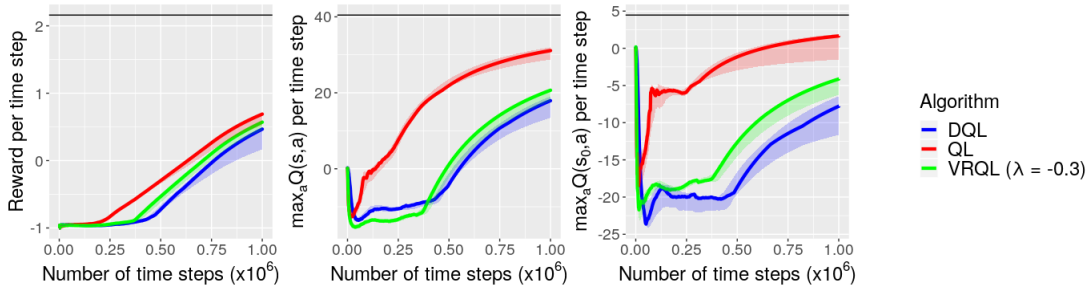
(a) $\lambda = +0.4$



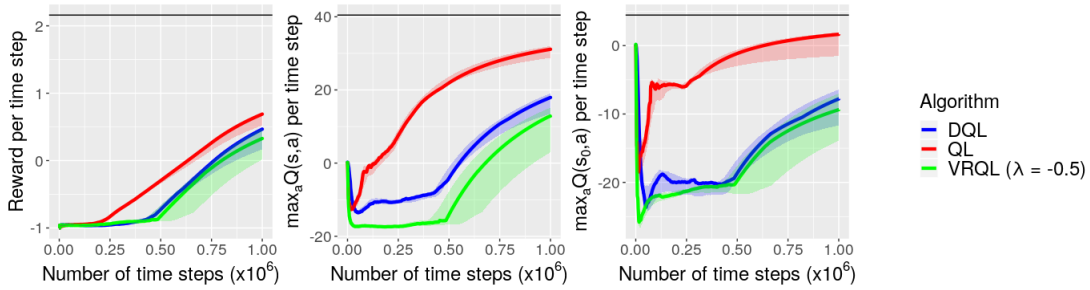
(b) $\lambda = +0.2$



(c) $\lambda = -0.1$



(d) λ = -0.3



(e) λ = -0.5

Notice that the uncertainty intervals for the quantities that correspond to Variation-resistant Q-learning are wider in the results obtained when λ was set to +0.6 and -0.5. This behavior is similar to the one we observed in the preliminary experiments that we mentioned in the previous section and suggests that the algorithm is more sensitive to the value of λ when function approximation is used.

Chapter 6

Discussion

In this chapter we conclude this thesis with a discussion and the answers to the research questions that were presented in section 1.1. In the first section we answer the research questions. In the second section we outline some possible directions for future work. Finally, in the third section we conclude this chapter and this thesis.

6.1 Answers to Research Questions

Under which conditions is overestimation bias harmful for reinforcement learning algorithms?

Based on our investigation, we concluded that overestimation bias is harmful for reinforcement learning algorithms when it encourages exploration of suboptimal actions. In the second chapter we presented an episodic finite Markov decision process where an algorithm that has overestimation bias and follows an ϵ-greedy policy would potentially not perform well, because it would be encouraged to explore suboptimal actions in the beginning of learning (see figure 2.4 and relevant discussion). Moreover, overestimation bias was harmful for Q-learning in the tasks of our first and second experiments. We explained that this happened because Q-learning overestimated the optimal action values that correspond to suboptimal actions and followed bad policies for more steps than the other two algorithms.

How can overestimation bias be reduced or removed when it is harmful for reinforcement learning algorithms?

During our investigation, we identified methods that reduce or remove overestimation bias and methods that control estimation bias. In principle, all these methods can be used to reduce or remove overestimation bias when it is harmful for reinforcement learning algorithms. However, Double Q-learning is the most well-tested and successful method to overcome the problems caused by overestimation bias. This does not mean that the other methods do not have appropriate use cases, but we need more empirical results to conclude that they are useful alternatives to Double Q-learning. In this thesis we proposed Variation-resistant Q-learning as a method to control and utilize estimation bias for better performance. We claimed that the method can either reduce or remove overestimation bias depending on the value of the variation resistance parameter and provided empirical results that support this

claim. Moreover, Variation-resistant Q-learning achieved superior performance in the tasks of our first and second experiments, in which overestimation bias is harmful. Nevertheless, we need more empirical results to conclude that the algorithm is a useful alternative to Double Q-learning.

Are there any conditions under which overestimation bias is desirable for reinforcement learning algorithms?

Based on our investigation, we concluded that overestimation bias is desirable for reinforcement learning algorithms when it encourages exploration of optimal actions. In the second chapter we presented an episodic finite Markov decision process where an algorithm that has overestimation bias and follows an ϵ-greedy policy would potentially perform well, because it would be encouraged to explore optimal actions in the beginning of learning (see figure 2.5 and relevant discussion). Furthermore, Q-learning and Variation-resistant Q-learning outperformed Double Q-learning in the task of our third experiment when the variation resistance parameter was set to higher values. Since Q-learning and Variation-resistant Q-learning computed higher estimates for the optimal action values that correspond to optimal actions, it seems that the two algorithms repeated the optimal actions more frequently and followed good policies for more steps than Double Q-learning. Moreover, when the variation resistance parameter was set to lower values in the same task, Variation-resistant Q-learning performed similarly to Double Q-learning, and the estimates of the two algorithms for the maximum optimal action values were approximately the same.

Assuming that there are conditions under which overestimation bias is desirable for reinforcement learning algorithms, how can overestimation bias be controlled and utilized for better performance?

During our investigation, we identified methods that control not only overestimation bias but also underestimation bias. We mentioned the limitations of these methods and proposed Variation-resistant Q-learning as an alternative. We claimed that the new method can control and utilize estimation bias for better performance and provided empirical results that support this claim. Unlike the other methods that control estimation bias, Variation-resistant Q-learning translates the values of uncertain action-value estimates in the update targets to influence the agent's exploration behavior in a more direct way. Moreover, the algorithm can arbitrarily overestimate more than Q-learning and can arbitrarily underestimate more than Double Q-learning. However, more empirical results are needed to determine the usefulness of Variation-resistant Q-learning compared to the other methods.

6.2 Future Work

One direction for future work would be to provide a better theoretical framework for Variation-resistant Q-learning. Although the algorithm seems to have behaved as expected in practice, there are limited theoretical guarantees for this method. Specifically, we think that it would be desirable to prove that the variation resistance parameter can control the estimation bias of the algorithm. Moreover, we think that the properties of the method should be analyzed further to determine

its advantages, limitations, and appropriate use cases. Additionally, we think that it is worth analyzing how function approximation affects the method, because it seems that the function approximation version of the algorithm is not as stable as its tabular version.

Another direction for future work would be to conduct more experiments in order to evaluate Variation-resistant Q-learning further. It would be interesting to test the performance of the algorithm on large-scale tasks (e.g. in the video game domain) or tasks that involve non-stationary environments. Moreover, we think that it is necessary to conduct experiments in which the tabular version of the algorithm would be tested on tasks that favor overestimation. This would remove the approximation error that is generated from function approximation and would allow for better conclusions to be made.

An additional direction for future work would be to improve Variation-resistant Q-learning. We think that it is worth investigating how the algorithm behaves when measures of statistical dispersion other than mean absolute deviation are used (e.g. median absolute deviation). Furthermore, it would be desirable to determine the variation resistance parameter automatically during learning without the need to set a value before the algorithm is executed.

One more direction for future work would be to develop an entirely new method that is based on the idea of controlling and utilizing estimation bias for better performance. An example would be to develop a method that can arbitrarily switch between overestimation and underestimation during learning. We think that such a method would be applicable to more use cases than Variation-resistant Q-learning.

6.3 Conclusion

In this thesis we investigated overestimation bias in reinforcement learning. The main conclusion of our investigation is that overestimation bias can have either a negative or positive effect on reinforcement learning algorithms depending on the reinforcement learning problem. Based on this conclusion, we proposed a new method to control and utilize estimation bias for better performance. Furthermore, we proved a convergence theorem for the tabular version of the new method and presented empirical results that indicate that the new method behaves as expected in practice.

Our purpose was to provide a better understanding of the effect of overestimation bias on reinforcement learning algorithms. Since overestimation bias was identified in the literature, it has been generally considered to have only a negative effect on reinforcement learning algorithms. We hope that this thesis will inspire the reinforcement learning community to reconsider the role of overestimation bias in reinforcement learning problems and investigate this topic further.

Bibliography

[1] Martın Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[2] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[3] Oron Anschel, Nir Baram, and Nahum Shimkin. "Averaged-DQN: Variance reduction and stabilization for deep reinforcement learning". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 176–185.

[4] Marc G Bellemare, Will Dabney, and Rémi Munos. "A distributional perspective on reinforcement learning". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 449–458.

[5] Richard Bellman. "Dynamic programming and stochastic control processes". In: *Information and control* 1.3 (1958), pp. 228–239.

[6] Dimitri P Bertsekas and John N Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

[7] François Chollet et al. *Keras*. https://keras.io. 2015.

[8] Carlo D'Eramo, Marcello Restelli, and Alessandro Nuara. "Estimating maximum expected value through gaussian approximation". In: *International Conference on Machine Learning*. 2016, pp. 1032–1040.

[9] Will Dabney et al. "Distributional reinforcement learning with quantile regression". In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[10] Eyal Even-Dar and Yishay Mansour. "Learning rates for Q-learning". In: *Journal of machine learning Research* 5.Dec (2003), pp. 1–25.

[11] Meire Fortunato et al. "Noisy networks for exploration". In: *arXiv preprint arXiv:1706.10295* (2017).

[12] Scott Fujimoto, Herke van Hoof, and David Meger. "Addressing function approximation error in actor-critic methods". In: *arXiv preprint arXiv:1802.09477* (2018).

[13] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[15] Geoffrey J Gordon. "Stable function approximation in dynamic programming". In: *Machine Learning Proceedings 1995*. Elsevier, 1995, pp. 261–268.

[16] Hado Philip van Hasselt. "Insights in reinforcement learning". PhD thesis. Utrecht University, 2011.

[17] Matteo Hessel et al. "Rainbow: Combining improvements in deep reinforcement learning". In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[18] Tommi Jaakkola, Michael I Jordan, and Satinder P Singh. "Convergence of stochastic iterative dynamic programming algorithms". In: *Advances in neural information processing systems*. 1994, pp. 703–710.

[19] Qingfeng Lan et al. "Maxmin Q-learning: Controlling the Estimation Bias of Q-learning". In: *arXiv preprint arXiv:2002.06487* (2020).

[20] Donghun Lee, Boris Defourny, and Warren B Powell. "Bias-corrected Q-learning to control max-operator bias in Q-learning". In: *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. IEEE. 2013, pp. 93–99.

[21] Donghun Lee and Warren B Powell. "Bias-corrected Q-learning with multi-state extension". In: *IEEE Transactions on Automatic Control* 64.10 (2019), pp. 4011–4023.

[22] A. LeNail. "NN-SVG: Publication-Ready Neural Network Architecture Schematics." In: *The Journal of Open Source Software* 4 (2019), p. 747.

[23] Marvin Minsky and Seymour Papert. "An introduction to computational geometry". In: *Cambridge tiass., HIT* (1969).

[24] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), p. 529.

[25] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[26] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[27] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[28] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, Department of Engineering Cambridge, England, 1994.

[29] Matthia Sabatelli et al. "Approximating two value functions instead of one: towards characterizing a new family of Deep Reinforcement Learning algorithms". In: *arXiv preprint arXiv:1909.01779* (2019).

[30] Matthia Sabatelli et al. "Deep quality-value (DQV) learning". In: *arXiv preprint arXiv:1810.00368* (2018).

[31] Arthur L Samuel. "Some studies in machine learning using the game of checkers. II—Recent progress". In: *IBM Journal of research and development* 11.6 (1967), pp. 601–617.

[32] Tom Schaul et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[33] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), p. 484.

[34] Satinder Singh et al. “Convergence results for single-step on-policy reinforcement-learning algorithms”. In: *Machine learning* 38.3 (2000), pp. 287–308.

[35] James E Smith and Robert L Winkler. “The optimizer’s curse: Skepticism and postdecision surprise in decision analysis”. In: *Management Science* 52.3 (2006), pp. 311–322.

[36] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[37] Csaba Szepesvári and Michael L Littman. “Generalized Markov decision processes: Dynamic-programming and reinforcement-learning algorithms”. In: *Proceedings of International Conference of Machine Learning*. Vol. 96. 1996.

[38] Gerald Tesauro. “TD-Gammon, a self-teaching backgammon program, achieves master-level play”. In: *Neural computation* 6.2 (1994), pp. 215–219.

[39] Sebastian Thrun and Anton Schwartz. “Issues in using function approximation for reinforcement learning”. In: *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*. 1993.

[40] Hado Van Hasselt. “Double Q-learning”. In: *Advances in Neural Information Processing Systems*. 2010, pp. 2613–2621.

[41] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep reinforcement learning with Double Q-learning”. In: *Thirtieth AAAI conference on artificial intelligence*. 2016.

[42] Hado Van Hasselt et al. “Deep reinforcement learning and the deadly triad”. In: *arXiv preprint arXiv:1812.02648* (2018).

[43] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[44] Ziyu Wang et al. “Dueling network architectures for deep reinforcement learning”. In: *arXiv preprint arXiv:1511.06581* (2015).

[45] Christopher JCH Watkins and Peter Dayan. “Q-learning”. In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[46] Christopher John Cornish Hellaby Watkins. “Learning from delayed rewards”. PhD thesis. King’s College, Cambridge, 1989.

[47] Marco A Wiering. “Explorations in efficient reinforcement learning”. PhD thesis. University of Amsterdam, 1999.

[48] Zongzhang Zhang, Zhiyuan Pan, and Mykel J Kochenderfer. “Weighted Double Q-learning.” In: *IJCAI*. 2017, pp. 3455–3461.

[49] Yan Zheng et al. “Weighted double deep multiagent reinforcement learning in stochastic cooperative environments”. In: *Pacific Rim International Conference on Artificial Intelligence*. Springer. 2018, pp. 421–429.

Appendix A

Convergence of Tabular Variation-resistant Q-learning

In this appendix we present and prove a convergence theorem for tabular Variation-resistant Q-learning. We first present the necessary preliminaries and then present and prove the theorem.

A.1 Preliminaries

Definition A.1.1. *An ergodic Markov decision process is a Markov decision process in which each state can be reached from any other state in a finite number of steps.*

Lemma A.1.1. *Let (β_t, Δ_t, F_t) be a stochastic process, where $\beta_t, \Delta_t, F_t : X \mapsto \mathbb{R}$ satisfy,*

$$\Delta_{t+1}(x_t) = (1 - \beta_t(x_t))\Delta_t(x_t) + \beta_t(x_t)F_t(x_t)$$

where $x_t \in X$ and $t = 0, 1, 2, \ldots$. Let P_t be a sequence of increasing σ-fields such that β_0 and Δ_0 are P_0-measurable and β_t, Δ_t, and F_{t-1} are P_t-measurable, with $t \geq 1$. Assume that the following conditions are satisfied:

1. *The set X is finite (i.e. $|X| < \infty$).*
2. *$\beta_t(x_t) \in [0, 1]$, $\sum_t \beta_t(x_t) = \infty$, $\sum_t \beta_t^2(x_t) < \infty$ w.p.1, and $\forall x \neq x_t : \beta_t(x) = 0$.*
3. *$\|\mathbb{E}\{F_t \mid P_t\}\| \leq \kappa \|\Delta_t\| + c_t$, where $\kappa \in [0, 1)$ and $c_t \to 0$ w.p.1.*
4. *$\mathbb{V}\{F_t(x_t) \mid P_t\} \leq C(1 + \kappa \|\Delta_t\|)^2$, where C is some constant.*

where $\mathbb{V}\{\cdot\}$ denotes the variance and $\|\cdot\|$ denotes the maximum norm. Then Δ_t converges to zero with probability one.

Proof. See [18, 37, 34]. □

A.2 Convergence Theorem

Theorem A.2.1. *In an ergodic Markov decision process, the approximate action-value function Q as updated by tabular Variation-resistant Q-learning in algorithm*

5 *converges to the optimal action-value function* q_* *with probability one if an infinite number of experience tuples of the form* $(S_t, A_t, R_{t+1}, S_{t+1})$ *are sampled by a learning policy for each state-action pair and if the following conditions are satisfied:*

1. *The Markov decision process is finite (i.e.* $|\mathcal{S} \times \mathcal{A} \times \mathcal{R}| < \infty$*).*
2. $\gamma \in [0, 1)$.
3. $\alpha_t(s, a) \in [0, 1]$, $\sum_t \alpha_t(s, a) = \infty$, $\sum_t \alpha_t^2(s, a) < \infty$ *w.p.*1, *and* $\forall s, a \neq S_t, A_t : \alpha_t(s, a) = 0$.

Proof. We apply lemma A.1.1 with $X = \mathcal{S} \times \mathcal{A}$, $\Delta_t = Q_t - q_*$, $\beta_t = \alpha_t$, $P_t = \{Q_0, \sigma_0, S_0, A_0, \alpha_0, R_1, S_1, \ldots, S_t, A_t\}$, and

$$F_t(S_t, A_t) = R_{t+1} + \gamma \left[Q_t(S_{t+1}, A_*) + \lambda\sigma_t(S_{t+1}, A_*)\right] - q_*(S_t, A_t)$$

where $A_* = \arg\max_{a'} Q_t(S_{t+1}, a')$. The first condition of the lemma is satisfied because $|\mathcal{S} \times \mathcal{A}| < \infty$. The second condition of the lemma is satisfied by the third condition of the theorem.

For the fourth condition of the lemma we have that,

$$\begin{aligned}
&\mathbb{E}\left\{F_t(S_t, A_t) \mid P_t\right\} \\
&= \sum_{s'} \sum_{r} p(s', r \mid S_t, A_t) \left[r + \gamma \left[Q_t(s', a_*) + \lambda\sigma_t(s', a_*)\right] - q_*(S_t, A_t)\right] \\
&= \sum_{s'} \sum_{r} p(s', r \mid S_t, A_t) \left[r + \gamma \left[Q_t(s', a_*) + \lambda\sigma_t(s', a_*)\right]\right] - q_*(S_t, A_t)
\end{aligned}$$

where $a_* = \arg\max_{a'} Q_t(s', a')$. Therefore, we have that,

$$\begin{aligned}
&F_t(S_t, A_t) - \mathbb{E}\left\{F_t(S_t, A_t) \mid P_t\right\} \\
&= R_{t+1} + \gamma \left[Q_t(S_{t+1}, A_*) + \lambda\sigma_t(S_{t+1}, A_*)\right] \\
&\quad - \sum_{s'} \sum_{r} p(s', r \mid S_t, A_t) \left[r + \gamma \left[Q_t(s', a_*) + \lambda\sigma_t(s', a_*)\right]\right] \\
&= R_{t+1} + \gamma \left[Q_t(S_{t+1}, A_*) + \lambda\sigma_t(S_{t+1}, A_*)\right] \\
&\quad - \mathbb{E}\left\{R_{t+1} + \gamma \left[Q_t(S_{t+1}, A_*) + \lambda\sigma_t(S_{t+1}, A_*)\right] \mid P_t\right\}
\end{aligned}$$

Therefore, it follows that,

$$\begin{aligned}
\mathbb{V}\left\{F_t(S_t, A_t) \mid P_t\right\} &= \mathbb{E}\left\{\left[F_t(S_t, A_t) - \mathbb{E}\left\{F_t(S_t, A_t) \mid P_t\right\}\right]^2 \mid P_t\right\} \\
&= \mathbb{V}\left\{R_{t+1} + \gamma \left[Q_t(S_{t+1}, A_*) + \lambda\sigma_t(S_{t+1}, A_*)\right] \mid P_t\right\} \\
&\leq C(1 + \gamma \left\|\Delta_t\right\|)^2
\end{aligned}$$

for some constant C, because $|\mathcal{R}| < \infty \implies \forall t : \mathbb{V}\left\{R_{t+1} \mid P_t\right\} < \infty \implies \forall t : \mathbb{V}\left\{R_{t+1} + \gamma \left[Q_t(S_{t+1}, A_*) + \lambda\sigma_t(S_{t+1}, A_*)\right] \mid P_t\right\} < \infty$.

For the third condition of the lemma we have that,

$$\begin{aligned}
F_t(S_t, A_t) &= R_{t+1} + \gamma \left[Q_t(S_{t+1}, A_*) + \lambda\sigma_t(S_{t+1}, A_*)\right] - q_*(S_t, A_t) \\
&= F_t'(S_t, A_t) + \gamma\lambda\sigma_t(S_{t+1}, A_*)
\end{aligned}$$

where $F_t'(S_t, A_t)$ is the value of $F_t(S_t, A_t)$ in the case of Q-learning. Since it is well known that $\forall t : \|\mathbb{E}\{F_t' \mid P_t\}\| \leq \gamma \|\Delta_t\|$, it follows that,

$$\|\mathbb{E}\{F_t \mid P_t\}\| = \|\mathbb{E}\{F_t' \mid P_t\}\| + \gamma\lambda \|\mathbb{E}\{\sigma_t \mid P_t\}\| \leq \gamma \|\Delta_t\| + \gamma\lambda \|\mathbb{E}\{\sigma_t \mid P_t\}\|$$

Therefore, it suffices to show that $c_t = \gamma\lambda \|\mathbb{E}\{\sigma_t \mid P_t\}\| \to 0$ w.p.1.

Since $\forall t, s, a : \sigma_t(s, a) \in [0, \infty)$, it suffices to show that,

$$\lim_{t\to\infty} \sigma_t(S_t, A_t) = 0 \iff \forall \epsilon > 0 \; \exists t_0 : \forall t \geq t_0 \implies \sigma_t(S_t, A_t) < \epsilon$$

Assume that time step t is such that the memory for each action value is full. We have that,

$$\sigma_t(S_t, A_t) = \frac{\sum_{i=1}^{n} \left| Q_{t_i}(S_t, A_t) - \overline{Q}_t(S_t, A_t) \right|}{n}$$

where $t_i < t$, $\forall i = 1, 2, \ldots, n$. After the update at time step t we have that,

$$\sigma_{t+1}(S_t, A_t) = \frac{\sum_{i=2}^{n+1} \left| Q_{t_i}(S_t, A_t) - \overline{Q}_{t+1}(S_t, A_t) \right|}{n}$$

where $t_{(n+1)} = t + 1$. Because of the third condition of the theorem, the differences between the $Q_{t_i}(S_t, A_t)$ values approach zero as $t \to \infty$. Therefore, given $\epsilon > 0$, $\exists t_0 : \forall t \geq t_0 \implies \sigma_t(S_t, A_t) < \epsilon \implies \lim_{t\to\infty} \sigma_t(S_t, A_t) = 0$.

Since all the conditions of lemma A.1.1 are satisfied, it holds that, $\forall s, a$: $Q_t(s, a) \to q_*(s, a)$ w.p.1. □